Making Parallel Programming Easy
Research Contributions from Illinois

Josep Torrellas (I2PC Director)
Sarita V. Adve
Vikram S. Adve
Danny Dig
Minh N. Do
Maria Jesus Garzaran
John C. Hart
Thomas S. Huang
Wen-mei W. Hwu (UPCRC Co-Director)
Samuel T. King
Darko Marinov
Klara Nahrstedt
David A. Padua
Madhusudan Parthasarathy
Sanjay J. Patel
Marc Snir (UPCRC Co-Director)

Illinois Parallelism Center
Department of Computer Science
Department of Electrical and Computer Engineering
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
September 2013

UPCRC Illinois
Universal Parallel Computing Research Center

I2PC
ILLINOIS-INTEL PARALLELISM CENTER (I2PC)
http://i2pc.cs.illinois.edu/

ILLINOIS™
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Contents

Chapter 1

Introduction: The Illinois Research Agenda

For many decades, the microprocessor industry has seen a steady growth in CPU performance, driven by Moore's Law [113] and Dennard scaling [42]. Unfortunately, as feature size decreased below 130nm over a decade ago, Dennard scaling ceased to apply, as static power became significant and voltage could not be decreased as fast as before. To keep power consumption in check, designers stopped increasing the clock rate and started to integrate multiple processors in one chip [41].

This technology shift has had major software implications. Before, single-threaded applications would see their performance increase over successive microprocessor generations with little or no need for software changes. Now, the performance of an application improves only if it can use an increasing number of concurrent threads. The problem is particularly acute for client (i.e., desktop and mobile) workloads which, unlike server ones, are turnaround-oriented — parallelism is used to reduce response time or handle a more complex problem. This is a difficult programming problem because it requires parallelization of many compute-intensive algorithms, often with fine-grain sharing of complex data structures.

A key question is whether today's multicore parallel computing context is fundamentally different from the traditional high-performance parallel computing context. There are, in fact, two fundamental differences: the importance of productivity and the market size. First, applications for desktop and mobile devices are developed under enormous competitive pressure to minimize time-to-market and enhance functionality, leaving less developer time for performance-oriented goals like parallelization. In this context, maximizing developer productivity becomes vital: application teams are willing to accept moderate speedups at low developer cost rather than invest the time to maximize speedups. Second, the client computing market is ten to a hundred times larger than the high-performance one. This justifies far greater investments by industry, which in turn can enable many high-level and specialized languages, libraries, frameworks, tools, and architectures addressing different subsets of the market and aiming at improving programmer productivity.

If, however, client applications do not leverage parallelism, then users will see no performance improvement as they buy a more powerful processor. They will have no incentive to upgrade their systems, and an industry strongly dependent on a continuous demand for increasing performance will be threatened. This is the problem addressed by the *Illinois Parallelism Center*, through the Universal Parallel Computing Research Center (UPCRC) funded by Intel and Microsoft during 2008-2010, and the Illinois-Intel Parallelism Center (I2PC) funded by Intel during 2011-2013. The Center focused on three questions:

- What applications will require the increasing performance that parallelism can bring to client processors?
- What programming models and tools will facilitate productive development of such applications?
- What computer architectures will leverage most efficiently the many cores that future manycores may have?

This book summarizes the research results of the Illinois Parallelism Center, and includes a few key papers resulting from the research.

1.1 Applications

In a world that increasingly relies on technology to facilitate interpersonal communication, we envision the killer client applications of the near future to be those that require high-quality, interactive tele-immersive environments with significant amount of local processing. In the *AvaScholar* project described in Chapters 3 and 4, PIs John Hart, Minh Do, Thomas Huang, Sanjay Patel and Klara Nahrstedt study such an application: an educational environment where an online instructor uses her hands to interact with real and virtual 3-D visual aids while, in real time, gauging the engagement of thousands of online students by expression recognition through their webcams. This application has posed many new challenges for parallel software engineering, system design, and visual computing algorithm development. For example, synchronous communication relies on minimizing latency, but the usual mobile-client/cloud-server relationship adds too much latency to client-client communication. Hence, the PIs have used the new approach of using clients as cloud processors. 3-D reconstruction of instructor and visual aids, along with student expression recognition required scaling up single-user laboratory computer vision algorithms. It also needed new parallel algorithms for graphics rendering and managing spatial data structures. The application has become so intriguing that the PIs plan to extend its use beyond the educational environment.

One of the students in the project, Wanmin Wu, received the ACM SIGMM Best Ph.D. Thesis Award in November 2012 for her thesis "Human-centric Control of Video Functions and Underlying Resources in 3D Tele-immersive Systems". She also received the Best Student Paper Award at ACM Multimedia in November 2011 [167]. PI Huang and his graduate student Usman Tariq won the First Prize in a competition for Automatic Person-Dependent Emotion Recognition at the IEEE International Conference on Automatic Face and Gesture Recognition in March 2011. A paper by student Liangliang Cao received the Best Paper Award at the International Workshop on Big Data Mining in August 2012 [34]. In addition, another student, Raoul Rivas, has finished his Ph.D. and will start working at Intel in the OS/power group.

• • •

Web browsing is a key client application where attaining high performance on mobile devices is critical. In the *Parallel Web Browser* project of Chapter 5, PI Samuel King studies how to parallelize web browsing for multicores. Rather than applying traditional techniques such as using parallel layout algorithms or applying task-level parallelism to the browser, he proposes that browser developers focus on parallelizing web pages. He presents the ADRENALINE prototype web browser [106], which consists of a server-side preprocessor and a client browser. The former decomposes existing web pages on the fly into loosely coupled subpages or mini pages; the browser then processes mini pages in parallel. Since each mini page is a "complete" web page, the browser can download, parse, and render this web content in parallel, while still using single-threaded, mature techniques in the client.

1.2 Software Development

Developing concurrent software is more difficult than developing sequential one. The programmer has to think through all of the potential interactions between multiple concurrent threads. Subtle, non-repeatable bugs occur because parallel programming today produces schedule-dependent, non-deterministic results. Low-level notations rather than high-level, structured abstractions are widespread. To address these problems, the Illinois Parallelism Center researched multiple directions.

• • •

In practice, the most widely-used approach to embrace parallelism is to retrofit a program incrementally by changing the existing code — each small step being a behavior-preserving transformation or a refactoring. In the *Refactoring* project described in Chapter 6, PI Danny Dig studies refactoring tools that allow developers to interactively and safely change large existing code bases. His research has been driven by two questions:

(i) what are the refactorings that occur most often in practice, and (ii) how can one automate refactorings to improve programmer productivity and software quality. He opened the area of interactive tools for retrofitting parallelism into sequential programs, which resulted in 11 publications that appeared in the top conferences in Software Engineering. Two of his papers received awards: a Best Paper Award at ICST 2013 [105] and an ACM SIGSOFT Distinguished Paper Award at ISSTA 2013 [137]. Some of Dig's refactorings are already shipping with the official release of the NETBEANS Integrated Development Environment (IDE), or are ongoing integration in the ECLIPSE IDE. Both IDEs are open-source and are used by millions of Java developers everyday. Dig has taught his refactoring principles and tools to more than 800 participants that attended our Center's industrial courses, summer schools, and conference tutorials.

• • •

In the context of deep memory hierarchies, the development of efficient parallel applications has become increasingly difficult. In the *Tiling Notations and Optimizations* project of Chapter 7, PIs David Padua and Maria Garzaran discuss Hierarchically Tiled Arrays (HTAs), an API that facilitates parallel programming while giving programmers the necessary control to attain good performance [20,32,72]. With HTAs, parallel programs are written as a sequence of high-level operations on arrays or sets, so that they resemble sequential programs — although there is parallelism inside each operator. Parallelism is controlled through partitioning of the data, which gives programmers a powerful abstraction to express data distribution and locality. This approach has advantages in program size, readability, portability, and control of determinacy. They also used tiles as first-class objects for the automatic generation of linear algebra solvers through an autotuning system [58]. Based on a description of the problem in the form of an equation, the system selects the best partition of arrays into tiles and the best shape of parallelism that conforms to this partition. Finally, the PIs used tiles to facilitate the process of compilation on multicores, distributed memory machines, and GPUs by automatically selecting the best tiling shape and removing unnecessary barriers [175].

Two of the Ph.D. students from this project have joined Intel after graduation: James Broadman works in the Software and Services Group, and Alexandre Duchateau works in the Programming Systems Group of Intel Labs. The HTA technology is being actively used in two Intel-led exascale computing projects: the DARPA-funded Runnemede project and the DOE-funded X-Stack Traleika Glacier project.

• • •

An important source of programmer hardship is that today's parallel programming notations (languages or libraries) permit concurrency errors that can produce schedule-dependent, non-deterministic results. In the project *Deterministic-by-default Parallel Programming* described in Chapter 8, PIs Vikram Adve, Sarita Adve, Madhusudan Parthasarathy, and Marc Snir observe that most compute-intensive algorithms used in client-side programs are, in fact, deterministic [26]. Unfortunately, today's parallel programming notations force developers to spend substantial time understanding and fixing the sources of unintentional non-determinism (i.e., concurrency errors). The DPJ project has developed powerful, largely compile-time, techniques based on a type and effect system to guarantee the absence of concurrency errors, and also the stronger property of deterministic results with sequential semantics [25, 27]. The Accord project has developed closely related verification-based approaches for specifying, inferring, and checking thread contracts for safe parallelism [92]. Moreover, for the many programs that mix non-deterministic with deterministic algorithms, DPJ provides a strong property called "determinism-by-default," which guarantees data race freedom, deadlock freedom, and strong atomicity to the entire program, as well as sequential semantics for deterministic subsets of the program [28]. The Tasks-with-effects programming model extends nearly all these guarantees to a much broader class of programs, including server, interactive programs, and other programs with unstructured parallelism [77]. These guarantees eliminate insidious concurrency errors and simplify reasoning about run-time behaviors of parallel code.

The student Robert Bocchino was awarded the ACM SIGPLAN Distinguished Dissertation Award for his Ph.D. thesis on DPJ. Alex Tzannes and PI V. Adve are working with Autodesk engineers to develop a static checker for C++ applications parallelized using Intel's Threading Building Blocks (TBB). The checker, called Annotations for Safe Parallelism (ASP), is based on the DPJ and Accord projects. PI V. Adve co-founded a series of workshops on Determinism and Correctness in Parallel Programming (WoDet). WoDet has been successful in bringing together leading researchers in the area of correctness techniques for parallel programming.

● ● ●

GPUs are an increasingly popular platform for client computing. It has long been held that performance programming for GPUs and CPUs require different source code development. In the *MCUDA: CUDA for Multicores* project of Chapter 9, PI Wen-mei Hwu demonstrates that, with good high-level algorithm and data optimization techniques, the same OpenCL kernel source code can be compiled for high-performance execution on both CPUs and GPUs. This is attained by a collection of novel thread serialization techniques that enhance vectorization, eliminate overhead in barrier synchronization, and coalesce privatized data. The early ACM LCPC 2008 paper [154] that describes the MCUDA compiler that first demonstrated this capability for the CUDA C language has over 170 citations according to Google Scholar. It also inspired the Intel and AMD OpenCL implementation for their multicore CPUs. The MCUDA project also led to the development of MxPA, a popular product from MulticoreWare that enables multi-platform high-performance execution of OpenCL applications in cloud, mobile, and consumer electronics platforms. Two papers that build on MCUDA to compile CUDA kernel code into FPGA logic for energy-efficient execution won Best Paper Awards at the IEEE Symposium on Application Specific Processors in April 2009 [124] and at the IEEE International Symposium on Field-Programmable Custom Computing Machines in May 2011 [125].

● ● ●

The task of programming is made harder because multicores are becoming heterogeneous and rely on sophisticated energy management support. In the *Scheduling for Energy Efficiency* project described in Chapter 10, PIs Maria Garzaran and David Padua develop scheduling algorithms to map vision applications onto heterogeneous mobile devices. Their algorithms take into account the properties of the different tasks to run, and the performance and energy consumption characteristics of the heterogeneous cores in the chip. Experiments on the Intel Ivy Bridge system show interesting trade-offs. A paper published by the PIs with Intel collaborators was selected as one of the 5 Best Papers in the Conference on Languages, Compilers, Tools and Theory for Embedded Systems in April 2011 [163].

1.3 Correctness

As long as today's parallel programming languages and environments remain popular, the onus of correctness will fall into the testing, verification, and debugging stages. Multiple concurrent threads can have a huge number of possible schedules. As a result, programmers find it hard to reason about all of the potential interactions between them. Unfortunately, a bug may manifest itself in only a small number of such schedules. In the *Verification and Testing Advances* project of Chapter 11, PIs Darko Marinov and Madhusudan Parthasarathy describe several novel techniques and tools to improve the testing and verification of parallel programs. One line of work, on predictive testing, developed techniques that efficiently and automatically identify small subclasses of interleavings that are likely to cause various kinds of bugs to manifest themselves. The *Penelope* tool has emerged as a mature platform for predictive testing for various kinds of bugs (e.g., data-races, atomicity violations, deadlocks, or null-pointer dereferences) [149, 150]. Another line of work allows specifying, for each test, a small set of schedules that should be explored for this particular test. Testing improvements also include novel approaches for automatically generating tests and exploring interleavings to find more bugs faster for evolving code. The open-source *IMUnit* testing tool [86, 87] is being considered for use at Google. The

MuTMuT paper [69] was invited for journal publication [68]. Ph.D. student Vilas Jagannath graduated based on this work [85].

• • •

In many cases, concurrency bugs are strongly timing dependent, and software instrumentation of the code for debugging them alters their timing and prevents them from manifesting. In the *Record&Replay and Debugging Architectures* project described in Chapter 12, PIs Samuel King and Josep Torrellas propose hardware architectures to aid program debugging and development without affecting the timing of execution. These architectures can be "always on", even during production runs. In particular, the PIs have designed and constructed a hardware prototype for Record&Replay (R&R) of parallel programs. The prototype, called *QuickRec* [127], was built jointly with the Intel team of Gilles Pokam. It is built with FPGAs and has a full Linux-based OS. It has special hardware that automatically records enough of the execution of a parallel program into a log, to be able to reproduce the execution exactly later on. The prototype can record and replay complete Intel-Architecture (IA) parallel applications. The PIs are currently exploring with Intel researchers many other uses of this R&R technology for debugging and security aids. The PIs have also designed other novel architectures for detecting and avoiding other concurrency defects such as data races [116], atomicity violations [114], sequential consistency violations [115], and determinism bugs. This work was featured in a Communications of the ACM Research Highlight [80] and has been published in many high-visibility conferences [79, 110, 112], in several cases jointly with Intel co-authors. Two Ph.D. students working in this project, Abdullah Muzahid and Radu Teodorescu, received Intel Ph.D. Fellowships.

1.4 Multicore Architectures

Using multiprocessor systems reduces the need for complex processor design, but increases the complexity of interconnecting them. Hence, forward-looking multiprocessor architectures must provide novel, simpler memory subsystems and interprocessor communication fabrics. At the same time, the hardware architecture must help provide a programmable environment for parallel software. The need for these characteristics resulted in two multicore architecture projects.

• • •

In the *Bulk Multicore Architecture for Programmability* project, described in Chapter 13, PI Josep Torrellas proposes a scalable shared-memory substrate designed to enable a programmable environment. Data sharing is automatically managed with scalable hardware cache coherence based on the novel primitives of continuous Chunks and Signatures. Chunks are groups of dynamically-contiguous instructions that execute atomically; signatures are registers that encode address footprints. Chunk operation helps programmability by enabling high-performance sequential consistency and novel chunk-based hardware primitives for parallel program development (e.g., efficient R&R as in Chapter 12). The Bulk Multicore can operate with no changes to currently-existing software stacks. At the same time, it keeps the hardware complexity in check by operating on chunks of instructions in bulk, as opposed to single instructions at a time. The Bulk Multicore can deliver higher performance than current systems with a compiler pass that marks chunks and aggressively optimizes the code inside a chunk. Such optimizations can be unsafe in current machines, such as moving code across synchronization operations or across potentially aliasing pointers [6, 7]. Such optimizations are possible with Bulk because chunks execute atomically. This work received a 2009 IEEE Micro Top Picks from Computer Architecture Conferences Award [161], and has been published in several visible venues, including an article in the Communications of the ACM [159]. Some of the publications are with Intel researchers. One Ph.D. student working in this project, Aditya Agrawal, received an Intel Ph.D. Fellowship.

• • •

Shared-memory is arguably the most widely used general-purpose multicore parallel programming model. While it provides the advantage of a global address space, shared-memory programs are known to be difficult to debug and maintain [102] due to unstructured parallel control, data races, and ubiquitous non-determinism. At the same time, designing performance-, power-, and complexity-scalable hardware for such a software model remains a major challenge. Directory-based cache coherence protocols are notoriously complex to verify [1], hard to extend, and hard to scale. In the *DeNovo* project described in Chapter 14, PIs Sarita Adve and Vikram Adve take the view that these problems are not inherent to a global address space paradigm. Instead, they occur due to undisciplined programming models that use arbitrary reads and writes for implicit and unstructured communication and synchronization. There has been a recent surge of research on more disciplined shared-memory programming models to address the software problem. The DeNovo project asks the question: "if software becomes more disciplined, can we build more performance-, power-, and complexity-scalable shared-memory hardware?" This work shows that the evolving software landscape represents a unique opportunity for a new multicore architecture paradigm. Compared to conventional hardware driven by "wild shared memory programming models," disciplined models can significantly simplify the hardware implementation, reduce communication traffic, and provide comparable or better performance with commensurate energy savings. The motivation and research agenda addressed by this work appeared in CACM in 2010, the first DeNovo paper won the Best Paper Award at PACT 2011, and two students won Qualcomm innovation fellowships in 2012 to apply these ideas to heterogeneous systems.

1.5 Training Efforts

Ultimately, the success of multiprocessors depends on the availability of programmers who understand the problems and potential solutions to parallel programming. Therefore, training has been an important part of the Center's activities.

Inside the University of Illinois, we have offered courses on multicore programming (e.g., [44]). We have revamped the Undergraduate Curriculum to include parallelism. We have integrated some of our research outcomes in courses, such as annotations for thread-safety, testing tools, parallel programming patterns, and refactoring tools.

In an effort spearheaded by PI Danny Dig, the Center has taught Multicore Parallel Programming courses outside the university. They include four one-week-long Summer Schools at Illinois, three one-week-long Technical Courses at Boeing, two conference tutorials, and one International Summer School in Singapore. These courses educated more than 800 participants. While the majority were professional programmers, they also included over 40 faculty from other universities. These courses have been highly rated by the participants. More than 80% of the participants in the Summer Schools and Technical Courses at Boeing have rated the instructor effectiveness and overall course quality as good or excellent. In anonymous surveys, Boeing engineers said "... *the multicore course presented by Prof. Dig was the best technical course I took at Boeing*".

PI David Padua lead an effort to assemble an Encyclopedia of Parallel Computing. With contributions from more than 300 authors worldwide, the encyclopedia was published in 2011 [123].

Acknowledgments

The authors thank Intel, Microsoft, and the University of Illinois for their continuous and generous support. We especially thank the members of the Joint Steering Committee from Intel and Microsoft. On the Intel side, the Committee was formed at various times by Ali-Reza Adl-Tabatabai, Allan Knies, Alon Naveh, Andrew Chien, Arch Robison, Bob Kuhn, Chris Ramming, David Kuck, Gad Sheaffer, Gans Srinivasa, Geoff Lowney, Henry Gabb, Jean-Pierre Giacalone, Jesse Fang, Jim Held, Joel Emer, Michael Wrinn, Opher Kahn, Paul Petersen, Pradeep Dubey, Sanjiv Shah, Tanay Karnik, Wei Li, and Youfeng Wu. On the Microsoft side, it was composed of Burton Smith, Daniel Reed, David Callahan, Dave Probert, James Larus, John Feo, Juan Vargas, Judith Bishop, Mark Lewin, Tony Hey, and Wolfram Schulte.

Chapter 2

Parallelism Center Personnel

The following individuals contributed to the work presented in this book over the last five years.

Students:

Abdullah Muzahid, Abid Malik, Adam Smith, Aditya Agrawal, Adrian Nistor, Albert Sidelnik, Alex Gyori, Alexandre Duchateau, Alexandria Shearer, Ben Ahrens, Binh Le, Brandon Moore, Bruno Virlet, Byn Choi, Carl Evans, Charles Tucker, Chien-Nan Chen, Chris Rodrigues, Cosmin Radoi, Dale Kim, Daniel Kubacki, Danny Johnson, Dennis Lin, Dongyun Jin, Ehsan Totoni, Francesco Sorrentino, Fredrik Kjolstad, Gabriel Acevedo, Greg Meyer, Haichuan Wang, Haohui Mai, Hui Xue, Huy Bui, Hyojin Sung, I-Jui Sung, James Brodman, Jared Hoberock, Jeffrey Overbey, Jia Guo, Jiangping Wang, John Kelm, John Mark Lau, John Poulakos, John Sartori, John Stratton, Joseph Sloan, Joshua Blackburn, Kyle Doreen, Lyle Franklin, Marios Nicolaides, Mariyam Khalid, Mark Faust, Matt Sinclair, Matthew Johnson, Matthieu Delahaye, Mihai Tarce, Milos Gligoric, Mohsen Vakilian, Nathan Dautenhahn, Neil Crago, Nick Bray, Nicolas Zea, Nima Honarmand, Pablo Montesinos, Patrick Simmons, Pooya Khorrami, Praateek Jindal, Qingzhou Luo, Quang Nguyen, Radu Teodorescu, Rajesh Bhasin, Rajesh Karmani, Rajesh Kumar, Rakesh Komuravelli, Raoul Rivas, Rob Smolinski, Robert Bocchino, Saeed Maleki, Semih Okur, Shanxiang Qi, Shuo Tang, Stephen Heumann, Swapnil Ghike, Traian Serbanuta, Usman Tariq, Victor Lu, Vijay Korthikanti, Vilas Jagannath, Voytek Truty, Vuong Le, Wei Han, William Tuohy, Wonjun Jang, Xing Zhou, Xuehai Qian, Yu Lin, Yuelu Duan, Yuntao Jia, Zhencheng Wang, and Zhenhuan Gao.

Research and Academic Professionals:

Alexandre Tzannes, Andrea Whitesell, Cheri Helregel, David Raila, Mark K. Smith, Megan Osfar, Mert Dikmen, Sherry Unkraut, and Wonsun Ahn.

Faculty:

Craig Zilles, Dan Roth, Danny Dig, Darko Marinov, David A. Padua, Grigore Rosu, Gul Agha, John C. Hart, Josep Torrellas (I2PC Director), Klara Nahrstedt, Madhusudan Parthasarathy, Marc Snir (UPCRC Co-Director), Maria Jesus Garzaran, Matt Frank, Minh N. Do, Rakesh Kumar, Ralph Johnson, Samuel T. King, Sanjay Kale, Sanjay J. Patel, Sarita V. Adve, Thomas S. Huang, Vikram S. Adve, Walid Abu-Sufah, and Wen-mei W. Hwu (UPCRC Co-Director).

Chapter 3

AvaScholar Instructor

3.1 Problem Addressed

The goals of the Illinois Parallelism Center were to discover and define how we would program in an era where performance improvements in consumer computing platforms came mainly from increased parallelism. In this era, serial applications would no longer run faster on newer computers, causing a contraction of the personal computing economy. Parallel programming had previously been relegated to high performance scientific computing. We as a community needed to prepare for a time when all programming needed to be parallel programming.

Research into new parallel programming tools needs applications to (1) motivate the need for higher performance and (2) serve as a testbed to demonstrate that new tools are effective for real world examples. In the past, the applications that drove consumer computing to its current state have been word processing and spreadsheets, with e-mail and browsing added more recently. Video games have also merged into the consumer computing application spectrum. When we combine this evolution of applications with our own expectations, we envision that networked visual applications will dominate the foreseeable applications landscape. Intel further demonstrated their commitment to this direction by initiating their first Intel Science and Technology Center with a focus on visual computing.

AvaScholar was designed as a testbed application that contained the elements of networked visual computing that we expect of future applications. It is a system designed to facilitate remote synchronous education, where a single instructor teaches a set of remote participants connected through standard consumer computing devices (e.g., desktop, laptop or mobile computers). This choice of application was aided by the fact that we are quite familiar with online education and have already used and developed a wide variety of systems for remote education and teleconferencing.

The AvaScholar system as shown in Figure 3.1 consists of two components. The AvaScholar Instructor component is a system that captures and reconstructs a real-time 3-D model of the instructor. This 3-D model can be combined with any 3-D models of visual aids and transmitted to the remote students for arbitrary-viewpoint observation, which built on our previous remote telepresence work [145].

The AvaScholar Student component uses the student's computer camera to report statistics on engagement and demographics, using soft biometric software to estimate race, sex, emotion and other indicators of the students as they participate in an online class session. In this chapter we focus on the AvaScholar Instructor component, whereas the next chapter details the AvaScholar Student.

Figure 3.2 illustrates the individual components of AvaScholar and their dependencies. At the top, the instructor module depends on surface reconstruction, whereas the student module relies on soft biometrics. We examined parallel methods for depth reconstruction using stereo images and using Kinect-style active depth, which can be improved through Kinect-fusion style ICP alignment. All of these techniques rely on fast kernels for nearest neighbors, for which we investigated parallel k-D tree algorithms, and image feature analysis, for

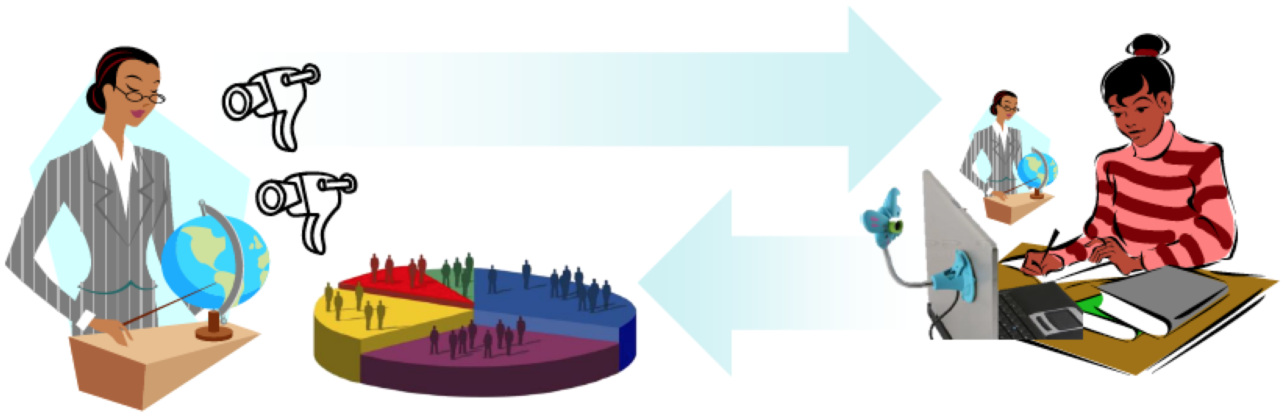


Figure 3.1: AvaScholar consists of the Instructor module (left), which transmits a 3-D version of the instructor interacting with visual aids to the Student module (right), which also measures student engagement via webcam soft biometrics and reports aggregate statistics back to the instructor.

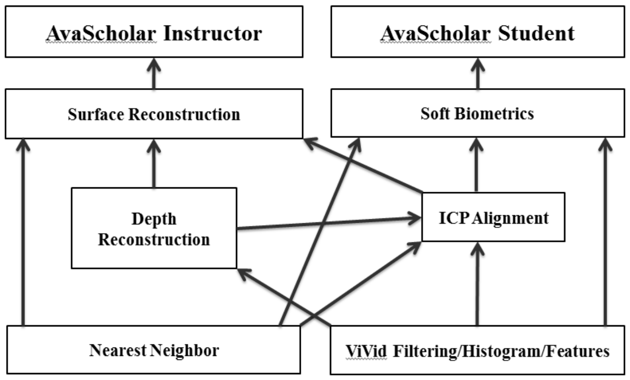


Figure 3.2: The “House of Cards” illustrating the different components of AvaScholar and their interdependence.

which we created the ViVid parallel image analysis library.

Nearest neighbor algorithms are particularly challenging for parallel programming. Figure 3.3 shows a spectrum of scalability for the parallelism of various visual computing processes, as measured by Intel production applications. While none scales perfectly, two of the curves fall well below the rest, and will pose the greatest challenge to parallelizing visual computing past 64 cores. These two scalability challenges are cloth and rigid-body physics. Both critically rely on collision detection, especially cloth to avoid self interpenetration. Collision detection itself relies on proximity queries, which largely rely on hierarchical spatial data structures to accommodate the varying range of scales found in modern video games and simulations. The construction and query of these hierarchies can lead to execution and memory divergence that inhibits scalable performance.

3.2 Contributions

Our work on visual computing has focused on areas that especially challenge scalable parallel programming. For manycore GPU programming, these scalability challenges are largely due to code divergence. Our first contribution examined shader performance when rendering photorealistic scenes. When rendering an image, each pixel is run through a program, called a “fragment shader,” that uses its geometric information to run a lighting simulation to determine what color it should be. While this lighting simulation can be uniform across a scene, leading to fast data parallelism as it is applied to pixels from various geometric contexts, most scenes depict a wide variety of different materials that each require unique code segments in the lighting simulation. Hence,

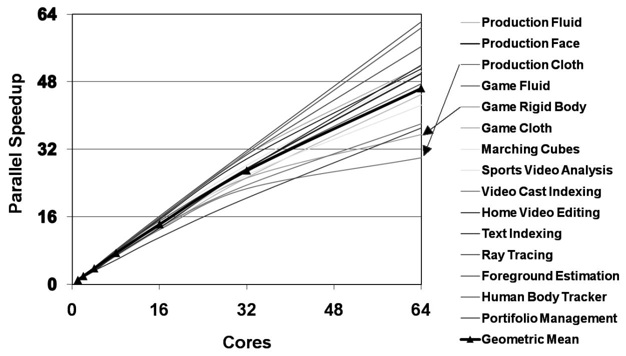


Figure 3.3: The parallel visual computing processes most problematic for scalability are cloth and rigid body, because both rely on collision detection. Source: Jim Held, Intel Fellow, personal communication .

materials lead to shader divergence when a scene is rendered. We sought to determine under what conditions it was better to coalesce shading requests by material to reduce code divergence and improve performance. We discovered that the cost of sorting shader requests by material justified the improvement in reduced divergence performance for scenes with several different shaders, or even a few complex shaders (e.g., wood or hair) [78].

We studied surface reconstruction, through parallelization of a state-of-the-art film-quality stereo depth algorithm from Disney's Zurich Research Center [17]. This algorithm measures the disparity between two pre-registered and aligned images by finding matching pixels in each scanline, taking special care that some features may occur in the middle of a pixel in one image and between pixels in the other image. The algorithm's disparity searchers also utilize several consistency passes that can lead to race conditions and divergence that we used as testbed code for our Center's safe parallel programming tools. Our parallel re-implementation of this algorithm on a 32-processor Intel Nehalem system accelerated it from its reported serial running time of 20 minutes to an optimized parallel running time of 20 seconds. Further scaling will be necessary to use this high-quality approach for real-time depth computation.

We also investigated parallel algorithms for processing depth images [53], such as those generated by the Kinect depth camera. We describe a propagation approach to filling in color and depth for the regions that can be seen from a user viewpoint but were occluded from the depth sensor. Two dimensional image propagation leads to significant special-case divergence and race conditions, which were overcome by reducing them to eight one-dimensional cases using epipolar geometry which could be collected and streamed efficiently.

Depth images are noisy and some areas are occluded in any single depth image. Kinect fusion and other techniques have shown that by aligning multiple depth images, a more complete and cleaner averaged model can be generated. A common method for geometric 3-D alignment is the Iterated Closest Point (ICP) algorithm, which moves two point clouds closer by averaging the displacement vector from each point in one point cloud to its nearest point in the other point cloud. We constructed a fast parallel manycore GPU method for ICP using a volumetric model (a 3-D array) that measured and recorded a distance function for the other point cloud [95]. As the number of depth images increases, this approach outperforms classical ICP.

We investigated several parallel nearest-neighbor methods. Such nearest-neighbor queries are commonplace in many graphics, vision and multimedia applications, including sparse data reconstruction, image stitching, machine learning, vector quantization and even the ICP algorithm. Our ParKD approach [38] sought to build k-D trees that optimized the Surface-Area Heuristic (SAH), which seeks to subdivide bounding boxes into smaller bounding boxes that minimize their surface area. We collaborated with DeNovo and DPJ researchers to investigate two approaches to parallel k-D tree construction: a nested parallel approach and an in-place method. The nested parallel approach was faster on our 32-core system but the in-place algorithm exhibited better scalability and we expect it to outperform the nested approach on finer-grain parallel systems.

We also revealed a vulnerability of existing parallel k-D tree construction algorithms that tend to serialize the construction of the upper levels of the k-D tree. This serialization creates an Amdahl's law condition that degrades the performance of such algorithms as the number of processors increases. We utilized a two-phase construction algorithm that streams through data in parallel in the upper levels of the k-D tree to find the optimal divisions, which avoids this serialization.

We also collaborated on a render server to reduce computational load on lightweight mobile clients [146]. Our implemented method for delivery of the AvaScholar Instructor is based on WebGL and runs in any compatible browser.

3.3 Lessons Learned

We learned the following:

1. The GPU performance gain resulting from collecting similar jobs often overcomes the cost of sorting the jobs. However, the resulting memory fragmentation can often degrade performance.

2. Volumetric ICP reduces the need for nearest-point queries by using a volume-based distance function, but this benefit does not outperform ICP until ten or more depth images are merged.

3. Parallelization of spatial hierarchies needs to include all nodes, including the upper nodes, for scalability.

4. In-place construction of spatial hierarchies reduces the amount of data transfer, but also reduces memory coherence.

5. Numerous opportunities exist for parallelism in visual computing that cannot be detected automatically but require domain knowledge, such as epipolar geometry.

3.4 Future Work

We continue to work on high-performance methods for stereo reconstruction, high-dimensional neighbor queries (e.g., FLANN) and image stitching, for a variety of concurrent projects. We also plan to continue to pursue AvaScholar as an remote synchronous educational tool, with applications for Massive Open Online Courses (MOOCs).

3.5 Key Papers and Other Material

We have included two papers in this book. The first one is "Parallel SAH k-D Tree Construction" from High Performance Graphics 2010 [38]. The other is "Immersive Visual Communication" from the IEEE Signal Processing Magazine 2011 [53].

Chapter 4

AvaScholar Student

4.1 Problem Addressed

In the previous chapter, we introduced AvaScholar Instructor, which supports seamless 3-D reconstruction and streaming of the instructor's scene to students. At the students' side, another vision system is designed to observe and understand the participation of the users. In a traditional lecturing setup, students' participation is monitored and responded directly by the instructor. This method is not efficient in large classroom setups and remote online lectures. In AvaScholar Student, we design a student site setup that includes a screen which shows the video cast from the instructor and a camera which captures the student's appearance dynamics. These visual data are then analyzed by local and distributed vision algorithms to estimate the participation behavior of the student. The types of visual data that we concentrate on in this project includes 3-D geometrical motion extracted by face tracking and dynamic appearance of the human faces analyzed by robust image representation.

The first problem we address is facial 3-D geometrical reconstruction and tracking. Using as input a single low-resolution 2-D camera, we developed a novel non-invasive, reliable, fully automatic realtime algorithm for reconstructing a 3-D facial model and tracking nonrigid motion. We designed a new optimization procedure called *Iterative Linearized Optimization* [99] to concurrently optimize both rigid and nonrigid facial motions in linear time from a set of visually-tracked landmark points [98]. The nonrigid motions are coded into MPEG 4's facial animation parameters. These coded facial motions can be used for a number of applications, including emotion recognition [101], attention detection [104] which conveys the inner state of the student, and performance-driven avatar animation [99], which allows very low bit-rate avatar-based communication between student and instructor. Figure 4.1 illustrates the workflow of the 3-D face tracking algorithm on an input of a 2-D video.

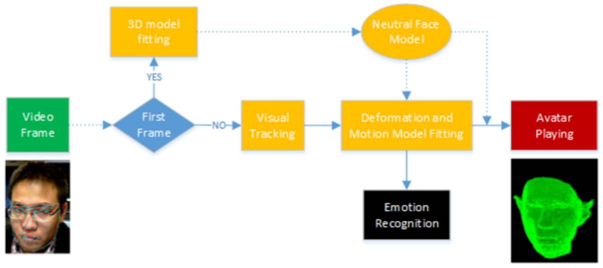


Figure 4.1: Workflow of our face tracking system.

In this chapter, we introduce one example application for face tracking, namely determining where a subject is focusing her gaze. Attention and interest are correlated with gaze fixation in a number of settings. Therefore, if a face tracker can estimate the location and duration of a subject's gaze, it may also be able to determine what is capturing the subject's attention. This is particularly useful in online classroom sessions when a student's gaze tends to drift from their computer monitors. Having software that can detect when a student is losing focus will be instrumental in constructing effective learning environments.

Our system for estimating a subject's gaze location can be split into two separate modules via a client-server model. First, a program on the client runs the face tracking application and extracts various different facial features of the subject over time. These features are bundled into packets and transmitted over TCP/IP to a server. The server receives the packets sent from the client and computes where on the monitor the subject is fixing her gaze. If the student's gaze is outside of the monitor's boundaries for a prolonged amount of time, then the subject is considered to be disengaged and a warning flag is raised. A graphical interpretation can be seen in Figure 4.2 below.

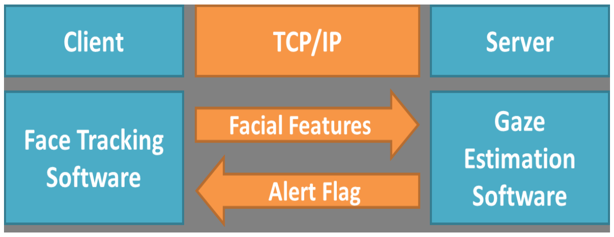


Figure 4.2: Attention detection system layout.

3-D facial tracking and its applications are designed to give real-time performance in a laptop PC computer. However, the end points of the online learning participants are not powerful parallel PCs anymore but mobile light-weight devices, such as smartphones and tablets. The problem becomes how to enable computationally-heavy tasks in AvaScholar such as face tracking on thin mobile devices. Hence, another goal of our work is to research the feasibility of using mobile devices as part of the AvaScholar system. Our approach to this problem is two-fold:

1. We employ **cloudlets**. The cloudlet concept was introduced by Satya et al. [141], who pointed out early on that the combination of heavy processing in clouds, and usage of mobile devices will need a novel computing model which allows people to get access to cloud computing resources with their mobile devices. The novel computing model encompasses a cloudlet — a local powerful server to which mobile devices can offload their computationally-intensive tasks. In our AvaScholar system, this translates into that each participant replaces its powerful parallel PC, equipped with cameras(s), microphone, speakers, and graphics, with a pair of devices: the mobile device and its corresponding cloudlet, called mobile-cloudlet. The mobile-cloudlet pair requires then a distributed solution for the computationally heavy tasks.

2. We employ **multi-threading** on each device of the mobile-cloudlet pair to further increase the concurrency and parallelism among computationally-heavy subtasks.

The challenges of our approach are

- Functional split (parallelization) of computationally-heavy tasks such as face tracking into individual subtasks.

- Decision of locality for each subtasks with respect to mobile smartphone/tablet and cloudlet components.

- Impact of the network connecting the smartphone/tablet and the cloudlet components.

While human behavior can be conveyed through geometrical structures of the face which are analyzed by face tracking, appearance dynamics of the facial image are also important clues for understanding the subject's inner state. To exploit this important type of clues, we propose a new method for non-frontal expression recognition. The increasing applications of facial expression recognition, especially those in Human Computer Interaction, have attracted a great amount of research work in this area in the past decade. However, much of the literature focuses on expression recognition from frontal or near-frontal face images [156]. Expression recognition from non-frontal faces is much more challenging. It is also of more practical utility, since it is not trivial in real applications to always have a frontal face. Nonetheless, there are only a handful of works in the literature working with non-frontal faces. We approach this problem by proposing an extension to the very popular image classification framework based upon the Bag-of-Words (BoW) models [157].

We develop a simple yet effective supervised learning method of GMM for Soft Vector Quantization (SVQ) applied to facial expression recognition. The objective function is smooth, and can be easily solved by gradient descent. We term the resulting image features as Supervised SVQ (SSVQ) features. Our extensive experiments on the multiview face images, generated from the BU-3DFE database for recognizing expressions, show that our approach significantly improves the resulting classification rate over the unsupervised training counterpart. When our method is combined with Spatial Pyramid Matching, it also outperforms the published state-of-art results, which were achieved with a much more complex model.

4.2 Contributions

4.2.1 3-D Face Modeling

The first contribution of AvaScholar Student is a fully automatic real time system for 3-D face model fitting and tracking from monocular 2-D videos. The system accurately fits facial images into a highly-detailed 3-D morphable model [100] which assists reliable tracking and allows realistic avatar rendering. The system is powered by a novel efficient tracking and fitting algorithm based on an iterative linearized procedure. The experiments on public datasets show that the system achieves high performance on face tracking and avatar rendering and is suitable for animation and telepresence applications.

4.2.2 Performance Driven Avatar

Based on the face-modeling algorithm, we develop a demonstration of a performance-driven avatar and attention estimation. After the initial fitting step, the shape of the face is recovered by applying the extracted identity parameters. The textures of the model's vertices are mapped from the images. The full textured model is then used as an animated avatar.

In the motion-fitting step, we analyze the video frame and estimate the facial motion represented by FAP values. These FAP values can be used to control any compatible face model to generate a facial animation that resembles the facial actions of a performer in the video. The deformed face is recovered on the FAP spanned subspace.

When the rendered avatar is of the performer, the system will be acting as a video encoding and decoding system. The avatar can also be of another subject or a cartoon character. In that case, we have a performance-driven animation. Figure 4.3 demonstrates an example of face tracking results on a video frame which is used to play the avatar of the subject, and another prebuilt avatar named ”Obama”.

4.2.3 Attention Detection

To detect the gaze point of the subject during tracking at the client module, several facial features are collected for transmission to the server. These features include the pixel locations of the eyes as well as the head pose via three rotation angles corresponding to pitch, yaw, and roll of the head. These parameters are then collected into a packet and transmitted to the server via the TCP/IP protocol, where they are then used to compute the world coordinates of the subject's gaze.

Upon receiving a packet from the client, the server performs three steps to determine the 3-D location of the subject's gaze. First, the 3-D world coordinates of the subject's eye are computed. Then, a direction vector

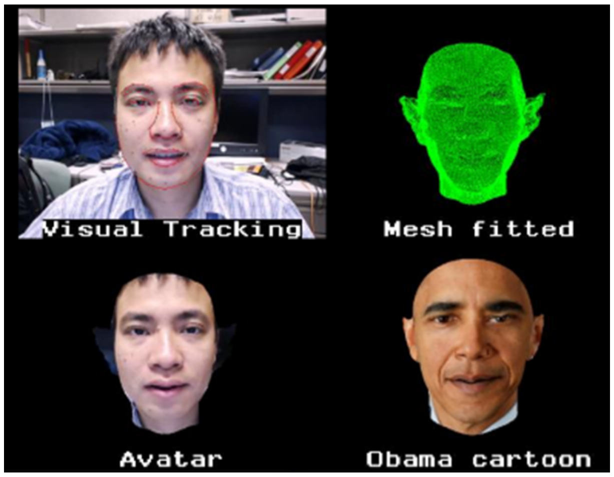


Figure 4.3: Example of 3-D performance-driven avatar.

originating from the eyes is constructed via the estimated pose. Finally, the gaze point is considered to be the intersection point of the z plane corresponding to the monitor and the line from the aforementioned line from the subject's eyes. The geometry model used in the process is shown in Figure 4.4.

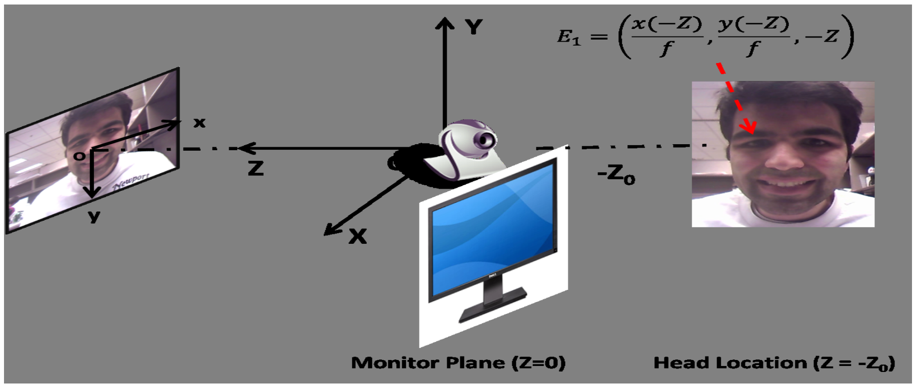

Y
$E_1 = \left(\frac{x(-Z)}{f}, \frac{y(-Z)}{f}, -Z\right)$
x
o
y
Z
X
$-Z_0$
Monitor Plane (Z=0)
Head Location ($Z = -Z_0$)

Figure 4.4: System visualization.

After estimating the location of the subject's gaze, the server checks whether the point is within the monitor's dimensions. If the subject's gaze is outside the monitor's boundary for a prolonged amount of time, then the server sends a signal to alert the subject that she is not engaged.

4.2.4 Mobile-Cloudlet Design Framework

For the cloud-based face tracking design, we have developed (1) a mobile-cloudlet design framework; (2) a scalable software architecture; and (3) an experimental validation using Android-class mobile devices such as Nexus 4 and Nexus 7.

The validation and experimental environment of our mobile-cloudlet framework is shown in Figure 4.5. To enable the execution of the face detection and tracking task in a distributed manner rather than on a single powerful parallel machine, we have considered multiple splitting options of the functional data flow. One option was to do face capturing, 3-D model fitting and face rendering on the mobile device, and visual tracking and motion model-fitting on the cloudlet. The second option was to do the face capturing and face rendering on the mobile device, and move the computation of 3-D model fitting, visual tracking, and deformation and motion model-fitting on the cloudlet.

The second option is best, since current mobile devices are not powerful enough and fast enough to do 3-D

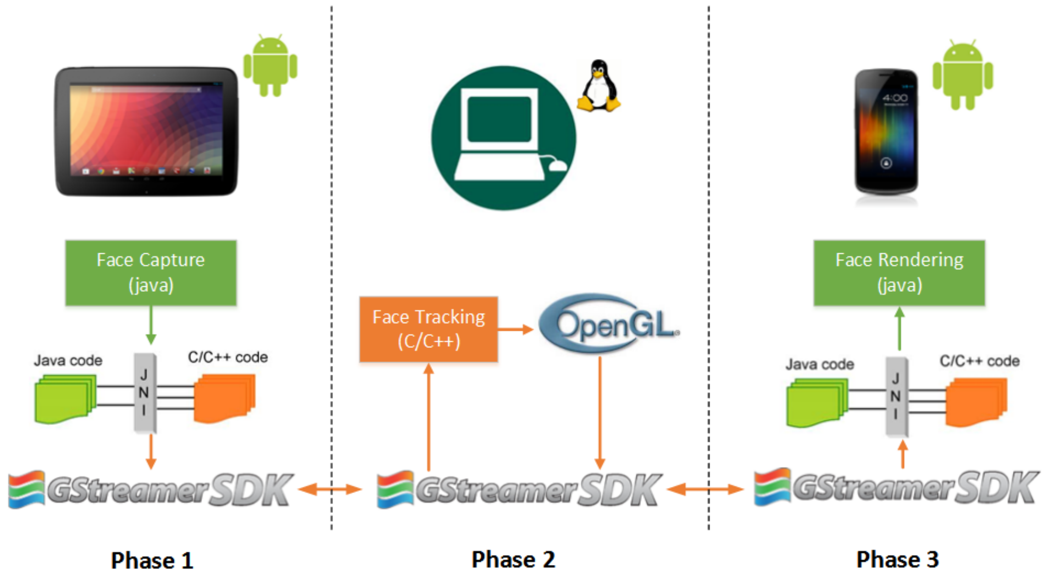


Figure 4.5: Overview of the software architecture. Green lines and blocks denote Java code; orange lines and blocks denote C/C++ code.

model fitting. On the other hand, we had the hypothesis that WiFi networks have the bandwidth availability to transmit video streams. Overall, this particular (mobile-cloudlet) architectural choice meant that we traded-off bandwidth for time.

In summary, the functional split and locality of subtasks in our architectural framework included the video capturing, as well as the simple face rendering for avatar display on the mobile device; the core of computation with 3-D model fitting upon first video frame and visual tracking with motion model fitting was included on the cloudlet. This functional split and locality of individual subtasks meant that raw video frames are sent from the mobile device to the cloudlet, and tracking-augmented frames are sent from the cloudlet to the mobile device.

4.2.5 Appearance-Based Emotion Recognition

We propose a novel supervised soft vector quantization model for appearance-based facial expression recognition. The discriminative training of GMM produces significant performance gains compared with the unsupervised counterpart. Combined with the spatial pyramid, our approach achieves state-of-the-art performance on the BU-3DFE facial expression database with simple linear classifiers.

4.3 Lessons Learned

We learned the following:

- The feature-based approach has a significantly-improved efficiency over the appearance-based approach for face-modeling algorithms, while preserving most of the details of facial structure and motion for animation applications.
- Face tracking can provide accurate-enough pose information for attention-level recognition, although for exact gaze detection, eye sight direction recognition is required.
- Faster processors and multicore/multi-GPU/CPU architectures on smartphones and tablets are necessary. The reason is that if one can do some of the video processing tasks at the mobile device, and send to the cloudlet only selected points for further processing, the bandwidth demand is smaller, and less computation is needed at the cloudlet, speeding up the overall face-detection and tracking process.

- High-bandwidth and time-sensitive wireless networks for smartphones and tablets are necessary. The reason is that if one has high-bandwidth availability for video streams, and capabilities for timing control exist, the RTT overhead can be kept very small, ensuring low RTT and EED between mobile device and cloudlet.

- The discriminative training of GMMs for expression recognition produces significant performance gains compared to the unsupervised counterpart.

4.4 Future Work

In the near future, the face-tracking engine will be updated to take advantage of the RGBD signal that may be available in depth cameras. The z dimension of the feature point will help estimate the motion parameters more accurately and efficiently.

The attention-detection module will be upgraded to accurately detect the eye sight by analyzing infrared images of the eyes. The eye sight will be combined with pose angles to determine the subject's point of gaze.

The cloud architecture will be further exploited for higher performance. We plan to investigate the relationship between mobile devices, cloudlets and clouds for multimedia support under diverse user behaviors. We will consider groups of mobile users sharing cloudlets when conducting face-detection and tracking and other multimedia-specific tasks.

For expression recognition, we will explore supervised training for full GMM parameters (mean, mixture weights, and covariance matrices) with proper regularization. Incorporating SPM in supervised training will also be investigated, to make each level of SPM more discriminative.

4.5 Key Papers and Other Material

We have included two papers in this book. The first one is "Maximum Margin GMM Learning for Facial Expression Recognition" from FG 2013 [157]. The other is "Expression Recognition from 3-D Dynamic Faces Using Robust Spatio-Temporal Shape Features", from FG 2011 [101].

Chapter 5

Parallel Web Browser

5.1 Problem Addressed

Web browsing on mobile devices is slow, yet recent reports from industry show that performance is critical [108, 170]. Google and Microsoft reported that a 200ms increase in page load latency times resulted in "strong negative impacts", and that even delays of under 500ms "impact business metrics" [142].

One source of overhead for web-based applications (web apps) is the network [164]. Engineers have attempted to mitigate this source of overhead with increased network bandwidth, prefetching, caching, content delivery networks, and by ordering network requests carefully.

A second and increasing source of overhead for web apps is the client CPU [57, 91]. Web browsers combine a parser (HTML), a layout engine, and a language environment (JavaScript), where the CPU sits squarely on the critical path [15, 63, 109]. Even though the serial performance of mobile CPUs continues to increase, the constraints on mobile device form factors and battery power impose fundamental limitations on further improvement.

Recent work proposes exploiting parallelism to improve browser performance on multi-core mobile platforms [30, 140], including parallel layout algorithms [15, 109], and applying task-level parallelism to the browser [71]. These special cases, however, only speed up web apps that make heavy use of specific features (e.g., Cascading Style Sheets (CSS)), or are limited to the tasks that the browser developers identify ahead of time. Unfortunately, years of sequential optimization, the sheer size of modern browsers (e.g., Firefox has over three million lines of code), and the fundamentally single-threaded event-driven programming model of modern browsers make it challenging to generalize this approach to refactor today's browsers into fully multi-threaded parallel applications.

5.2 Contributions

Our position is that *browser developers should focus on parallelizing web pages*. By taking a holistic approach, we anticipate an architecture that can work on a wide range of existing commodity browsers with only a few minor changes to their implementation, rather than a major refactoring of existing browsers or a re-implementation of these mature and feature-rich applications.

To back up our position, we present the design for *Adrenaline* [106], a prototype system that attempts to speed-up web apps for multi-core mobile devices, like smart phones and tablets. Adrenaline consists of two components, a server-side preprocessor and a client (i.e., browser) that renders pages concurrently on the mobile device. The Adrenaline server decomposes existing web pages on the fly into loosely coupled sub pages, or *mini pages*. The Adrenaline browser processes mini pages in parallel. Each mini page is a "complete" web page that consists of HTML, JavaScript, CSS, and so on, running in a separate process. Therefore, the Adrenaline browser can download, parse, and render this web content in parallel while still using a single-threaded and mature browser on the client.

Figure 5.1 shows the workflow when a user accesses `wikipedia.org` with the Adrenaline browser. First, the browser issues a request to the Adrenaline server. Second, the Adrenaline server fetches the contents of the web page, optimizes and decomposes it into mini pages. Third, the browser downloads, parses, and renders each of these mini pages in separate processes running in parallel. The browser is responsible for properly aggregating displays, synchronizing global data structures, and propagating DOM and UI events to maintain correct web semantics. In this example, the server decomposes the `wikipedia.org` page into four mini pages, and the browser runs four processes in parallel to render the page.

Component	% of CPU	4 cores	16 cores
V8	16%	1.13	1.17
X & Kernel	17%	1.14	1.19
Painting	10%	1.08	1.10
libc+Qt	25%	1.23	1.31
CSS	4%	1.03	1.04
Layout/Render	22%	1.20	1.27
Other	6%	1.05	1.06

Table 5.1: Breakdown of CPU time spent on web browsing. The last two columns predict the ideal speed-ups with Amdahl's law, assuming that either 4 or 16 cores are available.

Figure 5.1 shows the workflow of Adrenaline when accessing `wikipedia.org`. In the figure, each of the numbers, 1-4, show the four mini pages Adrenaline uses for this web page. The Adrenaline server acts as a proxy between the Adrenaline browser and the Internet. It fetches the web page, optimizes and decomposes it into mini pages, then sends them back to the Adrenaline browser. The Adrenaline browser downloads and renders mini pages in parallel using multiple processes. To preserve the proper visual and programmatic semantics, the Adrenaline browser aggregates the displays for all mini pages, forwards DOM and UI events between mini pages, and synchronizes DOM interactions. Solid lines between the Adrenaline browser and the Adrenaline server show the mappings of mini pages.

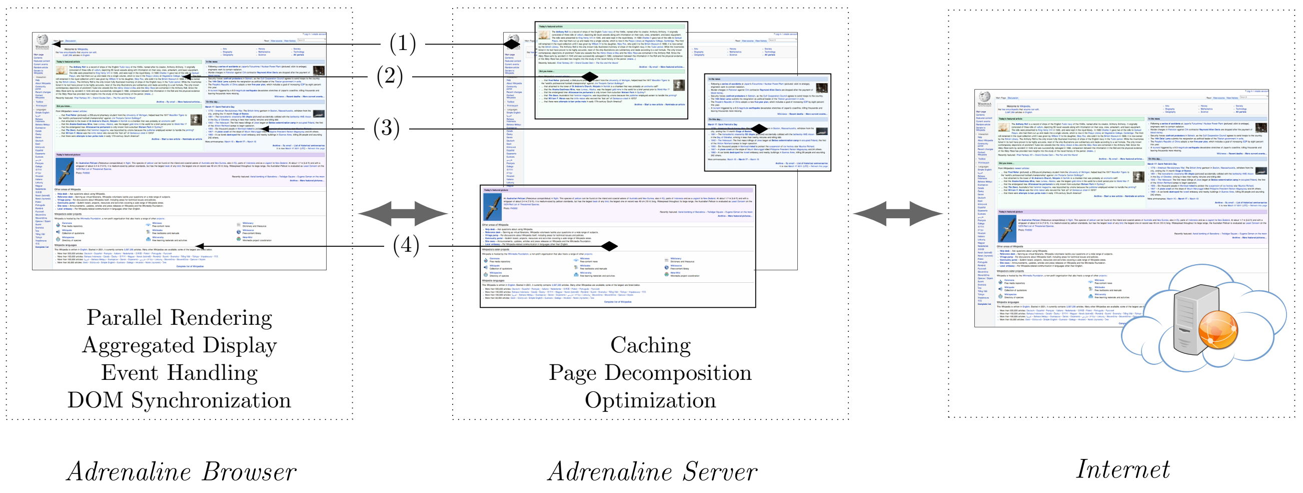


Figure 5.1: Workflow of Adrenaline when accessing `wikipedia.org`.

This architecture offers four unique advantages compared to other techniques for parallelizing web browsers. First, Adrenaline is a data parallel system. It parallelizes web pages, rather than specific components in web browsers. Conceptually, all components in a web browser can now be executed in parallel. Second, decomposition reduces the total amount of work from some tasks, particularly layout and rendering because of smaller working sets for each mini page. Third, careful decomposition could potentially remove serialization bottlenecks. Specifically, Adrenaline isolates JavaScript into a single mini page to allow tasks such as layout and rendering in other mini pages to run concurrently. Fourth, pre-processing the pages on the Adrenaline server creates opportunities to shift computation from the client to the server.

This architecture does also introduce two sources of overhead that the Adrenaline system must overcome. Fundamentally, the architecture places a proxy in between the Adrenaline browser and the Web. This additional component will add latency for individual network connections when compared to connecting to web sites directly. In addition, this architecture uses more resources on the mobile device through its use of multiple processes. Despite these inherent sources of overhead, the Adrenaline browser speeds up the overwhelming majority of sites we tested.

5.2.1 Results

We implemented the Adrenaline server as an HTTP proxy that fetches web pages and decomposes them on the fly automatically. This architecture mirrors closely the server-side architecture for other mobile browsers, like Opera mini, Skyfire, and Amazon Silk [12, 122, 147].

The Adrenaline browser uses the WebKit rendering engine and the V8 JavaScript engine. We use the Qt Toolkit to implement the platform-specific portions of the browser. Mini pages are implemented as browser plugins in Adrenaline to reuse existing mechanisms and to maintain visual compatibility. Our changes were rather minimal, and we believe that the same techniques are applicable to commodity browsers.

To test the performance of our prototype and to test the efficacy of the basic Adrenaline approach, we ran the Adrenaline browser on a Cortex-A9 CPU running at 400MHz and 768MB of DDR2 RAM. We tested Adrenaline on 170 of the most popular web sites (according to Alexa), and we compared against an unmodified version of a WebKit-based browser. To isolate the effects of our algorithms we mirror the web pages on our local network and connect to the server via an Ethernet connection.

Our preliminary experience with Adrenaline is encouraging. Adrenaline reduces the page load latency by 1.75s on average — more than the 0.5s latency reduction that industry considers meaningful [108, 142, 169]. Adrenaline improves the page load latency time by 1.54x on average across the entire workload. For one experiment, Adrenaline speeds up web browsing by 3.95x, reducing the page load latency time by 14.9s. Among the 170 popular web sites we tested, Adrenaline speeds up 151 out of 170 (89%) sites, and reduces the latency for 39 (23%) sites by two seconds or more.

5.3 Lessons Learned

In this work, we advocated that browser developers should think about parallelizing web pages, rather than individual components of web browsers. Based on our initial experience with Adrenaline, we believe that Adrenaline can improve significantly the performance of web browsing on mobile devices.

5.4 Future Work

We plan to further investigate the performance of Adrenaline under more realistic network conditions and hardware configurations. In addition, we plan to explore more heuristics on page decomposition, as well as providing APIs for web developers to express page-level parallelism. Finally, we plan to apply Adrenaline to a larger set of web sites to evaluate our techniques more comprehensively.

5.5 Key Papers and Other Material

We have included one paper in this book. It is “A Case for Parallelizing Web Pages” from HotPar 2012 [106].

Chapter 6

Refactoring

6.1 Problem Addressed

One approach to parallelize an existing sequential program is to rewrite it from scratch. However, this requires a lot of programmer effort. Another approach is to use an automatic parallelizing compiler [8, 10, 11, 96]. Despite continuous improvements on compilers, programmers still need to change their programs to make compilers work acceptably. Unfortunately, knowing where to introduce parallelism requires domain knowledge and understanding of the program's algorithms and data structures.

In practice, the most widely used approach is to parallelize a program *incrementally* by changing the existing code. Each small step can be seen as a behavior-preserving transformation, i.e., a *refactoring*. Programmers prefer this approach because it is safer: they maintain a working, deployable version of the program. Also, the incremental approach is more economical than rewriting from scratch.

Unfortunately, little is known about the kinds of refactorings that programmers use to change real-world programs for parallelism. The existing books on parallel programming and API documentation of parallel constructs primarily focus on designing parallel programs from scratch. Therefore, average programmers lack educational resources on how to retrofit existing programs for parallelism through refactoring. They also miss examples of successful projects that were refactored for parallelism.

Programmers also lack tools that automate these refactorings. Despite high-productivity parallel libraries, refactoring sequential code for parallelism is still tedious, because it requires changing many lines of code, and error-prone, because programmers need to ensure non-interference of parallel operations. For example, an expert parallel programmer who parallelized six simple divide-and-conquer algorithms like quicksort and mergesort using Java 7's `ForkJoinTask` spent on average 30 minutes and changed 50 lines per algorithm [48].

Automating refactorings is challenging as it requires complex code transformations that span multiple, non-adjacent program statements and requires deep inter-procedural analyses that globally reason about objects shared through the heap. A key problem is designing program analyses that are accurate yet fast enough to be used in an interactive tool.

In the past, under sequential programming, interactive refactoring tools have revolutionized how programmers approach software design. Without refactoring tools, programmers often over-designed, because it was expensive to change the design once it was implemented. Refactoring tools have enabled programmers to continuously explore the design space of large codebases, while preserving the existing behavior. Modern IDEs incorporate refactoring in their top menu, and often compete on the basis of refactoring support.

Looking forward, under parallel programming, we envision that refactoring tools for retrofitting parallelism can be similarly transformative. They will enable programmers to safely and efficiently explore the space of performance optimizations and parallel constructs, while preserving the existing functionality.

6.2 Contributions

In this project, we embarked on an effort to develop automated refactoring tools. We started by empirically studying refactorings used [49] or misused [105] in practice, and how developers are embracing parallel libraries [120]. Based on these findings, we built a Java refactoring toolset [46–48, 50, 66, 75, 93, 137, 162] that currently automates ten refactorings that fall into three categories. First, Refactorings for Thread-Safety make a program thread-safe, e.g., by synchronizing accesses to shared state via library classes. Second, Refactorings for Throughput add multi-threading via task and loop parallelism. Third, Refactorings for Scalability replace lock-based synchronization with accesses to lock-free, highly scalable data structures. Our empirical evaluation shows that our toolset is useful: it reduces the burden of analyzing and modifying code, it is fast enough to be used interactively, it correctly applies transformations that open-source developers applied incompletely, and the refactored code exhibits good speedup.

We also validated rigorously the research results by employing empirical methods (e.g., case studies, controlled experiments, and interviews) in the evaluation stage (did we built the *tool right*?) and also in the formative stage (are we building the *right tool*?). Next, we describe the specific contributions we made.

6.2.1 Empirical Studies with Actionable Items

• **Usage of Parallel Libraries [49]** Industry leaders hope to convert the hard problem of *using parallelism* into the easier problem of *using a parallel library*. Yet, we know little about how programmers adopt these libraries in practice. Without such knowledge, other programmers cannot educate themselves about the state of the practice, library designers are unaware of API misusage, researchers make wrong assumptions, and tool vendors do not support common usage of library constructs. We conducted the first study that analyzes the usage of parallel libraries in a large scale experiment. We analyzed 655 open-source applications that adopted Microsoft's new parallel libraries – Task Parallel Library (TPL) and Parallel Language Integrated Query (PLINQ) — comprising 17.6M lines of code written in C#. These applications are developed by 1609 programmers. Using this data, we answered 8 research questions and uncovered some interesting facts. For example, (i) for two of the fundamental parallel constructs, in at least 10% of the cases developers misuse them so that the code runs sequentially instead of concurrently, (ii) developers make their parallel code unnecessarily complex, and (iii) applications of different size have different adoption trends. Library designers confirmed that our findings are useful and will influence the future development of the libraries.

• **Usage and Misusage of CHECK-THEN-ACT Idioms [105]** Concurrent collections provide thread-safe, highly-scalable operations, and are widely used in practice. However, programmers can misuse these concurrent collections when composing two operations where a *check* on the collection (such as non-emptiness) precedes an *action* (such as removing an entry). Unless the whole composition is atomic, the program contains an atomicity violation bug. We conducted the first empirical study of CHECK-THEN-ACT idioms of Java concurrent collections in a large corpus of open-source applications. We cataloged nine commonly misused CHECK-THEN-ACT idioms and showed the correct usage. We quantitatively and qualitatively analyzed 28 widely-used open source Java projects that use Java concurrency collections – comprising 6.4M lines of code. We classified the commonly used idioms, the ones that are the most error-prone, and the evolution of the programs with respect to misused idioms. We implemented a tool, CTADETECTOR, to detect and correct misused CHECK-THEN-ACT idioms. Using CTADETECTOR, we found 282 buggy instances. We reported 155 to the developers, who examined 90 of them. The developers confirmed 60 as new bugs and accepted our patch. This shows that CHECK-THEN-ACT idioms are commonly misused in practice, and correcting them is important.

• **Usage of Concurrent Refactorings [49].** A major software maintenance task in the multicore era will be to make sequential programs concurrent. Must concurrency be designed into a program, or can it be retrofitted later? What are the most common transformations to retrofit concurrency into sequential programs? Are these transformations random, or do they belong to certain categories? How can we automate these transformations? To answer these questions, we analyzed the source code of five open-source Java projects and looked at a total of 14 versions. We analyzed qualitatively and quantitatively the concurrency-related transformations. We found that these transformations belong to four categories: transformations that improve the responsiveness, throughput, scalability, or correctness of the applications. In 73.9% of these transformations, concurrency was retrofitted on existing program elements. In 20.5% of the transformations, concurrency was designed into new program elements. Our findings educate software developers on how to parallelize sequential programs, and provide hints for tool vendors about what transformations are worth automating.

6.2.2 Refactoring and Analysis Tools

• **CONCURRENCER [47, 48]** The Java 5 package `java.util.concurrent (j.u.c.)` supports writing concurrent programs: much of the complexity of writing thread-safe and scalable programs is hidden in the library. To use this package, programmers still need to reengineer existing code. This is tedious because it requires changing many lines of code; it is error-prone because programmers can use the wrong APIs; and it is omission-prone because programmers can miss opportunities to use the enhanced APIs. Our tool, CONCURRENCER enables programmers to refactor sequential code into parallel code that uses three `j.u.c.` concurrent utilities. CONCURRENCER does not require any program annotations. Its transformations span multiple, non-adjacent, program statements. A find-and-replace tool cannot perform such transformations, which require program analysis. Empirical evaluation shows that CONCURRENCER refactors code effectively: CONCURRENCER correctly identifies and applies transformations that some open-source developers overlooked, and the converted code exhibits good speedup.

• **LAMDBAFICATOR [66, 75]** Java 8 introduces two functional features: lambda expressions and functional operations like `map` or `filter` that apply a lambda expression over the elements of a `Collection`. Refactoring existing code to use these new features enables explicit but unobtrusive parallelism and makes the code more succinct. However, refactoring is tedious: it requires changing many lines of code. It is also error-prone: the programmer must reason about control flow, data flow, and side effects. Fortunately, refactorings can be automated. We designed and implemented LAMBDAFICATOR, a tool which automates two refactorings. The first refactoring converts anonymous inner classes to lambda expressions. The second refactoring converts `for` loops that iterate over `Collections` to functional operations that use lambda expressions. Using 9 open-source projects, we applied these two refactorings 1263 and 1709 times, respectively. The results showed that LAMBDAFICATOR is useful: (i) it is widely applicable, (ii) it reduces the code bloat, (iii) it increases programmer productivity, and (iv) it is accurate.

• **IMMUTATOR [93]** It is common for object-oriented programs to have both mutable and immutable classes. Immutable classes make parallel programming simpler. Since threads cannot change the state of an immutable object, they can share it without synchronization. An immutable object is embarrassingly thread-safe. Sometimes programmers write immutable classes from scratch, other times they transform mutable into immutable classes. To transform a mutable class, programmers must find all methods that mutate its transitive state and all objects that can enter or escape the state of the class. The analyses are non-trivial and the rewriting is tedious. Fortunately, this can be automated. Our algorithm and tool, IMMUTATOR, enables the programmer to safely transform a mutable class into an immutable class. Two case studies and one controlled experiment showed that IMMUTATOR is useful. It (i) reduces the burden of making classes immutable, (ii) is fast enough to be used interactively, and (iii) is much safer than manual transformations.

• **ITERACE [137]** Despite significant progress in recent years, the important problem of static race detection remains open. Previous techniques took a general approach and looked for races by analyzing the effects induced by low-level concurrency constructs (e.g., `java.lang.Thread`). But constructs and libraries for expressing parallelism at a higher level (e.g., fork-join, futures, and parallel loops) are becoming available in all major programming languages. We claim that specializing an analysis to take advantage of the extra semantic information provided by the use of these constructs and libraries improves precision and scalability. We designed ITERACE, a set of techniques that are specialized to use the intrinsic thread, safety, and data-flow structure of collections and of the new loop-parallelism mechanism to be introduced in Java 8. Our evaluation showed that ITERACE is fast and precise enough to be practical. It scales to programs of hundreds of thousands of lines of code and it reports few race warnings, thus avoiding a common pitfall of static analyses. The tool revealed six bugs in real-world applications. We reported four of them; one had already been fixed, and three were new and the developers confirmed and fixed them.

• **DPJIZER [162]** We have also developed refactoring tools to port existing code to new parallel programming languages. We participated in the development of Deterministic Parallel Java (DPJ) (Chapter 8), a language that aims to make parallel programming deterministic by default. At the heart of DPJ is a type and effect system that ensures that the parallel tasks are non-interfering. Our tool, DPJIZER, reduces the burden of writing annotations, by automatically inferring the method effects using a constraint-based algorithm.

6.2.3 Dissemination of Results

1. **Product Releases**. One of our tools, LAMDAFICATOR, is already shipping with the official release of the NETBEANS Integrated Development Environment (IDE). Another tool, CONCURRENCER is ongoing integration in the ECLIPSE IDE. Both NETBEANS and ECLIPSE are open-source IDEs, and are used by millions of Java developers everyday. We are also committed to work with the Microsoft Visual Studio team to integrate refactoring tools into the official release of Visual Studio.

2. **Tool Releases**. All our tools and empirical data are released as open-source code. In cases when the tools are not shipping with the official release of an IDE, we package them as plugins for the IDE, which are easily installable. Thus, researchers and software engineers can immediately use our results.

3. **Publications.** Our research results have been published in the major software engineering conferences (FSE'13, ISSTA'13, ICST'13, ICSE'12, FSE'12, ICSE'11, IEEE Software'11, ASE'09, ICSE'09, OOPSLA'09). Two of our papers have received awards: our ICST'13 paper [105] got the Best Paper Award, and our ISSTA'13 paper [137] got the ACM SIGSOFT Distinguished Paper Award.

4. **Education and Outreach**. We have taught the refactoring tools and techniques in the undergraduate/-graduate software engineering courses (CS427/527) at the University of Illinois, as well as a special topics class on Multicore Programming [44]. In addition to traditional audiences, we have reached over 800 professional programmers at three one-week long Industrial Training courses at Boeing, four Summer Schools at Illinois [83] and one in Singapore, and two conference tutorials [43, 45]. We have also developed resources for educating developers. Our `http://learnparallelism.net` shows thousands of relevant examples of how to use parallel APIs. Our website received more than 34,000 visits within a year.

6.3 Lessons Learned

Building this refactoring toolset taught us several lessons. First, programmers often use parallel libraries, thus refactoring tools need to support such libraries. Second, to keep the programmer engaged, refactoring tools need to finish in less than thirty seconds. Thus, they must use efficient, on-demand program analyses. Third, program analysis libraries and IDEs with excellent AST rewriting capabilities are essential for building refactoring tools. Fourth, once a program is parallel, it must remain maintainable, i.e., readable and portable. Fifth, refactoring tools must interact with other tools in the parallel toolbox, such as profilers and data-race detectors. Sixth, even carefully designed library APIs can still be misused by developers, so empirical studies are crucial in discovering common usage errors.

6.4 Future Work

There are new forces that shape computing. First, mobile devices and web applications represent the environment and the applications where end-users spend most of their time. Second, JavaScript is the *lingua franca* of the Internet. Third, mobile devices are all moving into the multicore direction. Given the confluence of these three trends, refactorings that enable mobile and web programmers to transform their code so that their applications can run faster on the multicore devices can have a tremendous impact.

We will implement refactorings that improve responsiveness, e.g., by extracting a lengthy computation that is currently scheduled on the GUI Event Dispatch Thread into an asynchronous task, running in a background thread. Such refactorings are extremely important for mobile devices, where users' perception of responsiveness makes the difference between a usable and an unusable application.

A second line of work, on refactoring for parallel JavaScript, has huge rewards too. We are currently studying the River Trail implementation of Intel's flavor of `ParallelArray` for JavaScript. We are also surveying the literature to find the unique challenges and state-of-the-art program analysis for dealing with JavaScript. We are collaborating with a team at Intel Labs on this project.

6.5 Key Papers and Other Material

We include two representative papers in this book, one on the empirical work and another on the refactoring tools. The first one is "How Do Developers Use Parallel Libraries?" from FSE 2012. This paper presents a large-scale study on the usage of Microsoft Parallel Libraries. Our study has several practical implications. First, it is a tremendous resource for educating developers. Our `http://learnparallelism.net` shows thousands of relevant examples of how to use parallel APIs. We received more than 34,000 visitors within the last year. Second, it provides value for researchers, tool vendors, and the software testing/verification community. Microsoft library designers confirmed that our findings are useful and will influence the future development of their libraries.

The second paper is "Transformation for Class Immutability" from ICSE 2011. This paper illustrates the challenges of implementing one refactoring, namely converting a mutable into an immutable class. Since its state cannot be mutated once an object is properly constructed, the immutable class is thread-safe. Immutability plays an important role in other fields, e.g., in computer security, memory optimizations, and distributed computing. Our refactoring tool, IMMUTATOR, uses a demand-driven points-to analysis to find mutator methods and objects that enter into or escape from the object's transitive state. Our 3-pronged evaluation involves running IMMUTATOR on 346 classes from popular open-source projects, a case-study of how open-source developers refactor manually, and a controlled experiment. The evaluation shows that IMMUTATOR is useful.

All refactoring tools and the experimental data are freely available for download at:
`http://refactoring.info/tools/`.

Chapter 7

Tiling: Notations and Optimization Techniques

7.1 Problem Addressed

A widely-used program optimization strategy consists of partitioning arrays and organizing computations in terms of the subarrays resulting from the partition. This strategy was, to the best of our knowledge, first introduced in the 1950s [139] for enhancing locality, and is universally used today for this purpose [54, 74, 165]. In parallel computing, the way partitioning is done [64, 90, 103] determines the performance of both distributed and shared-memory computations.

In this project, we studied three topics in the area of data partitioning:

- Notations that explicitly manipulate data blocks.
- The use of partitioning to generate multiple implementations of linear algebra algorithms.
- Compiler techniques for the automatic partitioning of arrays.

7.2 Contributions

This section outlines our contributions in each of the three areas.

7.2.1 Programming with Tiles

Our work on programming notations is based on the assumption that two key characteristics facilitate the development, maintenance, and portability of parallel programs: (i) a global view of the data and explicit representation of the data partition, and (ii) a single thread of execution and parallelism encapsulation.

Today's codes typically do not possess these two characteristics. Thus, data partitioning is typically represented implicitly with locality-enhancement strategies, often relying on control flow to schedule array accesses one block at a time and implicit data partitioning being the norm in MPI codes. Also, multiple threads of control, and the difficulties they create, are pervasive in today's parallel programs.

Our work in this area started with the design of Hierarchically Tiled Arrays (HTAs) [20, 21, 72], which are partitioned arrays whose components could be scalars or lower-level partitioned arrays. The blocks or tiles of the HTAs can be referenced explicitly and used to control locality, data distribution or both. For distributed-memory computations, the outermost tiles are typically distributed across processors for parallelism, and the inner tiles are used to improve locality within a processor. Figure 7.1 depicts an HTA with two levels of partitioning. The figure also shows how elements of an HTA can be referenced. In the figure, braces are used to index tiles and parentheses to index scalar elements. As can be seen, all or just some of the tile indices can be omitted to enable global indexing of the scalar elements (i.e., to ignore tiling).

We can illustrate the use of HTAs as a notation to enhance locality using matrix-matrix multiplication whose tiled version takes the following form in MATLAB notation:

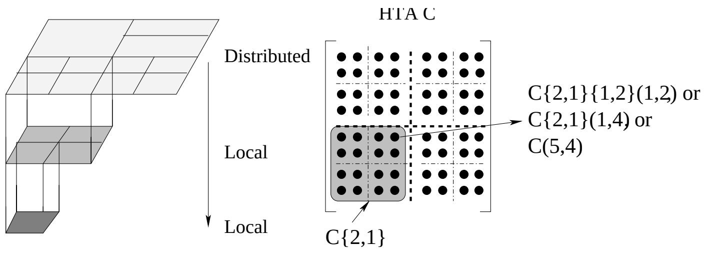


Figure 7.1: Pictorial view of a hierarchically tiled array.

```
for i=1:n
    for j=1:n
        for k=1:n
            c{i,j} = c{i,j} + a{i,k} * b{k,j};
```

Here, the operation * represents matrix-matrix multiplication, not element-by-element product, and a, b, and c are two dimensional HTAs. Clearly, even in this simple case, our notation simplifies coding since, to achieve the same result using scalar notation would require six levels of nesting.

With HTAs, data parallel computations are represented as element-by-element operations. The operations within each tile, in the case of unary operations, or within corresponding tiles, in the case of binary operations, can be executed in parallel with each other. Thus, if the operation sin(C) was applied on the HTA C depicted in Figure 7.1, it would be 16-way parallel if the leaf tiles were assigned to different processors, or just 4-way parallel if only the top level tiles were distributed. Similarly, assuming a second HTA A, with identical shape and distribution as those of C, the operation A+C could be 16-way or just 4-way parallel.

There is no communication in these operations. Sin is a unary operation and tiles are identically distributed in the addition operation. Communication is needed when the corresponding tiles in a binary operation are in different nodes, or when we are executing assignments between tiles that are in different processors. Consider for example the 3×3 HTA V, of Figure 7.2. If each tile of V were assigned to a different node of a distributed memory machine, the assignment V{2:n,:}(1,:)=V{1:n-1,:}(n,:) would copy all the elements in the last row of each tile to the first row of their neighbor tile below it.

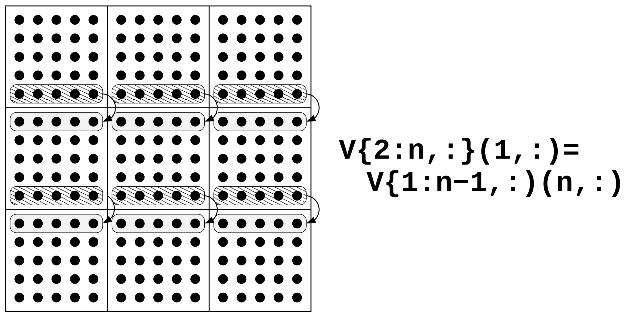


Figure 7.2: Assignment of the elements in the last row of tiles to the first row of their neighbors below.

Besides assignment and arithmetic operations, several other HTA methods that operate at the tile level facilitate the expression of communication and parallel computation. These include circshift, transpose, repmat, permute, reduce and hmap, as well as the standard arithmetic operators. For example, the MATLAB circshift function implements circular shifts of array elements. The HTA version shifts instead whole tiles of HTAs, which could require communication.

For example, in the following tiled version of Cannon's algorithm (assuming that c is initially all zeroes):

```
for i=2:n {
    a{i:n,:} = cshift(a(i:n,:},dim=2,shift=1);   !Communication
    b{:,i:n} = cshift(b{:,i:n},dim=1,shift=1)    !Communication
    }
for k=1:n {
    c{:,:} = c{:,:}+a{:,:}*b{:,:};              !Parallel computation
    a{:,:} = cshift(a{:,:},dim=2);               !Communication
    b{:,:} = cshift(b{:,:},dim=1);               !Communication
    }
```

all the parallelism is in the statement

```
c{:,:} = c{:,:}+a{:,:}*b{:,:};
```

which represents n^2 independent tile operations. In this case, HTA also simplifies coding as the code above is significantly simpler than its MPI counterpart.

Particularly useful in many parallel computations are the `reduce` and `hmap`. `Reduce` applies associative operations between components of an HTA and `hmap` applies the same function to each tile of an HTA or to the corresponding tiles of different HTAs when the `hmap` implements an n-ary operation. Other more complex operators such as `scan`, `mapReduce` and operators for dynamic partitioning and overlapped tiling are also useful [21, 72].

The tiled notation can also be applied successfully to data structures other than arrays. The PhD thesis of James Brodman [31] is a study of notations to tile sets for parallel symbolic computations.

7.2.2 HYDRA: Automatic Tuning from Equations

During the last decade there has been growing interest in autotuning systems which automatically explore the space of algorithms and implementations seeking high performance. An influential autotuning system, SPIRAL, which targets signal processing algorithms starts with formulas and transforms them to generate the space of possible versions [131, 168]. SPIRAL then does empirical search, looking for the best-performing version.

Although there has been much work on autotuning for linear algebra [22, 166], existing experimental systems typically focus on the autotuning of the basic linear algebra operations, and do not manipulate formulas. Building autotuning systems around the manipulation of formulas has several advantages. One is that, with such systems, it is possible to extend programming languages with formulas. This has great potential for programmer productivity.

The Ph.D. dissertation of Alexandre Duchateau [59] describes a system, HYDRA, which starts with linear algebra equations and uses data partitioning to generate several solvers for the equation. Like SPIRAL, HYDRA then applies empirical search to identify the most efficient of all the generated solvers. To illustrate how Hydra operates, consider first an extremely simple equation such as $X = A * B$, where A and B are known, square, $n \times n$ matrices and X is the unknown. Clearly, solving this equation is tantamount to carrying out matrix-matrix multiplication, but we must represent it as an equation because that is the type of input that HYDRA accepts.

By partitioning A and B, we can generate different ways of finding X. For example, if A is partitioned in the vertical direction into the two submatrices, $A(1 : n/2, :)$ and $A(n/2 + 1 : n, :)$, which we represent in our notation as $A\{1\}$ and $A\{2\}$ and B is not partitioned, the original equation could be transformed into two equations $X\{1\} = A\{1\} * B$ and $X\{2\} = A\{2\} * B$. On the other hand, if B is partitioned along the horizontal direction into the two submatrices, $B(:, 1 : n/2)$ and $B(:, n/2 + 1)$, the original equation can be translated into four equations $X\{i, j\} = A\{i\} * B\{j\}$, with $i, j = 1, 2$. Thus, each way of partitioning the matrices produces a different way of solving the equation. In this case, all resulting equations are the same as the original equation except that the sizes are smaller.

This process of partitioning can be repeated recursively. In theory, we could go all the way until the resulting equations only involve scalars, but this would typically not be a good idea. We assume, instead, that there is an implementation of matrix-matrix multiplication that can be used to solve the final equations. This implementation does not have to be sophisticated since the partitioning strategy would naturally introduce locality and parallelism. The equations can be solved in parallel since they are independent from each other.

When introducing parallelism, HYDRA sometimes must enforce a partial order of execution. Consider the equation $L * X * U - X = C$, where L is lower triangular, U is upper triangular and X is the unknown. If the matrices are partitioned into 2×2 submatrices, HYDRA transforms the original equation into four equations:

1. $L\{1,1\} * X\{1,1\} * U\{1,1\} - X\{1,1\} = C\{1,1\}$
2. $L\{2,1\} * X\{1,1\} * U\{1,1\} + L\{2,2\} * X\{2,1\} * U\{1,1\} - X\{2,1\} = C\{2,1\}$
3. $L\{1,1\} * X\{1,1\} * U\{1,2\} + L\{1,1\} * X\{1,2\} * U\{2,2\} - X\{1,2\} = C\{1,2\}$, and
4. $L\{2,1\} * X\{1,1\} * U\{1,2\} + L\{2,2\} * X\{2,1\} * U\{1,2\} + L\{2,1\} * X\{1,2\} * U\{2,2\} + L\{2,2\} * X\{2,2\} * U\{2,2\} - X\{2,2\} = C\{2,2\}$

To solve the second and third equations, equation 1 must be solved first. To solve equation 4, the other three must be solved first. HYDRA was implemented and evaluated for a shared-memory multiprocessor with serial MKL routines used as the basis for the computation.

7.2.3 Automatic Selection of Block Shapes

The kernels of several important applications take the form of stencil computations. For these computations, data partitioning is of great importance for performance on all classes of machines. Consequently, in collaboration with Intel's Jean-Pierre Giacalone, Robert Kuhn, and Yang Ni, we studied automatic partitioning strategies for stencil computations on multicores. The main objectives were to improve performance and energy efficiency.

When there are multiple consecutive applications of stencil computations as in the code

```
parallel for (int i=1:n)      A[i] = . . .
parallel for (int i=2:n-1)  . . . = A[i-1]+A[i]+A[i+1]
```

there could be a significant number of cache misses, depending on how the iterations are scheduled and how many processors are used for the execution of the two loops. Furthermore, for correctness, a barrier is typically assumed at the end of the first loop, which further hinders performance.

To avoid both of these problems, the computation can be organized as depicted in Figure 7.3-(a), where each processor computes redundantly the array elements it needs for successive application of the stencil computation. The array elements that are used in the redundant computations are shown darker. The problem with this approach is the need to do redundant computations. The Ph.D. thesis of Xing Zhou [174] introduced a strategy that implements the overlapped tiling in a hierarchical fashion (Figure 7.3-(b)), to attenuate the effect of this redundancy. The strategy was implemented on top of an experimental OpenCL compiler developed at Intel Labs. Its effect was evaluated using multicores. Several other block selection strategies are studied in Zhou's Ph.D. thesis.

7.3 Lessons Learned

The three projects discussed above had positive outcomes. The evaluations [14, 72] show the significant programmability advantage of the proposed notation over conventional notations such as OpenMP and MPI. While ease of programming affects performance, the impact can be reduced by refactoring the code manually or automatically [65]. The HYDRA autotuning system produces in several cases parallel codes that match and even outperform the performance of manually-written MKL codes [58]. Finally, the automatic blockshape selection techniques led to significant performance improvements over the unoptimized versions [175].

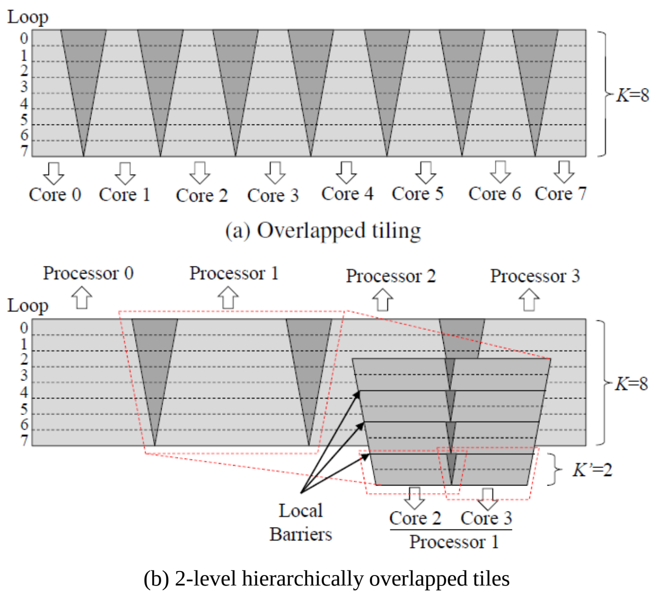


Figure 7.3: Overlapped tiling and hierarchically overlapped tiles.

7.4 Future Work

There are numerous open problems related to each of the three projects discussed in this paper. A particularly important direction is to merge the work on autotuning with that of high-level notations. In this way, it would be possible to include equations and other formulas in the code. This would not only improve readability and portability but also help automatic optimization.

7.5 Key Papers and Other Material

We include two papers in this book. The first one is “Hydra: Automatic Algorithm Exploration from Linear Algebra Equations” from CGO 2013 [58]. The other is “Hierarchical Overlapped Tiling” from CGO 2012 [175].

Chapter 8

Deterministic-by-default Parallel Programming

8.1 Problem Addressed

Parallelism is needed to utilize modern hardware, but it introduces numerous correctness challenges. Parallel programs can suffer from problems such as data races, atomicity violations, insufficient ordering synchronization, and deadlocks. The first three problems all lead to unintentional nondeterminism, i.e., producing different observable results in the different executions for the same input. Moreover, these classes of concurrency errors are often difficult to reproduce and debug after they occur, because they are often schedule-dependent and non-deterministic. For large production software, where debugging, testing and maintenance are expensive and challenging tasks, reproducibility of results – obtaining *the same externally visible results for every execution on a given input* – is essential for reasoning about the correctness of software. For parallel programming to become ubiquitous, therefore, it is essential that concurrency errors that produce unintentional nondeterminism be eliminated or made extremely unlikely to occur.

In practice, most *algorithms* are deterministic, and their sequential reference implementations are deterministic as well. It is highly inefficient and unproductive to write such algorithms in a parallel notation (language or library) that allows non-deterministic, schedule-dependent behavior due to programming bugs and then expect programmers to find and eliminate those bugs. We expect programmers would have *much higher productivity* if the parallel notation guaranteed the absence of such bugs in the first place, i.e., guaranteed deterministic results.

A purely deterministic programming notation would likely be too restrictive, however. Some parallel algorithms permit non-deterministic results, i.e., different results for a given input. Examples include branch-and-bound search (where any solution that meets a certain cost criterion is acceptable), some parallel graph algorithms (where there are multiple ways of extracting subgraphs with a particular property, e.g., spanning tree), and large internet servers (which service large numbers separate requests from external users). Such algorithms do not expect a guarantee of determinism, and there may be no sequential equivalent, but the developers can still benefit from other strong guarantees. Moreover, such algorithms are commonly combined with more conventional deterministic parallel algorithms in the same program. Deterministic parallelism may be encapsulated within a larger non-deterministic parallel structure (e.g., parallel physics simulations within a non-deterministic multi-player game), or vice versa (e.g., a non-deterministic branch-and-bound optimization followed by a deterministic post-processing of the results, all performed as a step of a larger deterministic simulation).

8.2 Contributions

One major focus of our research has been techniques to enable disciplined parallel programming models that *guarantee the absence of concurrency errors*, rather than requiring complicated debugging after the fact.

A key outcome of our work has been to define what guarantees a disciplined parallel programming model should provide. First, the model should enable the programmer to distinguish explicitly between deterministic and non-deterministic parallel constructs, so that it is clear to both other programmers and to programming tools where non-deterministic results are intentional and acceptable. Second, the model should guarantee deterministic results with sequential equivalence for all portions of a program that do not include an explicit non-deterministic parallel construct. Third, for programs with nondeterministic constructs, the programming model should guarantee data race freedom and also guarantee atomicity with strong isolation for the entire program, even when using nondeterministic constructs. Collectively, we refer to these guarantees as "determinism-by-default." *These guarantees are stronger than* any *previously existing parallel programming notation we know of that supports both deterministic and non-deterministic parallel algorithms.*

Our second key contribution has been to show that the above guarantees are not only possible, but can be achieved with a purely static type system for a large class of fork-join parallel programs, even in an expressive object-oriented base language such as Java. We have implemented this type system as an extension of Java we call Deterministic Parallel Java (DPJ), which guarantees determinism-by-default to conforming parallel applications. We have also explored a pure logical annotation language, called ACCORD, that allows annotating programs written in existing languages (such as standard Java), and provides automatic mechanisms to use these annotations to prove the program race-free and deterministic [92]. Moreover, we have developed Annotations for Safe Parallelism (ASP), an annotation language and checker for C/C++ programs that combines the ideas behind DPJ and ACCORD, which aims to bring determinism-by-default to very large scale real-world applications (containing multiple millions of lines of code). One of the major challenges to achieving that is *automatically inferring* the required annotations with little input from the programmer.

Our third key contribution has been a hybrid static-dynamic scheduling strategy we call Tasks With Effects (TWE) that provides nearly the full determinism-by-default guarantees (except for certain deadlocks) to a much broader class of programs with unstructured parallelism. We have implemented TWE as an extension of the DPJ language, together with a novel scheduler that exploits the structure of effect summaries for scalability.

Our fourth key contribution has been an automatic inference algorithm that infers roughly half of the type annotations used in the DPJ, ASP and TWE languages fully automatically, given the other half. We are now extending this inference to nearly the complete set of annotations, and will incorporate it into the ASP system.

The research on the DPJ component of this work won the lead author, Robert Bocchino, an *ACM SIGPLAN Distinguished Dissertation Award*. The following subsections describe the contributions in detail.

8.2.1 Deterministic Parallel Java: Strong Static Safety Guarantees

The Deterministic Parallel Java (DPJ) language uses static checking to guarantee determinism-by-default for programs with well-structured parallelism (nested fork-join, i.e., where task creation and joins are properly nested) [25, 29]. DPJ requires the programmer to write annotations that partition the heap into named sets of memory locations called *regions*, and to specify the *effects* of each method in terms of the regions read and written. The DPJ compiler statically verifies that these effect specifications are correct, and the region-based type annotations are type-safe. It then uses the method effect summaries and region information to infer the effects of every task, and then to check that no two parallel tasks can have conflicting effects, i.e., that the two tasks can run safely in parallel. The fork-join assumption is needed solely to infer which tasks may run in parallel with each other, without expensive interprocedural analysis. The DPJ region annotations and effect summaries use a flexible hierarchical notation called Region Path Lists (RPLs) to support a wide range of parallel data access patterns; the RPL notation is key to expressing a wide range of parallel data structures and idioms.

Experiments show that deterministic DPJ programs achieve speedups comparable to equivalent multi-threaded Java programs with no safety guarantees: in fact, there are *no run-time overheads* for deterministic programs. Figure 8.1 shows the speedups for a range of benchmarks; they are relative to an equivalent sequential algorithm with no parallelism overheads. For three of these benchmarks – Monte Carlo, IDEA encryption and Barnes-Hut – hand-coded Java threads programs that lack any safety guarantees were available, and the speedups with DPJ were close to or better than the Java speedups in all three cases. DPJ achieves good speedups even for highly irregular pointer-based codes such as Barnes-Hut and CollisionTree. Moreover, our experience has shown that a wide range of parallel idioms can be expressed in DPJ, such as parallel loops over arrays (including arrays of pointers to distinct objects), parallel traversals of trees with (non-conflicting) updates, and parallel divide-and-conquer algorithms with (non-conflicting) in-place array updates. This is the first time that static checking of such potentially irregular parallel algorithms has been shown to be feasible.[1]

One major restriction in DPJ is that it does not support unstructured parallelism, common in concurrent programs such as servers and interactive applications. A second restriction is that algorithms that traverse and update data structures, e.g., arrays or trees, in parallel cannot restructure those data structures after they are constructed: they must create new copies, which can incur unnecessary run-time overhead. Both these limitations are addressed by the TWEJava language, described below.

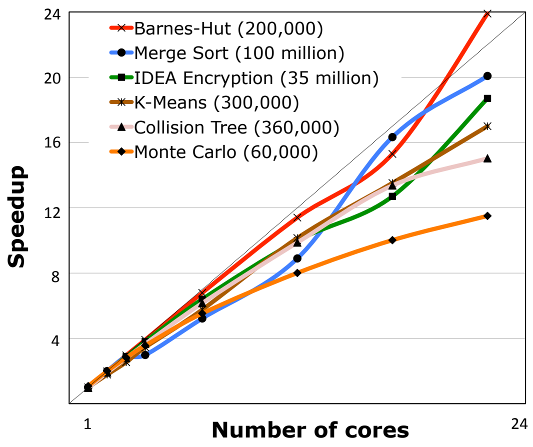


Figure 8.1: Speedups for DPJ benchmarks on a 24-core Dell R900.

DPJ also supports programs where the programmer explicitly mixes nondeterministic and deterministic parallel constructs, by adding atomic blocks for protecting potentially conflicting accesses to shared data [28]. DPJ still ensures the full guarantees of determinism-by-default. In particular, it guarantees determinism for all parts of a program that do not encounter an explicit non-deterministic parallel construct in some execution. Also, DPJ guarantees data race freedom and atomicity with strong isolation for the entire program, even when using nondeterministic constructs. Together, *these guarantees are the strongest parallelism guarantees we know of in* any *parallel programming language that allows both deterministic and non-deterministic semantics*.

8.2.2 Tasks With Effects: Supporting Flexible Concurrent Programs

Second, we have developed the Tasks With Effects (TWE) programming model, which enables us to give nearly all the determinism-by-default safety guarantees for programs that may include arbitrary, unstructured parallelism and concurrency constructs [77]. TWE also supports more flexible parallel idioms such as restructuring of concurrent data structures while still permitting parallel traversals and updates. Some examples of such programs are interactive applications and servers that may begin a concurrent task in response to user input or a client request, programs structured in terms of modules or "actors" that communicate asynchronously, and programs that may rebalance a binary search tree or permute a parallel array.

The key to allowing arbitrary, unstructured parallelism or arbitrary restructuring of parallel data structures is to use a run-time scheduler to determine which tasks are safe to run in parallel. In the TWE model, a program is composed of tasks with effect summaries that are available to the run-time scheduler. The effect summaries are inferred using some combination of user-specified effect summaries on methods (similar to DPJ programs) or interprocedural effect inference (similar to the DPJizer project, below). The scheduler ensures that no two tasks with conflicting effects are run concurrently. By using statically checkable effect specifications on tasks, such as those used in DPJ, no run-time checks are needed on individual memory accesses. This combination of static and dynamic elements efficiently guarantees that concurrently-running tasks are isolated from each other, with non-conflicting memory accesses. This gives a guarantee of data race freedom, and also guarantees atomicity for portions of tasks that do not create or wait for other tasks.

Our TWEJava language implements this model for Java programs, using the DPJ effect system for the method and task effect summaries. Experiments show that TWEJava supports more general concurrency struc-

[1]Automatic parallelization techniques based on interprocedural static analyses such as shape analysis have attempted some of these before, but they are extremely *conservative*.

tures than DPJ or any other language limited to nested fork-join parallelism, while still delivering guarantees nearly as strong as DPJ. We initially developed a relatively simple TWEJava scheduler that holds all tasks in a single queue protected by a single lock. This could give reasonable performance for some programs with relatively few tasks that needed to be scheduled, but its scalability was limited when dealing with codes that used fine-grained parallelism [77]. We have recently developed a new scheduler that exploits the hierarchical structure of effect summaries to minimize effect comparisons between tasks. The new scheduler is far more scalable than the previous one for TWEJava, and gives fairly good speedups (measured up to 80 threads) for several benchmarks, including the K-Means benchmark which uses fine-grain critical sections.

8.2.3 Logical annotations for safe parallelism using Accord

In the work in ACCORD [92], we have explored building an annotation language for thread contracts, using which a programmer can annotate a program in a conventional programming language (like standard Java) in order to prove that the program is race-free and/or deterministic. Intuitively, in the annotation language ACCORD, every parallel-for loop gets annotated to logically specify precisely the read/write set of locations of each parallel iteration. Furthermore, each method is also annotated to specify the read/write set of locations that the method will read and write to. Using these annotations, we can build algorithmic methods, using logical constraint solvers like SMT solvers, in order to automatically verify whether the annotations imply race-freedom, by checking whether the read/write sets of each thread is disjoint from the write-set of other threads. In order to verify that the annotations themselves are correct (i.e., whether the program meets the annotations), we can either achieve this using sound logic tools for verifying purely *sequential programs* or resort to incomplete methods such as testing. Using ACCORD, we have verified a large number of both interesting programs as well as large programs and showed that the annotation mechanism is very effective in both capturing the concurrency as well as proving race-freedom. For instance, using ACCORD annotations, we have annotated programs in the Spec OMP2001 benchmark, which are thousands of lines long, and both proved such programs race-free as well as discovered races in some programs which have eluded detection for more than 10 years.

8.2.4 Annotations for Safe Parallelism: Scaling to Large Production Software

We have taken the lessons and techniques from both DPJ and ACCORD and are applying them to C++ code at a much larger scale. We have developed an annotation language called Annotations for Safe Parallelism (ASP), which can provide nearly[2] as strong guarantees as DPJ, for C++ programs parallelized with any language or library approach that is restricted to nested fork-join parallelism, e.g., TBB, OpenMP, or Cilk.

The key new challenges for ASP are to minimize or eliminate the annotation burden for very large applications, and to support large reusable parallel libraries. First, the annotation burden of DPJ is not acceptable for applications of millions of lines of code, because of the number of annotations that would be needed, and also because they require understanding new and sophisticated language concepts. Second, DPJ focuses on checking explicitly parallel code but in practice, large software is developed and compiled in modules or libraries, many of which are used in a parallel client context and may not contain all the parallelism internally (e.g., functions that are called within a parallel loop). ASP uses module API annotations that describe legal uses of the module by describing what functions are allowed to be called in a parallel context and under which conditions, and it can enforce that both the internal library implementation as well as the client code abide by these rules.

We have implemented an ASP annotation checker for C++ by extending the Clang C++ front-end. We use the rich attribute syntax introduced by the latest C++11 standard to express the region and effect annotations. Our motivation was that any standards-compliant C++ compiler should be able to parse (and if necessary, ignore) ASP annotations because it is very hard for software project teams to switch compilers.

[2]ASP assumes there are no type safety or memory safety *errors* in the underlying C++ code, and does not attempt to detect or prevent such errors.

We are collaborating with engineers at Autodesk to design the ASP checker to support very large Autodesk libraries and applications. Autodesk has recognized the value of static analysis tools and ASP in particular by designating it an official internal project and assigning part of the time of three engineers to our joint project.

We currently are using this checker along with a series of *default* annotation rules in order to check without any manual annotations that global and static variables are not written to unless the function is explicitly permitted to do so by an annotation. This simple property was important enough to Autodesk that they had implemented their own checker, Hippocrates. But Hippocrates could not detect modifications to global and static variables through pointers and aliases, whereas our checker, by using the DPJ region and effects, is able to track and report those accesses as well. We have provided Autodesk with our checker, and they are in the process of testing it over large modules of one of their products, evaluating its usefulness. Meanwhile, we are also experimenting with using ASP on medium- and large open-source applications parallelized with TBB.

8.2.5 Automatic Inference of Region and Effect Annotations

To make it easier to develop large programs using DPJ, we have worked to reduce the need to manually write annotations. We have developed a tool called DPJizer that can automatically infer the effect specifications of almost all methods in a DPJ program where the programmer has included only region annotations and not effect annotations [162]. DPJizer uses an interprocedural analysis to gather *constraints* on the effect summaries of individual methods, using information from parallel constructs and from region type annotations. It then uses a custom constraint solver to attempt to determine a set of method effect summaries that satisfy these constraints; if it fails to do so, DPJizer marks the chosen parallelism as unsafe with the given region types. DPJizer supports the full range of DPJ regions and effects, including hierarchical RPLs, region-parameterized arrays, and "wildcard" RPLs that are necessary for many recursive and divide-and-conquer algorithms.

In the ASP project, we are now extending the inference algorithm in multiple ways. First, we are extending it to infer region annotations as well as effect annotations, using only the parallel constructs as inputs (since the goal is to check the correctness of a chosen parallelization of some code). We have developed and prototyped a preliminary constraint-inference algorithm for combined region and effect inference, using Prolog to encode and solve the constraints. Second, we are extending the algorithm to support higher-level "intent" annotations on module interfaces, e.g., an annotation that says a particular library entry point must be "*thread-safe*," and using those to infer the necessary region and effect annotations within the module automatically.

8.3 Lessons Learned

This project has shown that it is feasible to guarantee strong safety properties such as determinism by default (including determinism, data race-freedom, and strong atomicity) for a wide range of parallel programs using only static checking of effects. Dynamic effect checking in a scheduler greatly expands the range of programs that can enjoy nearly as strong safety guarantees. Automated effect inference significantly reduces the burden of using these mechanisms, and we expect we can reduce this even further by extending the inference capabilities.

8.4 Future Work

We are currently working to optimize the TWEJava run-time scheduler so that good performance and scalability can be achieved in programs that use TWEJava tasks for fine-grained critical sections. We are also working on task scheduling optimizations in TWEJava's run-time scheduler, e.g. to optimize locality. We are also extending the effect inference algorithm in DPJizer to include region and effect inference.

8.5 Key Papers and Other Material

We include two papers: the OOPSLA 2009 paper [25] gives an overview of DPJ, and the PPoPP 2013 paper [77] describes the tasks-with-effects model. All DPJ-related information is available at `http://dpj.cs.illinois.edu`.

Chapter 9

MCUDA: CUDA for Multicores

9.1 Problem Addressed

Heterogeneous computing systems are becoming very prevalent in many market segments. The number of accelerated supercomputers in the Top500-ranked list of supercomputers has been steadily increasing since 2009 [158], and currently occupy half of the top ten spots. The current #1 entry in the Top500 ranking of supercomputers uses more wide-vector Intel Xeon Phi™accelerator processors than CPU processors [158]. Processors marketed towards consumer laptop and workstation systems often integrate heterogeneous CPU and GPU components on the same silicon die. Correspondingly, there has been a growing adoption of parallel programming languages for different parallelism patterns, such as CUDA and OpenCL for data-parallelism, Cilk for task parallelism, and OpenMP for both task- and loop-level parallelism.

Divergent programming models and architectures are already causing significant costs in software development. Optimization is difficult for any architecture, and explicit heterogeneity increases that difficulty even for a single platform. An application developer must target program segments to the appropriate system component, balancing the relative strengths and weaknesses of each, and then optimize each program segment for the targeted architecture. Also, software typically outlives systems, so even when a specific platform is targeted at the time of initial development, the software must be continually updated to rebalance the distribution of tasks among heterogeneous components as it migrates to new platforms. In reality, long-term software development costs match the development costs for multi-platform support.

When faced with the challenge of targeting multiple heterogeneous systems, developers today typically make one of two choices. Some accept that each performance-sensitive program component must be capable of running on any CPU or GPU with high performance. Those developers bear the high cost of writing high-performance implementations for each architecture class (CPU, GPU, and possibly others) and guarantee performance on any platform through exhaustive specialization. Other developers lack the motivation or resources to pursue such a high-cost path, and choose instead to target some lowest common denominator among platforms, usually the CPU. These two choices drain development effort or leave significant performance opportunities on the table, respectively. The root problem is a lack of *performance portability*, or the ability to write a single software implementation that can be targeted to multiple architecture classes with high performance on all of them.

9.2 Contributions

In this project, we started by developing the MCUDA tool, to implement the CUDA programming model on CPU processors, demonstrating feasibility for the concept of executing “GPU” programming languages on CPUs efficiently. The initial goal of the MCUDA project was to find and contribute the missing pieces to an ecosystem for a specific programming language and enable performance portability in a real and usable way. As our work continued, the Khronos Group specifically designed OpenCL with portability between GPUs and CPUs in mind. Yet, despite well-resourced implementations of OpenCL for x86 from both Intel and AMD, the emerging OpenCL ecosystem failed to deliver satisfactory performance portability initially, as noted by application programmers [143], leading to continued work on "better" programming models [126]. The MCUDA project therefore took on a second objective: to understand why high-quality language implementations for both CPU and GPU architectures failed to provide performance portability.

Based on preexisting infrastructure supporting the CUDA language, we first sought to find and remedy any performance inefficiencies that we could find within the MCUDA tool, to determine the limits of its translation methodology. Its core methodology involved aggregating many small GPU tasks into a single CUDA thread, using compiler transformation techniques. The consequences of such an aggregation were the explicit instantiation of many copies of each CUDA thread's private variables, and the repeated execution of the entire kernel program for each unique CUDA thread index. However, there were many "private" variables that always stored the same value, regardless of CUDA thread index. During the aggregation process, we could identify such variables, and keep only one copy to hold the value for many CUDA threads. The data-redundancy removal often exposed computational redundancy across CUDA threads as well. And yet, even with all of this redundancy removal, there were many cases where the generated code was orders of magnitude slower than well-written C code.

As we worked on compiler optimizations, we also analyzed emerging heterogeneous applications in more depth. We studied the programming guidelines as published by GPU manufacturers, and the ways in which the programming community applied those guidelines to their applications. As a result, we created a characterization of optimization patterns for GPGPU programming, drawn from our informal survey of the GPU Computing Gems contributions [81, 82], and from a detailed analysis of the Parboil benchmarks. Table 9.1 summarizes our proposed collection of optimization patterns, and the goals or benefits accomplished by each.

Technique	Memory Contention	Bandwidth Demand	Locality Utilization	Instruction Efficiency	Load Imbalance	CPU
Tiling		X	X			
Privatization	X		X			
Scatter to Gather Conversion	X					
Binning		X	X	X		X
Regularization				X	X	X
Compaction		X				
Data Layout Transformation	X		X			
Granularity Coarsening	X	X	X	X		

Table 9.1: Issues addressed by each optimization pattern.

The programming and optimization principles we observed were very general, and reflected more about the challenges of targeting massive parallelism and wide SIMD than GPU architectures specifically. These optimization patterns represent portable performance principles for a variety of parallel architectures, which have been applied to systems from supercomputers to many-core CPUs. Some of these principles are even important for sequential workloads, whereas others are only relevant in the context of parallel execution and contention. While other performance principles specific to GPU architectures were observed, they tended to be secondary concerns, applied only after the primary principles were exploited fully.

As a result of a deeper understanding of the performance assumptions carried by OpenCL developers, we developed the next generation of compilation tools for GPU computing languages, this time focusing on OpenCL for convenient comparison with industry practice. By targeting the C Extensions for Array Notation (CEAN) from Intel, we were able to address two fundamental issues with the original MCUDA methodology: a lack of explicit vectorization opportunities, and a misuse of the CPU cache hierarchy. CEAN addressed both of these issues simultaneously, by delivering memory access orderings that took better advantage of spatial locality, and by unleashing the C compiler's vectorization capabilities from the constraints of unannotated C code. Figure 9.1 shows the improvement over the initial MCUDA methodology for a collection of kernels from the Parboil [153] and Rodinia [37] benchmark suites. Figure 9.2 shows the same benchmarks, comparing the CEAN methodology with existing industry implementations.

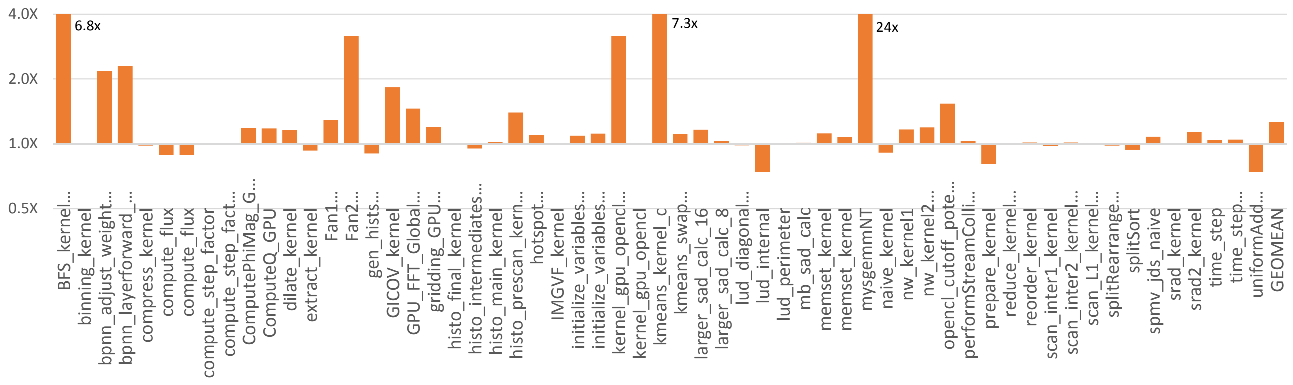


Figure 9.1: The performance effects of using vector-based compilation (CEAN) rather than region-based compilation (original MCUDA). The graph shows the change in performance from the baseline MCUDA methodology. Higher is better for the CEAN target.

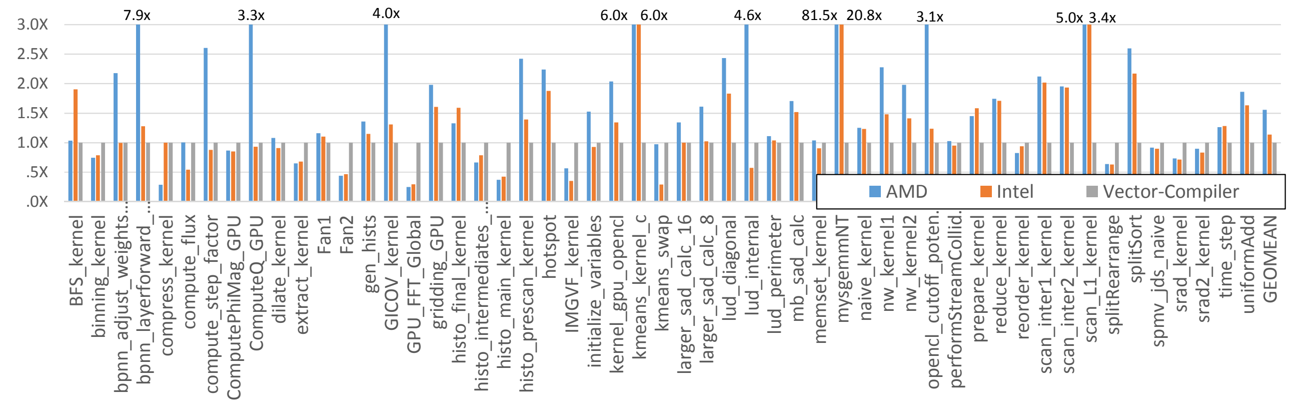


Figure 9.2: The performance effects of using vector-based compilation (CEAN) rather than runtime library methods of managing work-item synchronization. The graph shows the comparative execution time between widely-used runtime-based implementations from industry. The results are normalized to the CEAN target execution time; lower execution times are better.

These kernels are taken from benchmark suites typically used to evaluate GPU performance. The significant performance gains achieved for several classes of benchmarks and programming patterns highlights just how much performance portability was absent from the ecosystem prior to our contributions. Developers could not rely on their GPU-optimized kernels getting reasonable performance on the CPU platforms. In the same results, a few limitations of the current system are exposed, such as the 2-4x slowdown on `GPU_FFT_Global`, `histo_main_kernel`, and `compress_kernel`. These kernels all exhibit meaningful levels of control divergence, which are not handled well by our initial approach to CEAN code generation, as discussed in Section 9.4.

The version of our OpenCL compiler and runtime targeting CEAN is still under development, and has not yet been publicly released. The initial MCUDA toolset is available for download from the IMPACT website under the Illinois Open Source License.

9.3 Lessons Learned

As a result of this research, we have gained the following insights related to heterogeneous computation and compiler development.

OpenCL was specifically designed with portability between GPUs and CPUs in mind, and both AMD and Intel released high-quality x86 CPU implementations. Yet the ecosystem fails to deliver satisfactory performance portability, primarily because a language specification, by itself, is insufficient for establishing performance portability.

Performance portability requires not only an agreed-on functional language specification, but a clear specification or convention for expressing a program's performance-related attributes. In practice, a vendor implementing a language may think first of how the language's functional components might most naturally map to the specific architecture. That implementation will then define some best practices for programmers targeting that architecture through that implementation. The problem is that when the best practices for different platforms diverge, the potential for performance portability is lost. To follow divergent best practices for different platforms, programmers have no choice but to write specialized implementations for each.

We believe that the OpenCL community is beginning to converge on some unified conventions that can be exploited for performance portability. The performance convention embodies preferred methods for expressing data parallelism, task parallelism, spatial locality, and temporal locality, and covers a broad range of application-level optimizations required by developers. To achieve performance portability, vendor implementations of the language must determine how to adopt those abstract performance expressions to their architecture, in addition to obeying the functional specification of the language. We have demonstrated that a CPU implementation of OpenCL following those guidelines outperforms the currently available implementations, given typical workloads written for GPU acceleration. Through our experience executing source code on CPUs and GPUs, we conclude that the high-level optimization principles applied to these programs were meaningful for a variety of architectures, and that language implementations can deliver performance portability effectively.

9.4 Future Work

Programming conventions have limitations. There is some measurable cost paid to conform not only to a language specification but to a particular performance model. Our methodology completely falls back to scalar code generation when, in an irregular workload, regular control flow cannot be statically guaranteed. We are currently working on more advanced CEAN code generation methodologies that may be able to recover some of that lost performance potential.

9.5 Key Papers and Other Material

We include two papers in this book. The first one is "Efficient Compilation of Fine-Grained SPMD-threaded Programs for Multicore CPUs" from CGO 2010 [151]. It explains the entire core methodology. The other is "Performance Portability in Accelerated Parallel Kernels", Technical Report IMPACT-13-01 [152]. It describes our initial attempts at targeting CEAN.

Chapter 10

Scheduling for Energy Efficiency

10.1 Problem Addressed

Designers of mobile devices face the challenge of providing the user with more processing power while increasing battery life. In this work, we have looked at the opportunities offered by heterogeneous systems to address this challenge. In a heterogeneous system, multiple classes of processors with dynamic voltage and frequency scaling functionality are embedded in the same chip. With such a system, it is possible to maximize performance while minimizing power consumption if tasks are mapped to the class of processors where they execute the most efficiently. Thus, we have proposed a scheduling algorithm to determine the scheduling of tasks in a real-time context on a heterogeneous system-on-a-chip that has dynamic voltage and frequency scaling functionality. Our algorithm minimizes energy consumption while meeting the deadlines. In addition, to verify some of the assumptions made in this algorithm, we evaluated performance and energy of a visual object-detection application running on the heterogeneous Intel Ivy Bridge processor [40]. Different mapping strategies were evaluated and execution times and energy consumption were measured for each of them.

10.2 Contributions

We highlight two main contributions. The first one (Section 10.2.1) is a scheduling algorithm that takes advantage of the different types of processors in an heterogeneous device. The second one (Section 10.2.2) is our experiences when running a vision application in an Ivy Bridge processor.

10.2.1 Scheduling

We propose a new scheduling mechanism that minimizes the energy consumed by streaming computations under the constraint of a minimum output rate. We assume that there could be multiple classes of processors embedded in the mobile device and that they have dynamic voltage and frequency scaling (DVFS) functionality. Since the maximum possible frequency is not typically required to achieve the desired output rate, energy consumption can often be minimized by lowering the voltage and hence the frequency as much as the real time constraint allows. This minimization process is complicated by three situations. First, only a discrete number of frequencies are possible in today's machines. Second, changing the voltage/frequency consumes time, which means that such changes must be applied judiciously. Third, energy efficiency can be controlled not only by DVFS but also by choosing the class of processor on which to map each task, since different processors are efficient for different types of tasks. A GPU will excel in vector operations but will be inferior to a conventional CPU for applications rich in control flow.

The technique that we propose takes the form of a two-step heuristic that first chooses what class of processors to map each task to [163]. To make this decision, we introduce the concept of cross-platform task heterogeneity, which measures which tasks have a larger variation in execution time based on the type of processor where they run. Tasks with a higher value of cross-platform heterogeneity are scheduled first, to give them more choices. Then, after tasks have been assigned to a processor type, a homogeneous scheduling algorithm is used to determine the voltage and frequency scaling within each homogeneous subsystem. Our scheduling techniques apply frequency scaling at the granularity of the tasks. This enables us to place the code for frequency scaling at natural locations.

Our experimental results show that our scheduling algorithm outperforms previous algorithms, such as Greedy and LR [171]. It has a high success rate at finding a feasible schedule and, for most of the cases we studied, the resulting schedule is within 5% of the optimal one. It also offers very stable results when the architecture heterogeneity and task uniformity vary. Additionally, the memory usage is linear with the size of the input.

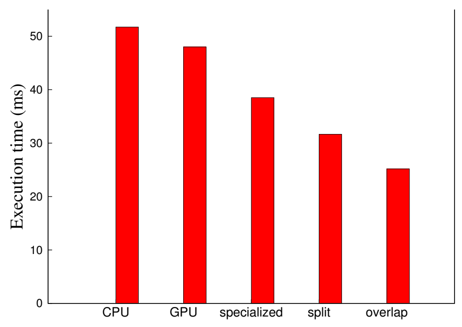
10.2.2 Experimental Results

To verify some of the assumptions we made when developing the scheduling algorithm described in the previous section, we have run some experiments on a heterogeneous Intel Ivy Bridge processor used for an Ultrabook. This processor has a 1.7 GHz dual-core and an Intel HD Graphics 4000 with 16 GPU execution units running at 350 Mhz. For the evaluation, we have chosen an object detection algorithm [51] for finding objects with a specific shape or appearance in unconstrained visual input. Our object detector uses the ViVid library [1]. ViVid compiles several atomic functions common to many vision algorithms and is representative of many typical applications on mobile devices that work with the streaming visual input from the camera and are fairly computationally demanding. As most vision applications (e.g., recognition, tracking, stabilization), Vivid consists of a pipeline of a small number of kernels. To improve performance and energy efficiency, we optimized the code and evaluated all the possible mapping strategies. We also evaluated the use of DVFS.

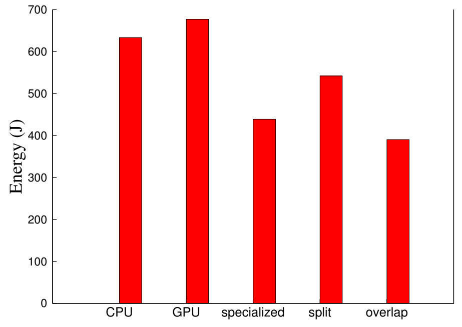
ViVid consists of three main kernels: Blockwise Distance (Filter kernel), Cell Histogram Kernel, and Pairwise Distance (Classifier Kernel). Each input frame goes through the three kernels, but there are no dependences across frames. ViVid was originally written in C and used OpenMP for parallelism. There was also a CUDA version to run on the NVIDIA GPUs. To run in the GPU of the Ivy Bridge processor, we had to translate the original C code into OpenCL. Then, each kernel was optimized and executed on both the CPU and the GPU. We found that while the filter and histogram kernels run faster in the GPU, the classifier runs faster in the CPU. Similarly, the filter and histogram kernels consume less energy when running in the GPU, while the classifier consumes less when running in the CPU. It is interesting to notice that these results contradict the general belief that all these highly-parallel kernels run faster in the GPU. While this statement might be true for larger GPUs, for the smaller GPUs of heterogeneous systems, both GPU and CPU are necessary.

After understanding the different trade-offs between GPU and CPU for each kernel, the natural question is how to utilize the heterogeneous system for an application to achieve best performance and energy efficiency. To answer this question, we evaluated different approaches. The results are shown in Figures 10.1(a) and 10.1(b), which show the execution time and energy, respectively. *CPU* and *GPU* correspond to running all the kernels in the CPU or in the GPU; *specialized* corresponds to an execution where the filter and histogram are mapped to the GPU (where they run faster and are more energy efficient) whereas the classifier is mapped to the CPU (where it is faster and more energy efficient). In *specialized*, when the GPU is executing filter or histogram, the CPU is idle, and when the CPU runs the classifier, the GPU is idle. *Overlap* corresponds to an execution similar to software pipelining that can be applied to streaming applications, where parallelism can be exploited across multiple input images or frames, like multiple frames of a video. In *overlap*, filter and histogram form the first stage of the pipeline, operating on a frame in the GPU, while classifier is the second stage of the pipeline, running on the CPU and operating on the GPU's results. Note that with *overlap*, since the CPU's work takes around 3 times longer than the GPU, the GPU will be idle for about two-thirds of the execution time. These strategies underutilize either the CPU or the GPU because of data dependencies. Therefore, one can split the image between the CPU and the GPU for maximum utilization, shown as *split* in the figures.

[1]http://www.github.com/mertdikmen/vivid



(a) Execution time



(b) Energy consumption

Figure 10.1: Running the full application on the CPU or GPU, or utilizing both using different approaches.

The results in Figure 10.1(a) show that *specialized* is more than 25% faster than just running on the CPU (20% faster than the GPU), as one would expect. *Split* is about 39% faster than *CPU*. *Overlap* obtains the best performance by running the kernels on the best type of processor, but trying to keep them busy by software pipelining. With respect to energy, the results in Figure 10.1(b) show that *specialized* and *overlap* consume the least energy because they run each kernel where it runs the best. On the other hand, using only the CPU or the GPU is not energy efficient. Note that *split* is a very fast method but it consumes much more power also, so it is not the most energy efficient in the end. *Specialized* and *overlap* are 35% and 42% more energy efficient than the GPU-only method, respectively. They are also 19% and 28% more energy efficient than *split*, respectively.

Our performance is superior or similar to recent works using much more capable discrete GPUs [18, 129, 173]. However, notice that real-time vision applications need to run at a certain number of frames per second. For instance, we can run at around 40 frames per second (fps) with *overlap* and 31 fps with *split*, while 10 fps might be enough for many object detection purposes. The extra available computation power can be used for more analysis or for other applications (e.g., if vision is only the interface for some other purpose). Alternatively, one might consider DVFS to save energy. However, our results show that the Ivy Bridge processor's DVFS is not very effective for these compute-intensive codes. We applied DVFS to our application and could only save 5% of the energy, while sacrificing 9% performance. The reason is that it makes the runtime so much longer (for compute-intensive codes) that it offsets the power savings. Thus, running the application for a while and then idling the processor seems to be the best solution for saving energy. In this case, savings will depend on the sleep and wakeup latencies of the processor in the specific usage. However, we expect DVFS support to improve significantly in future devices, as vendors consider it in earlier steps of the processor design. This will be very important for the energy efficiency of many vision applications similar to ours.

10.3 Lessons Learned

Our experimental results show that mapping kernels to the most appropriate device is important and programmers need to consider the different choices. To that end, we have provided an algorithm that can help determine the mapping for such a heterogeneous system.

Our experimental results for a vision application with a small number of kernels show that mapping the kernels to the most efficient device and overlapping the computation of the kernels is the best approach. For our ViVid application, it is 1.91 times faster and 42% more energy-efficient than a GPU-only solution. It is also 1.26 times faster, and 28% more energy-efficient than an input-splitting method. Notice that even for the few codes that use GPUs, this approach is not currently used, as most vision codes run significantly faster in the large discrete GPUs. However, in the case of smaller on-chip GPU, our experimental results show that both CPU and GPU need to be involved in computation.

We have also observed that DVFS does not help the energy efficiency much in current systems, but we expect it to be useful for our applications on future machines with better DVFS support.

10.4 Future Work

We are currently translating other vision applications, such as image recognition and image tracking. One goal is to understand the optimizations that we need to apply to generate highly-efficient code for both CPU and GPU. The second goal is to better understand the different scheduling strategies. For instance, we are currently investigating techniques that can improve the results that we obtained with the *overlap* execution. With this strategy, the running times are dominated by the execution time of the slowest stage of the pipeline. New scheduling techniques that take advantage of the idle time in the shorter stages of the pipeline should reduce energy consumption.

10.5 Key Papers and Other Material

In this book, we include the paper "Scheduling of Stream-Based Real-Time Applications for Heterogeneous Systems" from LCTES 2011 [163]. This paper was selected as one of the five Best Papers of the conference. It was produced in collaboration with Intel researchers Jean Pierre Giacalone and Bob Kuhn.

Chapter 11

Verification and Testing Advances

11.1 Problem Addressed

The currently-dominant programming model for parallel code is that of shared memory, where multiple threads of computation communicate by reading and writing shared data objects. Multithreaded code is notoriously hard to get right, with common bugs including dataraces, atomicity violations, or deadlocks. While new parallel programming models are emerging, programmers need help right now to develop more reliable multithreaded code.

Software testing is the most widely-used method for increasing software reliability. Researchers and practitioners have developed a number of automated testing techniques and tools that have been adopted by programmers as an aid in developing more reliable code. Some of the techniques and tools most successful in practice include unit-testing frameworks (such as JUnit or NUnit), regression test selection (which determines what tests to rerun after making a change in code), and test prioritization (which determines in what order to run the tests to find bugs faster). The existing approaches work fairly well for sequential code, but unfortunately do not work nearly as well for multithreaded code.

To significantly improve the testing of multithreaded code, we aimed to develop a set of new techniques and tools for multithreaded tests. A multithreaded test is a piece of code that creates and executes two or more threads (and/or invokes code under test that itself creates and executes two or more threads). Executing a test follows some schedule/interleaving for the execution of the multiple threads (and different schedules can give different results).

In one approach that we have taken, the key idea is to allow *specifying a set of relevant schedules* for each test. Note that even the traditional tests that *do not explicitly* specify a set of schedules actually *do implicitly* specify a set of schedules, namely the set of *all possible* schedules. We addressed several important challenges for multithreaded tests: (1) test schedules: how to describe a set of schedules and which schedules from a given set to explore? (2) test generation: how to automatically generate multithreaded tests, especially schedules for a given code? (3) regression testing: how to select/prioritize the rerunning of the multithreaded tests when the code under test changes?

In another approach, called *predictive testing*, we have tackled the problem of automatically finding particular interleavings to test the program on that are likely to contain errors. In predictive testing, we take one arbitrary run of the program under test, and use the execution to predict alternate interleavings that are likely to expose more errors, and then test those by carefully scheduling the predicted interleavings. We have, over the years, built a very robust and extensible predictive testing tool called PENELOPE that finds effective interleavings to expose a variety of errors, ranging from dataraces to atomicity violations, deadlocks, and null-pointer dereferences.

11.2 Contributions

11.2.1 General Testing of Multithreaded Code

In the area of general testing of multithreaded code, we made several contributions:

- We developed Ballerina [117], a novel technique for automated random generation of efficient multithreaded tests that effectively trigger concurrency bugs. As mentioned, a multithreaded unit test creates two or more threads, each executing one or more methods on shared objects of the class under test. Such unit tests can be generated at random, but basic random generation produces tests that are either slow or do not trigger concurrency bugs. Worse, such tests have many false alarms, which require human effort to filter out. Ballerina makes tests efficient by having only two threads, each executing a single, randomly selected method. Ballerina increases chances that such simple parallel code finds bugs by appending it to more complex, randomly generated sequential code. We also proposed a clustering technique to reduce the manual effort in inspecting failures of automatically-generated multithreaded tests.

- We developed IMUnit [86, 87], a novel approach to specifying and executing schedules for multithreaded tests. When developers want to enforce a particular schedule for their multithreaded test execution, they commonly use time delays (e.g., Thread.sleep in Java). Unfortunately, this approach can produce false positives (tests failing when there is no bug in the code) or negatives (tests passing when they could execute a bug in the code), and can result in unnecessarily long testing time. We introduced a new language that allows explicit specification of schedules as orderings on events encountered during test execution. We developed a tool that automatically instruments the code to control test execution to follow the specified schedule. We also developed a tool that helps developers migrate their legacy, sleep-based tests into event-based tests in IMUnit. The migration tool uses novel techniques for inferring events and schedules from the executions of sleep-based tests.

- We developed CAPP [89], a novel technique that uses information about the changes in software evolution to prioritize the exploration of schedules in a multithreaded regression test. Successful software evolves over time as developers add more features, respond to requirements changes, and fix faults. Regression testing is the most widely-used method for ensuring the validity of evolving software. As regression test suites grow over time, it becomes expensive to execute them. The problem is exacerbated when test suites contain multithreaded tests. These tests are generally long running, as they explore many different thread schedules searching for concurrency faults such as dataraces, atomicity violations, and deadlocks. While many techniques have been proposed for regression test prioritization, selection, and minimization for sequential tests, there was not much work for multithreaded code. We implemented CAPP in two frameworks for systematic exploration of multithreaded Java code, stateful JPF and stateless ReEx [88].

- We developed MuTMuT [68,69], a general framework for efficient exploration that can reduce the time for mutation testing of multithreaded code. While CAPP addressed code changes in the actual code evolution, the first step of our work considered code changes in mutation testing. Mutation testing is a method for measuring and improving the quality of test suites. Given a system under test and a test suite, mutations are systematically inserted into the system, and the test suite is executed to determine which mutants it detects. When the test suite does not detect a non-equivalent mutant, the test suite can be improved by adding a test case that detects that mutant. A major cost of mutation testing is the time required to execute the test suite on all the mutants. This cost is even greater when the system under test is multithreaded: not only are test cases from the test suite executed on many mutants, but also each test case is executed, or more precisely, explored, for multiple possible thread schedules. We presented several techniques within the general MuTMuT framework.

- We developed Light64 [118] and InstantCheck [119], two novel approaches for hardware-supported testing of parallel code. Light64 focuses on dataraces, which are common bugs but, unfortunately, checking for races is often skipped in systematic testers because it introduces substantial runtime overhead if done purely in software. Recent techniques proposed for race detection in hardware require significant hardware support. In contrast, Light64 has both small runtime overhead and very lightweight hardware requirements. Light64 is based on the observation that two thread interleavings in which racing accesses are flipped will very likely exhibit some deviation in their program execution history. InstantCheck focuses on checking determinism. During code testing, InstantCheck can check whether the code under test ends up in a deterministic state in various runs. The idea is to compute hashes of the memory state and compare them for different test runs that have the same input. InstantCheck has a very small runtime overhead while requiring only a minor hardware extension.

11.2.2 Predictive Testing and the PENELOPE Framework

In the area of predictive testing, we made several advances to predict as well as to test multithreaded code:

- We have developed, over time, a robust and extensible tool called PENELOPE [150] for testing multithreaded Java programs. PENELOPE works by automatically transforming Java byte-code to insert monitors and schedulers into the code. The predictive approach calls for three phases: (a) an initial monitoring phase that monitors precisely the interleaving executed on an arbitrary run of the program, (b) a prediction phase that predicts alternate interleavings that are likely to expose errors of various kinds, and (c) a rescheduling phase that accurately replays the predicted interleavings. Our tool performs the phases (a) and (c) above, allowing developers to plug in any prediction tool for Phase (b) so as to build their own predicting tools [60–62, 149].

- We have built a variety of predicting routines for various generic errors in programs, including dataraces, atomicity violations, and deadlocks. Furthermore, since there are a large number of interleavings that are hard for a programmer to consider, several generic errors for sequential programs also manifest in the concurrent setting, where they manifest themselves only on a small set of interleavings. We have explored predicting routines for null-pointer dereferences as well [60–62, 149].

- The prediction of alternate interleavings for the various kinds of errors is a hard algorithmic problem. We have developed techniques based mostly on lock-sets and *acquisition histories*, the latter being more sophisticated information on lock acquisition patterns. The algorithms we build are often extremely efficient (polynomial time) [61, 149].

- In more recent work, we have also explored techniques based on constraint solvers in order to predict alternate interleavings, in particular predict interleavings to expose null-pointer dereferences [61]. Here, using state-of-the-art SMT solvers, we can search for interleavings whose feasibility and accuracy in exposing bugs are greater [61].

11.3 Lessons Learned

Our results show that the techniques and tools we developed can improve testing of multithreaded code: they find more bugs and/or find bugs faster than previous techniques, make it easier to write tests, and even automatically generate tests in some cases. While many of the experiments that we have performed used previous-known bugs to compare techniques, we have also found previously-unknown bugs in popular open-source code (e.g., Apache Pool and Log4j Java projects). Some of the bugs we found and reported were already confirmed and fixed by the developers. These results confirm the widely-held belief that parallel code is prone to bugs, and that is it important to develop techniques to find, eliminate, and/or prevent these bugs. As the types of bugs vary (e.g., dataraces vs. deadlocks), we need a multi-pronged approach with various techniques for these types.

11.4 Future Work

The future work is to continue improving the techniques and tools for testing parallel code. An important aspect of the future work is to get these tools included in the actual software development process. For example, Google is currently considering to include our IMUNIT tool [86] in their testing.

11.5 Key Papers and Other Material

We include two papers in this book. The first one is "Improved Multithreaded Unit Testing" from ESEC/FSE 2011 [87]. This paper discusses the IMUnit language for specifying and enforcing schedules in multithreaded tests. Because multithreaded tests depend on schedules, being able to easily and explicitly specify schedules enables proper explorations of these schedules. The code released from this project includes the IMUNIT [86] and REEX [88] tools.

The second paper is "Predicting Null-Pointer Dereferences in Concurrent Programs" from FSE 2012 [61]. This paper, though it concentrates on finding null-pointer dereferences, outlines the general predictive technique and framework, and the use of SMT solvers, to predict interleavings for various kinds of errors. The PENELOPE tool has also been released [150].

Chapter 12

Record&Replay and Debugging Architectures

12.1 Problem Addressed

Despite the best efforts of programmers, parallel code often contains concurrency bugs. What is worse, debugging parallel code is very time consuming. There are at least two reasons for this. First, concurrency bugs are typically hard to find and isolate. They are often strongly dependent on how the different threads interleave. Small timing changes often prevent a bug from manifesting again.

A second reason is that there are many different types of concurrency bugs. There is no cost-effective universal approach to debug all concurrency bugs. Instead, for each type of bug, there are different approaches and tools. Even for the same bug, there are different approaches depending on the desired effectiveness versus overhead design point. Examples of concurrency bugs are data races, atomicity violations, and Sequential Consistency Violations (SCVs). An SCV occurs when the accesses of multiple threads form two or more data races that overlap in a way that the instructions cannot be ordered in any sequentially-consistent order.

Most of the existing tools and techniques to handle these bugs are software-based. In this chapter, we introduce techniques that, to keep the execution overhead negligible, rely on some hardware structures in the processor or memory system. Our techniques do not alter the timing of the execution and, therefore, the programmer sees the same bugs as in a production run.

Unfortunately, even if we do not alter the timing of the execution, it is not guaranteed that a bug will reoccur in a re-execution. The reason is that machines are typically nondeterministic. Hence, we also introduce novel architectures to Record and deterministically Replay (R&R) multithreaded applications. Under R&R, as a parallel program executes, there is special hardware and OS components that record in a log any nondeterministic event that occurs; then, in a second run, these special components guide and force the execution to follow the exact same paths. With this hardware and OS support, bugs can be perfectly reproduced.

12.2 Contributions

The main contribution of this project has been the design and construction of a multicore hardware prototype for R&R called *QuickRec* [127]. The prototype is built with FPGAs, has a full Linux-based OS, and records and replays complete Intel-Architecture (IA) applications. It was built jointly with Intel researchers. Moreover, we have also designed other schemes for R&R [79, 80, 110–112, 128] that introduce novel ideas. In particular, we have developed the first OS that virtualizes R&R hardware [112].

In addition, we have designed novel architectures for detecting and avoiding other concurrency defects such as data races [116, 118, 132], atomicity violations [114], sequential consistency violations (SCVs) [115, 136], and determinism bugs [119] with negligible execution overhead. These techniques do not alter the timing of parallel application execution.

We have extensively discussed these techniques with Intel architects and worked together with several of them. Together with Intel, we produced the QuickRec [127] system and several joint papers [79, 127, 128]. Our work is documented in 16 technical papers [35, 36, 79, 80, 110–112, 114–116, 118, 119, 127, 128, 132, 136]. They include one Communications of the ACM Research Highlight [80] and 11 papers in the ISCA, MICRO, HPCA, or ASPLOS top conferences.

The following sections give an overview of the QuickRec prototype (Section 12.2.1), additional R&R architectures that we designed (Section 12.2.2), and architectures for detecting and avoiding concurrency bugs (Section 12.2.3).

12.2.1 The QuickRec Prototype

Together with Intel researchers, we designed and built *QuickRec* [127], the first multicore IA-based prototype of R&R for multithreaded programs. QuickRec is based on *QuickIA*, an Intel emulation platform for rapid prototyping of new IA extensions. QuickRec is composed of a Xeon server platform with FPGA-emulated second-generation Pentium cores, and *Capo3*, a full software stack for managing the recording hardware from within a modified Linux kernel. Figure 12.1 shows different aspects of QuickRec: a photograph of the prototype (left), the architecture of the processor-emulation platform (center), and an overview of the extended Pentium core, where circled numbers identify the main CPU touch points (right).

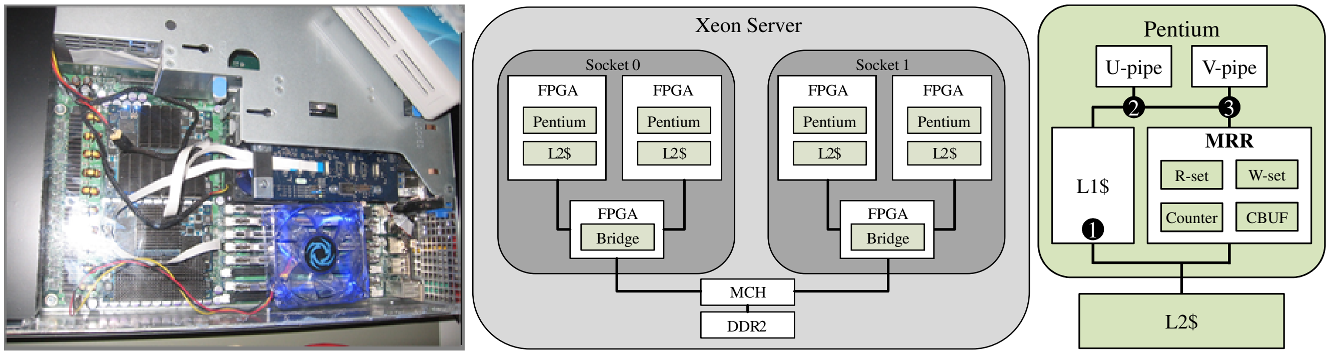


Figure 12.1: The QuickRec prototype for R&R.

The QuickRec recording system dynamically divides a thread's execution into regions of consecutive dynamic instructions called *Chunks*. It adds two Bloom filters [23] next to the L1 cache to capture the read and write sets of the memory accesses in a chunk. Specifically, the line addresses of the locations accessed by loads and stores are hashed in the Bloom filters and inserted into their respective set (R-set and W-set in the figure) at retirement and at global observation time, respectively. A thread's chunk is terminated when the hardware observes a memory conflict (i.e., a data dependence) with a remote thread. Conflicts are detected in hardware, by checking the addresses of incoming coherence transactions against addresses in the read and write sets. When a conflict is detected, the hardware logs a count with the current chunk size into an internal chunk buffer (CBUF), along with a timestamp that provides a total order of chunks across cores. The chunk-size count is the number of retired instructions in the chunk. After a chunk is terminated, the read and write sets are cleared, and the chunk-size count is reset.

CBUF is organized into four entries, where each is as large as a cache line. When a CBUF entry is full, the hardware flushes it to a dedicated memory region called CMEM. This is done lazily during idle bus cycles to minimize the performance impact.

There are several subtle issues related to capturing the ordering of instructions. One of them is the fact that the IA memory model allows a load to retire before a prior store to a different address has committed, hence ordering the load before the prior store in memory. A second issue is that an instruction may perform multiple memory accesses before completing execution, and the different memory accesses may end up in two or more different chunks. QuickRec handles all of these issues.

The Capo3 OS records the inputs to the program's processes, manages the replay logs, and virtualizes the hardware components. It is needed to make the R&R system practical for real IA multicore systems.

12.2.2 Additional R&R Architectures Designed

The DeLorean Model of R&R Based on Chunks. The *DeLorean* [110] ISCA 2008 paper introduces a new approach to R&R that provides substantial advances in log size reduction. DeLorean uses a new execution substrate: one were processors execute large blocks of instructions called chunks. To capture a multithreaded execution, DeLorean only needs to record chunk sizes and the total order in which chunks from different processors commit — not individual shared-memory dependences between threads, as was the state of the art before. This reduces the log size substantially.

In addition, in DeLorean, each processor executes chunks atomically as in the Bulk Multicore architecture (Chapter 13). As a result, the memory accesses of a processor within a chunk can overlap and reorder. This allows DeLorean to record execution at the speed of the most aggressive consistency models used today — and replay at a comparable speed. This is in contrast to the other schemes, which needed conservative memory consistency models to record. Finally, DeLorean offers different execution modes that provide different tradeoffs between performance and log size.

The Capo OS for Virtualizing R&R. The *Capo* [112] ASPLOS 2009 paper presents the first set of OS abstractions and software-hardware interface for practical hardware-assisted R&R. Past R&R work had focused only on the hardware implementation of the basic primitives for recording and, sometimes, replay. It did not address key issues such as how to separate software that is being recorded or replayed from software that should execute without being recorded or replayed, or from other software that should be recorded or replayed separately. Practical R&R systems require a software component to manage large logs, and a way to mix standard execution, recorded execution, and replayed execution of different applications in the same machine concurrently.

Capo introduces the abstraction of *Replay Sphere*, which allows designers to separate the responsibilities of hardware and software components. A replay sphere is a group of threads (together with their address spaces) that are recorded and replayed as a cohesive unit. All the threads that belong to the same process must run within the same replay sphere. It is possible, however, to include different processes within the same replay sphere. To evaluate Capo, we design and build an implementation of it in Linux.

The Cyrus Advanced R&R System. The *Cyrus* [79] ASPLOS 2013 paper presents the first hardware-assisted approach for unintrusive, application-level R&R that explicitly targets high-speed replay. Fast replay is an enabling property in R&R systems for debugging, intrusion analysis, and fault tolerance. Application-level R&R means that only an application (or a set of them) is recorded, rather than the whole machine state. Unintrusive hardware design is fundamental for acceptance of R&R hardware. It prohibits any big system-level hardware changes, such as changes to the cache coherence protocol. Since many multiprocessors use snoopy cache coherence, we require that our design is compatible with (and does not modify) snoopy protocols.

Cyrus introduces the concept of an on-the-fly software *Backend Pass* during recording which, as the log is being generated, consumes it and transforms it. This pass fixes-up the log, exposes a higher degree of parallelism for replay, and can also flexibly trade-off replay parallelism for log size.

12.2.3 Architectures for Detecting & Avoiding Concurrency Bugs

We proposed architectures for debugging data races [116, 118, 132], atomicity violations [114], sequential consistency violations (SCV) [115, 136], and non-determinism [119] with negligible execution overhead. These techniques do not alter the timing of parallel application execution.

Debugging Data Races: SigRace, Light64, and Pacman. *SigRace* [116] is a novel approach to hardware-assisted data race detection that overcomes key limitations of previous schemes. Previous hardware-assisted data race detectors detect races by augmenting the cache state and the coherence protocol. They tag each cache line with a timestamp or a lockset, perform additional operations on local/external access to the cache, and piggyback information on cache coherence protocol messages. Unfortunately, L1 caches and coherence protocol units are critical hardware structures. In addition, if a line is displaced or invalidated from the cache, these systems typically lose the ability to detect races involving the line.

SigRace, instead, relies on hardware address signatures. As a processor runs, the addresses of the data that it accesses are automatically encoded in signatures. At certain times, the signatures are automatically passed to a hardware module that intersects them to those of other processors. If the intersection is not null, a data race may have occurred. With SigRace, there are no changes to the cache or the cache coherence protocol messages, and there are no critical-path operations performed on local/external access to the cache. Moreover, lines can be displaced or invalidated from caches without affecting SigRace's ability to detect data races.

Light64 [118] is a novel technique for detecting data races during systematic testing of multithreaded programs. It encodes execution histories with a hardware hashing module. More details have been presented in Chapter 11.

Pacman [132] is a scheme to detect Asymmetric data races. An asymmetric data race occurs when at least one of the racing threads is inside a synchronization-protected critical section. In this case, while the thread (call it safe) is accessing shared variables inside a critical section, a second thread (call it unsafe) races in, corrupting the state or reading inconsistent state. The idea behind Pacman is to use the hardware cache coherence protocol in a multiprocessor to temporarily protect the variables that a thread is accessing in a critical section. The hardware performs two concurrent actions. One is to record the addresses of (a subset of) the variables that the safe thread is accessing while executing a critical section. The second is to reject any requests from the unsafe threads that conflict with these variables, until the safe thread leaves the critical section. We use a hardware signature to encode such addresses.

Debugging Atomicity Violations: AtomTracker. *AtomTracker* [114] is a comprehensive approach to infer Atomic Regions (AR) and to detect AR violations. It is the first scheme to (1) automatically infer generic non-nested ARs and (2) automatically detect violations of them at runtime with negligible execution overhead. No programmer input or annotations are needed.

AtomTracker has two parts: one that automatically infers ARs (AtomTracker-I) and one that automatically detects violations of their atomicity (AtomTracker-D). AtomTracker-I infers generic ARs by analyzing annotation-free memory traces of test runs of the program. AtomTracker-I uses a novel algorithm that works by greedily joining successive references of a thread into an AR if the other threads do not conflict.

AtomTracker-D takes the set of ARs and detects violations of their atomicity at runtime. AtomTracker-D uses a new algorithm for atomicity violation detection. It checks if concurrently-executing ARs can be made to appear to execute in sequence. Moreover, we present a hardware implementation of AtomTracker-D in a shared-memory multiprocessor that leverages cache coherence state transitions.

Debugging Sequential Consistency Violations: Vulcan and Volition. A Sequential Consistency Violation (SCV) occurs when the memory operations of a program have executed in an order that does not conform to any SC interleaving. An SCV is caused by two or more overlapping data races where the dependences end up ordered in a cycle [144].

With *Vulcan* [115] and *Volition* [136], we develop highly-precise approaches to detect SCVs in relaxed-consistency machines. In addition, they deliver information to debug the SCV, use no other input than the program executable, and have a negligible execution overhead. They are hardware-only solutions that look for cycles of inter-thread data dependences at runtime. The idea is to rely on the cache coherence protocol to dynamically record the observed inter-thread data dependences, while checking whether they form cycles. These dependences are kept around only for as long as they can participate in a cycle, and are discarded soon after.

Our techniques tag each monitored reference issued by each processor with a per-processor Sequence Number (SN). Then, they use cache coherence protocol transactions to pass these SNs between processors in a communication. The hardware checks for a dependence cycle. When a dependence closes a cycle and causes an SCV, an exception is raised. This provides valuable information for debugging the SCV. Both schemes work with realistic multiple-word cache lines.

Vulcan [115] relies on a snoopy-based coherence protocol. Moreover, the design presented only identifies 2-processor SCVs. With Volition [136], we have taken a different approach, focusing on scalability and on identifying SCV cycles with an arbitrary number of processors. The resulting Volition design is scalable, as it works with a directory-based cache-coherence protocol and its hardware does not need all-to-all structures.

Debugging Non-Determinism: InstantCheck. *InstantCheck* [119] is a novel technique that checks determinism in the execution of a multithreaded program with a very small runtime overhead, while requiring only a minor hardware extension. The idea is to compute a hash of the memory state and compare the hashes of different runs that have the same input. More details have been presented in Chapter 11.

12.3 Lessons Learned

We have learned several lessons from the QuickRec prototype [127]. The three most relevant are: (1) The prototype demonstrates that chunk-based recording can be implemented with low-enough implementation complexity and few-enough touch points to make it attractive to processor vendors; (2) By far, the biggest challenge in implementing QuickRec R&R is dealing with the idiosyncrasies of the Intel Architecture, including its memory consistency model and its CISC nature; (3) The main performance overhead is in the software layer, collecting and managing the input logs; with a slightly-improved software stack, R&R can be used in an "always-on" manner, enabling a potentially-large number of new R&R uses.

The main lesson in the work on architectures for debugging concurrency bugs is that there are many good opportunities: simple hardware that adds negligible execution overhead can help debug code from a variety of bug types without interfering with the timing of the execution. We believe that improving the programmability of the architecture in this way is an excellent use of transistors.

12.4 Future Work

There are several avenues for future work in hardware-assisted R&R, as discussed elsewhere [127]. In particular, emphasis should be placed on the replay aspect of R&R. We need approaches that are tolerant of, and abstract away, the micro-architectural details of the recording platform. Otherwise, proprietary details will stifle the development of replay support. In addition, we need to develop and demonstrate many uses of the R&R technology that solve real problems of multicore users. The areas of parallel program development tools and security aids seem particularly attractive.

Regarding the architectures for debugging concurrency bugs, we note that, currently, there is a different architectural mechanism for each type of bug. Our future work is to find primitive mechanisms that can be used for many types of bugs.

12.5 Key Papers and Other Material

We include two papers in this book. One is the ISCA 2013 paper on QuickRec [127], which discusses many issues that arise when building a real hardware prototype for R&R with a complete Linux-based OS (*Capo3*). The source code of Capo3 can be found at http://iacoma.cs.uiuc.edu/capo. The other paper is the ISCA 2009 paper on SigRace [116], which describes and evaluates this new approach for data-race detection. The other papers cited above have a similar flavor for other environments or types of bugs.

Chapter 13

The Bulk Multicore Architecture for Programmability

13.1 Problem Addressed

Traditionally, parallel architectures have been designed for performance and, recently, for energy-efficient performance. Given the increasing community of parallel programmers, and the difficulty of programming in parallel, it is now crucial that the hardware architecture also assists in providing a programmable environment.

In practice, programmability is a difficult metric to define and measure. At the hardware-architecture level, it implies several things. First, the architecture should be able to attain high efficiency while relieving the programmer from managing low-level tasks. Second, the architecture should help minimize the chance of (parallel) programming errors. Finally, the architecture should support a broad software base, and not introduce significant constraints on the programming model, compiler support, or application structure required.

A second concern as we scale the multicore size is that of hardware complexity. In large multicores, there are many memory accesses in progress at any given time. However, individual cores are still required to commit one instruction at a time, in order, providing the architectural state of the processor after every single instruction — although most likely no other processor or unit in the machine needs it.

The goal of the Bulk Multicore Architecture is to provide a scalable shared-memory substrate that enables a highly-programmable environment. At the same time, it minimizes complexity and delivers high performance.

13.2 Contributions

The contribution of this project has been the design of the Bulk Multicore Architecture for programmability, including its novel basic operation, improvements to make it more scalable and usable, and its innovative compilation support. We have also developed additional mechanisms for performance and programmability.

The Bulk Multicore delivers a programmable environment through a three-pronged approach. First, data sharing is managed with a scalable hardware cache coherence based on two novel primitives: continuous Chunks and Signatures. A *Chunk* is a group of dynamically-contiguous instructions (e.g., 5,000) that are executed atomically; a *Signature* is a register that accumulates hash-encoded addresses using a Bloom filter [23].

Second, to help minimize the chance of parallel-programming errors, the Bulk Multicore provides high-performance Sequential memory Consistency (SC), and also introduces several novel hardware primitives that leverage chunks. These primitives enable low-overhead data-race detection, deterministic replay of parallel programs, and high-speed disambiguation of sets of addresses. These primitives have an overhead low enough to be ”always on” during production runs.

Third, the Bulk Multicore is a general-purpose architecture, capable of running both novel software as well as currently-existing software stacks — including those for which performance is paramount.

At the same time, the Bulk Multicore keeps the hardware complexity in check, and delivers high performance. Complexity is minimized by operating on chunks as opposed to single instructions and, as we will see, moving the support for memory consistency away from the core. The higher performance is attained by aggressively reordering instructions and memory accesses inside chunks — both transparently by the hardware, and with a purposely-designed compiler layer.

We have extensively discussed the Bulk Multicore architecture with Intel architects. One result of this work is 11 papers in the top conferences [5–7, 55, 56, 133–135, 160] and journals [159, 161] — some co-authored by Intel personnel. The basic Bulk Multicore architecture appeared in the Communications of the ACM [159]. Nine of the papers appeared in the ISCA, MICRO, HPCA, or ASPLOS top conferences. One of them received a 2009 IEEE Micro Top Picks from Computer Architecture Conferences Award [161]. Also, the chunk-based deterministic replay resulting from Bulk (Section 12) resulted in a hardware prototype developed together with Intel [127] and was selected as a Research Highlight in the Communications of the ACM [80].

13.2.1 Basic Bulk Architecture

The Bulk Multicore [159] eliminates the need to efficiently support per-instruction in-order commit. The default execution mode of a processor running a thread is to commit chunks of instructions from the thread at a time. This continuous "chunked" execution is, by default, a hardware mechanism invisible to the software. However, the compiler can also exploit it. Chunk execution improves programmability and performance.

Each chunk executes on the processor atomically and in isolation. Atomic execution means that none of the chunk's actions are made visible to the rest of the system (processors or main memory) until the chunk completes and commits. Execution in isolation means that, if the chunk reads a location and, before it commits, a second chunk in another processor that has written to the location commits, then the local chunk is squashed and must re-execute. Intuitively, processors continuously execute hardware-initiated transactions.

To execute chunks inexpensively, the Bulk Multicore introduces hardware address signatures. Figure 13.1(a) outlines a simple signature implementation. The bits of an incoming address go through a fixed permutation to reduce collisions and are then separated into bit fields C_i. Each field is hashed and accumulated into a bit field V_j in the register. A signature, therefore, represents a set of addresses.

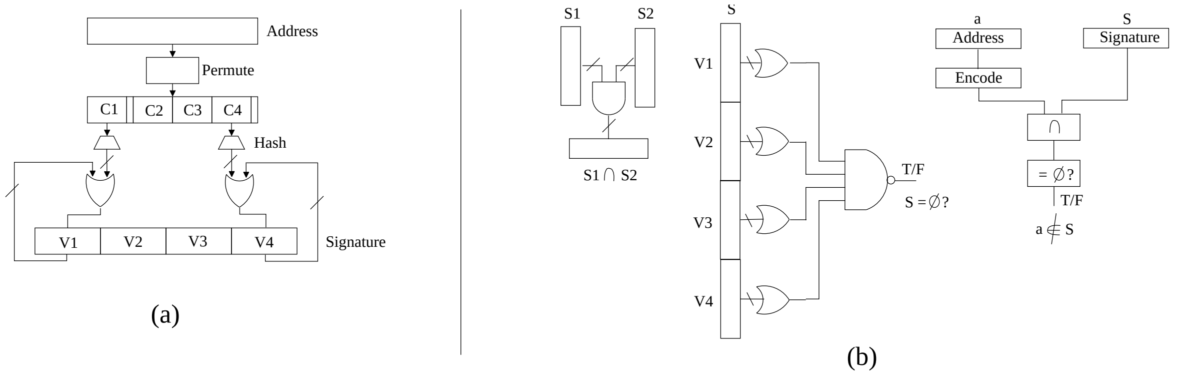


Figure 13.1: Signature (a) and operations on signatures (b).

The hardware in each processor automatically accumulates the addresses read and written by a chunk into a read (*R*) and a write (*W*) signature. Then, simple software-invisible functional units operate efficiently on signatures (Figure 13.1(b)). They perform operations such as intersection, test for null signature, or test for address membership. Intersection finds the addresses common to two signatures by performing a bit-wise AND of them. The resulting signature is empty if any of its bit-fields is all zeros. Testing whether an address *a* is present in a signature involves encoding *a* into a signature, intersecting it with the original signature and then testing the result for an empty signature.

Atomic chunk execution is supported by buffering the state generated by the chunk in the cache. No update is propagated outside the cache while the chunk is executing. When the chunk completes, it proceeds to commit. A successful commit involves sending the chunk's *W* signature to the subset of sharer processors as indicated by the directory. The written lines remain dirty in the cache. The *W* signature carries enough information to both invalidate stale lines from the other coherent caches and enforce the isolation of all the concurrently executing chunks. To enforce the latter, when a processor receives an incoming signature W_{inc}, its hardware intersects W_{inc} against the local R_{loc} and W_{loc} signatures. If any of the two intersections is not null, it means (conservatively) that the local chunk has accessed a data element written by the committing chunk. Consequently, the local chunk is squashed and restarted.

Figure 13.2 shows an example. Thread $T0$ executes a chunk that writes variables *B* and *C* without sending invalidations. Signature $W0$ stores the hashed addresses of *B* and *C*. Concurrently, Thread $T1$ issues reads for *B* and *C*, which (by construction) load the non speculative values of the variables — those before $T0$'s updates. When $T0$'s chunk commits, the hardware sends $W0$ to $T1$. At $T1$, the hardware intersects $W0$ with the ongoing chunk's $R1$ and $W1$. If $W0 \cap R1$ or $W0 \cap W1$ are not null, $T1$'s chunk is squashed.

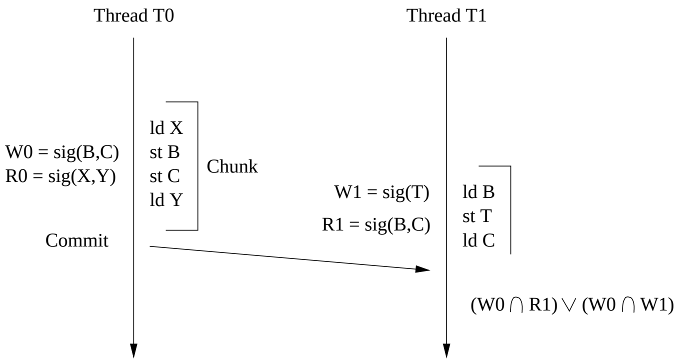


Figure 13.2: Using signatures to execute chunks atomically and in isolation.

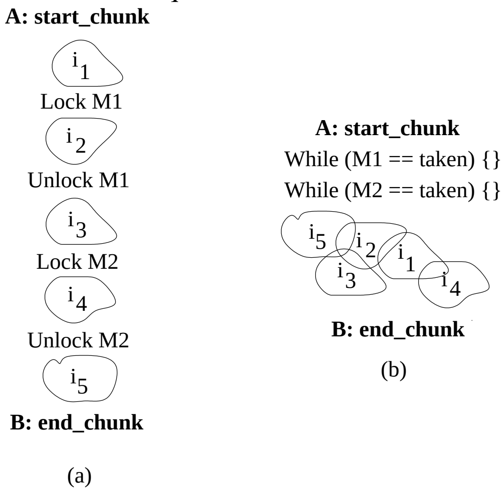


Figure 13.3: Compiler-driven chunking for high performance. In the figure, each i_j represents a set of instructions.

The commit of chunks is serialized across the whole machine. Serialization is enforced by an arbiter module. *W* signatures are sent to the arbiter. The arbiter either acknowledges that the chunk can be considered committed or, if there is another chunk with a conflicting signature currently committing, requests a retry.

Programmability Advantages. Since chunks execute atomically and in isolation, commit in program order in each processor, and there is a global commit order of chunks, the Bulk Multicore supports SC at the chunk level — and, therefore, at the instruction level. This makes software development and debugging less painful. Moreover, it allows the hardware to work well with software correctness tools, since most of them assume SC.

Chunks also free development and debugging tools from having to record or be concerned with individual loads and stores; they only need to keep per-chunk state. This in turn allows the development of novel hardware primitives for program development that can be "always on". For example, we have proposed primitives for deterministic replay [110] or data-race detection [130]. More obviously, chunks provide support for thread-level speculation and transactional memory.

Performance Advantages. The Bulk Multicore delivers high performance because its hardware can reorder and overlap memory access within a chunk. Inside a chunk, fences are no-ops and synchronization instructions do not limit reordering. This is because the intermediate state of a chunk cannot be observed by any other processor. Furthermore, if the compiler marks chunk boundaries in the static program, it can perform aggressive code optimizations within a chunk. Such optimizations may be illegal in conventional processors, but are acceptable in Bulk as long as, by the time the chunk is about to commit, the final state is correct (Section 13.2.3). The result is higher performance than conventional multicores.

Complexity Advantages. The Bulk Multicore reduces complexity by largely decoupling memory-consistency enforcement from processor structures. Specifically, thanks to the use of chunks, SC enforcement is performed with simple signature intersections outside of the processor core. Conventional processors, instead, need intrusive processor hardware to check that individual local accesses issued speculatively with respect to the memory consistency model are not observed by other processors [159].

13.2.2 Improving Bulk Scalability and Usability

Distribution of Chunk Commit. The basic Bulk Multicore uses a centralized arbiter for commit, which compares the address signatures of chunks attempting to commit concurrently. In a large multicore, a single arbiter becomes a bottleneck. To solve this problem, *ScalableBulk* [133] introduces a highly-overlapped, scalable chunk commit protocol. This protocol uses the directory modules of a distributed directory protocol as a distributed arbiter. The commit operation uses no centralized structure, and the committing processor communicates only with the relevant directory modules. Importantly, multiple chunks from different processors can concurrently commit even if they communicate with the same directories, as long as their signatures do not overlap.

Support for Simultaneous Multithreaded (SMT) Processors. The basic Bulk Multicore uses single-context cores. However, SMT cores are especially attractive for Bulk. Specifically, the cost of interaction between the multiple contexts of the same core is very low. Hence, SMT cores can easily support the concurrent execution of dependent chunks from different contexts. Consequently, we designed *BulkSMT* [134], the first SMT design that supports chunked (or transactional) execution. BulkSMT has three configurations that behave differently when two threads have a data conflict: *Squash-on-conflict* squashes one of the threads, *Stall-on-conflict* stalls one of the threads until the other commits, and *Order-on-conflict* records the order of the dependence and will order the chunk commits in the same order. We also showed how to design a multicore of BulkSMTs.

High-Speed Disambiguation of Sets of Addresses. We extended Bulk with *SoftSig* [160, 161], a small register file of signatures and corresponding ISA that can be used by the software for code analysis and optimization. SoftSig allows the programmer or compiler to automatically encode a set of addresses into a signature, and then check if they overlap with a second set of addresses.

SoftSig provides three primitives. The first one collects the addresses accessed in a code region into a signature. The second one checks the overlap between a signature and the addresses accessed in a code region. The third one checks the overlap between the signature and the incoming coherence messages received while executing a code region. These primitives are the basis of many potential code optimizations.

13.2.3 The Bulk Compilation Support

We developed *BulkCompiler* [5–7], a compiler that ensures that the Bulk Architecture delivers high performance and supports SC. BulkCompiler forms the chunks in an intelligent manner, following two main principles. The first one is to place chunk boundaries around code that the compiler can optimize. The second one is to terminate a chunk before a point that is likely to conflict with a concurrently-executing chunk.

Forming a Chunk Around Code that Can Be Optimized. BulkCompiler looks for code regions that it can optimize aggressively. The optimizations can be illegal under certain conditions, as long as the state is correct when the chunk is about to commit.

One example is code around low-contention critical sections [7], which are common in Java as synchronized blocks. BulkCompiler includes one or several of these sections and their surrounding code in the same chunk (Figure 13.3(a)). Then, each Lock operation is replaced with a spinning loop with plain loads, which checks if the synchronization variable is taken. All the Unlock operations are removed. Next, BulkCompiler moves the spins to the top of the chunk — subject to data and control dependences — to better prepare the code for compiler optimization. Finally, with the synchronizations removed, BulkCompiler uses conventional optimizations to aggressively reorder and optimize the code inside the chunk. The resulting code is shown in Figure 13.3(b), where the overlapping sets of instructions denote the beneficial effects of compiler optimization.

Since the chunk is executed atomically, there is no need to acquire and release any lock. However, the chunk still needs to check if any lock is taken. A lock can be taken if another thread, after failed attempts to execute its own chunk, reverted to a non-speculative Safe Version of the code, where it grabbed the lock explicitly. In this case, when the owner of the lock releases the lock, the spinning chunk will get squashed.

This transformation improves performance in two ways. First, it replaces costly synchronization operations with cheaper loads. Second, it eliminates the constraints on instruction reordering imposed by synchronizations. Indeed, even under relaxed memory models, compilers cannot move instructions across synchronizations. After BulkCompiler removes the synchronizations, BulkCompiler can reorder the code. Hence, BulkCompiler can attain higher performance than conventional compilers for relaxed-consistency machines while supporting SC.

Terminating a Chunk to Avoid Squashes. BulkCompiler terminates chunks to minimize squashes [5]. Squashes due to data conflicts occur primarily because threads communicate. Hence, BulkCompiler identifies the code locations where threads communicate. It focuses on the first communication in a code region where two threads may perform multiple communications. We call these operations *Squash Hazards* [5]. Based on the common types of Squash Hazards in popular applications, BulkCompiler introduces squash-removing algorithms tailored to them. These algorithms consist of simple code transformations, typically embedded inside synchronization macros, that create chunk boundaries for minimal squashes [5].

Alias Speculation Using Atomic Region Support. *DeAliaser* is a BulkCompiler pass that performs alias speculation [6]. It performs aggressive optimizations inside chunks. Unlike conventional alias analyses, which need to prove aliasing properties, DeAliaser works with aliasing properties that are true most of the time. It detects the cases when they are incorrect at runtime, and just rolls back the execution.

DeAliaser adds a few instructions to enable code motion (hoisting and sinking) inside chunks. The result is optimizations on pointer-intensive code that are currently not possible with conventional compilers.

13.3 Lessons Learned

The main lesson learned is that Bulk's novel chunk-based execution model has advantages in programmability, performance, and complexity reduction. Some of these issues were not obvious at the beginning. Also, these advantages are attained while supporting existing software — in addition to code generated by BulkCompiler.

A second lesson is that the hardware-supported atomic-region primitive has a high potential for improved code generation. We are only beginning to understand it, as we use it for optimizing low-contention critical sections and pointer-intensive code. Many new tools can be built.

A final lesson is that Bloom-filter signatures have a broad use in computer architecture and code optimization. They encode address footprints efficiently and can be managed with signature-based functional units.

13.4 Future Work

A key area for future work is building a more extensive software infrastructure for the Bulk Multicore. This effort includes new code transformations that use chunks, language support for chunk-based execution, and tailored operating system designs. Another area is designing program development tools that take advantage of chunk execution to reduce their state or complexity. Finally, we need to understand the benefits of providing SC execution in the Bulk Multicore and how the programmer can take advantage of it.

13.5 Key Papers and Other Material

We include two papers in this book. The CACM 2009 paper [159] gives an overview of the basic Bulk Multicore architecture, and discusses programmability, performance, and hardware complexity issues. The MICRO 2009 paper [7] describes the basic BulkCompiler. Subsequent papers discuss more advanced topics.

Chapter 14

DeNovo: Rethinking Memory Systems for Disciplined Parallelism

14.1 Problem Addressed

Shared-memory is arguably the most widely used general-purpose multicore parallel programming model. It provides the advantage of a global address space, potentially promising a straightforward adaptation of the sequential model. Unfortunately, shared-memory programs are known to be difficult to debug and maintain [102]. Unstructured parallel control, data races, and ubiquitous non-determinism make shared-memory programs hard to understand, and forfeit safety, modularity, and composability. At the same time, designing performance-, power-, and complexity-scalable hardware for such a software model remains a major challenge. Directory-based cache coherence protocols are notoriously complex to verify [1], hard to extend, and hard to scale, and remain an active area of research (e.g., [76, 138, 172]). More fundamentally, despite decades of research, it has been difficult to define an acceptable memory semantics (the memory model) for current popular systems, resulting in a call for rethinking current languages and hardware [2].

The above problems have led some researchers to promote abandoning shared-memory altogether, giving up on the significant advantages of a global address space [84, 102]. The DeNovo project takes the view that these problems are not inherent to a global address space paradigm. Instead, they occur due to undisciplined programming models that use arbitrary reads and writes for implicit and unstructured communication and synchronization. This results in "wild shared-memory" behaviors with unintended data races, non-determinism, and implicit side effects. The same phenomena also result in complex hardware that must assume that any memory access may trigger communication, and performance- and power-inefficient hardware that is unable to exploit communication patterns known to the programmer but obfuscated by the programming model.

There has been a recent surge of research on disciplined shared-memory programming models to address the software problem (e.g., [4, 9, 13, 19, 24, 25, 28, 33, 67, 73, 97, 121]). The DeNovo project asks the question: *if software becomes more disciplined, can we build more performance-, power-, and complexity-scalable shared-memory hardware?* Our work shows that the evolving software landscape represents a unique opportunity for a new multicore architecture paradigm. Compared to conventional hardware driven by "wild shared memory programming models," disciplined models can significantly simplify the hardware implementation, reduce communication traffic, and provide comparable or better performance with commensurate energy savings.

14.2 Contributions

DeNovo involved a collaboration with languages and applications researchers. We first briefly describe our work on disciplined software and applications (details in Chapters 8 and 3), and then our hardware contributions.

14.2.1 Disciplined Shared-Memory Software

As an exemplar of disciplined shared memory models, we initially used the Deterministic Parallel Java (DPJ) language to drive the DeNovo design. DPJ programs have the following properties [25, 28].
(1) Structured parallel control that clearly demarcates parallel sections of the code (or *parallel phases*).

(2) Explicit specification of side effects of parallel sections through a region based type and effects system. The programmer assigns every object field/array element to a "region" and annotates every method with read and write "effects," indicating which (possibly non-contiguous) regions will be read or written in a parallel section.
(3) Guaranteed data-race-freedom and deterministic-by-default semantics. This is achieved by ensuring that concurrent tasks do not have conflicting effects in deterministic code sections.
(4) Strong safety properties for non-deterministic code sections; e.g., data-race-freedom, strong isolation, and composition with non-deterministic code sections. This is achieved by ensuring that conflicting accesses in concurrent tasks are confined to *atomic sections* and their regions and effects are explicitly annotated as *atomic*.

The DPJ language design was driven entirely by considerations for concurrency safety in software. Sections 14.2.3 and 14.2.4 describe how DeNovo uses these properties to enable simpler, faster, and more energy-efficient hardware. Although much of the DeNovo work has been driven by DPJ, we have recently begun work on a language-neutral virtual ISA to capture properties that were deemed to be the most important to DeNovo at the ISA level, thereby decoupling DeNovo from any specific language [3] (Section 14.2.5).

14.2.2 Applications

To ensure DeNovo's relevance to future real client applications, we collaborated with the AvaScholar project on key requirements for visual computing. Rapid construction of hierarchical spatial data structures is a common task in such applications, and the k-D Tree is a commonly used data structure. The highest quality k-D tree can be constructed using a surface area heuristic (SAH) based optimization; however, parallelizing SAH-based algorithms was previously thought to be difficult. We developed two parallel algorithms that provided the best known speedups for precise SAH-based k-D trees [38]. These algorithms make different tradeoffs between the total work done and the amount of data movement, providing different scalability characteristics.

Our experience with these algorithms inspired key optimizations for DeNovo. They exposed common problems with object-oriented programming techniques using array-of-struct (AoS) style data structures that make inefficient use of cache capacity and network bandwidth. One of the algorithms converted some AoS structures to the more cache and bandwidth friendly struct-of-arrays (SoA) style, but this required considerable programming complexity. This experience motivated hardware strategies in DeNovo that effectively provide AoS to SoA transformations without requiring software changes (flexible communication granularity in [39] and ongoing work on region caches, where only the required structure fields or regions are transferred and stored).

14.2.3 DeNovo Architecture for Deterministic Codes

We first discuss DeNovo for deterministic codes. Conventional directory coherence protocols ensure reads return updated values by tracking all current sharers and invalidating them on a write, incurring significant storage (sharer-list) and network traffic (invalidations and acknowledgments) overhead. These protocols also have many transient states, making them hard to verify and extend. DeNovo uses software information to achieve lower overheads and simpler design.

DeNovo observes that coherence requires ensuring (1) a read hit in a private cache never sees "stale" data and (2) a read miss always knows where to get "up-to-date" data. For the first part (no stale values on a hit), DeNovo exploits the software knowledge of which regions are written in a parallel phase of the program. Before a new phase, each core issues a self-invalidation to invalidate any data in its private cache that could have been written in the previous phase by another core. Data-race-freedom implies that if a core reads some data in the subsequent phase, no other core could have concurrently written that data, ensuring a read never returns a stale value. This eliminates the need for tracking sharer lists in directories and the ensuing invalidation and acknowledgment messages, significantly simplifying the protocol.

For the second part (locating up-to-date data on a miss), writers "register" themselves (at word granularity) in a structure similar to the directory, which we call the "registry." However, unlike directories, the registry does not need additional storage overhead in the presence of shared last level caches. The registry needs to store

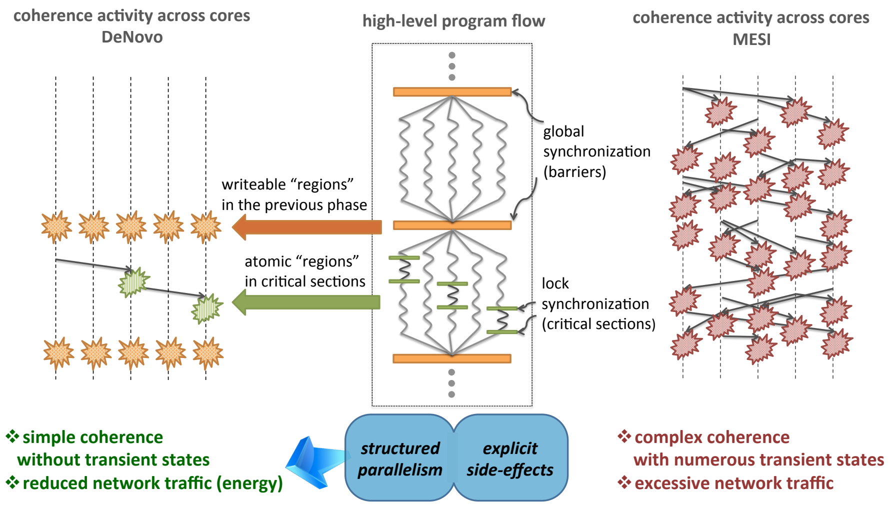

coherence activity across cores DeNovo
high-level program flow
coherence activity across cores MESI
global synchronization (barriers)
writeable "regions" in the previous phase
atomic "regions" in critical sections
lock synchronization (critical sections)
❖ simple coherence without transient states
❖ reduced network traffic (energy)
structured parallelism
explicit side-effects
❖ complex coherence with numerous transient states
❖ excessive network traffic

Figure 14.1: Conceptual Comparison of the Complexity and Network Activities of DeNovo vs. MESI coherence.

either the up-to-date value of the data or the location of only one up-to-date copy of the data – this can be easily stored in the shared last level data arrays.

Because there are no data races, the DeNovo protocol does not have any transient states – it has exactly three stable states (invalid, valid, and registered). Figure 14.1 gives a high level illustration for coherence activities for DeNovo and MESI and the following summarizes the advantages of such a protocol.

Simplicity: To quantify DeNovo's simplicity, we compared it with a conventional MESI protocol using the Murphi model checking tool for verification [52]. For MESI, we used the implementation in the Wisconsin GEMS simulator [107] as an example of a publicly available, state-of-the-art, mature implementation. We found six bugs in MESI that involved subtle protocol races and took several days to analyze and fix. We found three bugs in DeNovo that were mostly an incorrect translation from our high-level specification to implementation and were straightforward to fix. This was surprising since the GEMS MESI protocol was mature and had been used by many researchers while the DeNovo protocol was new and immature. The debugged MESI showed 15X more reachable states compared to DeNovo, with a verification time difference of 20X (173 seconds for MESI vs. 8.7 seconds for DeNovo). These results attest to the complexity of the MESI protocol and the relative simplicity of DeNovo. Further details on our verification effort can be found in [94].

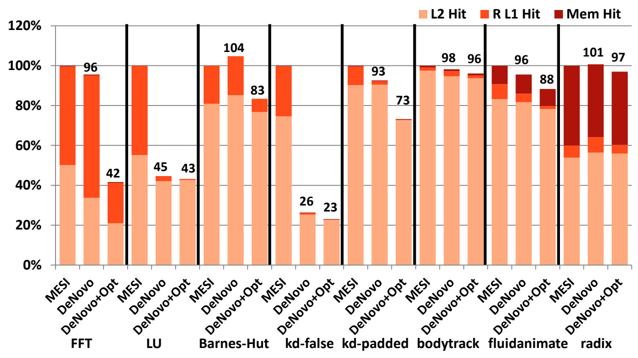
Extensibility: We implemented two optimizations [39]: (1) Direct cache-to-cache transfer: Data in a remote cache may be sent to another cache without indirection through the registry via producer-prediction. (2) Flexible communication granularity based on regions rather than cache lines: This decouples the communication granularity from the address and coherence granularity via a programmer-defined *communication space* based on regions. It allows programmers to define sets (regions) of memory locations that should be transferred together (vs. a contiguous cache line). Neither optimization required adding any new protocol states to DeNovo; since there are no sharer lists and no transient states, valid data can be freely transferred from one cache to another. This is in contrast to current protocols that require significant effort to incorporate and verify such optimizations.

Storage overhead: DeNovo incurs no directory storage overhead with shared last-level caches, a source of unscalability in current systems. Accounting for some increase in DeNovo's overhead (e.g., for regions), DeNovo overhead wins after a few tens of cores and is scalable beyond (constant per cache line).

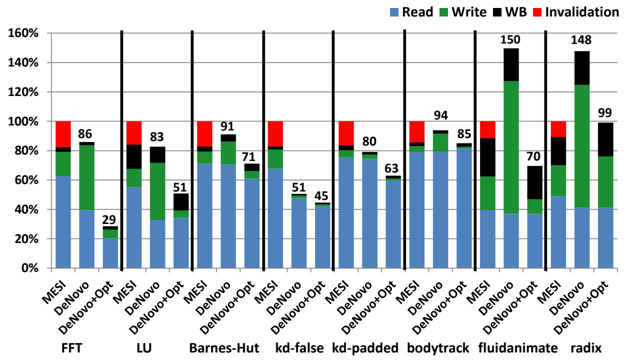
Performance and energy: In our evaluations, the base DeNovo protocol performed about the same or better than MESI for a range of applications [39]. Figure 14.2 shows data memory stall time and network traffic, the two aspects of the execution directly targeted by DeNovo. For each application, the figure shows results for MESI (as implemented in GEMS), the base DeNovo protocol, and DeNovo with optimizations. DeNovo's memory stall time is always comparable to or better than MESI, with an improvement of up to 77%. The gains are the most for applications with false sharing (because DeNovo keeps coherence state at word granularity and hence does not incur false sharing) and AoS style data accesses (because of the flexible communication granularity optimization). These gains are accompanied by reductions in the cache miss rate (as shown in [39]) and network traffic; we therefore expect commensurate reductions in energy.

The network traffic graph shows that DeNovo reduces traffic significantly in many cases; however, for two cases, traffic increases because of word granularity registration requests. We added a simple optimization, write combining, which aggregates individual registration requests for words in a given cache line into a single request (similar to a combining write buffer). This optimization is included in the bar labeled DeNovo+opt. With all the optimizations, DeNovo reduces network traffic by up to 71% for the applications studied.

In recent work [148], we have performed a more systematic analysis of network traffic. Our analysis shows that MESI exhibits much wasted data movement (almost 70% in some cases) which is difficult to eliminate. DeNovo (after the above optimizations) still incurs some waste (much less than MESI), but has potential to reduce this waste with further optimizations that we are exploring in ongoing work [148].



(a) Data memory stall time



(b) Network traffic (flit-crossing)

Figure 14.2: Performance of DeNovo vs. MESI.

14.2.4 Beyond Deterministic Codes

So far, we have focused on deterministic codes. Our recent and ongoing work extends DeNovo to cover more general codes, without sacrificing its advantages.

DeNovoND for disciplined non-determinism: As discussed earlier, DPJ permits disciplined non-determinism by permitting conflicting concurrent accesses, but constraining them to occur within well defined atomic sections with explicitly declared atomic regions and effects [28]. We have shown that modest extensions to DeNovo can allow this form of non-determinism without sacrificing its advantages. The key insight is to use small and simple hardware Bloom filters to track and communicate only the non-deterministic accesses (i.e., those identified as atomic) across explicit lock transfers. The locks themselves are implemented using ideas similar to queue-based locks or QOSB [70], without requiring maintaining sharer's lists and invalidation messages. The resulting system, DeNovoND, provides comparable or better performance than MESI for several applications designed for lock synchronization, and shows 33% less network traffic on average, implying potential energy savings [155].

Lock-free synchronization: We are currently working on characterizing general (lock-free) synchronization patterns. We expect that a combination of statically specified effects-driven (as in [39]) and dynamic signature-driven (as in [155]) invalidations can be used to correctly implement such patterns on DeNovo with modest extensions, while preserving the advantages of no sharer's list, no transient states, reduced network bandwidth waste, and the performance benefits of DeNovo. We expect that the ability to correctly implement such general synchronization patterns will enable us to support most code patterns of interest.

Legacy codes: Although we expect software will become more disciplined (e.g., recent work has even explored alternatives for operating systems [16]), we intend to support legacy codes (which will not contain our required annotations but will obey state-of-the-art memory models such as the Java and C++ models requiring explicit synchronization). We could handle such codes through a combination of conservative self-invalidations (using signatures or simply invalidating the entire cache) and an appropriate writethrough policy at synchronization points. Another alternative is to support such codes temporarily using a small on-chip hardware cache-coherent cluster. The goal here is to provide a transition path for such codes, and not necessarily the best performance. In the long-term, we expect legacy software to remain, but only as part of larger, disciplined software. For such a case, we expect our conservative approach to be more effective; e.g., the use of signatures to drive invalidations over small code sections may be viable. Determining the best transition path is part of our ongoing work.

14.2.5 Heterogeneous systems

So far we have focused on homogeneous multicores; however, heterogeneity is a natural path for improved energy efficiency. Key impediments to programmability for such systems include disparate memory systems and ISAs. We are currently working on applying DeNovo's approach to unifying the memory systems of heterogeneous compute elements found in modern SOCs [3]. DeNovo's software-driven coherence protocol with its flexible, region based customization of communication and cache capacity allocation as well as direct cache to cache transfer strategy maps well to the memory and communication demands of heterogeneous systems. To address the disparate ISAs, we are working on a common virtual ISA that can capture the needed forms of parallelism and the information DeNovo requires for its memory system in a language neutral way [3].

14.3 Lessons Learned

We started from the hypothesis that co-designing shared-memory hardware and software would address imminent issues in hardware complexity and efficiency. Our most important lesson is that our hypothesis holds not only in the baseline system we initially had in mind, but applies in the same strong way to more general or fundamentally different (heterogeneous) systems and applications. The following gives more specifics.
(1) Current shared-memory systems exhibit significant inefficiencies, largely from a software-oblivious design.
(2) A software-driven approach is important. Once we had a clear vision of what would be desirable shared-memory languages and applications, it drove a clear vision for much simpler and efficient hardware.
(3) When taking a software-driven approach, starting from constrained software and then widening the space worked well. We started with only deterministic software, which helped to isolate what is really important to implement in hardware and what can be obtained from software. As we added more software complexity, i.e., non-determinism, the additional hardware support required was minimal.
(4) Even for heterogeneous systems which have distinctively different system characteristics from conventional homogeneous multicore systems, our approach seems to apply well.
(5) Overall, a careful software-driven design led to hardware that is simple, fast, and energy-efficient. The software ideas themselves were motivated to make shared-memory programming easier.

14.4 Future Work

We are currently working on: (1) completing DeNovo support for lock-free synchronization and legacy codes; (2) applying DeNovo principles to tightly integrate the memory systems for heterogeneous systems for better programmability and efficiency; (3) using regions to drive cache allocation; (4) applying DeNovo ideas to main memory; and (5) the language-neutral virtual ISA to capture information required by DeNovo.

14.5 Key Papers and Other Material

We include two papers in this book. The PACT 2011 paper [39] describes the DeNovo architecture for deterministic codes (Section 14.2.3). This work won the best paper award at the conference. The ASPLOS 2013 [155] paper extends DeNovo to support safe non-determinism, enabling the use of lock based codes (Section 14.2.4).

Chapter 15

Concluding Remarks

Five years of research have allowed us to gain many insights on the technical issues related to the use of multicores. As described in the previous pages, some of the research we have done is already influencing products; other research, while promising, will require many years to come to fruition. Meanwhile, multicore software and hardware technology continues to evolve, new market segments continue to rise while others decline, and new exciting ideas continue to appear.

Application domains for clients are still in flux. The use of multi-modal user interfaces (with voice and gesture recognition), one of the focus areas of our research, is slowly making headway into cellphones and tablets — although voice recognition in systems such as Siri is cloud-based. Cameras, including those on cellphones and tablets, have increasingly sophisticated image-processing capabilities, e.g., for face recognition, scene recognition, digital red-eye reduction, and image deblurring.

Software frameworks and tools remain disparate, and it seems clear that no single solution is emerging. For example, multi-modal user interfaces are made available to application developers via toolkits that encapsulate and hide from them the lower-level image-processing layers. In effect, parallelism is handled by specialists, while most application developers use embedded domain-specific languages to develop applications.

Multiprocessor computer architectures also continue to evolve. The increased chip densities are used to integrate full systems on one chip. For example, chips used in smartphones combine general-purpose cores with specialized components such as GPU, DSP, modem, video, image processor, GPS, and display controller. The functionality of components other than the general-purpose cores is accessed by application developers through vendor-provided libraries. Further integration will result in 3-D chips and processing near memory.

In addition, acute energy consumption concerns are affecting architectures. Multicores are becoming heterogeneous, enabling different parts of an application to run on the component that supports that part most efficiently. Since it is feasible to integrate more compute engines than can be simultaneously powered, we may need to keep different parts of the chip powered down at all times. Alternatively (or in addition), new approaches such as ultra-low voltage operation will require rethinking the architecture.

Looking forward, continued growth in microprocessor performance will be increasingly dependent on innovations at the architecture level and above. Semiconductor technology cycles will lengthen and the design of special purpose architectures will become more worthwhile. The future decade is likely to be one of fast change and market dislocations for the microprocessor industry; it is also likely to be one of fast innovation, offering renewed opportunities for revolutionary research.

Bibliography

[1] D. Abts, S. Scott, and D. J. Lilja. So Many States, So Little Time: Verifying Memory Coherence in the Cray X1. In *International Parallel & Distributed Processing Symposium*. IEEE, 2003.

[2] S. V. Adve and H.-J. Boehm. Memory models: A case for rethinking parallel languages and hardware. *Communications of the ACM*, 53(8):90–101, 2010.

[3] V. Adve, S. Adve, R. Komuravelli, M. D. Sinclair, and P. Srivastava. Virtual instruction set computing for heterogeneous systems. In *USENIX Workshop on Hot Topics in Parallelism (HotPar)*, June 2012.

[4] V. S. Adve and L. Ceze. *Workshop on Deterministic Multiprocessing and Parallel Programming, U-Washington*, 2009.

[5] R. Agarwal and J. Torrellas. FlexBulk: Intelligently Forming Atomic Blocks in Blocked-Execution Multiprocessors to Minimize Squashes. In *International Symposium on Computer Architecture*, pages 33–44, June 2011.

[6] W. Ahn, Y. Duan, and J. Torrellas. DeAliaser: Alias Speculation Using Atomic Region Support. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 167–180, March 2013.

[7] W. Ahn, S. Qi, J.-W. Lee, M. Nicolaides, X. Fang, J. Torrellas, D. Wong, and S. Midkiff. BulkCompiler: High-Performance Sequential Consistency through Cooperative Compiler and Hardware Support. In *International Symposium on Microarchitecture*, pages 133–144, December 2009.

[8] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the PTRAN analysis system for multiprocessing. *Journal of Parallel and Distributed Computing*, 5(5):617–640, 1988.

[9] M. D. Allen, S. Sridharan, and G. S. Sohi. Serialization Sets: A Dynamic Dependence-based Parallel Execution Model. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 85–96, 2009.

[10] R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Symposium on Principles of Programming Languages (POPL)*, pages 63–76, 1987.

[11] S. P. Amarasinghe, J.-A. M. Anderson, M. S. Lam, and A. W. Lim. An overview of a compiler for scalable parallel machines. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 253–272, 1993.

[12] Amazon.com, Inc. Amazon Silk Browser. `http://amazonsilk.wordpress.com/`, Sept 2011.

[13] Z. Anderson, D. Gay, R. Ennals, and E. Brewer. SharC: Checking Data Sharing Strategies for Multithreaded C. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 149–158. ACM, 2008.

[14] D. Andrade, B. Fraguela, J. Brodman, and D. Padua. Task-parallel versus data-parallel library-based programming in multicore systems. In *International Conference on Parallel, Distributed and Network-based Processing*, pages 101–110, 2009.

[15] C. Badea, M. R. Haghighat, A. Nicolau, and A. V. Veidenbaum. Towards Parallelizing the Layout Engine of Firefox. In *USENIX Workshop on Hot Topics in Parallelism (HotPar)*, 2010.

[16] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Symposium on Operating systems Principles*, pages 29–44, 2009.

[17] T. Beeler, B. Bickel, P. Beardsley, B. Sumner, and M. Gross. High-quality single-shot capture of facial geometry. *ACM Transactions on Graphics (TOG)*, 29:40:1–40:9, July 2010.

[18] C. Beleznai, D. Schreiber, and M. Rauter. Pedestrian detection using GPU-accelerated multiple cue computation. In *Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 58–65, 2011.

[19] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe multithreaded programming for C/C++. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 81–96, 2009.

[20] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarán, D. Padua, and C. von Praun. Programming for Parallelism and Locality with Hierarchically Tiled Arrays. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 48–57, 2006.

[21] G. Bikshandi, J. Guo, C. von Praun, G. Tanase, B. B. Fraguela, M. J. Garzarán, D. Padua, and L. Rauchwerger. Design and Use of htalib - a Library for Hierarchically Tiled Arrays. In *Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 17–32, 2006.

[22] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *International Conference on Supercomputing*, pages 340–347. ACM, 1997.

[23] B. Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 11(7):422–426, July 1970.

[24] R. D. Blumofe et al. Cilk: An Efficient Multithreaded Runtime System. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, 1995.

[25] R. Bocchino, Jr., V. Adve, D. Dig, S. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A Type and Effect System for Deterministic Parallel Java. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 97–116, 2009.

[26] R. L. Bocchino, V. Adve, S. Adve, and M. Snir. Parallel programming must be deterministic by default. In *USENIX Workshop on Hot Topics in Parallelism (HotPar)*, 2009.

[27] R. L. Bocchino and V. S. Adve. Types, regions, and effects for safe programming with object-oriented parallel frameworks. In *European Conference on Object-Oriented Programming*, 2011.

[28] R. L. Bocchino, Jr., S. Heumann, N. Honarmand, S. V. Adve, V. S. Adve, A. Welc, and T. Shpeisman. Safe Nondeterminism in a Deterministic-by-Default Parallel Language. In *Symposium on Principles of Programming Languages (POPL)*, pages 535–548, 2011.

[29] R. L. Bocchino Jr. *An Effect System and Language for Deterministic-by-Default Parallel Programming*. PhD thesis, University of Illinois, Urbana-Champaign, IL, 2010.

[30] Broadcom Inc. BCM28150 - 1080p 4G HSPA+ smartphone processor, 2011. `http://www.broadcom.com/products/Cellular/3G-Baseband-Processors/BCM28150`.

[31] J. C. Brodman. *Data Parallelism with Hierarchically Tiled Objects*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2011.

[32] J. C. Brodman, B. B. Fraguela, M. J. Garzarán, and D. Padua. New abstractions for data parallel programming. In *USENIX Workshop on Hot Topics in Parallelism (HotPar)*, Berkeley, CA, USA, 2009. USENIX Association.

[33] Z. Budimlic, A. Chandramowlishwaran, K. Knobe, G. Lowney, V. Sarkar, and L. Treggiari. Multi-core implementations of the concurrent collections programming model. In *Workshop on Compilers for Parallel Computers (CPC)*, 2009.

[34] L. Cao, H. D. Kim, M.-H. Tsai, B. Cho, Z. Li, I. Gupta, C. Zhai, and T. S. Huang. Delta-SimRank Computing on MapReduce. In *International Workshop on Big Data Mining*, August 2012.

[35] L. Ceze, C. von Praun, C. Cascaval, P. Montesinos, and J. Torrellas. Concurrency Control with Data Coloring. In *Workshop on Memory Systems Performance and Correctness (MSPC)*, pages 6–10. ACM, March 2008.

[36] L. Ceze, C. von Praun, C. Cascaval, P. Montesinos, and J. Torrellas. Programming and Debugging Shared Memory Programs with Data Coloring. In *Workshop on Compilers for Parallel Computing (CPC)*, January 2009.

[37] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. *IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[38] B. Choi, R. Komuravelli, V. Lu, R. L. Bochino, H. Sung, S. V. Adve, and J. C. Hart. Parallel SAH k-D tree construction. *Conference on High Performance Graphics*, pages 77–86, June 2010.

[39] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou. DeNovo: Rethinking the memory hierarchy for disciplined parallelism. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 155–166, October 2011.

[40] S. Damaraju, V. George, S. Jahagirdar, T. Khondker, R. Milstrey, S. Sarkar, S. Siers, I. Stolero, and A. Subbiah. A 22nm IA multi-CPU and GPU system-on-chip. In *Digest of Technical Papers - IEEE International Solid-State Circuits Conference*, volume 55, pages 56–57, 2012.

[41] A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, and M. Horowitz. CPU DB: Recording microprocessor history. *Queue*, 10(4):10:10–10:27, Apr. 2012.

[42] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of Ion-Implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[43] D. Dig. ICSM Conference Tutorial on Refactoring for Parallelism. `http://icsm2010.upt.ro/program/tutorials/dig`.

[44] D. Dig. Multicore Parallel Programming with Java (CS498DD)– course webpage. `https://wiki.engr.illinois.edu/display/cs498dd/Home`.

[45] D. Dig. OOPSLA Conference Tutorial on Refactoring for Parallelism. `http://tinyurl.com/SPLASH-Tutorial`.

[46] D. Dig. A refactoring approach to parallelism. *IEEE Software*, 28(1):17–22, January-February 2011.

[47] D. Dig, J. Marrero, and M. Ernst. Concurrencer: A tool for retrofitting concurrency into sequential Java applications via concurrent libraries. In *ICSE Companion*, pages 399–400, May 2009.

[48] D. Dig, J. Marrero, and M. D. Ernst. Refactoring sequential Java code for concurrency via concurrent libraries. In *International Conference on Software Engineering (ICSE)*, pages 397–407, May 2009.

[49] D. Dig, J. Marrero, and M. D. Ernst. How do programs become more concurrent? A story of program transformations. In *International Workshop on Multicore Software Engineering (IWMSE)*, pages 1–8, May 2011.

[50] D. Dig, M. Tarce, C. Radoi, M. Minea, and R. E. Johnson. ReLooper: Refactoring for loop parallelism in Java. In *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA Companion)*, pages 793–794, October 2009.

[51] M. Dikmen, D. Hoiem, and T. S. Huang. A data driven method for feature transformation. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3314–3321. IEEE, 2012.

[52] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol Verification as a Hardware Design Aid. In *International Conference on Computer Design (ICCD)*, pages 522–525, Washington, DC, USA, 1992. IEEE Computer Society.

[53] M. N. Do, Q. H. Nguyen, H. T. Nguyen, D. Kubacki, and S. J. Patel. Immersive visual communication. *IEEE Signal Processing Magazine*, 28:58–66, Jan. 2011.

[54] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, Mar. 1990.

[55] Y. Duan, A. Muzahid, and J. Torrellas. WeeFence: Toward Making Fences Free in TSO. In *International Symposium on Computer Architecture*, June 2013.

[56] Y. Duan, X. Zhou, W. Ahn, and J. Torrellas. BulkCompactor: Optimized Deterministic Execution via Conflict-Aware Commit of Atomic Blocks. In *International Symposium on High-Performance Computer Architecture*, pages 1–12, February 2012.

[57] S. Dubey. AJAX performance measurement methodology for internet explorer 8 beta 2. CODE Magazine, 2008. `http://www.code-magazine.com/Article.aspx?quickid=0811102`.

[58] A. Duchateau, D. Padua, and D. Barthou. Hydra: Automatic algorithm exploration from linear algebra equations. In *International Symposium on Code Generation and Optimization (CGO)*, pages 1–10, 2013.

[59] A. X. Duchateau. *Automatic Algorithm Derivation and Exploration in Linear Algebra for Parallelism and Locality*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2013.

[60] A. Farzan and P. Madhusudan. The complexity of predicting atomicity violations. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 155–169, Berlin, Heidelberg, 2009. Springer-Verlag.

[61] A. Farzan, P. Madhusudan, N. Razavi, and F. Sorrentino. Predicting null-pointer dereferences in concurrent programs. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 47:1–47:11, New York, NY, USA, 2012. ACM.

[62] A. Farzan, P. Madhusudan, and F. Sorrentino. Meta-analysis for atomicity violations under nested locking. In *International Conference on Computer Aided Verification (CAV)*, pages 248–262, Berlin, Heidelberg, 2009. Springer-Verlag.

[63] E. Fortuna, O. Anderson, L. Ceze, and S. Eggers. A limit study of JavaScript parallelism. In *International Symposium on Workload Characterization (IISWC)*. IEEE, 2010.

[64] G. Fox. Square matrix decompositions-symmetric, local, scattered. Technical Report Hm-97 / C3P-97, Cal Tech, Pasadena, California, August 1984.

[65] B. B. Fraguela, G. Bikshandi, J. Guo, M. J. Garzáran, D. Padua, and C. von Praun. Optimization techniques for efficient HTA programs. *Parallel Computing*, 38(9):465 – 484, 2012.

[66] L. Franklin, A. Gyori, J. Lahoda, and D. Dig. LAMBDAFICATOR: From imperative to functional programming through automated refactoring. In *International Conference on Software Engineering (ICSE)*, pages 1287–1290, May 2013.

[67] A. Ghuloum et al. Ct: A Flexible Parallel Programming Model for Tera-Scale Architectures. Intel White Paper, 2007.

[68] M. Gligoric, V. Jagannath, Q. Luo, and D. Marinov. Efficient mutation testing of multithreaded code. *Software Testing, Verification and Reliability (STVR)*, 23(5):375–403, Aug. 2013.

[69] M. Gligoric, V. Jagannath, and D. Marinov. MuTMuT: Efficient exploration for mutation testing of multithreaded code. In *International Conference on Software Testing, Verification, and Validation (ICST)*, pages 55–64, Paris, France, Apr. 2010.

[70] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1989.

[71] C. Grier, S. Tang, and S. T. King. Secure web browsing with the OP web browser. In *IEEE Symposium on Security and Privacy*, pages 402–416, 2008.

[72] J. Guo, G. Bikshandi, B. B. Fraguela, M. J. Garzarán, and D. Padua. Programming with Tiles. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 111–122, 2008.

[73] N. Gustafsson. Axum: Language Overview. Microsoft Language Specification, 2009.

[74] F. Gustavson. High-performance linear algebra algorithms using new generalized data structures for matrices. *IBM Journal of Research and Development*, 47(1):31–55, 2003.

[75] A. Gyori, L. Franklin, D. Dig, and J. Lahoda. From imperative to functional programming through automated refactoring. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 543–553, August 2013.

[76] N. Hardavellas et al. Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches. In *International Symposium on Computer Architecture*, pages 184–195, 2009.

[77] S. Heumann, V. Adve, and S. Wang. The tasks with effects model for safe concurrency. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 239–250, 2013.

[78] J. Hoberock, V. Lu, Y. Jia, and J. C. Hart. Stream compaction for deferred shading. *Conference on High Performance Graphics*, pages 173–180, Aug. 2009.

[79] N. Honarmand, N. Dautenhahn, J. Torrellas, S. T. King, G. Pokam, and C. Pereira. Cyrus: Unintrusive Application-Level Record-Replay for Replay Parallelism. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 193–206, March 2013.

[80] D. R. Hower, P. Montesinos, L. Ceze, M. D. Hill, and J. Torrellas. Two Hardware-Based Approaches for Deterministic Multiprocessor Replay. *Communications of the ACM*, 52(6):93–100, 2009.

[81] W.-m. W. Hwu, editor. *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, San Francisco, CA, USA, Feb. 2011.

[82] W.-m. W. Hwu, editor. *GPU Computing Gems Jade Edition*. Morgan Kaufmann, San Francisco, CA, USA, Oct. 2011.

[83] Illinois faculty and invited guests. I2PC Summer School on Multicore Parallel Programming with Java. `http://i2pc.cs.illinois.edu/summer/2012/schedule.html`.

[84] Intel. The SCC Platform Overview. `http://techresearch.intel.com/spaw2/uploads/files/SCC_Platform_Overview.pdf`.

[85] V. Jagannath. *Improved Regression Testing of Multithreaded Programs*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, May 2012.

[86] V. Jagannath, M. Gligoric, D. Jin, and Q. Luo. IMUnit: Improved multithreaded unit testing. `http://mir.cs.illinois.edu/imunit`.

[87] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Roşu, and D. Marinov. Improved multithreaded unit testing. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 223–233, Szeged, Hungary, Sept. 2011.

[88] V. Jagannath, Q. Luo, and M. Kirn. ReEx: Re-execution based exploration of multithreaded Java programs. `http://mir.cs.illinois.edu/reex`.

[89] V. Jagannath, Q. Luo, and D. Marinov. Change-aware preemption prioritization. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 133–143, Toronto, Canada, July 2011.

[90] S. L. Johnsson. Data permutations and basic linear algebra computations on ensemble architectures. Technical Report YALEU/DCS/RR-367, Yale University, New Haven, Connecticut, February 1985.

[91] C. G. Jones, R. Liu, L. Meyerovich, K. Asanovic, and R. Bodik. Parallelizing the web browser. In *USENIX Workshop on Hot Topics in Parallelism (HotPar)*, 2009.

[92] R. K. Karmani, P. Madhusudan, and B. M. Moore. Thread contracts for safe parallelism. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 125–134, New York, NY, USA, 2011. ACM.

[93] F. Kjolstad, D. Dig, G. Acevedo, and M. Snir. Transformation for Class Immutability. In *International Conference on Software Engineering (ICSE)*, pages 61–70, May 2011.

[94] R. Komuravelli. Verification and Performance of the DeNovo Cache Coherence Protocol. Master's thesis, University of Illinois @ Urbana-Champaign, May 2010.

[95] D. B. Kubacki. Signed distance registration for depth image sequence. Master's thesis, University of Illinois, 2011.

[96] D. J. Kuck. Automatic program restructuring for high-speed computation. In *Conference on Analysing Problem Classes and Programming for Parallel Computing (CONPAR)*, pages 66–84, 1981.

[97] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 211–222, 2007.

[98] V. Le, J. Brandt, Z. Lin, L. D. Bourdev, and T. S. Huang. Interactive facial feature localization. In *European Conference on Computer Visions*, pages 679–692, 2012.

[99] V. Le and T. S. Huang. Iterative linearized optimization for efficient 3D face tracking and animation. Submitted for publication, 2013.

[100] V. Le, H. Tang, L. Cao, and T. S. Huang. Accurate and efficient reconstruction of 3D faces from stereo images. In *IEEE International Conference on Image Processing (ICIP)*, pages 4265–4268, 2010.

[101] V. Le, H. Tang, and T. S. Huang. Expression recognition from 3D dynamic faces using robust spatio-temporal shape features. In *IEEE International Conference on Automatic Face & Gesture Recognition*, pages 414–421, 2011.

[102] E. A. Lee. The Problem with Threads. *Computer*, 39(5):33–42, 2006.

[103] W. Lichtenstein and S. L. Johnsson. Block-cyclic dense linear algebra. *SIAM Jounrnal of Scientific Computing*, 14(6):1259–1288, Nov. 1993.

[104] D. J. Lin, V. Le, and T. S. Huang. Human-computer interaction. In T. B. Moeslund, editor, *Visual Analysis of Humans*, pages 493–510. Springer, 2011.

[105] Y. Lin and D. Dig. CHECK-THEN-ACT misuse of Java concurrent collections. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 164–173, March 2013.

[106] H. Mai, S. Tang, S. T. King, C. Cascaval, and P. Montesinos. A Case for Parallelizing Web Pages. In *USENIX Workshop on Hot Topics in Parallelism (HotPar)*, June 2012.

[107] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Computer Architecture News*, 33(4):92–99, 2005.

[108] M. Mayer. Google I/O '08 keynote, 2008. `http://www.youtube.com/watch?v=6x0cAzQ7PVs`.

[109] L. A. Meyerovich and R. Bodík. Fast and parallel webpage layout. In *Proceedings of the International Conference on World Wide Web (WWW)*, 2010.

[110] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In *International Symposium on Computer Architecture*, pages 289–300, June 2008.

[111] P. Montesinos, M. Hicks, W. Ahn, S. T. King, and J. Torrellas. Lessons Learned During the Development of the CapoOne Deterministic Multiprocessor Replay System. In *Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*, June 2009.

[112] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: A Software-Hardware Interface for Practical Deterministic Multiprocessor Replay. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 73–84, March 2009.

[113] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.

[114] A. Muzahid, N. Otsuki, and J. Torrellas. AtomTracker: A Comprehensive Approach to Atomic Region Inference and Violation Detection. In *International Symposium on Microarchitecture*, pages 287–297, December 2010.

[115] A. Muzahid, S. Qi, and J. Torrellas. Vulcan: Hardware Support for Detecting Sequential Consistency Violations Dynamically. In *International Symposium on Microarchitecture*, pages 363–375, December 2012.

[116] A. Muzahid, D. Suarez, S. Qi, and J. Torrellas. SigRace: Signature-Based Data Race Detection. In *International Symposium on Computer Architecture*, pages 337–348, June 2009.

[117] A. Nistor, Q. Luo, M. Pradel, T. R. Gross, and D. Marinov. Ballerina: Automatic generation and clustering of efficient random unit tests for multithreaded code. In *International Conference on Software Engineering (ICSE)*, pages 727–737, Zurich, Switzerland, June 2012.

[118] A. Nistor, D. Marinov, and J. Torrellas. Light64: Lightweight hardware support for race detection during systematic testing of parallel programs. In *International Symposium on Microarchitecture (MICRO)*, pages 541–552, New York City, NY, Dec. 2009.

[119] A. Nistor, D. Marinov, and J. Torrellas. InstantCheck: Checking the determinism of parallel programs using on-the-fly incremental hashing. In *International Symposium on Microarchitecture (MICRO)*, pages 251–262, Atlanta, GA, Dec. 2010.

[120] S. Okur and D. Dig. How do developers use parallel libraries? In *Symposium on the Foundations of Software Engineering (FSE)*, pages 54–64, November 2012.

[121] M. Olszewski et al. Kendo: Efficient deterministic multithreading in software. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 97–108, 2009.

[122] Opera Software. `http://www.opera.com/mobile`.

[123] D. Padua, editor. *Encyclopedia of Parallel Computing*. Springer, 2011.

[124] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W.-M. W. Hwu. FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs. In *7th IEEE Symposium on Application Specific Processors*, pages 35–42, Apr. 2009.

[125] A. Papakonstantinou, Y. Liang, J. Stratton, K. Gururaj, D. Chen, W.-M. W. Hwu, and J. Cong. Multilevel granularity parallelism synthesis on FPGAs. In *International Symposium on Field-Programmable Custom Computing Machines*, May 2011.

[126] M. Pharr and W. Mark. ispc: A SPMD compiler for high-performance CPU programming. In *Proceedings of the IEEE conference on Innovative and Parallel Computing*, 2012.

[127] G. Pokam, K. Danne, C. Pereira, R. Kassa, T. Kranich, S. Hu, J. Gottschlich, N. Honarmand, N. Dautenhahn, S. King, and J. Torrellas. QuickRec: Prototyping an Intel Architecture Extension for Record and Replay of Multithreaded Programs. In *International Symposium on Computer Architecture*, June 2013.

[128] G. Pokam, C. Pereira, K. Danne, L. Yang, S. King, and J. Torrellas. Hardware and Software Approaches for Deterministic Multiprocessor Replay of Concurrent Programs. *Intel Technology Journal, Issue on Addressing the Challenges of Tera-Scale Computing*, 13(4):20–41, December 2009.

[129] V. Prisacariu and I. Reid. fastHOG-a real-time GPU implementation of HOG. *University of Oxford Technical Report*, 2310(09), 2009.

[130] M. Prvulovic and J. Torrellas. ReEnact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *International Symposium on Computer Architecture*, pages 110–121, June 2003.

[131] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[132] S. Qi, N. Otsuki, L. Orosa, A. Muzahid, and J. Torrellas. Pacman: Tolerating Asymmetric Data Races with Unintrusive Hardware. In *International Symposium on High-Performance Computer Architecture*, February 2012.

[133] X. Qian, W. Ahn, and J. Torrellas. ScalableBulk: Scalable Cache Coherence for Atomic Blocks in a Lazy Environment. In *International Symposium on Microarchitecture*, pages 447–458, December 2010.

[134] X. Qian, B. Sahelices, and J. Torrellas. BulkSMT: Designing SMT Processors for Atomic-Block Execution. In *International Symposium on High-Performance Computer Architecture*, February 2012.

[135] X. Qian, B. Sahelices, J. Torrellas, and D. Qian. Scalable and Fast Commit of Atomic Blocks in a Lazy Multiprocessor Environment. In *International Symposium on Microarchitecture*, December 2013.

[136] X. Qian, B. Sahelices, J. Torrellas, and D. Qian. Volition: Scalable and Precise Sequential Consistency Violation Detection. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 535–548, March 2013.

[137] C. Radoi and D. Dig. Practical Static Race Detection for Java Parallel Loops. In *International Symposium on Software Testing and Analysis, (ISSTA)*, pages 178–190, July 2013.

[138] A. Raghavan, C. Blundell, and M. M. Martin. Token tenure: PATCHing token counting using directory-based cache coherence. In *International Symposium on Microarchitecture (MICRO)*, pages 47–58, 2008.

[139] H. Rubinstein and J. D. Rutledge. High order matrix computations on the UNIVAC. In *Proceedings of the 1952 ACM National Meeting (Pittsburgh)*, ACM '52, pages 181–186, New York, NY, USA, 1952. ACM.

[140] Samsung Inc. Samsung Introduces High Performance, Low Power Dual CORTEX - A9 Application Processor for Mobile Devices, September 2010. `http://www.samsung.com/global/business/semiconductor/newsView.do?news_id=1195`.

[141] M. Satyanarayanan. Fundamental challenges in mobile computing. In *Fifteenth ACM Symposium on Principles of Distributed Computing*, pages 1–7, 1996.

[142] E. Schurman and J. Brutlag. Performance related changes and their user impact. `http://velocityconf.com/velocity2009`.

[143] S. Seo, G. Jo, and J. Lee. Performance characterization of the NAS parallel benchmarks in OpenCL. In *International Symposium on Workload Characterization (IISWC)*, pages 137–148, nov 2011.

[144] D. Shasha and M. Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Transactions on Programming Languages and Systems*, pages 282–312, April 1988.

[145] R. M. Sheppard, M. Kamali, R. Rivas, M. Tamai, Z. Yang, W. Wu, and K. Nahrstedt. Advancing interactive collaborative mediums through tele-immersive dance (TED): a symbiotic creativity and design environment for art and computer science. *ACM international conference on Multimedia*, pages 579–588, Oct. 2008.

[146] S. Shi, M. Kamali, K. Nahrstedt, J. C. Hart, and R. H. Campbell. A high-quality low-delay remote rendering system for 3D video. *ACM international conference on Multimedia*, pages 601–610, Oct. 2010.

[147] SkyFire Labs, Inc. `http://www.skyfire.com`.

[148] R. Smolinski. Eliminating On-Chip Traffic Waste: Are We There Yet? Master's thesis, University of Illinois @ Urbana-Champaign, May 2013.

[149] F. Sorrentino, A. Farzan, and P. Madhusudan. PENELOPE: Weaving threads to expose atomicity violations. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 37–46, New York, NY, USA, 2010. ACM.

[150] F. Sorrentino, A. Farzan, P. Madhusudan, and N. Razavi. PENELOPE: A testing tool for concurrent software. `http://www.cs.illinois.edu/~madhu/penelope/`.

[151] J. A. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu, and W.-m. W. Hwu. Efficient compilation of fine-grained SPMD-threaded programs for multicore CPUs. In *international symposium on Code Generation and Optimization (CGO)*, pages 111–119. ACM, 2010.

[152] J. A. Stratton, H.-S. Kim, T. B. Jablin, and W.-M. W. Hwu. Performance portability in accelerated parallel kernels. Technical Report IMPACT-13-01, University of Illinois at Urbana-Champaign, Urbana, May 2013.

[153] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L. Chang, G. Liu, and W.-M. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, Urbana, Mar. 2012.

[154] J. A. Stratton, S. S. Stone, and W.-m. W. Hwu. MCUDA: An effective implementation of CUDA kernels for multi-core CPUs. In *Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 16–30, July 2008.

[155] H. Sung, R. Komuravelli, and S. V. Adve. DeNovoND: Efficient hardware support for disciplined non-determinism. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 13–26, October 2013.

[156] U. Tariq, K.-H. Lin, Z. Li, X. Zhou, Z. Wang, V. Le, T. S. Huang, X. Lv, and T. X. Han. Recognizing emotions from an ensemble of features. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 42(4):1017–1026, 2012.

[157] U. Tariq, J. Yang, and T. S. Huang. Maximum margin GMM learning for facial expression recognition. In *IEEE International Conference on Automatic Face & Gesture Recognition*, pages 1–6, 2013.

[158] Top500.org. Top500 list, June 2013. `http://www.top500.org/list/2013/06/`.

[159] J. Torrellas, L. Ceze, J. Tuck, C. Cascaval, P. Montesinos, W. Ahn, and M. Prvulovic. The Bulk Multicore Architecture for Improved Programmability. *Communications of the ACM*, 52(12):58–65, 2009.

[160] J. Tuck, W. Ahn, L. Ceze, and J. Torrellas. SoftSig: Software-Exposed Hardware Signatures for Code Analysis and Optimization. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2008.

[161] J. Tuck, W. Ahn, L. Ceze, and J. Torrellas. SoftSig: Software-Exposed Hardware Signatures for Code Analysis and Optimization. In *IEEE Micro Special Issue: Micro's Top Picks from Computer Architecture Conferences*, January-February 2009.

[162] M. Vakilian, D. Dig, R. Bocchino, J. Overbey, V. Adve, and R. Johnson. Inferring method effect summaries for nested heap regions. In *International Conference on Automated Software Engineering (ASE)*, pages 421–432, Novermber 2009.

[163] B. Virlet, X. Zhou, J. P. Giacalone, B. Kuhn, M. J. Garzarán, and D. Padua. Scheduling of Stream-Based Real-Time Applications for Heterogeneous Systems. In *Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, April 2011.

[164] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. Why are web browsers slow on smartphones? In *Workshop on Mobile Computing Systems and Applications*, pages 91–96. ACM, 2011.

[165] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, Jan. 2001.

[166] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *International Conference for High-performance Computing, Networking, Storage and Analysis (SC)*, pages 38:1–38:12, New York, NY, USA, 2007. ACM.

[167] W. Wu, A. Arefin, G. Kurillo, P. Agarwal, K. Nahrstedt, and R. Bajcsy. Color-plus-depth level-of-detail in 3D tele-immersive video: A psychophysical approach. In *ACM Multimedia Conference*, pages 13–22, 2011.

[168] J. Xiong, J. Johnson, R. Johnson, and D. Padua. SPL: A language and compiler for DSP algorithms. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 298–308, New York, NY, USA, 2001. ACM.

[169] C.-Y. Yang, J.-J. Chen, T.-W. Kuo, and L. Thiele. An approximation scheme for energy-efficient scheduling of real-time tasks in heterogeneous multiprocessor systems. In *Conference of Design, Automation, and Test in Europe (DATE)*, pages 694–699, 2009.

[170] Z. Yang. Every millisecond counts, 2009. `http://www.facebook.com/note.php?note_id=122869103919`.

[171] Y. Yu and V. K. Prasanna. Power-aware resource allocation for independent tasks in heterogeneous real-time systems. In *International Conference on Parallel and Distributed Systems (ICPADS)*, pages 341–348. IEEE Computer Society, 2002.

[172] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos. A tagless coherence directory. In *International Symposium on Microarchitecture (MICRO)*, pages 423–434. IEEE, 2009.

[173] L. Zhang and R. Nevatia. Efficient scan-window based object detection using GPGPU. In *Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 1–7, 2008.

[174] X. Zhou. *Tiling Optimizations for Stencil Computations*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2013.

[175] X. Zhou, J.-P. Giacalone, M. J. Garzarán, R. H. Kuhn, Y. Ni, and D. Padua. Hierarchical overlapped tiling. In *International Symposium on Code Generation and Optimization (CGO)*, pages 207–218, New York, NY, USA, 2012. ACM.

Chapter 16

Key Papers

High Performance Graphics (2010)
M. Doggett, S. Laine, and W. Hunt (Editors)

Parallel SAH k-D Tree Construction

Byn Choi, Rakesh Komuravelli, Victor Lu, Hyojin Sung, Robert L. Bocchino, Sarita V. Adve, and John C. Hart

University of Illinois at Urbana-Champaign
sds@cs.illinois.edu

Abstract

The k-D tree is a well-studied acceleration data structure for ray tracing. It is used to organize primitives in a scene to allow efficient execution of intersection operations between rays and the primitives. The highest quality k-D tree can be obtained using greedy cost optimization based on a surface area heuristc (SAH). While the high quality enables very fast ray tracing times, a key drawback is that the k-D tree construction time remains prohibitively expensive. This cost is unreasonable for rendering dynamic scenes for future visual computing applications on emerging multicore systems. Much work has therefore been focused on faster parallel k-D tree construction performance at the expense of approximating or ignoring SAH computation, which produces k-D trees that degrade rendering time. In this paper, we present two new parallel algorithms for building precise SAH-optimized k-D trees, with different tradeoffs between the total work done and parallel scalability. The algorithms achieve up to 8× *speedup on 32 cores, without degrading tree quality and rendering time, yielding the best reported speedups so far for precise-SAH k-D tree construction.*

1. Introduction

We foresee an evolution of visual experiences into shared online visual simulations whose user-generated content (including self-scanned avatars) changes dynamically and unpredictably. Unlike modern videogames, which achieve lush visual effects through heavy precomputation of predefined content, the real-time rendering, meshing, and simulation of dynamic content will require the rapid construction and update of hierarchical spatial data structures. For example, these spatial data structures are well known rendering accelerators for both ray tracing [WHG84] and rasterization [GKM93], and form integral components of recent parallel real time ray tracers [WSS05, CHCH06, SCS*08, LP08, GDS*08]. However, existing parallel algorithms designed to rapidly build dynamic spatial hierarchies will soon face a serious roadblock as processor parallelism continues to grow.

The previous work summarized in Sec. 2 and the emergent pattern analyzed in Sec. 3 reveal that parallel hierarchy construction algorithms load balance well when the frontier of hierarchy nodes needing processing exceed the number of parallel processors, but struggle with the initial stages of construction when the hierarchy contains too few nodes. Some parallel approaches suffer reduced throughput at these initial levels [Ben06, PGSS06, HMS06], whereas others use alternative subdivision heuristics that can reduce hierarchy quality [SSK07,ZHWG08,LGS*09]. Fig. 1 shows that as processor parallelism continues to scale up, the number of initial steps in parallel hierarchy construction grows, and current subdivision heuristic sacrifices made to maintain throughput cause increasing degradation in tree quality and ultimately rendering rates.

This paper presents two new parallel algorithms for improving throughput when constructing these initial upper levels of a k-D tree. The first algorithm, "nested," is a depth-first task parallelization of sequential k-D tree construction that nests geometry-level parallelism within the node-level parallelism for these upper level nodes. The second algorithm, "in-place," builds the upper nodes of the hierarchy breadth-first, one level at a time, storing in each triangle the node(s) it belongs to at that level. This reduces geometry data movement and allows an entire level's nodes to be computed across a single data parallel geometry stream.

These new algorithms regain throughput without sacrificing spatial hierarchy quality, as measured by rendering performance gains. They compute a precise surface area heuristic (SAH) that subdivides geometry into regions of small surface area that contain many triangles [GS87, MB90]. Spatial hierarchies formed by subdividing at the spatial me-

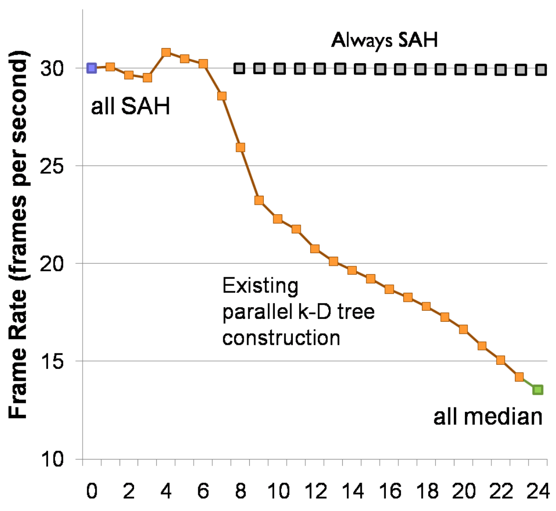


Figure 1: *Degradation in hierarchy quality using spatial median vs. precise-SAH to select splitting planes for the upper level nodes. The vertical axis indicates the rendering rate in normalized frames per second for ray tracing the fairy scene on the Dunnington machine described in Sec. 7 (the performance of the best configuration is normalized to a target of 30 fps). The horizontal axis indicates the depth at which current parallel kD-tree construction algorithms switch from using a spatial median to using SAH. This switch occurs when the depth approximately equals* $\log_2$ *number of processors. As the number of processors continue to double biannually, the hierarchies generated by existing parallel algorithms eventually degrades rendering performance, whereas the rendering rate remains constant for our (fully SAH) parallel k-D trees.*

dian (e.g., the octree) [ZHWG08] or at the object median (e.g., into children of approximately equal numbers of primitives) [SSK07] can be computed faster than SAH but the resulting hierarchies render slower than SAH hierarchies, as shown in Fig. 1 for the spatial median.

Hierarchy quality can be further justified by the relationship between hierarchy rendering time and construction time. Recent renderers that focus on real-time direct ray tracing of dynamic content currently experience about a 1:1 rendering-to-construction speed ratio. For these, some approaches justify a degraded hierarchy quality that increases the rendering time by a corresponding decrease in hierarchy construction time, and given a few processors, the upper-level nodes may not even incur a quality degradation. Such a relationship might continue as the triangle count grows but only to a ceiling level on the order of one REYES-micropolygon triangle per pixel since frame rates and display resolutions remain fairly constant. As processor parallelism nevertheless continues to grow, we will see increased global illumination Monte-Carlo effects and hundreds of rays per pixel which would cause the rendering-to-construction speed ratio to grow to 100:1 such that even a 1% degradation in rendering rate could not be tolerated by a hierarchy construction acceleration.

Our implementation is designed to measure the efficiency and throughput of the *parallelism* of our approach, as opposed to the raw *performance* of SAH k-D tree construction and rendering. For example, we compute SAH directly at the endpoints of triangle extents in each direction, and do not implement "binned SAH" approximations or "split clipping" triangle subdivision which would affect raw performance but their impact on scalability results from less work for binning [WBS07] and similar but greater dynamic growth in per-level triangles for split clipping. We believe our parallel construction algorithms are general enough to permit both enhancements in a production enviroment. We similarly focus on the construction of k-D trees, but believe our parallel algorithms can also be adapted to bounding volume hierarchies (BVHs). BVHs can be constructed and maintained more efficiently [WBS07, WIP08, LGS*09] but k-D trees better accelerate ray tracing [Hav00] which would make them the preferred choice for high rendering-to-construction speed ratio applications.

Sec. 7 examines the results of our two approaches on a 32-core shared-memory CPU platform for five input models, indicating scalability of these difficult upper levels up to depth 8. For these configurations, the algorithms achieve speedups of up to 8X, with in-place outperforming nested for two input models and vice versa for the other three. A deeper analysis of the scalability of the two algorithms reveals that while nested performs less work overall, in-place has better parallel scalability and is likely a better choice for future machines with larger core counts. To our knowledge, these results represent the first multicore speedups on the upper levels of k-D tree construction using precise SAH, and the best parallel approach for working with these levels in general.

2. Related Work

Wald and Havran [WH06] describe an optimal sequential $O(n\log n)$ SAH k-D tree construction algorithm that initially sorts the geometry bounding box extents in the three coordinate axes, peforms linear-time sorted-order coordinate sweeps to compute the SAH to find the best partitioning plane, and maintains this sorted order as the bounding boxes and their constituent geometries are moved and subdivided. We describe this algorithm in more detail in Section 4 and use it as our baseline state-of-the-art sequential algorithm. Our contribution is to develop a parallel approach that produces the same k-D tree as this sequential algorithm but at a much higher level of performance.

Some have accelerated SAH computation by approximation, replacing the initial $O(n\log n)$ sort with an $O(n)$ binned radix sort along each axis, and interpolating the SAH measured only between triangle bins [PGSS06, HMS06, SSK07] for both sequential and parallel acceleration. Even with a binned approximate sort, the k-D tree construction cost nevertheless remains $O(n \log n)$ since all n of the triangles are processed for each of the $\log n$ levels.

Many have worked on parallel SAH k-D tree construction. Several versions use a single thread to create the top levels of the tree until each subtree can be assigned to each core in a 2- or 4-core system [Ben06, PGSS06, HMS06], limiting 4-core speedup to only 2.5×.

Shevtsov et al. [SSK07] also implemented a 4-core parallel SAH k-D tree builder, but used a parallel triangle-count median instead of SAH to find splitting planes at the top levels of the tree, which degraded k-D tree quality by 15%. They did not report a construction time speedup, but they did report a 4-core speedup of 3.9 for a construction combined with rendering, which includes millions of k-D tree traversals. This algorithm was also used for Larrabee's real-time ray tracer [SCS*08], which reports the real-time construction of a 25MB k-D tree of a 234K triangle scene rendered with 4M rays and similar scalability for total time-to-render.

Kun Zhou et al. [ZHWG08] built k-D trees on the GPU, using a data-parallel spatial median algorithm for the upper levels of the tree, to a level where each node's subtree could be generated by each of the GPU's streaming processors. Their 128-core GPU version achieved speedups of $6 \sim 15\times$ over a single-core CPU, and of $3 \sim 6\times$ over 16-cores of the GPU for scenes ranging from 11K to 252K triangles. Their speedups improved for larger models, but their SAH and median approximations degraded the k-D trees and corresponding rendering times of these larger models, by as much as 10% for scenes over 100K triangles. Like Zhou et al.'s GPU algorithm, both our nested and in-place algorithms use scan primitives for data parallelism, but our new algorithms compute SAH precisely at all levels and propagate information differently from level to level.

Several authors have also examined the construction of dynamic bounding volume hierarchies. Wald et al. [WBS07] explore BVH maintenance for dynamic scenes for real-time rendering, showing them to be faster to construct but slower to render than similar k-D tree approaches. Wald [Wal07] describes a binned SAH BVH approach using "horizontal" and "vertical" parallelism, which resembles the node and geometry parallelism described in the next section, and reports CPU bandwidth limitations (as does our results section).

Lauterbach et al. [LGS*09] constructed a dynamic BVH on the GPU, using a breadth-first approximated SAH computation using GPU work queues optimized for SIMD processing by compaction. Similar to previous k-D tree approaches, they observe low utilization for the upper-level nodes and instead sort along a space filling curve to organize the upper levels into a linearized grid-like structure that serves effectively as a flattened spatial median tree.

3. Parallel Patterns for k-D Trees

Software patterns emerge from recurring program designs [GHJV95], and have evolved to include parallel programming [MSM04]. Fig. 2 illustrates patterns for parallel k-D tree construction that emerge from the analysis of previous work.

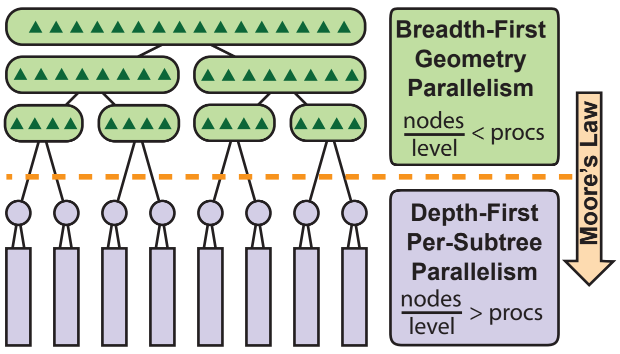

Breadth-First Geometry Parallelism
$\frac{\text{nodes}}{\text{level}} <$ procs
Depth-First Per-Subtree Parallelism
$\frac{\text{nodes}}{\text{level}} >$ procs
Moore's Law

Figure 2: *Parallel k-D Tree Patterns. Each level of the upper (green) portion of the tree has fewer nodes than cores, so multiple cores must cooperate on node creation leading to a breadth-first stream process that organizes all of the triangles into the current level's nodes. When the number of nodes at a level meets or exceeds the number of cores, then each node's subtree can be processed per core independently. The dashed dividing line (orange) where the number of nodes equals the number of processors descends one level every 1.5 to 2 years, indicating that the upper (green) pattern will eventually dominate k-D tree construction.*

The initial phases of a breadth-first top-down hierarchy construction consist of cases where large amounts of geometry need to be analyzed and divided among a few nodes. These cases suggest an approach where scene geometry is streamed across any number of processors whose goal is to analyze the geometry to determine the best partition, and categorize the geometry based on that partition. Previous serial and parallel versions of this streaming approach to SAH computation [WH06, PGSS06, HMS06, SSK07] all share this same pattern at the top of their hierarchies (as do breadth-first GPU constructions based on median finding [ZHWG08, GHGH08]), which can be efficiently parallelized by the techniques discussed in this paper.

Once the hierarchy has descended to a level whose number of nodes exceeds the number of cores or threads, then a node-parallel construction with depth-first traversal per node becomes appropriate. Here each subtree is assigned to a separate thread and is computed independently. Even on the GPU this parallelism is independent in that it needs no interprocessor communication, though the processes would run in SIMD lock step. If the subtrees vary in size, then load balancing via task over-decomposition/work stealing or other methods can be employed.

The most recent parallel SAH k-D tree construction algorithms ignore SAH in the top half of the tree, instead using the triangle count median [SSK07] or the spatial median [ZHWG08]. We see from Figure 1 that using median splitting planes for upper levels in a k-D tree degrades tree quality and rendering times significantly. In contrast, all the algorithms described in the rest of this paper compute precise SAH at all levels of the tree for high tree quality and rendering performance.

4. State-of-the-Art Sequential Algorithm

We begin by summarizing the best known sequential algorithm for precise SAH k-D tree construction [WH06]. Algorithm 1 shows that it finds the best SAH splitting plane for each node by an axis-aligned sweep across each of the three axes. It takes as input three pre-sorted lists (one per axis) of "events" (edges of the axis-aligned bounding box, one pair per triangle), and an axis-aligned bounding box representing the space covered by the node. The bounding box of the root node consists of the per-coordinate minima and maxima of the triangle vertices. For a descendant node, this bounding box is refined by intersection with the node's ancestry of splitting planes.

This single-thread sequential version builds a k-D tree in depth-first order, as revealed by the tail recursion. It achieves its $O(n \log n)$ efficiency due to its three axial sweeps through $E[axis]$ that compute SAH for each of the $O(n)$ events for each of the $O(\log n)$ levels of the k-D tree.

The SAH need only be evaluated at each *event* where the sweep encounters a new triangle or passes the end of a triangle [Hav00, (p. 57)]. Each event contains three members: its 1-D *position* along the axis, its type (START or END), and a reference to the triangle generating the event.

The three event lists $E[x], E[y], E[z]$ are each provided in position sorted order, and when two events share the same positions, in type order, where START < END. These three sorts are a pre-process and also require $O(n \log n)$ time.

The algorithm consists of three phases. The first phase, FINDBESTPLANE, determines the axis, position, and corresponding event index of the splitting plane yielding the lowest SAH cost over the events in E. FINDBESTPLANE evaluates SAH at each event position (redundantly computing SAH even for coplanar events). The SAH evaluation at each event utilizes the triangle counts n_L, n_R to the left and right of the current splitting plane, which are maintained and updated as the sweep passes each event in each axis' sorted list. The SAH computation utilizes constants C_I, the cost of ray intersection, and C_T, the cost of traversal. Triangles that intersect the splitting plane are added to both sides. When the splitting plane sweep passes an END event, one less triangle

Algorithm 1: Sequential k-D Tree Construction

```
BuildTree($E_{x,y,z}$, $\square$) returns $Node$
/* $E[axis]$ - sorted events, $\square$ - Extent                    */
$C \leftarrow \infty$ ;                                      // SAH cost
foreach $axis' \in \{x,y,z\}$ do
    FindBestPlane($E[axis']$, $\square$) $\longrightarrow (pos', C', i')$
    if $C' < C$ then $(C, pos, axis, i_{split}) \longleftarrow (C', pos', axis', i')$
if $C > C_I \times |E[axis]|$ then return Leaf Node
ClassifyTriangles($E[axis]$, $i_{split}$)
FilterGeom($E, pos, axis$) $\longrightarrow (E_L, E_R)$
Subdivide $\square$ into $\square_L, \square_R$ at $pos$ along $axis$.
$Node_L \longleftarrow$ BuildTree($E_L, \square_L$)
$Node_R \longleftarrow$ BuildTree($E_R, \square_R$)
return $Node(pos, axis, Node_L, Node_R)$
```

```
FindBestPlane($E[axis]$, $\square$) returns $(pos', C', i')$
$C' \longleftarrow \infty, S \longleftarrow$ surface area of $\square$, $n_L \longleftarrow 0, n_R \longleftarrow \frac{|E[axis]|}{2}$
foreach $e_i \in E[axis]$ do
    if $e_i$.type is END then decr $n_R$
    let $S_L, S_R$ be surface areas of $\square$ split at $e_i$.pos
    $C \longleftarrow C_T + C_I(n_L \frac{S_L}{S} + n_R \frac{S_R}{S})$ ;            // SAH
    if $C < C'$ then $(pos', C', i') \longleftarrow (e_i.\text{pos}, C, i)$
    if $e_i$.type is START then incr $n_L$
return $(pos', C', i')$
```

```
ClassifyTriangles($E[axis]$, $i_{split}$)
/* Lbit, Rbit cleared for every $\triangle$ by prev. sweep          */
for $i \longleftarrow 0 \ldots i_{split}$ do
    if $e_i$.type is START then set $E[axis][i].\triangle$.Lbit
for $i \longleftarrow i_{split} \ldots |E[axis]| - 1$ do
    if $e_i$.type is END then set $E[axis][i].\triangle$.Rbit
```

```
FilterGeom($E$) returns $(E_L, E_R)$
foreach $axis \in \{x,y,z\}$ do
    foreach $e \in E[axis]$ do
        if $e.\triangle$.Lbit then $E_L[axis]$.append($e$)
        if $e.\triangle$.Rbit then $E_R[axis]$.append($e$)
return $(E_L, E_R)$ // $E_L, E_R$ sorted
```

is on its right side, and when it passes a START event, one more triangle is on its left side.

The next two phases divide the event lists into (not necessarily disjoint) subsets left and right of the splitting plane. CLASSIFYTRIANGLES sweeps over the triangles, marking them as left or right, or both if they intersect the splitting plane. FILTERGEOMETRY divides the event lists into two portions, duplicating the splitting-plane straddling events, and maintaining the sorted order of the events for each axis.

5. Nested Parallel Algorithm

As Figure 2 illustrates, an obvious source of parallelism comes from independent nodes in the tree. Given two children of a node, the sub-trees under each child can be built indepedently (node-level parallelism). The problem with solely pursuing this approach is the lack of parallelism at the top levels of the tree. Unfortunately, at the top levels of the tree, each node has a larger number of events than at the bottom; the lack of node-level parallelism at these levels becomes a severe bottleneck. To alleviate this problem, we exploit a second source of parallelism: we parallelize the work on the large number of events (triangles) within a given node, referred to as geometry-level parallelism. Thus, our parallel algorithm nests two levels of parallelism. This is similar to the nested parallelism popularized by the NESL programming language [BHC*93, Ble95].

Expressing node-level parallelism is relatively straightforward in lightweight task programming environments such as Cilk [BJK*95] or Intel's Threading Building Blocks (TBB) [Int09] that allow recursive creation of light-weight tasks that are load balanced through a work stealing task scheduler. (We use TBB for our code.)

Within the computation of each node, we again use light-weight tasks to parallelize each of the major functions in the sequential computation (Algorithm 1) – FINDBESTPLANE, CLASSIFYTRIANGLES, and FILTERGEOM – as follows.

5.1. FINDBESTPLANE

Figure 3 depicts how FINDBESTPLANE works. Given an array of events (the top row of boxes, S=START E=END), the sequential "1 thread" box shows how FINDBESTPLANE in Algorithm 1 proceeds. The left-to-right sorted axis sweep maintains a running count of N_L and N_R, immediately incrementing N_L for each START event, and decrementing the next N_R for each END event. Recall that some triangles straddle the splitting plane and are counted in both N_L and N_R, and this post-decrement processing of END events accounts for such triangles. The remaining values needed for SAH evaluation are constants and $O(1)$ surface area computations. Hence as each event is processed, the current N_L, N_R counts generate the current SAH, which is compared against the previous minimal SAH to determine the minimal SAH splitting plane at the end of the sweep.

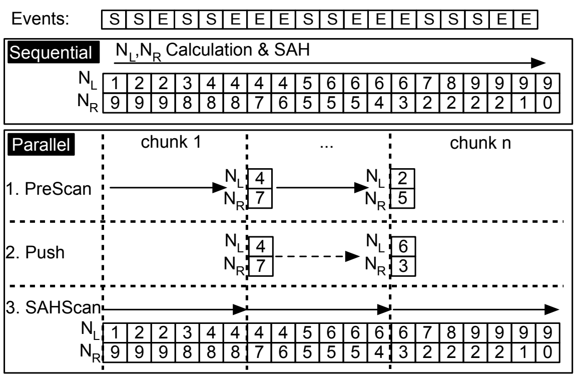

Events:
S S E S S E E E S S E E E S S S E E
Sequential
N_L,N_R Calculation & SAH
N_L 1 2 2 3 4 4 4 4 5 6 6 6 6 7 8 9 9 9 9
N_R 9 9 9 8 8 8 7 6 5 5 5 4 3 2 2 2 2 1 0
Parallel
chunk 1
...
chunk n
1. PreScan
N_L 4
N_R 7
N_L 2
N_R 5
2. Push
N_L 4
N_R 7
N_L 6
N_R 3
3. SAHScan
N_L 1 2 2 3 4 4 4 4 5 6 6 6 6 7 8 9 9 9 9
N_R 9 9 9 8 8 8 7 6 5 5 5 4 3 2 2 2 2 1 0

Figure 3: *Parallel SAH.*

We parallelize FINDBESTPLANE using a parallel prefix style operation [HS86], with three sub-phases: `PreScan`, `Push`, and `SAHScan` as illustrated in the lower (parallel) box of Fig. 3. We first decompose the event list into n contiguous chunks, allocating one chunk per task. For the `PreScan` phase, each of $n-1$ tasks counts the number of START and END edges in its corresponding chunk. (The last chunk need not be PreScanned.) Next, a single thread executes the `Push` phase, adding the total N_L, N_R of previous chunks to the current chunk totals, yielding correct N_L, N_R values at the beginning of each chunk. (In a typical parallel prefix, this is also done in parallel, but we did not find that necessary for the relatively few cores in our system.) For the final `SAHScan` phase, each of the n tasks processes its corresponding chunk, propagating its starting N_L, N_R values through the chunk and computing the minimum SAH value for its chunk. A final (sequential) reduction yields the minimum SAH across all n chunks.

5.2. CLASSIFYTRIANGLES

The CLASSIFYTRIANGLES phase classifies whether a triangle will fall into the left and/or right child of the current node, depending on its position with respect to the splitting plane. We can parallelize this phase by sweeping through the event array corresponding to the splitting plane axis, finding the corresponding triangle index for the event, and updating the right or left membership bit of the triangle. This is conceptually a parallelizable computation across the events; however, we found that it incurs significant false-sharing making it not profitable to parallelize. Our experiments reported in Sec. 7, therefore, do not parallelize this phase.

5.3. FILTERGEOM

The FILTERGEOM phase divides (for each of x, y, and z axes) one big array of events into two smaller arrays, duplicating some entries corresponding to plane straddling triangles, while preserving the sorted ordering from the original. On the face of it, this splitting with potential duplication of geometries into two sorted arrays of unknown length may appear to have limited parallelism (the length of the new arrays is currently unknown because some triangles may need to be duplicated). However, we can use the same observations as for parallelizing the FINDBESTPLANE phase here. We map the above to a parallel prefix style computation again, performing a parallel `PreScan`, a short sequential `Push`, and a parallel `FilterScan`. The parallel `PreScan` determines how many triangles in its chunk need to go to the left and right arrays. The `Push` accumulates all of the per-chunk information so that each chunk now knows how many triangles to its left will enter each of the two new arrays. This gives each chunk the correct starting location in the new arrays. All chunks can thus proceed in parallel to update their own independent portions of the two new arrays, creating a fully sorted pair of arrays in parallel in the `FilterScan` phase. (Note that the information about whether an event goes to the left or right new array is obtained from the *Lbit* and *Rbit* flags of the triangle corresponding to the event, as set in the CLASSIFYTRIANGLES phase.)

6. In-Place Parallel Algorithm

One major drawback of the state-of-the-art sequential Alg. 1 in Sec. 4 is that the division and distribution of triangle and event lists from a node to its two children require a lot of data movement. Worse yet, there exists a slight growth in the aggregate working set size due to triangles intersecting the splitting plane, which is proportional to the square root of the number of triangles in the node [WH06]. Since the parallel version in Sec. 5 essentially follows the structure of the sequential algorithm, it inherits these problems as well.

In an attempt to eliminate the cost of this data movement, we developed a new "in-place" algorithm. This algorithm is based on the insight that, although each node can contain many triangles, each triangle belongs to a small number of nodes at any given time during the construction of the top-levels of the tree. Our experiments revealed that triangles usually belong to a single node (most don't intersect splitting planes) and even in the worst case they belong to no more than eleven nodes for the tree depth of eight for the inputs used in this paper.

Our "in-place" algorithm overcomes the expense of data movement by letting the triangles keep track of which of a level's nodes they belong to. This is in contrast to the previous approach that required nodes to keep track of which triangles they contained. When FILTERGEOM processes each level, it moves triangle and event data from the parent node into its two child nodes. In "in-place," we instead update the "membership" of each triangle.

Zhou et al. [ZHWG08] employ an analogous strategy of keeping events (split candidates) in-place during a small node precise SAH construction phase, but the strategy relies on a bit mask representation of triangle sets which is only feasible for small numbers of triangles and is hence only viable for lower level construction. In contrast, our approach keeps events in-place throughout top level construction as well.

This new approach has the following implications:

1. The triangle data structure and the axial event elements are not moved in memory. Instead, the triangle's "nodes" membership field is updated.
2. A post-process at the end of k-D tree construction is necessary to produce the output in a desired format, which involves scanning the entire array of triangles and collecting them into appropriate node containers.
3. Since event elements remain fixed in memory, no resorting of any form is necessary at any stage.
4. Triangles can be more easily organized in a struct-of-arrays instead of an array-of-structs for a more cache-friendly memory access pattern. This particular optimization is not as easily applicable in the previous nested parallel algorithm due to the FILTERGEOM phase that mutates the array structure. The ordering must be preserved at the object granularity, which is difficult to achieve with the array of objects in struct-of-arrays format.
5. The in-place algorithm operates one level of the tree at a time, with sweeps on the entire array (instead of chopping the array into increasingly smaller pieces). This type of access pattern incurs worse cache behavior but is arguably more amenable to SIMD instructions and GPUs – this tradeoff remains to be studied since we do not focus on SIMD or GPUs in this paper.

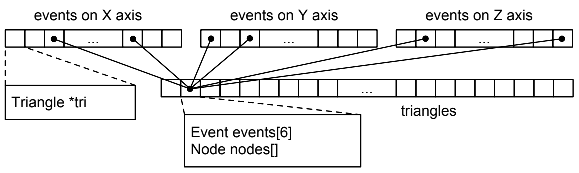


Figure 4: *Data structures used in the in-place algorithm.*

6.1. Algorithm

The algorithm operates on the data structure shown in Fig. 4. The three axial event arrays hold the events in position sorted order, and each event includes a pointer to the triangle that generated it. Each element of the triangle array contains pointers to the six events it generates, and a list of the current level's nodes to which it belongs.

One of the major differences between the nested-parallel approach in Sec. 5 and the in-place approach is that the latter is constructed in a breadth-first search manner, which makes more geometry parallelism available to tasks. The in-place approach processes the entire triangle stream and updates all the nodes of the current level, whereas the nested-parallel version switches between geometry processing and node construction phases. Therefore, it is a good choice for the geometry-parallel upper levels of k-D tree construction, and it should terminate when the number of nodes at the current level meets or exceeds the number of processing cores. From that point, subtrees can be constructed independently in parallel by each processor.

Alg. 2 outlines this approach. Current level's nodes are called "live," and each of them are considered for an SAH-guided split. It consists of four main phases:

FINDBESTPLANE Expanded from the FINDBESTPLANE phase in Sec. 5, this phase considers all live nodes in parallel instead of just one node. This phase outputs a splitting plane for each live node that is not to become a leaf.

NEWGEN This phase extends the tree by one level, creating two child nodes for each split live node. The decision to extend the tree is made dynamically since the SAH-based k-D trees are usually sparse.

ClassifyTriangles This phase updates each triangle's node list using the next generation nodes created in NewGen.

Fill This phase occurs once at the very end of the tree-building process, outside the main loop. It is essentially a glue phase that translates the generated tree into the format of the trees generated by the sequential and the nested parallel algorithms.

Algorithm 2: Outline of the in-place algorithm.

```
Data: List of triangles (T) in the scene
Result: Pointer to the root of the constructed kd-tree
live ← {root ← new kdTreeNode() };
foreach △ ∈ $T$ do
    △.nodes ← {root};
while nodes at current level < cores do
    // FindBestPlane phase (84.84% of time)
    foreach $e \in E[x] \cup E[y] \cup E[z]$ do
        foreach node ∈ $e$.△.nodes do
            SAH ← CalculateSAH($e$, node.extent);
            if SAH is better than node.bestSAH then
                node.bestEdge ← e ;
                node.bestSAH ← SAH ;
    // Newgen phase (0.04% of time)
    nextLive ← {};
    foreach node ∈ live do
        if node.bestEdge found then
            nextLive += (node.left ← new kdTreeNode()) ;
            nextLive += (node.right ← new kdTreeNode()) ;
    // ClassifyTriangles phase (14.60% of time)
    foreach △ ∈ $T$ do
        oldNodes ← △.nodes ;
        clear △.nodes ;
        foreach node ∈ oldNodes do
            if no node.bestEdge found then
                // leaf node
                insert △ in node.triangles ;
            else
                if △ left of node.bestEdge then
                    insert node.left in △.nodes ;
                if △ right of node.bestEdge then
                    insert node.right in △.nodes ;
    live ← nextLive;
// Fill phase (0.52% of time)
foreach △ ∈ $T$ do
    foreach node in △.nodes do
        insert △ in node.triangles ;
return root
```

6.2. Parallelization

As shown in Alg. 2, FindBestPlane and ClassifyTriangles phases together account for virtually all the build time. Therefore, we focused on parallelizing these two phases.

As in the nested-parallel algorithm, we employ the parallel prefix operators to compute FindBestPlane. However, instead of a single pair of n_L, n_R, we maintain a list of pairs, one for each live node. In the nested algorithm, the goal of FindBestPlane was to find one best plane that splits the given node. However, in the in-place algorithm, the end goal is to find a best plane for each live node.

ClassifyTriangles phase is fully-parallel, since all of the information needed to update the node membership of each triangle object is found locally. Therefore, each thread can operate on a subsection of the triangle array in isolation.

7. Results

Methodology and metrics. We demonstrate the algorithms using the five test models shown in Fig. 5 for triangle counts varying from 60K to 1M. We measured the performance of the geometry parallel construction of the top eight levels of the tree, which on completion yields 256 subtree tasks that can be processed independently in parallel.

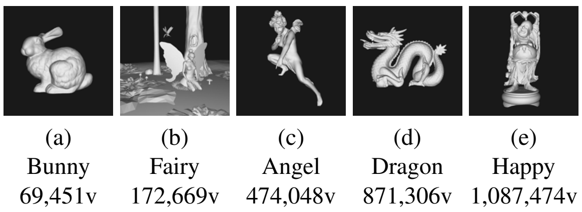


Figure 5: *Test models with triangle counts. Bunny, Dragon, and Happy courtesy of Stanford U., Angel courtesy of Georgia Tech, and Fairy courtesy of U. Utah.*

We performed experiments on the two machines shown in Table 1, which we refer to by Intel's product codename "Beckton" and "Dunnington." Both machines run CentOS 5.4. Beckton represents the state of the art, while results obtained using Dunnington are used to show how the algorithms exploit increased resources on new generations of machines (e.g., larger caches and memory bandwidth). We did not utilize Beckton's hyperthreading capability as we experimentally concluded that there were no significant advantages. We compiled the executables with GCC 4.1.2 with `-O3 -funroll-loops -fomit-frame-pointer` flags and linked against Intel TBB 2.2.

We present results in terms of speedup, measured both in absolute and self-relative terms. Absolute speedup numbers are measured using, as a $1\times$ baseline, our optimized implementation of the sequential algorithm (Alg. 1), which outperformed Manta's sequential k-D tree builder [SBB*06]. We report self-relative speedups solely to understand parallel scalability of the algorithms. These use the single-thread runs of the parallel nested and in-place implementations as their $1\times$ baseline. These single-thread versions do the same "work" as the parallel versions, including the unnecessary

Processor	Xeon E7450 ("Dunnington")	Xeon X7550 ("Beckton")
Microarchitecture	Core	Nehalem
Core Count	24	32
Socket Count	4	4
Last-level Shared Cache Size	12 MB (L2)	18 MB (L3)
Frequency	2.4 GHz	2.0 GHz
Memory Bandwidth	1x	9x
Memory Size	48 GB	64GB

Table 1: *Experimental Setup*

		Nested		In-Place	
Model	Best-serial	1-core	32-core	1-core	32-core
Bunny	0.304	0.455	0.068	0.512	0.050
Fairy	0.737	1.10	0.146	1.50	0.116
Angel	2.16	3.09	0.337	6.98	0.387
Dragon	3.75	5.50	0.654	8.63	0.744
Happy	4.67	6.89	0.835	11.8	0.951

Table 2: *Running times, in seconds, on Beckton.*

prescan portions of the parallelized phases. For reference, Table 2 lists running times, in seconds, for the best-serial, nested, and in-place algorithms on the Beckton machine, which also clarifies the difference between best-serial algorithm performance and one-core parallel algorithm performance.

Performance on state-of-the-art machine. Fig. 6 shows the absolute speedups of nested (left) and in-place (right), measured on the Beckton machine. Nested achieves nearly 8x speedup on Angel and in-place reaches 7x on Fairy. These represent the best parallel speedup for the upper levels of precise-SAH k-D tree construction to date.

The absolute speedup plot shows that for smaller Bunny (scanned) and Fairy (gaming, varying-sized triangle) inputs, in-place performs better than nested, whereas nested outpeforms in-place on larger (scanned, uniform-sized triangles) inputs. The performance of both algorithms saturates as the number of cores increase. Nested gives increasing performance up to 20 threads, whereas in-place gives increasing performance through 24 threads. In fact, nested's performance degrades significantly from the peak in all cases. Thus, although nested outperforms in-place for three out of five cases on the evaluated machine, the results indicate that in-place is more scalable. We next investigate in more detail the scalability of the two algorithms and the implications for future machines.

Scalability and performance on future machines. Fig. 7 shows the self-relative speedup of nested (left) and in-place (right) over our five inputs. This metric removes the impact of the increased amount of work done in the parallel algorithms (compared to the best sequential algorithm). By fixing the amount of work done across different thread counts, it provides us with a deeper insight on how effectively each algorithm exploits parallelism. The higher the self-relative speedup, the higher the potential for future larger machines with more cores to mitigate the cost of the increased work with increased parallelism. To further understand the effectiveness of the two algorithms in exploiting additional resources in new generations of machines (e.g., larger caches and memory bandwidth), we show self-relative speedups for

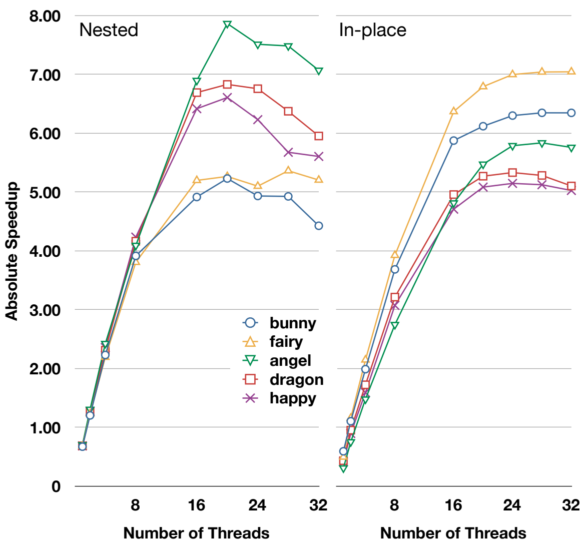


Figure 6: *Absolute speedup of the nested and in-place parallel algorithms for five inputs on the Beckton machine.*

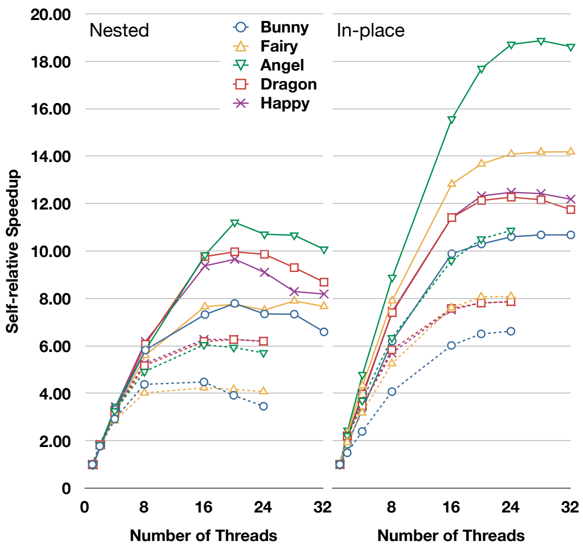


Figure 7: *Self-relative speedup of the nested and in-place parallel algorithms for five inputs, on the Beckton (solid lines) and Dunnington (dashed lines) machines.*

both the newer Beckton (solid line) and the older Dunnington (dashed lines) machines.

The figure immediately shows that in-place is more effective at exploiting parallelism than nested for all inputs on both machines. Although both algorithms perform better on the newer machine, in-place is better able to exploit the resources of the newer machine. Fig. 8 quantifies this effect by showing the ratio of the best speedup of in-place relative to nested for both machines (>1 implies that in-place is faster). The figure clearly shows that for the two inputs where in-place starts out better on the older machine, its performance advantage increases further on the new machine. Conversely, for the cases where nested starts better, its performance advantage reduces on the new machine. Although in-place performance does not yet catch up with nested on the new machine for these cases, the following analysis shows that it is likely that in-place will continue to show higher scalability than nested in newer machines, potentially outperforming it for all cases.

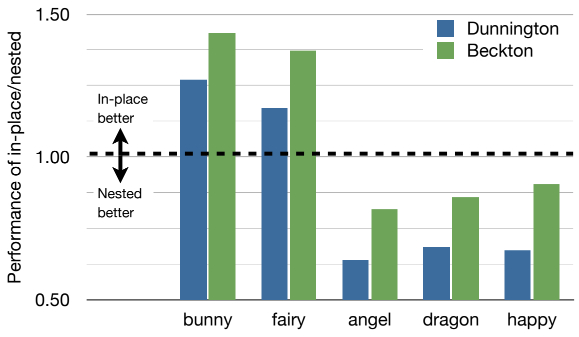


Figure 8: *Performance of in-place relative to nested on the Dunnington and Beckton machines, on all five inputs (> 1 means in-place is better).*

The main bottleneck to scalability for nested is its hard-to-parallelize CLASSIFYTRIANGLES phase. Amdahl's law [Amd67] states that the theoretical maximum speedup attainable using N threads for a program whose parallelizeable fraction is P is given by $1/((1-P)+(P/N))$. Table 3 indicates these maximum absolute (and self-relative) speedups for nested, based on measurements of the fraction of the execution time spent in CLASSIFYTRIANGLES on the Beckton machine.

For example, nested achieves close to 8x absolute speedup on Angel using 20 threads, whereas Table 3 indicates the theoretical maximum speedup of the nested algorithm is slightly less than 10.1x using 20 threads. Thus, nested is already seeing most of its theoretical maximum speedup. The degradation beyond that point is likely due to the increased communication and parallelization overhead with larger number of threads that is not mitigated enough by the increased parallelism.

The in-place algorithm, on the other hand, does not suffer

Input	24 threads	32 threads	∞ threads
bunny	11.5 (14.2)	12.9 (16.5)	21.0 (33.1)
fairy	11.6 (14.4)	13.1 (16.8)	21.6 (34.4)
angel	10.1 (13.5)	11.2 (15.6)	16.7 (29.6)
dragon	9.4 (13.4)	10.3 (15.5)	14.7 (29.3)
happy	9.4 (13.2)	10.3 (15.2)	14.8 (28.1)

Table 3: *Theoretical maximum absolute (and self-relative) speedups achievable by the nested algorithm, based on parallelizable fraction on the Beckton machine.*

from such a bottleneck since it does not contain any significant sequential portion. The performance saturation at larger core counts seen in in-place is likely due to limited system resources; e.g., cache size and memory bandwidth. To investigate this hypothesis, we ran our experiments with all threads scheduled in as few sockets as possible (the default scheduler spreads the threads among the sockets) – this had the positive effect of more cache sharing for smaller input sizes and the negative effect of reduced available pin bandwidth for larger input sizes. We found that the performance of our algorithms was indeed sensitive to the thread placement, showing both the above positive and negative effects (detailed results not shown here).

In summary, we believe that higher core counts coupled with larger caches and memory bandwidth in future machines will allow in-place to continue seeing performance improvements. The performance scalability for nested, however, is likely to be limited by its serial bottleneck.

8. Conclusion

We have presented and analyzed a pair of algorithms designed to address the lack of scalability and/or lack of quality in the upper levels of spatial hierarchies construction. Using our prototype implementations, we showed that our two algorithms, nested and in-place, can achieve speedups of up to 8x and 7x, respectively, over the best sequential performance on a state-of-the-art 32-core cache-coherent shared-memory machine. To our knowledge, these algorithms provide the best known speedups for precise SAH-based high quality k-D tree construction, relative to a sequential case that is better than the best publicly available code.

Each algorithm outperforms the other on some of our inputs for the current state-of-the-art machine, but the in-place approach showed better scalability. Using data obtained from two machines that are a product generation apart, we show that in-place is more effective in harnessing the additional system resources of new machine generations (e.g., cache size and memory bandwidth) than nested. We showed that nested is limited in scalability by a sequential Amdahl's law bottleneck. Overall, we conclude that the in-place algorithm has more potential to scale in future generation multicore hardware.

An interesting future research topic is a GPU implementation of the in-place algorithm. The streaming nature of the in-place algorithm makes it more amenable to a GPU's SIMD-style computation model than the nested algorithm's inherent recursive approach. We are currently investigating various ways to map in-place onto a GPU, and are not aware of any prior work on fully precise SAH based high quality k-D tree construction on the GPU platform.

Another topic for further research centers around the bandwidth limitations of hierarchical data structures identified here and by previous publications, both on CPU and GPU platforms. We have identified some optimizations for the current implementations to improve locality, but these remain to be fully explored.

Acknowledgments Thanks to Ingo Wald for helpful discussions. This work was funded by the Universal Parallel Computing Research Center at the University of Illinois at Urbana-Champaign. The Center is sponsored by Intel Corporation and Microsoft Corporation.

References

[Amd67] AMDAHL G. M.: Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proc. Spring Joint Computer Conf.* (1967), pp. 483–485.

[Ben06] BENTHIN C.: *Realtime Ray Tracing on Current CPU Architectures*. PhD thesis, Saarland University, 2006.

[BHC*93] BLELLOCH G. E., HARDWICK J. C., CHATTERJEE S., SIPELSTEIN J., ZAGHA M.: Implementation of a Portable Nested Data-parallel Language. In *Proc. Symp. on Principles and Practice of Parallel Programming* (1993), pp. 102–111.

[BJK*95] BLUMOFE R. D., JOERG C. F., KUSZMAUL B. C., LEISERSON C. E., RANDALL K. H., ZHOU Y.: Cilk: An Efficient Multithreaded Runtime System. 207–216.

[Ble95] BLELLOCH G. E.: *NESL: A Nested Data-Parallel Language*. Tech. rep., Pittsburgh, PA, USA, 1995.

[CHCH06] CARR N. A., HOBEROCK J., CRANE K., HART J. C.: Fast GPU Ray Tracing of Dynamic Meshes using Geometry Images. In *Proc. Graphics Interface* (2006), pp. 203–209.

[GDS*08] GOVINDARAJU V., DJEU P., SANKARALINGAM K., VERNON M., MARK W. R.: Toward a Multicore Architecture for Real-Time Ray-Tracing. In *Proc. Intl. Symp. on Microarchitecture* (2008), pp. 176–187.

[GHGH08] GODIYAL A., HOBEROCK J., GARLAND M., HART J. C.: Rapid Multipole Graph Drawing on the GPU. In *Proc. Graph Drawing* (2008), pp. 90–101.

[GHJV95] GAMMA E., HELM R., JOHNSON R., VLISSIDES J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GKM93] GREENE N., KASS M., MILLER G.: Hierarchical Z-buffer Visibility. In *Proc. SIGGRAPH* (1993), pp. 231–238.

[GS87] GOLDSMITH J., SALMON J.: Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics and Applications 7*, 5 (1987), 14–20.

[Hav00] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[HMS06] HUNT W., MARK W. R., STOLL G.: Fast k-D Tree Construction with an Adaptive Error-Bounded Heuristic. In *Proc. Interactive Ray Tracing* (2006), pp. 81–88.

[HS86] HILLIS W. D., STEELE JR. G. L.: Data Parallel Algorithms. *CACM 29*, 12 (1986), 1170–1183.

[Int09] INTEL: *Intel (R) Threading Building Block Manual*, 2009. Document number 315415-001US.

[LGS*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast BVH Construction on GPUs. *Computer Graphics Forum 28*, 2 (2009), 375–384.

[LP08] LUEBKE D., PARKER S.: *Interactive Ray Tracing with CUDA*. Tech. rep., 2008.

[MB90] MACDONALD J. D., BOOTH K. S.: Heuristics for Ray Tracing using Space Subdivision. *Visual Computer 6*, 3 (1990), 153–65.

[MSM04] MATTSON T. G., SANDERS B. A., MASSINGILL B. L.: *Patterns for Parallel Programming*. Addison-Wesley, 2004.

[PGSS06] POPOV S., GÜNTHER J., SEIDEL H.-P., SLUSALLEK P.: Experiences with Streaming Construction of SAH KD-Trees. In *Proc. Interactive Ray Tracing* (2006), pp. 89–94.

[SBB*06] STEPHENS A., BOULOS S., BIGLER J., WALD I., PARKER S.: An Application of Scalable Massive Model Interaction using Shared Memory Systems. In *Proc. EG Symp. on Parallel Graphics and Vis.* (2006), pp. 19–26.

[SCS*08] SEILER L., CARMEAN D., SPRANGLE E., FORSYTH T., ABRASH M., DUBEY P., JUNKINS S., LAKE A., SUGERMAN J., CAVIN R., ESPASA R., GROCHOWSKI E., JUAN T., HANRAHAN P.: Larrabee: A Many-core x86 Architecture for Visual Computing. *ACM Trans. Graph. 27*, 3 (2008), 18:1–18:15.

[SSK07] SHEVTSOV M., SOUPIKOV A., KAPUSTIN A.: Highly Parallel Fast k-D Tree Construction for Interactive Ray Tracing of Dynamic Scenes. *Computer Graphics Forum 26*, 3 (2007), 395–404.

[Wal07] WALD I.: On Fast Construction of SAH based Bounding Volume Hierarchies. In *Proc. Interactive Ray Tracing* (2007).

[WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Trans. Graph. 26*, 1 (2007), 6.

[WH06] WALD I., HAVRAN V.: On Building Fast k-D Trees for Ray Tracing and On Doing That in $O(NlogN)$. In *Proc. Interactive Ray Tracing* (2006), pp. 61–69.

[WHG84] WEGHORST H., HOOPER G., GREENBERG D. P.: Improved Computational Methods for Ray Tracing. *ACM Trans. Graph. 3*, 1 (1984), 52–69.

[WIP08] WALD I., IZE T., PARKER S. G.: Fast, Parallel, and Asynchronous Construction of BVHs for Ray Tracing Animated Scenes. *Computers & Graphics 32*, 1 (2008), 3–13.

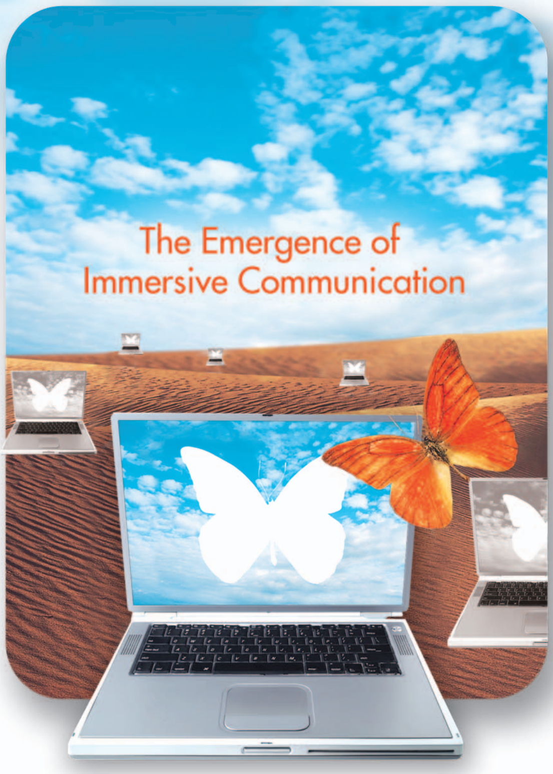
[WSS05] WOOP S., SCHMITTLER J., SLUSALLEK P.: RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. *ACM Trans. Graph. 24*, 3 (2005), 434–444.

[ZHWG08] ZHOU K., HOU Q., WANG R., GUO B.: Real-time k-D Tree Construction on Graphics Hardware. In *Proc. SIGGRAPH Asia* (2008), vol. 27, pp. 1–11.

[Minh N. Do, Quang H. Nguyen, Ha T. Nguyen, Daniel Kubacki, and Sanjay J. Patel]

Immersive Visual Communication

[An introduction of the propagation algorithm and analysis for image-based rendering with depth cameras]



The National Academy of Engineering recently identified 14 grand challenges for engineering in the 21st century (www.engineeringchallenges.org). We believe that the continuing advances in ubiquitous sensing, processing, and computing provide the potential to tackle two of these 14 grand challenges: specifically, enhancing virtual reality and advancing personalized learning.

INTRODUCTION

Recently, the ubiquity of digital cameras has made a great impact on visual communication as can be seen from the explosive growth of visual contents on the Internet and the default inclusion of a digital camera on cell phones and laptops. Two recent developments in sensing and computing have the potential to revolutionize visual communication further by enabling immersive and interactive capabilities. The first development is the emergence of lower-priced, fast, and robust cameras for measuring depth [1]. Depth measurements provide a perfect complementary information to the traditional color imaging in capturing the three-dimensional (3-D) scene. The second development is the general-purpose parallel computing platforms such as graphics processing units (GPUs) that can significantly speedup many visual computing tasks [2] and bring them to the real-time realm. These developments and high demand for immersive communication present a tremendous opportunity for the signal processing field.

In particular, we envision systems, called remote reality, which can record real scenes and render 3-D free-viewpoint videos and augment it with virtual reality. Such a system can provide immersive and interactive 3-D viewing experiences for personalized distance learning and immersive communication. With recorded 3-D visual information, users can freely choose their viewpoints as if each of them had a virtual mobile camera. When users want to get a closer look at some part of a remote scene, they simply have to move the virtual camera to a suitable location. With depth keying, the video background can be removed and replaced by other interactive backgrounds. Moreover, 3-D free-viewpoint videos can be merged with objects of virtual 3-D worlds. Then users become integral parts of a virtual world, which frees them from some of the constraints imposed by current telecommunication systems.

Digital Object Identifier 10.1109/MSP.2010.939075
Date of publication: 17 December 2010

1053-5888/11/$26.00©2011IEEE

The first key problem in remote reality is to find an efficient representation for generating free-viewpoint 3-D videos, because this has major impact on subsequent processing, transmitting, and rendering steps. Image-based rendering (IBR) is the process of synthesizing novel views from prerendered or preacquired reference images of a scene [3]. Obviating the need to create a full geometric 3-D model, IBR is relatively inexpensive compared to traditional rendering while still providing high photorealism.

WE ENVISION SYSTEMS, CALLED REMOTE REALITY, WHICH CAN RECORD REAL SCENES AND RENDER 3-D FREE-VIEWPOINT VIDEOS AND AUGMENT IT WITH VIRTUAL REALITY.

Depth IBR (DIBR) combines color images with per-pixel depth information of the scene to synthesize novel views. Depth information can be obtained by stereo matching algorithms [4]. However, these algorithms are usually complicated, inaccurate, and inapplicable for real-time applications. Thanks to the recent developments of new range sensors [1] that measure time delay between transmission of a light pulse and detection of the reflected signal on an entire frame at once, depth information can be obtained in real time from depth cameras. This makes the DIBR problem less computationally intense and more robust than other techniques. Furthermore, it helps to significantly reduce the required number of cameras and transmitting data.

A problem with DIBR techniques is that the resolution of depth images from depth cameras is often low, whereas the technology for color cameras is more mature. Hence, the need for integrating and exploiting the synergy between color cameras and depth cameras becomes significant. Moreover, because the geometric information in DIBR is usually captured in real time from the physical world instead of from modeling a synthetic world (which also makes DIBR more photorealistic), the obtained data always suffer from noise and insufficient sampling effects. Therefore, the need for coupling image processing techniques with rendering techniques is a must.

The fusion of depth and color information in DIBR raises a fundamental problem of analyzing the effects of different input factors on the rendering quality. The answer to this problem is crucial for both theoretical and practical purposes; we cannot effectively control the rendering quality and the cost of DIBR systems without accurate quantitative analysis of the rendering quality.

Finally, the need for processing and integrating acquired depth and color videos significantly increases the computations and is infeasible for real-time applications without parallelism, which makes algorithm and architecture codesign critical. Besides, since rendering with full geometric information (color and depth) has been optimized for GPUs, GPUs are considered to be the ideal computing platform for the DIBR problem. For these reasons, we should consciously develop processing algorithms that are suitable for the GPU platform.

REVIEW OF EXISTING FREE-VIEWPOINT RENDERING SOLUTIONS

Many IBR algorithms have been proposed to synthesize new views from actual acquired (referred to as reference) images of a scene [3]. One earlier approach is to use a large number (from tens to more than a hundred) of regular color cameras to compensate for the lack of geometry [6]–[8]. In such a system, new-view images are obtained by simply interpolating in the ray domain. However, the use of large number cameras put a heavy burden on the calibration, storage, and transmission tasks. Furthermore, the bulky setup required for large number of cameras limits the deployment of such systems.

An alternative approach for IBR is to use explicit depth information in addition with color images to synthesize novel views. If the per-pixel depth information is available, the warping equation [9] can be used to transfer information from actual color pixels to the virtual image plane. Zitnick et al. [4] demonstrated that high-quality and real-time new-view rendering can be achieved with depth plus color images using a modest number of cameras. However, their system requires intensive and off-line stereo matching computation to estimate depth information. A generalized framework of DIBR for 3-D TV applications was summarized in [10], which also includes the issues of compression and transmission of IBR data.

Assuming that per-pixel color plus depth information is available at reference views, several algorithms have been recently developed for view synthesis under the DIBR framework [11]–[14]. Generally, these algorithms are based on the following steps:

1) Forward warp reference views to the new view.
2) Process warped images to eliminate artifacts due to warping and noise.
3) Blend several warped images in the new view image plane.
4) Inpaint or fill holes due to disocclusion.

The first and third steps are quite straightforward and standard. However, with slightly noisy DIBR data, they create visible artifacts. Most of these artifacts appear around object boundaries. Hence, the second and forth steps are crucial in reducing these artifacts. We refer to [14] for a detail discussion and comparison of various proposed artifact reduction algorithms.

With the recent progress of depth cameras technologies [1], the depth information the depth information can be directly measured by depth cameras instead of stereo matching. In practice, depth cameras often provide the depth images with lower resolution and poorer quality than those of the color images. Therefore, the combination of several high quality color cameras with a few depth cameras becomes an interesting setup for IBR.

One way to increase the resolution and enhance the quality of depth images is to exploit the abundant information in high-quality color images. Several methods have been proposed to tackle this issue, including the Markov model approach [15] and the iterative bilateral filtering coupling with subpixel estimation [16]. However, these methods are computationally complex, hence, they are not yet suitable for real-time applications.

ONE WAY TO INCREASE THE RESOLUTION AND ENHANCE THE QUALITY OF DEPTH IMAGES IS TO EXPLOIT THE ABUNDANT INFORMATION IN HIGH-QUALITY COLOR IMAGES.

While many IBR methods have been proposed, little research has addressed the fundamental questions on the impact of various configuration parameters on the rendering quality of IBR algorithms, such as the number of actual cameras and their geometrical positions as well as their resolution and image quality. A mathematical framework to study this problem is the concept of the plenoptic function [17] that describes the light intensity passing through every viewpoint, in every direction for all time, and for every wavelength. McMillan and Bishop [18] recognized that the IBR problem is to reconstruct the plenoptic function (which leads to virtual images) using a set of discrete samples (i.e., actual images). Using a simplified domain of the plenoptic function, the light field, Chai et al. [19] analyzed the minimum number of images necessary to guarantee a given rendering quality.

In the analysis of IBR data, most existing literature addresses the Fourier domain because the IBR data exhibit fan-type structure in the frequency domain. However, Do et al. [20] showed that, in general, the plenoptic is not bandlimited unless the surface of the scene is flat. Another limit of the frequency-based approach is that it can not provide local analysis of the rendering quality. To address these drawbacks, Nguyen and Do [5] analyzed the rendering quality of novel views in the spatial domain using the framework of nonuniform sampling and interpolation. The mean absolute error (MAE) of the rendered images can be bounded based on the configuration parameters such as depth and color errors (e.g., due to lossy coding), scene geometry and texture, number of actual cameras and their positions, and resolution.

IMAGE-BASED RENDERING WITH DEPTH CAMERAS USING THE PROPAGATION ALGORITHM

To effectively represent 3-D free-viewpoint videos in real time using commodity cameras, we proposed [21] the 3-D propagation algorithm that consists of three main steps (see Figure 1). The first two steps are used to propagate the actual depth measurements from depth cameras to color cameras and use color information to enhance the depth quality. The last step—rendering—is the same with the DIBR view synthesis process that is reviewed in the previous section.

The 3-D propagation algorithm allows arbitrary configurations of color and depth cameras in 3-D. In addition, it adapts well with any combination of a few high-resolution color cameras and low-resolution depth cameras, which allows performing low-cost and light DIBR systems. Moreover, the propagation of available depth information to color cameras' image planes allow the development of effective algorithms to integrate depth and color information.

DEPTH PROPAGATION

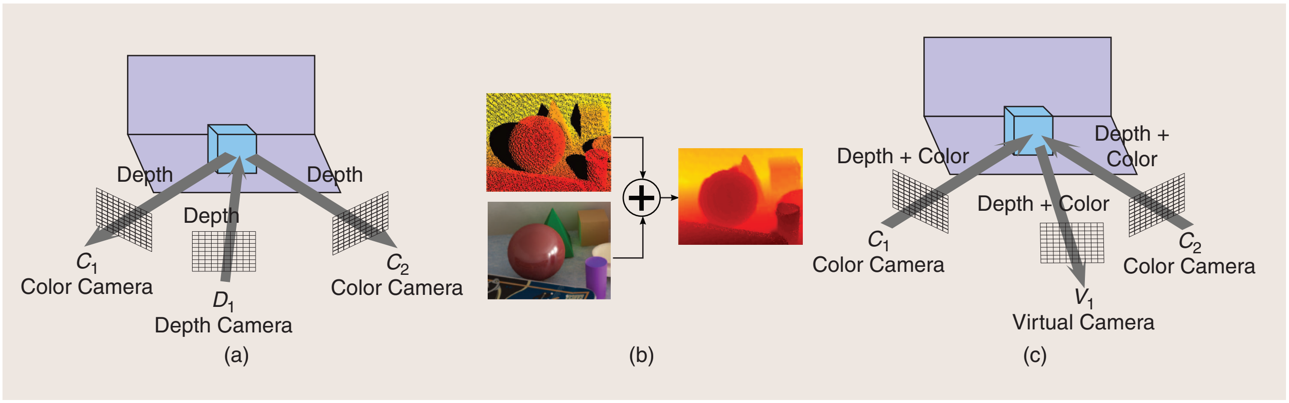
Depth information from the depth camera is propagated to every color camera's image plane using the warping equation in [9]. Since the depth resolution is usually much smaller than the color resolution, and occluded parts of the scene in the depth view are revealed in the other views, the propagated depth image usually has a large number of missing depth pixels. An example result is shown in Figure 2(a).

COLOR-BASED DEPTH FILLING

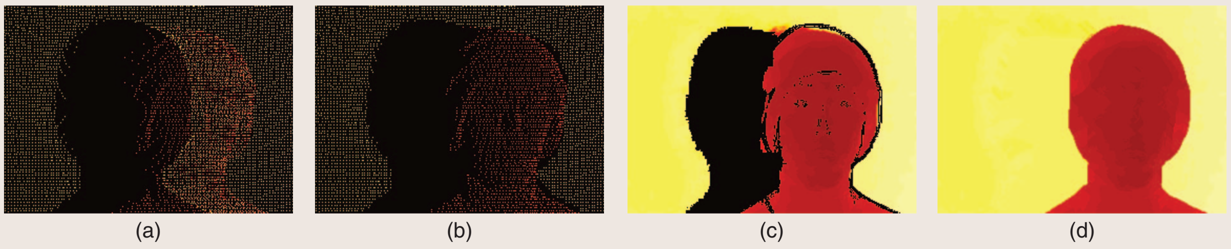
In this step, the missing depth pixels in the propagated image are efficiently filled using the color image at each color view. The block diagram is described in Figure 3.

OCCLUSION REMOVAL

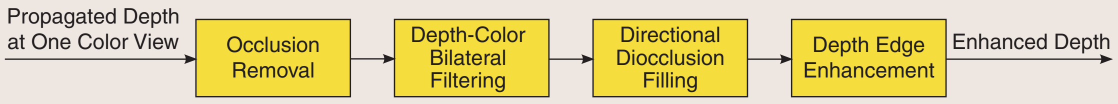
As shown in Figure 2(a), some background sample points (in brighter color) visible in the depth image should be occluded



[FIG1] Three main steps in the 3-D propagation algorithm: (a) depth propagation, (b) color-base depth filling, and (c) rendering.



[FIG2] **Steps for the color-based depth filling algorithm. Refer to the section "Color-Based Depth Filling" for a detailed description of these steps: (a) propagated depth, (b) occlusion removal, (c) CBDF, and (d) disocclusion and edge enhancement.**


Propagated Depth at One Color View
Occlusion Removal
Depth-Color Bilateral Filtering
Directional Disocclusion Filling
Depth Edge Enhancement
Enhanced Depth

[FIG3] **Block diagram of the color-based depth filling step.**

by the foreground (pixels with darker color) in the propagated depth image but are still visible. This significantly degrades the interpolation quality. We can remove these occluded pixels based on the smoothness of surfaces. If a point A in the propagated depth image is locally surrounded by neighboring points whose depth values are σ smaller than the depth of A, then A is recognized to be occluded by the surface composed of those neighbors. In that case, the depth value of A is set to unknown. An example result is shown in Figure 2(b).

COLOR-BASED BILATERAL DEPTH FILTERING

The color-based bilateral depth filtering (CBDF) is defined as follows:

$$d_A = \frac{1}{W_A} \sum_{B \in S_A} G_{\sigma_s}(|x_A - x_B|) \cdot G_{\sigma_r}(|I_A - I_B|) \cdot d_B \quad (1)$$

$$W_A = \sum_{B \in S_A} G_{\sigma_s}(|x_A - x_B|) \cdot G_{\sigma_r}(|I_A - I_B|), \quad (2)$$

where d_A, I_A, and x_A are the depth value, the color value, and the 2-D coordinate of point A. S_A is set of neighboring pixels of A, $G_\sigma(|x|) = \exp(-|x|^2/2\sigma^2)$ is the Gaussian kernel with variance σ^2, and W_A is the normalizing term.

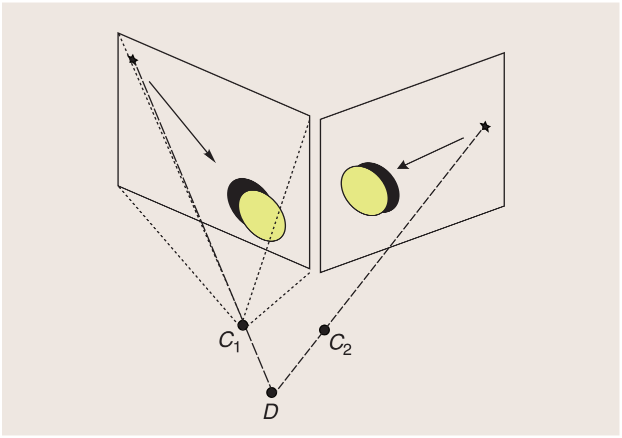
The idea of using color differences as a range filter to interpolate depth value is based on the observation that whenever a depth edge appears, there is almost always a corresponding color edge due to color differences between objects or between foreground and background. The CBDF also works well with textured surfaces since it counts only pixels on that surface which have similar color to the interpolated pixel. If surfaces have the same color, the color does not give any new information and the CBDF works as a simple interpolation scheme such as bilinear or bicubic. Therefore, by integrating known depth and color information, the proposed CBDF effectively interpolates unknown depth pixels while keeping sharp depth edges. An example result after this step is shown in Figure 2(c).

DIRECTIONAL DISOCCLUSION FILLING

To fill the disocclusion areas, a filling direction needs to be specified. Otherwise, if the filling is performed from all directions, depth edges are spread out. Based on the observation, as illustrated in Figure 4, which shows that the disocclusion areas are caused by the change of camera position, we choose the filling direction as the vector pointing from the epipole point (the projection of the depth camera position onto the image plane of the color camera) to the center of the propagated depth image.

DEPTH EDGE ENHANCEMENT

The sharpness of depth edges is extremely important. In the rendering step, a slightly blurred edge may blow up to a significant and visually annoying smearing artifact. The purpose of this stage is to correct and sharpen edges in the



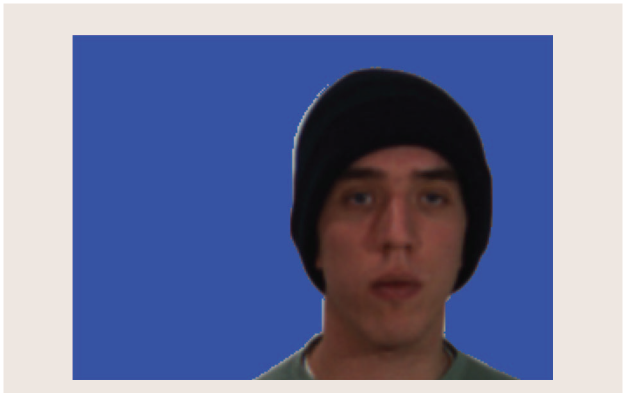
[FIG4] **Directional disocclusion filling. D, C_1, and C_2 are the camera centers of the depth and two color cameras. Stars indicate the epipole points. Black regions indicate the disocclusion areas. The arrows indicate the chosen directional disocclusion filling.**

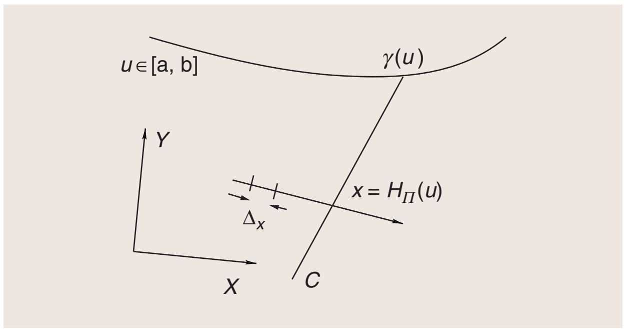
propagated depth images at color views. The proposed depth edge enhancement stage includes two parts. First, depth edge gradients are detected with the Sobel operator. Then, pixels with significant edge gradients are marked as undetermined depth pixels and their depth values need to be recalculated. Next, for each undetermined depth pixel, a block-based search is used to find the best pixel with known depth that matches in the color domain. Once the best candidate is chosen, its depth value is assigned to the unknown pixel. An example result after this step is shown in Figure 2(d).

[THE 3-D PROPAGATION ALGORITHM ALLOWS ARBITRARY CONFIGURATIONS OF COLOR AND DEPTH CAMERAS IN 3-D.]

RENDERING

Finally, depth and color information at each color view are propagated into the virtual view using the same technique as in the depth propagation step in Figure 1. The occlusion removal stage is performed for each propagated view. The final rendered images are blended and smoothed with a Gaussian filter. Note that most of the unknown color pixels in this step are caused by nonuniform resampling since the color cameras are intentionally installed in a way to capture the whole scene from different views and, therefore, reduce as much as possible the disocclusion areas. An example result after this step is shown in Figure 5.


[FIG5] **Background subtraction using propagated depth information.**



[FIG6] **The 2-D scene-camera model. The scene surface is modeled as a parameterized curve $\gamma(u)$ for $u \in [a, b] \subset \mathbb{R}$. The scene-to-image mapping $x = H_\Pi(u)$ maps a surface point u to an image point x. The texture function $T(u)$ is "painted" on the surface. The camera resolution is characterized by the pixel interval Δ_x on the image plane.**

ERROR ANALYSIS OF DEPTH IMAGE-BASED RENDERING

In this section, we present the analysis of the rendering quality based on the IBR configurations such as depth and color estimate errors, the scene geometry and texture, as well as the number of actual cameras and their positions and resolution. The analysis presented in this section is simplified from the results in [5]. We focus on the 2-D setting with no occlusion to present the results with better clarity.

PROBLEM SETTING

Figure 6 depicts the studied 2-D scene-camera model. The surface of the scene is modeled as a 2-D parameterized curve $\gamma(u) : [a, b] \to \mathbb{R}^2$. Each value of $u \in [a, b]$ corresponds to a surface point $\gamma(u) = [X(u), Y(u)]^T$. The color (or texture) "painted" on the surface is the function $T(u) : [a, b] \to \mathbb{R}$. Given a parameterization of the scene, the texture function $T(u)$ is independent of the cameras and the scene geometry $\gamma(u)$.

Using the pinhole camera model, there is a mapping from surface points $\gamma(u)$ to image points $x = H_\Pi(u)$, where $H_\Pi(u)$ is the scene-to-image mapping. Given an image point x in the image plane, the color at x is $f(x) = T(H_\Pi^{-1}(x))$. The image of the scene at a camera Π are discrete samples of the color function $f(x)$ with sample interval Δ_x.

We assume that at all actual pixels, both the color and the depth, are available. Let ε_X, ε_Y, and ε_T be the errors (due to measurement or lossy coding) of $X(u)$, $Y(u)$, and $T(u)$, respectively. We assume that these estimate errors are bounded

$$|\varepsilon_X| \le E_D, \qquad |\varepsilon_Y| \le E_D, \qquad |\varepsilon_T| \le E_T.$$

In the next sections, we will analyze the rendering quality for the case where N actual cameras $\{C_i, \Pi_i\}_{i=1}^N$ are used to render the image at a virtual camera (C_v, Π_v).

MAIN RESULT

In this section, we first give the mathematical definition of terms that will be used later in Theorem 1 to bound the MAE of the rendered image. As we define, we will also provide the physical meaning of the terms to appreciate the result.

The multiple-view term of order k is defined as

$$Y_k = \int_a^b \left(\sum_{i=1}^{N} \Pi_i(u) \right)^{1-k} (\Pi_v(u))^k du. \tag{3}$$

This term measures the impact of the actual cameras on the virtual camera based on their relative geometrical positions.

The depth jittering term

$$B_v = \sup_{u \in [a,b], i=1,\dots,N} \left\{ \frac{\|C_v - C_i\|_2}{d(u)^2} \right\}, \tag{4}$$

where $d(u)$ is the depth at surface point $\gamma(u)$ to the virtual image plane. This quantity measures how geometrically deviated the virtual camera is from the actual cameras.

Based on the above definitions, the following theorem presents a bound on the rendering quality of the propagation algorithm.

Theorem 1 [5]: The MAE of the virtual image using the propagation algorithm is bounded by

$$\mathrm{MAE} \le \frac{3Y_3}{4Y_1}\Delta_x^2 \|f_v''\|_\infty + E_T + E_D B_v \|f_v'\|_\infty. \tag{5}$$

We note that the first term in (5) is related to the interpolation error in the texture regions. The second term is related to the quality of the actual color cameras. The third term measures the impact of the depth and its estimate.

A MAJOR ADVANTAGE OF OUR PROPOSED ALGORITHM IS THAT IT CAN BE EASILY MAPPED ONTO DATA PARALLEL ARCHITECTURES SUCH AS MODERN GPUs.

INTERPRETATIONS

In this section, we provide interpretations of the result in the section "Results." The idea is to look at each component of the error bound in (5) and find its physical meaning and implications on IBR applications. These interpretations should only be considered as "rules of thumb" when designing IBR systems. For rigorous analysis, we refer to the original paper [5].

- Rule 1: In texture regions, the density of actual pixels counts. It can be shown that $Y_3 = \mathcal{O}(N^{-2})$. Hence, the first term in (5) behaves as $\mathcal{O}(\Delta_x^2/N^2)$. In other words, increasing the resolution of actual cameras has a similar effect to having more actual cameras in texture regions.
- Rule 2: The impact of the actual camera quality on the MAE is linear. It is intuitive to see that the quality of actual cameras effect directly to the rendering quality. However, the result in (5) also reveals that the rendering quality is linearly proportional to the actual camera quality (for both color and depth).
- Rule 3: Use neighboring actual cameras when depth is inaccurate. To reduce the impact of depth errors, i.e., the third term, two options are available. The first option is obvious, to equip with better depth cameras, which is to reduce E_D. The second option is to reduce B_v, or equivalently to use information from actual cameras that are close to the virtual camera.

FAST PROCESSING USING GENERAL-PURPOSE PARALLEL COMPUTING

A major advantage of the algorithm outlined in the section "Image-Based Rendering with Depth Cameras Using Propagation Algorithm" is that it can be easily mapped onto data parallel architectures such as modern GPUs. In this section, we briefly describe the parallelism of each processing step of our algorithm, and the high-level mapping onto the Nvidia CUDA architecture for GPU-based computing.

The occlusion removal and CBDF stages are purely parallel as each pixel in the desired view can be computed independently. In the depth propagation stage, copying the depth values in the reference view to appropriate pixels in the desired view is more complex from a parallelism perspective since, at some pixels, this is not a one-to-one mapping. This operation requires some form of synchronization to prevent concurrent writes to the same pixel and can be accomplished with the use of atomic memory operations, or alternatively, with the use of Z-buffering hardware available on modern GPUs.

The disocclusion filling stage also has a sequential component since calculating unknown depth information is dependent on previously interpolated values. However, this dependence exists only on one-dimensional (1-D) lines emanating from the epipole point, and thus the problem can be expressed as a parallel set of 1-D filters. First, find the epipole point position and categorize into one of eight following subsets: top, bottom, left, right, top left, top right, bottom left, or bottom right, corresponding to eight sets of par allel lines for every 45° angle. The parallel lines in each set need to pass through all pixels in the depth image. For each set of parallel lines, all pixel coordinates of each line can be precomputed and stored in a lookup table.

The 1-D CBDF is performed with each line proceeding in parallel, which can be easily mapped onto the GPU architecture. The depth edge enhancement stage is simply a series of independent window-based operators and, hence, is naturally parallel. The final rendering step is quite similar to the first and second part of the algorithm except for the inclusion of a median filter. However, the median filter is another window-based operator and, hence, is suitable for parallelism.

Regarding the parallel scalability of our algorithm: our experiments show that there is ample data parallelism to take advantage of the heavily threaded 128-core modern GPU architecture. Our technique scales further with image size, and higher resolution images will create additional parallel work for future data parallel architectures that support still higher degrees of parallelism. Furthermore, with the use of additional cameras, the data parallel computational load increases still further, creating additional work that can be gainfully accelerated on future data parallel architectures.

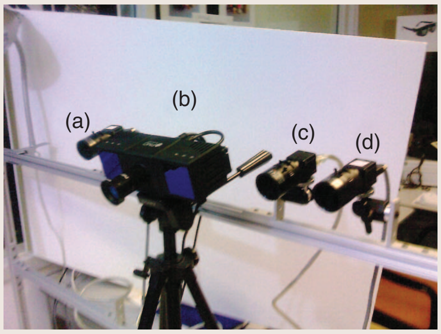
To check the efficiency of the parallelism, we compare the CPU-based implementation and the preliminary GPU-based implementation of the depth propagation stage and the CBDF stage. The experiment was run on the platform of Intel Core2 Duo E8400 3.0 GHz and a Nvidia GeForce 9800GT 600 MHz with 112 processing cores.

AN EFFICIENT REPRESENTATION OF DIBR DATA IS IMPORTANT TO FACILITATING THE PROCESSING, TRANSMITTING, AND RENDERING OF DIBR DATA.

Table 1 shows the comparison results of two representative processing steps in the 3-D propagation algorithm. The depth propagation step, as mentioned above, requires memory synchronization to prevent concurrent writes and thus limits the effectiveness of multithread. As a result, we observe the least speedup in the GPU implementation compared to the CPU implementation. The CBDF step, in contrast, is highly parallel and benefits a very large speedup with GPU. Since the CBDF step consumes the most computing time in the CPU implementation, the mapping from CPU to GPU significantly speeds up the overall computing time of the 3-D propagation algorithm and has potential to achieve real-time performance.

[TABLE 1] TIMING COMPARISON (IN MILLISECONDS) OF SEQUENTIAL CPU-BASED AND GPU-BASED IMPLEMENTATIONS FOR THE DEPTH PROPAGATION STAGE AND THE CBDF STAGE. THE IMAGE RESOLUTION IS 800 × 600 AND THE FILTER KERNEL SIZE IS 11 × 11.

	HARDWARE	DEPTH PROP.	CBDF
CPU	INTEL CORE 2 DUO E8400, 3.0 GHZ	38	1041
GPU	NVIDIA GEFORCE 9800 GT, 600 MHZ	24	14
SPEEDUP		1.6X	74.4X



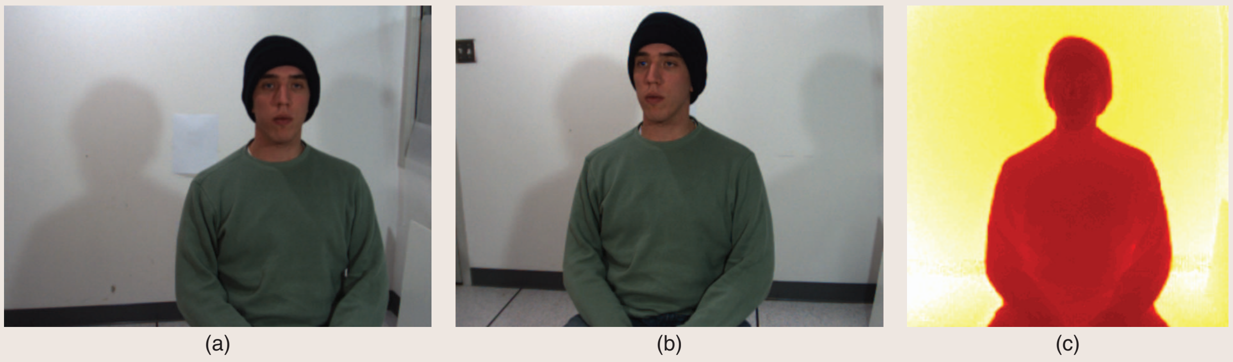
[FIG7] The real camera setup used in our experiments. The (b) depth camera is positioned in the center with a color camera approximately 10 in to the left (a) and right (d). The third color camera (c) is used as a ground truth for the virtual camera.

NUMERICAL EXPERIMENTS

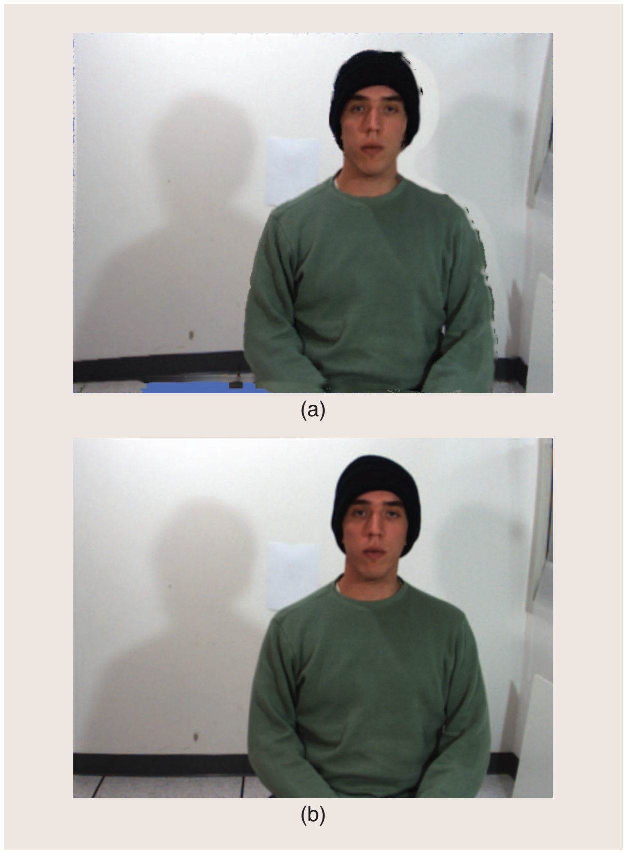
In this section, we provide results from the implementation of the above algorithm using actual depth and color cameras. For our experiment, we used one PMD CamCube depth camera with a resolution of 204 × 204 and three Point Grey Flea2 color research cameras, each with a resolution of 640 × 480. The cameras were arranged similar to a typical teleconference setup. Two color cameras were placed approximately 20 in apart with a depth camera in the middle. A third camera was used to provide a ground truth for the virtual camera. Figure 7 shows the setup. Note that these camera are not necessarily on a consistent baseline, which demonstrates the greater generality for camera setups. The captured scene is of a person approximately 4 ft away from the camera setup. Figure 8 shows the input images for the algorithm; Figure 8(c) is an example of an image captured by the depth camera. This experiment is chosen to demonstrate that DIBR can be utilized to correct the eye-gaze problem of teleconference systems.

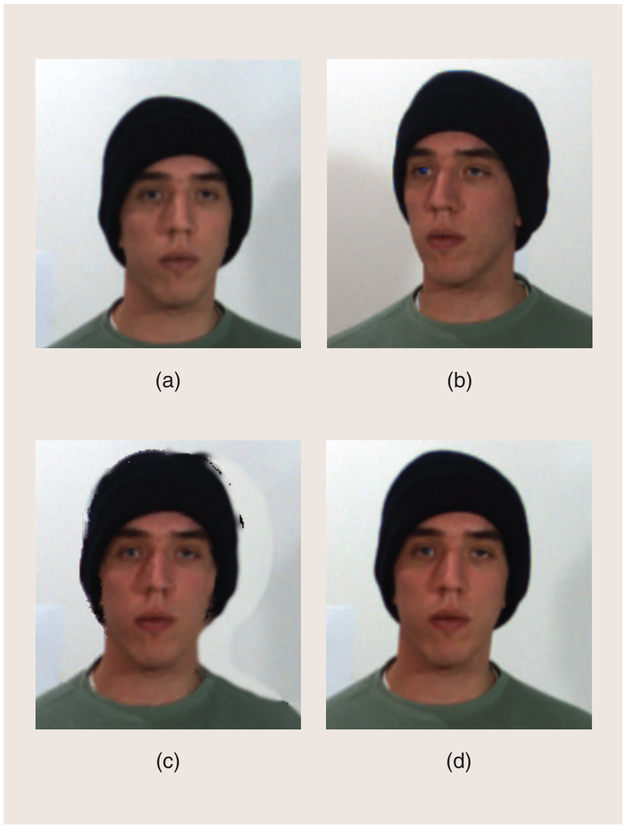
CALIBRATION

To fuse depth and color information, the cameras are calibrated using the classical checkerboard calibration technique, which is implemented using OpenCV's camera calibration and



[FIG8] Images used as input for the DIBR algorithm. The color images have a resolution of 640 × 480 and the depth image has a resolution of 204 × 204. (a) Input left color view, (b) input right color view, and (c) input depth image.


[FIG9] **A visual comparison between the (a) rendered virtual view and (b) the ground truth virtual view.**


[FIG10] **Closeup of the eyes to show eye-gaze correction using DBIR. (a) Left view, (b) right view, (c) rendered view, and (d) ground truth.**

stereo reconstruction toolbox. For depth camera, the intensity image is used for calibration.

To utilize the propagation equation, it was necessary to correct for the distortion of each camera. This was also implemented in OpenCV using the distortion coefficients determined in the camera calibration. The distortion propagates due to the depth enhancements made using the distorted color images.

RESULTS

A comparison between the rendered virtual view and the ground truth virtual view can be seen in Figure 9. Visually, the rendered image and the ground truth are very similar. Figure 10 shows a closeup comparison of input color views, rendered virtual view, and ground truth. We can clearly see the value of the DIBR system for providing an eye-gaze corrected view for video conferencing. Note that the location of the rendered view can be changed freely and dynamically, for example, by following a gaze-tracking system.

Figure 2 displays a close up of the color-based depth filling process detailed in the section "Color-Based Depth Filling." Note how the sparse propagated depth map is enhanced to a full depth map using the color information. Figure 5 makes evident the edge accuracy of the color-based depth filling. It also demonstrates another application of depth information for background subtraction. We can correct for eye gaze and remove/replace background for video conferencing using the same setup and our algorithms.

DISCUSSIONS AND CONCLUSIONS

In this article, we present a brief introduction of our algorithm and analysis for IBR with depth cameras. Our algorithm includes various techniques to render virtual images from a set of actual color and depth cameras, as well as parallel processing for real-time applications. We also give rigorous analysis of the rendering quality based on the camera configurations, such as depth, color quality, and geometrical positions of the virtual cameras. Our algorithms and analysis are general for any camera configuration. The proposed algorithm produces excellent rendering quality, such as correct eye gazing, as demonstrated in the experimental results.

For future work, important open problems are necessary to make DIBR applications practical. An efficient representation of DIBR data is important to facilitating the processing, transmitting, and rendering of DIBR data. The problem of compressions will be in demand for DIBR applications to serve a large number of users. The key to this problem will be how to use the redundancy between color and depth images. This redundancy can also be used in the processing and rendering steps. Finally, more precise analysis is necessary for DIBR applications to effectively control the quality and cost of DIBR applications.

ACKNOWLEDGMENTS
This work was supported in part by the U.S. National Science Foundation under grant CCF-0312432 and by the Universal Parallel Computing Research Center (UPCRC) at the University of Illinois, Urbana-Champaign. The UPCRC is sponsored by Intel Corporation and Microsoft Corporation.

AUTHORS
Minh N. Do (minhdo@illinois.edu) is an associate professor in the Department of Electrical and Computer Engineering at the University of Illinois, Urbana-Champaign (UIUC). He received a Ph.D. degree in communication systems from the Swiss Federal Institute of Technology Lausanne (EPFL). His research interests include image and multidimensional signal processing, computational imaging, wavelets and multiscale geometric analysis, and visual information representation. He received a Best Doctoral Thesis Award from EPFL, a CAREER Award from the National Science Foundation, a Xerox Award for Faculty Research from UIUC, and an IEEE Signal Processing Society Young Author Best Paper Award.

Quang H. Nguyen (quang@nuvixa.com) received the M.Sc. degree in electrical and computer engineering from the University of Illinois, Urbana-Champaign in 2009. His research interest includes DIBR and image processing using depth sensors. Currently, he is an R&D software engineer at Nuvixa Inc.

Ha T. Nguyen (nguyenthaiha678@gmail.com) received his engineering diploma from the Ecole Polytechnique, France, and a Ph.D. degree in electrical and computer engineering from the University of Illinois, Urbana-Champaign. Since 2007, he has been with the Media Processing Technologies Lab at Sony Electronics, San Jose, California. His research interests include multiple-view geometry, wavelets, directional representations, image and multidimensional signal processing, and video coding. He received the 1996 International Mathematical Olympiad Gold Medal, and he was a Best Student Paper Contest winner at the 2005 IEEE International Conference on Audio, Speech, and Signal Processing.

Daniel Kubacki (daniel.kubacki@gmail.com) received his B.S. degree in electrical engineering from Penn State Erie, The Behrend College in 2007. He is currently pursuing a Ph.D. degree in electrical and computer engineering at the University of Illinois, Urbana-Champaign. His research interests include image processing, multidimensional signal processing, and computer vision. Specifically, he is interested in applications utilizing depth cameras and methods to integrate and represent information gathered from calibrated depth and color videos.

Sanjay J. Patel (sjp@illinois.edu) is an associate professor of electrical and computer engineering and Willett Faculty Scholar at the University of Illinois, Urbana-Champaign. From 2004 to 2008, he served as the chief architect and chief technology officer at AGEIA Technologies, prior to its acquisition by Nvidia Corp. He earned his bachelors degree (1990), M.Sc. degree (1992), and Ph.D. degree (1999) in computer science and engineering from the University of Michigan, Ann Arbor.

REFERENCES
[1] A. Kolb, E. Barth, R. Koch, and R. Larsen, "Time-of-flight sensors in computer graphics," in *Eurographics (State-of-the-Art Report)*, Mar. 2009, pp. 119–134.

[2] D. Lin, X. Huang, Q. Nguyen, J. Blackburn, C. Rodrigues, T. Huang, M. Do, S. Patel, and W.-M. Hwu, "Parallelization of video processing: From programming models to applications," *IEEE Signal Processing Mag.*, vol. 26, no. 4, pp. 103–112, 2009.

[3] H.-Y. Shum, S.-C. Chan, and S. B. Kang, *Image-Based Rendering*. New York: Springer-Verlag, 2007.

[4] C. L. Zitnick, S. B. Kang, M. Uyttendaele, and R. Szeliski, "High-quality video view interpolation using a layered representation," in *Proc. SIGGRAPH*, 2004, pp. 600–608.

[5] H. T. Nguyen and M. N. Do, "Error analysis for image-based rendering with depth information," *IEEE Trans. Image Processing.*, vol. 18, pp. 703–716, Apr. 2009.

[6] S. Gortler, R. Grzeszczuk, R. Szeliski, and M. Cohen, "The lumigraph," in *Proc. SIGGRAPH*, 1996, pp. 43–54.

[7] M. Levoy and P. Hanrahan, "Light field rendering," in *Proc. SIGGRAPH*, 1996, pp. 31–40.

[8] M. Tanimoto, "Overview of free viewpoint television," *Signal Processing: Image Commun.*, vol. 21, no. 6, pp. 454–461, 2006.

[9] L. McMillan, "An image-based approach to three-dimensional computer graphics," Ph.D. dissertation, Dept. Comp. Sci., Univ. North Carolina, Chapel Hill, 1997.

[10] C. Fehn and R. Pastoor, "Interactive 3DTV—Concepts and key technologies," *Proc. IEEE*, vol. 94, pp. 524–538, Mar. 2006.

[11] P. Kauff, N. Atzpadin, C. Fehn, M. Müller, O. Schreer, A. Smolic, and R. Tanger, "Depth map creation and image-based rendering for advanced 3DTV services providing interoperability and scalability," *Image Commun.*, vol. 22, no. 2, pp. 217–234, 2007.

[12] Y. Mori, N. Fukushima, T. Yendo, T. Fujii, and M. Tanimoto, "View generation with 3D warping using depth information for FTV," *Image Commun.*, vol. 24, no. 1–2, pp. 65–72, 2009.

[13] S. Zinger, L. Do, and P. H. N. de With, "Free-viewpoint depth image-based rendering," *J. Vis. Commun. Image Represent.*, vol. 21, no. 5–6, pp. 533–541, 2010.

[14] L. Yang, T. Yendo, M. P. Tehrani, T. Fujii, and M. Tanimoto, "Artifact reduction using reliability reasoning for image generation of ftv," *J. Vis. Commun. Image Represent.*, vol. 21, no. 5–6, pp. 542–560, 2010.

[15] J. Diebel and S. Thrun, "An application of Markov random fields to range sensing," in *Proc. Conf. Neural Information Processing Systems*, 2005, pp. 291–298.

[16] Q. Yang, R. Yang, J. Davis, and D. Nistér, "Spatial-depth super resolution for range images," in *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, June 2007, pp. 1–8.

[17] E. H. Adelson and J. R. Bergen, "The plenoptic function and the elements of early vision," in *Computational Models of Visual Processing*, M. Landy and J. A. Movshon, Eds. Cambridge, MA: MIT Press, 1991, pp. 3–20.

[18] L. McMillan and G. Bishop, "Plenoptic modeling: An image-based rendering system," in *Proc. SIGGRAPH*, 1995, pp. 39–46.

[19] J.-X. Chai, X. Tong, S.-C. Chan, and H.-Y. Shum, "Plenoptic sampling," in *Proc. SIGGRAPH*, 2000, pp. 307–318.

[20] M. N. Do, D. Marchand-Maillet, and M. Vetterli, "On the bandlimitedness of the plenoptic function," in *Proc. IEEE Int. Conf. Image Processing*, Genova, Italy, Sept. 2005, pp. III–17–20.

[21] Q. H. Nguyen, M. N. Do, and S. J. Patel, "Depth image-based rendering from multiple cameras with 3D propagation algorithm," in *Proc. Int. Conf. Immersive Telecommunications*, Berkeley, CA, May 2009, pp. 1–6. [SP]

Maximum Margin GMM Learning for Facial Expression Recognition

Usman Tariq, Jianchao Yang and Thomas S. Huang

Abstract— **Expression recognition from non-frontal faces is a challenging research area with growing interest. In this paper, we explore discriminative learning of Gaussian Mixture Models for multi-view facial expression recognition. Adopting the BoW model from image categorization, our image descriptors are computed using Soft Vector Quantization based on the Gaussian Mixture Model. We do extensive experiments on recognizing six universal facial expressions from face images with a range of seven pan angles ($-45^\circ \sim +45^\circ$) and five tilt angles ($-30^\circ \sim +30^\circ$) generated from the BU-3dFE facial expression database. Our results show that our approach not only significantly improves the resulting classification rate over unsupervised training but also outperforms the published state-of-the-art results, when combined with Spatial Pyramid Matching.**

I. INTRODUCTION

The increasing applications of facial expression recognition, especially those in Human Computer Interaction, have attracted a great amount of research work in this area in the past decade. However, much of the literature focuses on expression recognition from frontal or near-frontal face images [1], [2]. Expression recognition from non-frontal faces is much more challenging. It is also of more practical utility, since it is not trivial in real applications to always have a frontal face [3]. Nonetheless, there are only a handful of works in the literature working with non-frontal faces. We approach towards this problem by proposing an extension to the very popular image classification framework based upon Bag-of-Words (BoW) models.

In a typical bag-of-words representation in image classification, features (raw patches or some other descriptors) are first sampled from an image [4]. These feature vectors are then quantized into one of the pre-learned *visual words*. This Vector Quantization (VQ) procedure allows us to represent each image by a histogram of such *words*, often known as the bag-of-words representation [4]. Various extensions have been proposed to the BoW model. For instance, features may be softly assigned (Soft Vector Quantization (SVQ)) [5], [6] instead of hard quantization, or one may do some other pooling operations such as "max pooling" over the feature vector assignments [7] instead of "average pooling" in BoW.

In such a framework, learning the visual words (or a descriptor model) and classifier are the two fundamental problems [8]. Most of the existing approaches resort to unsupervised clustering mechanisms to learn the BoW model. The goal here, is to keep sufficient information with which the original feature can be reconstructed with fidelity, which is achieved by minimizing a reconstruction loss. Two such examples are K-means and sparse coding [9]. The criterion can also be to maximize the data likelihood, such as, learning a Gaussian Mixture Model (GMM) for SVQ [5]. However, such schemes may not be optimal if classification is the final goal. A better strategy would be to incorporate the class labels while building such models [8]. This can be done by linking the model parameters to the classification loss function [10], [11], which has shown promising improvements over the unsupervised counterparts.

In this paper, we develop a simple yet effective supervised learning method of GMM for soft vector quantization (SVQ) applied to facial expression recognition. The objective function is smooth and can be easily solved by gradient descent. We term the resulting image features as supervised SVQ (SSVQ) features. Our extensive experiments on the multi-view face images, generated from the BU-3DFE database (Section II) for recognizing expressions, show that our approach significantly improves the resulting classification rate over the unsupervised training counterpart. Our method when, combined with Spatial Pyramid Matching [12], also outperforms the published state-of-art results, which were achieved with a much more complex model.

A. Related Works

Most existing works focus on recognizing six basic expressions that are universal and recognizable across different cultures. These include anger (AN), fear (FE), disgust (DI), sad (SA), happy (HA) and surprise (SU) [2]. Some of the notable works in expression recognition focusing on frontal or near-frontal faces include [13], [14], [15], [16], [17], [18], [19], [20], [21]. For a comprehensive survey of the works in expression recognition please refer to [1] and [22]. In the following, we shall briefly review the papers that concentrate on non-frontal view facial expression recognition along with the papers that deal with supervised training of image descriptor models.

The works on non-frontal view expression recognition can be classified based upon the types of features employed. Some works use geometric features, e.g., Hu et al. [23] and Rudovic et al. [24], [25] use displacement or mapping of manually labeled key points to the neutral or frontal face views of the same subject. Whereas, some researchers extract various low-level features (e.g., SIFT) on pre-labeled landmark points and use them for further processing [2].

This work was partly supported by Intel under the Avascholar project of the Illinois-Intel Parallelism Center (I2PC) and the ONR grant N00014-12-1-0122.

Usman Tariq and Thomas S. Huang are with the Department of Electrical and Computer Engineering, Coordinated Science Laboratory and Beckman Institute for Advanced Science and Technology, University of Illinois at Urbana-Champaign, 405 N. Mathews Ave., Urbana, IL 61801, USA. Email: {utariq2, t-huang1}@illinois.edu

Jianchao Yang is with the Adobe Systems Incorporated, San Jose, CA 95110, USA. Email: jiayang@adobe.com

Some of such works include those by Hu et al. [26] and Zheng et al. [27].

Note that the aforementioned approaches require the facial key-points location information, which needs to be prelabeled. However, in real applications, key-points need to be automatically detected, which is a big challenge itself in the case of non-frontal faces. To address this issue, there have been some attempts which do not require key-point locations; they rather extract dense features on detected faces[1]. The prominent examples in this category include works by Moore and Bowden [28], [29], Zheng et al. [30] and Tang et al. [31]. Moore and Bowden [28], [29] extract LBP features and its variants from non-overlapping patches. While, Zheng et al. [30] and Tang et al. [31] extract dense SIFT features on overlapping image patches. Zheng et al. [30] use regional covariance matrices for the image-level representation. Tang et al. [31], after dense feature extraction, represent the images with super vectors which are learnt based on ergodic hidden markov models (HMM).

It is worthwhile to mention that the BU3D-FE database [32] has become the de-facto standard for works in this area. Many works use five pan angle views rendered from the database (0°, 30°, 45°, 60°and 90°) [26], [23], [27], [28], [29]. However, in real-world situations, we have variations in both pan and tilt angles. Thus, in more recent works [30], [31], people are working with a range of both pan and tilt angles.

The recent years have also seen a growing interest in supervised dictionary (descriptor model) learning. Such approaches may be classified into the following four categories [8].

The first category deals with computing multiple dictionaries. The works by Perronnin [33] and Zhang et al. [34] come under this category. The second type comprises the works which learn dictionaries by modifying an initial dictionary under various criteria, for instance by using mutual information to merge visual words [35]. Another criteria may be the intra-class compactness and inter-class discrimination power [36]. However since only the merging process is considered, one has to begin with a large enough dictionary so that it contains sufficient discriminative power to begin with.

The third category learns a dictionary by working with descriptor-level discrimination. However, as noted in [8], this assumption is quite strong because of the overlap amongst local regions of images from different categories. Some example works in this category are [37], [38], [39] and [40].

The fourth class of algorithms learn the descriptor models/dictionaries with image-level discriminative criteria. Previous example works in this category include [8] and [10]. Our proposed work on SSVQ also falls into this category. However, unlike the work in [8], we employ soft assignment coding to encode an image with a GMM, which can better describe the underlying multi-modal distribution of the local descriptors. Compared with [10], our algorithm is faster, because the coding in [10] needs to solve many Lasso problems. For the supervised training, the differentiable logistic loss function is used, and thus our objective function is differentiable without any conditions, while [8] relies on sub-gradient and [10] has to make some assumption for computing the gradient.

Unlike many previous works, our work neither requires key-point localization nor needs a neutral face. Our proposed method gives significant improvement over unsupervised GMM learning. This work also beats the state-of-the-art performance in the same experimental setting as [30] and [31].

In the following, we first describe the BU-3DFE database used in this work in Section II. Then we present our novel approach for supervised training of GMMs in Section III. Multi-view expression recognition experiments are conducted in Section IV, which is then followed by discussion in Section V. Finally, Section VI concludes our paper.

II. Database

The database used in this work is the publicly available BU3D-FE database [32]. It has 3D face scan and associated texture images of 100 subjects, each performing 6 expressions at four intensity levels. The facial expressions presented in this database include anger (AN), disgust (DI), fear (FE), happy (HA), sad (SA) and surprise (SU). Each subject also has a neutral face scan. Thus, there are a total of 2500 3D faces. The dataset is quite diverse and contains subjects of both gender with various races. Interested readers are referred to [32] for further details.

We used an openGL based tool from the database creators to render multiple views. We generated views with seven pan angles (0°, ±15°, ±30°, ±45°) and five tilt angles (0°, ±15°, ±30°). These views were generated for each subject with 6 expressions and the highest expression intensity, resulting in an image dataset with $5 \times 7 \times \times 6 \times 100 = 21000$ images. Some sample images of a subject in various pan and tilt angles are shown in Figure 1.

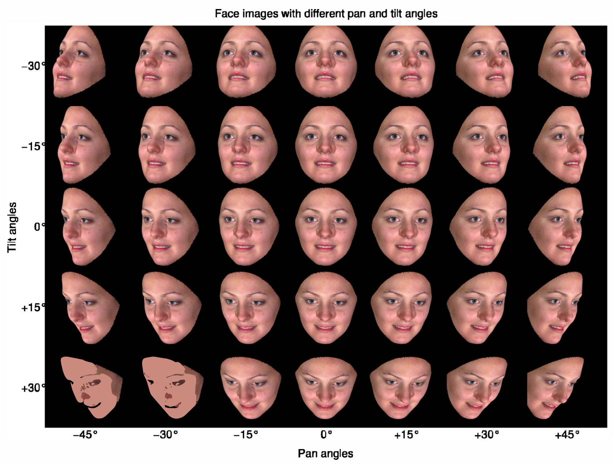


Fig. 1. Rendered facial images of a subject with various pan and tilt angles.

[1]Extraction of dense features essentially implies computing features on an entire image region from overlapping or non-overlapping image patches.

III. Supervised Soft Vector Quantization (SSVQ)

We begin our discussion by outlining soft vector quantization (SVQ). We shall then introduce supervised soft vector quantization (SSVQ) in Section III-B.

A. SVQ

Suppose, $S = \{I^d, c^d\}_{d=1}^{D}$ is a corpus of training images, where $I^d = \{z_1^d, \dots, z_{N_d}^d\}$ and $c_d \in \{-1, 1\}$ are labels. Let $V = \{v_1, v_2, \dots, v_k\}$ be a matrix whose columns represent visual words. In this case, for (hard) VQ, the feature vector z_i^d from the d^{th} image can be represented as a K-dimensional vector ϕ_i^d where,

$$\phi_i^d[k] = \begin{cases} 1 & \text{if } k = \operatorname*{argmin}_m \|z_i^d - v_m\|_2, \\ 0 & \text{otherwise} \end{cases}$$

Now given a GMM (with K components and parameters Θ), the posterior of a p-dimensional feature vector z_i^d is given as,

$$p(z_i^d|\Theta) = \sum_{k=1}^{K} \pi_k \mathcal{N}(z_i^d; \mu_k, \Sigma_k)$$

In SVQ, we may represent the feature vector z_i^d as ϕ_i^d, where,

$$\phi_i^d[k] = \frac{\pi_k \mathcal{N}(z_i^d; \mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(z_i^d; \mu_j, \Sigma_j)} \tag{1}$$

$$= p(k|z_i^d) \tag{2}$$

and $\mathcal{N}(z_i^d; \mu_j, \Sigma_j)$ is the gaussian pdf evaluated at z_i^d.

The image level descriptor can then be represented by a histogram,

$$\Phi^d = \frac{1}{N_d} \sum_{i=1}^{N_d} \phi_i^d \tag{3}$$

B. SSVQ

As outlined earlier, the basic idea here is to reduce the loss function in a classifier, given a training set, by modifying the image descriptor model (which is a GMM in this case). We use logistic regression in our framework. The loss function for L2-regularized logistic regression can be given as [41],

$$L = \frac{1}{2}(w^T w) + A \sum_{d=1}^{D} \log(1 + \exp[-c^d w^T \Phi^d]) \tag{4}$$

Here A, is a scalar, pre-selected by cross-validation on the training set. The derivative of L w.r.t. μ_k can be written as

$$\frac{\partial L}{\partial \mu_k} = A \sum_{d=1}^{D} \frac{-c^d \exp[-c^d w^T \Phi^d]}{1 + \exp[-c^d w^T \Phi^d]} w^T \frac{\partial \Phi^d}{\partial \mu_k} \tag{5}$$

To compute the derivative in equation (5), we need to compute the derivative for each ϕ_i^d,

$$\frac{\partial \Phi^d}{\partial \mu_k} = \frac{1}{N_d} \sum_{i=1}^{N_d} \frac{\partial \phi_i^d}{\partial \mu_k} \tag{6}$$

Note that,

$$\frac{\partial \phi_i^d}{\partial \mu_k} = \left[\dots, \frac{\partial \phi_i^d[k]}{\partial \mu_k}, \dots, \frac{\partial \phi_i^d[m]}{\partial \mu_k}, \dots\right]^T, \text{where } m \neq k \tag{7}$$

Now consider from equation (1),

$$\frac{\partial \phi_i^d[k]}{\partial \mu_k} = \frac{\partial}{\partial \mu_k}\left[\frac{\pi_k \mathcal{N}(z_i^d; \mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(z_i^d; \mu_j, \Sigma_j)}\right] = \frac{\partial p(k|z_i^d)}{\partial \mu_k}$$

After some derivation we get

$$\frac{\partial \phi_i^d[k]}{\partial \mu_k} = (p(k|z_i^d) - (p(k|z_i^d))^2)[z_i^d - \mu_k]^T \Sigma_k^{-1}$$

Using equation (2) we get,

$$\frac{\partial \phi_i^d[k]}{\partial \mu_k} = (\phi_i^d[k] - (\phi_i^d[k])^2)[z_i^d - \mu_k]^T \Sigma_k^{-1} \tag{8}$$

Similarly for the case when $m \neq k$, we have

$$\begin{aligned} \frac{\partial \phi_i^d[m]}{\partial \mu_k} &= \frac{\partial}{\partial \mu_k}\left[\frac{\pi_k \mathcal{N}(z_i^d; \mu_m, \Sigma_m)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(z_i^d; \mu_j, \Sigma_j)}\right] \\ &= \frac{\partial p(m|z_i^d)}{\partial \mu_k} \\ &= -p(k|z_i^d) p(m|z_i^d)[z_i^d - \mu_k]^T \Sigma_k^{-1} \end{aligned}$$

And this essentially implies,

$$\frac{\partial \phi_i^d[m]}{\partial \mu_k} = -\phi_i^d[k]\phi_i^d[m][z_i^d - \mu_k]^T \Sigma_k^{-1} \tag{9}$$

Please note that each of the equations (8) and (9) represents a $1 \times p$ vector, where p is the dimension of μ_k. Thus, $\frac{\partial \phi_i^d}{\partial \mu_k}$ is a $K \times p$ matrix, where K is the number of mixtures in GMM.

Equations (8) and (9) are then used to compute $\frac{\partial \Phi_i^d}{\partial \mu_k}$ in equation (6). Stochastic gradient descent with online learning can be used to update $\mu_k, k \in \{1, \dots, K\}$

$$\mu_k^{(t+1)} = \mu_k^{(t)} - \lambda^{(t)} \left(\frac{\partial L^d}{\partial \mu_k}\right)^T \tag{10}$$

where,

$$\frac{\partial L^d}{\partial \mu_k} = A \frac{-c^d}{1 + \exp(c^d w^T \Phi^d)} w^T \left(\frac{\partial \Phi^d}{\partial \mu_k}\right), \tag{11}$$

$$\lambda^{(t)} = \frac{\lambda^{(t_0)}}{\sqrt{n/N_d + 1}} \tag{12}$$

C. Multi-class SSVQ

Suppose we have a multi-class problem with $M > 2$ classes. Then we can have M regressors, trained in a one-vs-rest (OVR) fashion. The motivation for the OVR setting is that it is efficient, requires lesser computation and it gives comparable performance when compared with other multi-class classifier learning methodologies [41]. Now the regressors may be arranged in an $M \times K$ matrix W. Thus the derivative of the loss function in the multi-class setting for a single training sample can be given as (derived from equation (11)),

$$\frac{\partial L^d}{\partial \mu_k} = A \sum_{i=1}^{M} \frac{-y[i]}{1+\exp(y[i]W[i,:]^T\Phi^d)} W[i,:] \left(\frac{\partial \Phi^d}{\partial \mu_k}\right) \quad (13)$$

where,

$$y[i] = \begin{cases} +1, & \text{if } \Phi^d \in i^{th} \text{ class} \\ -1, & \text{otherwise} \end{cases}$$

$$W[i,:] = \text{ regressor trained on } i^{th} \text{ class vs rest}$$

Stochastic gradient descent is then used with online learning to update μ_k, $k \in \{1,\ldots,K\}$ in a similar fashion as for the binary class problem. Finally, equations (1) and (3) are used to compute the image level descriptors using the new discriminative GMM with updated means, which are then used for training and testing.

The algorithmic framework for the discriminative GMM training in SSVQ is given as follows:

Algorithm 1 Pseudocode for the discriminative GMM training for SSVQ

```
Require: A training database, $S = \{I^d, c^d\}_{d=1}^{D}$, where $I^d = \{z_1^d, \ldots, z_{N_d}^d\}$; $\lambda^{(t_0)}$; a GMM learnt on part of the training set, with parameters $\Theta = \{\pi_1,\ldots,\pi_K;\mu_1,\ldots,\mu_K;\Sigma_1,\ldots,\Sigma_K\}$
1:  $n \leftarrow 0$
2:  for $t = 1$ to $MaxIter$ do
3:    for $i = 1$ to $N_d$ do
4:      $\lambda^{(t)} \leftarrow \frac{\lambda^{(t_0)}}{\sqrt{n/N_d+1}}$
5:      for $k = 1$ to $K$ do
6:        $\mu_k^{(t+1)} \leftarrow \mu_k^{(t)} - \lambda^{(t)} \left(\frac{\partial L^d}{\partial \mu_k}\right)^T$
7:      end for
8:      $n \leftarrow n+1$
9:    end for
10:   Retrain regressor(s)
11: end for
```

IV. Multi-view Expression Recognition Experiments and Results

We do 5-fold subject independent cross validation on multi-view faces with 7 pan angles and 5 tilt angles generated from the BU-3DFE database (21,000 images in total). In each fold, around 80% images are used for training and 20% images for validation. The subjects in the training set do not appear in the validation set. The details of the database can be found in Section II.

The images are scaled so that the maximum image dimension has at most 200 pixels. We then extract SIFT features on dense sampling grid with 3 pixel shifts in horizontal and vertical directions with fixed scale (16×16 pixels) and orientation (0°). The initial GMMs are also learnt in a fold-independent setting, in the traditional unsupervised manner, using the Expectation Maximization algorithm. The GMM for each fold has 1024 mixtures to balance the computational cost and performance. The initial learning rate λ was set to be a small value (1e-6) in stochastic learning. To further speed up the algorithm, we reduce the SIFT feature dimension to 70 with PCA. The supervised GMM parameter updates are only for the means of the Gaussian mixtures, although the covariance matrixes can also be updated in principle. The optimization is run for twelve iterations for early stopping to avoid overfitting. We do not use the Spatial Pyramid (SPM) [12] while doing supervised training. However, we later combine SSVQ and SPM to obtain the final image representation, which achieves the state-of-the-art performance.

Figure 2 shows objective function value (eq. (4)) decreases, averaged for the expression classes, with supervised iterations for each of the folds. The last figure also shows the average of the five training folds. One can notice that the objective value, in general, reduces for all the cases. Figure 3 shows the data log likelihood with GMM as a function of optimization iterations. Interestingly, the log likelihood decreases for the five folds alike, meaning the supervised iterations are moving the model in a direction which makes it more discriminative rather than generative.

Table I shows how the performance increases with supervised training along with comparisons with earlier works in the same experimental setting. Tables II, III and IV respectively show the confusion matrices for SVQ, SSVQ and SSVQ+SPM (with max-pooling).

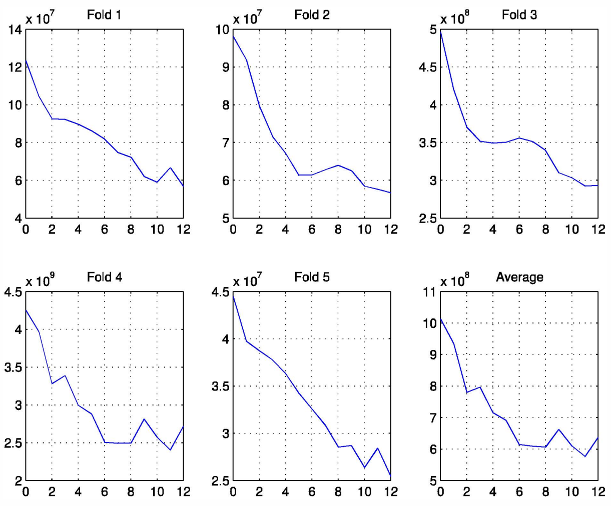


Fig. 2. Decrease in objective function value with the supervised iterations for each of the training folds and average. The horizontal axis represents the number of iterations while the vertical axis represents the objective function value

TABLE I
Comparison in terms of classification rate

Zheng et al. [30]	68.20%
Tang et al. [31]	75.30%
SVQ	63.28%
SSVQ [ours]	**69.81%**
SSVQ+SPM [ours]	**76.16%**
SSVQ+SPM (max pooling) [ours]	**76.34%**

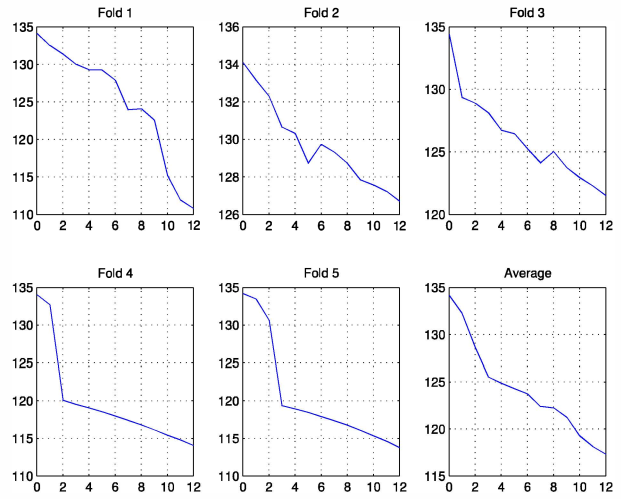


Fig. 3. Effect of supervised iterations on data log likelihood. The horizontal axis represents the number of iterations while the vertical axis represents the data log likelihood

TABLE II

CLASSIFICATION CONFUSION MATRIX FOR RECOGNITION PERFORMANCE WITH SVQ

SVQ		Predicted					
		AN	DI	FE	HA	SA	SU
Ground Truth	AN	**52.6**	14.7	5.9	3.1	21.7	1.9
	DI	13.2	**63.2**	8.1	4.5	6.3	4.7
	FE	8.1	9.8	**47.6**	12.8	14.2	7.5
	HA	3.0	3.5	10.0	**79.5**	2.1	1.9
	SA	23.2	6.9	13.2	3.0	**50.7**	2.9
	SU	1.8	1.9	5.0	2.7	2.5	**86.1**

TABLE III

CLASSIFICATION CONFUSION MATRIX FOR RECOGNITION PERFORMANCE WITH SSVQ

SSVQ		Predicted					
		AN	DI	FE	HA	SA	SU
Ground Truth	AN	**59.4**	11.5	4.0	1.9	21.5	1.7
	DI	10.9	**72.1**	5.2	3.2	5.7	2.9
	FE	7.6	7.7	**52.9**	12.2	11.6	7.9
	HA	0.7	2.2	8.0	**85.8**	1.6	1.7
	SA	22.2	5.1	10.2	1.9	**58.3**	2.3
	SU	0.3	1.9	4.1	1.3	1.9	**90.3**

TABLE IV

CLASSIFICATION CONFUSION MATRIX FOR RECOGNITION PERFORMANCE WITH SSVQ + SPM (MAX-POOLING)

SSVQ+SPM(max-pooling)		Predicted					
		AN	DI	FE	HA	SA	SU
Ground Truth	AN	**67.7**	8.9	4.2	1.3	17.3	0.7
	DI	7.5	**79.3**	5.3	2.8	2.7	2.4
	FE	7.5	7.5	**59.1**	9.9	9.5	6.5
	HA	0.3	0.8	5.6	**91.7**	0.4	1.2
	SA	21.2	3.3	7.1	0.7	**66.0**	1.7
	SU	0.2	1.1	2.5	1.3	0.6	**94.3**

V. DISCUSSION

As shown, our supervised training not only reduces the objective function value on all the folds (Figure 2) but also gives a significant increase in testing classification rate of 6.5% with 12 iterations compared with SVQ (Table I). Note that for our SSVQ, the feature vectors are only 1024 dimensional. It is particularly desirable to have compact feature descriptors when dealing with very large image databases. One can also notice from Tables II and III that supervised training helps all the expression classes from as low as 4.2% for Surprise to as high as 8.9% for Disgust.

The data log likelihood, on the other hand decreases with supervised training, as shown in Figure 3. Although this might be expected since the parameter updates are making the GMMs more discriminative rather than generative, it could also be a sign of overfitting, especially when training examples are limited. In our experiments, we haven't observed overfitting within 12 iterations. But it would be worthwhile to add the data log likelihood as a regularization term in the general form, where the derivation will follow section III-B similarly.

Table IV reports the confusion matrix for the case when SSVQ is combined with SPM and max-pooling [7] (instead of mean pooling in equation (3)) is used. Here, all the classes enjoy much better true recognition rate by incorporating the spatial information. Across all the three confusion matrices, Surprise, Happy and Disgust are the expressions with the highest recognition rates, while Fear is the worst performer. This is consistent with some previous works in facial expression recognition, such as [42]. One can also note from Table I that the combination of SSVQ and SPM, with either mean or max pooling, also achieves state-of-the-art performance, although our model is much simpler than [31].

VI. CONCLUDING REMARKS

Inspired by the BoW model popularly used in image categorization, we propose a novel supervised soft vector quantization model for facial expression recognition. The discriminative training of GMM produces significant performance gain compared with the unsupervised counterpart. Combining the spatial pyramid, our approach achieves state-of-the-art performance on the BU-3dFE facial expression database with simple linear classifier. For future works, we will explore supervised training for full GMM parameters (mean, mixture weights, and covariance matrices) with proper regularization. Incorporation of SPM in supervised training should also be investigated to make each level of SPM more discriminative. The framework is generic and can easily be applied to other classification tasks as well, such as object recognition, face recognition, speaker identification, and audio event recognition.

REFERENCES

[1] Z. Zeng, M. Pantic, G. I. Roisman, and T. S. Huang, "A survey of affect recognition methods: Audio, visual, and spontaneous expressions," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 31, no. 1, pp. 39–58, 2009.

[2] W. Zheng, H. Tang, and T. S. Huang, "Emotion recognition from non-frontal facial images," in *Advances in Emotion Recognition*, A. Konar and A. Chakraborty, Eds. Wiley, 2012, (in press).

[3] U. Tariq, J. Yang, and T. S. Huang, *Multi-view facial expression recognition analysis with generic sparse coding feature*, ser. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2012, vol. 7585 LNCS, no. PART 3.

[4] Y. Zhang, R. Jin, and Z.-H. Zhou, "Understanding bag-of-words model: a statistical framework," *International Journal of Machine Learning and Cybernetics*, vol. 1, pp. 43–52, 2010.

[5] E. Alpaydin, "Soft vector quantization and the em algorithm," *Neural Networks*, vol. 11, no. 3, pp. 467–477, 1998.

[6] L. Liu, L. Wang, and X. Liu, "In defense of soft-assignment coding," in *CVPR*, 2011.

[7] J. Yang, K. Yu, Y. Gong, and T. Huang, "Linear spatial pyramid matching using sparse coding for image classification," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, june 2009, pp. 1794 –1801.

[8] X. C. Lian, Z. Li, B. L. Lu, and L. Zhang, *Max-margin dictionary learning for multiclass image categorization*, ser. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2010, vol. 6314 LNCS, no. PART 4.

[9] H. Lee, A. Battle, R. Raina, and A. Y. Ng, "Efficient sparse coding algorithms," in *In NIPS*. NIPS, 2007, pp. 801–808.

[10] J. Yang, K. Yu, and T. Huang, "Supervised translation-invariant sparse coding," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2010, pp. 3517–3524.

[11] Y.-L. Boureau, F. Bach, Y. LeCun, and J. Ponce, "Learning mid-level features for recognition," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2010.

[12] S. Lazebnik, C. Schmid, and J. Ponce, "Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2, 2006, pp. 2169–2178.

[13] M. Pantic and M. S. Bartlett, "Machine analysis of facial expressions," in *Face Recognition*, K. Delac and M. Grgic, Eds. I-Tech Education and Publishing, 2007, pp. 377–416.

[14] S. Lucey, A. B. Ashraf, and J. F. Cohen, "Investigating spontaneous facial action recognition through aam representations of the face," in *Face Recognition*, K. Delac and M. Grgic, Eds. I-Tech Education and Publishing, 2007, pp. 275–286.

[15] Y. Chang, C. Hu, R. Feris, and M. Turk, "Manifold based analysis of facial expression," *Image and Vision Computing*, vol. 24, no. 6, pp. 605–614, 2006.

[16] M. F. Valstar, H. Gunes, and M. Pantic, "How to distinguish posed from spontaneous smiles using geometric features," in *Proceedings of the 9th International Conference on Multimodal Interfaces, ICMI'07*, 2007, pp. 38–45.

[17] K. Anderson and P. W. McOwan, "A real-time automated system for the recognition of human facial expressions," *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, vol. 36, no. 1, pp. 96–105, 2006.

[18] M. Valstar, M. Pantic, and I. Patras, "Motion history for facial action detection in video," in *Conference Proceedings - IEEE International Conference on Systems, Man and Cybernetics*, vol. 1, 2004, pp. 635–640.

[19] A. Dhall, A. Asthana, R. Goecke, and T. Gedeon, "Emotion recognition using phog and lpq features," in *2011 IEEE International Conference on Automatic Face and Gesture Recognition and Workshops, FG 2011*, 2011, pp. 878–883.

[20] Z. Zeng, J. Tu, B. M. Pianfetti Jr., and T. S. Huang, "Audio-visual affective expression recognition through multistream fused hmm," *IEEE Transactions on Multimedia*, vol. 10, no. 4, pp. 570–577, 2008.

[21] Y. Tian, T. Kanade, and J. F. Conn, "Recognizing action units for facial expression analysis," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 23, no. 2, pp. 97–115, 2001.

[22] B. Fasel and J. Luettin, "Automatic facial expression analysis: A survey," *Pattern Recognition*, vol. 36, no. 1, pp. 259–275, 2003.

[23] Y. Hu, Z. Zeng, L. Yin, X. Wei, J. Tu, and T. S. Huang, "A study of non-frontal-view facial expressions recognition," in *Proceedings - International Conference on Pattern Recognition*, 2008.

[24] O. Rudovic, I. Patras, and M. Pantic, "Regression-based multi-view facial expression recognition," in *Proceedings - International Conference on Pattern Recognition*, 2010, pp. 4121–4124.

[25] ——, *Coupled Gaussian process regression for pose-invariant facial expression recognition*, ser. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2010, vol. 6312 LNCS, no. PART 2.

[26] Y. Hu, Z. Zeng, L. Yin, X. Wei, X. Zhou, and T. S. Huang, "Multi-view facial expression recognition," in *2008 8th IEEE International Conference on Automatic Face and Gesture Recognition, FG 2008*, 2008.

[27] W. Zheng, H. Tang, Z. Lin, and T. S. Huang, "A novel approach to expression recognition from non-frontal face images," in *Proceedings of the IEEE International Conference on Computer Vision*, 2009, pp. 1901–1908.

[28] S. Moore and R. Bowden, "The effects of pose on facial expression recognition," in *British Machine Vision Conference*, 2009.

[29] ——, "Local binary patterns for multi-view facial expression recognition," *Computer Vision and Image Understanding*, vol. 115, no. 4, pp. 541–558, 2011.

[30] W. Zheng, H. Tang, Z. Lin, and T. S. Huang, *Emotion recognition from arbitrary view facial images*, ser. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2010, vol. 6316 LNCS, no. PART 6.

[31] H. Tang, M. Hasegawa-Johnson, and T. Huang, "Non-frontal view facial expression recognition based on ergodic hidden markov model supervectors," in *2010 IEEE International Conference on Multimedia and Expo, ICME 2010*, 2010, pp. 1202–1207.

[32] L. Yin, X. Wei, Y. Sun, J. Wang, and M. J. Rosato, "A 3d facial expression database for facial behavior research," in *FGR 2006: Proceedings of the 7th International Conference on Automatic Face and Gesture Recognition*, vol. 2006, 2006, pp. 211–216.

[33] F. Perronnin, "Universal and adapted vocabularies for generic visual categorization," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 30, no. 7, pp. 1243–1256, 2008.

[34] W. Zhang, A. Surve, X. Fern, and T. Dietterich, "Learning non-redundant codebooks for classifying complex objects," in *Proceedings of the 26th International Conference On Machine Learning, ICML 2009*, 2009, pp. 1241–1248.

[35] B. Fulkerson, A. Vedaldi, and S. Soatto, *Localizing objects with smart dictionaries*, ser. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2008, vol. 5302 LNCS, no. PART 1.

[36] J. Winn, A. Criminisi, and T. Minka, "Object categorization by learned universal visual dictionary," in *Proceedings of the IEEE International Conference on Computer Vision*, vol. II, 2005, pp. 1800–1807.

[37] S. Lazebnik and M. Raginsky, "Supervised learning of quantizer codebooks by information loss minimization," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 31, no. 7, pp. 1294–1309, 2009.

[38] J. Shotton, M. Johnson, and R. Cipolla, "Semantic texton forests for image categorization and segmentation," in *26th IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, 2008.

[39] J. Mairal, F. Bach, J. Ponce, G. Sapiro, and A. Zisserman, "Supervised dictionary learning," in *Advances in Neural Information Processing Systems 21 - Proceedings of the 2008 Conference*, 2009, pp. 1033–1040.

[40] ——, "Discriminative learned dictionaries for local image analysis," in *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, june 2008, pp. 1–8.

[41] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin, "LIBLINEAR: A library for large linear classification," *Journal of Machine Learning Research*, vol. 9, pp. 1871–1874, 2008.

[42] U. Tariq, K.-H. Lin, Z. Li, X. Zhou, Z. Wang, V. Le, T. Huang, X. Lv, and T. Han, "Recognizing emotions from an ensemble of features," *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, vol. 42, no. 4, pp. 1017 –1026, aug. 2012.

Expression Recognition from 3D Dynamic Faces using Robust Spatio-temporal Shape Features

Vuong Le, Hao Tang and Thomas S. Huang
Beckman Institute for Advanced Science and Technology
Department of Electrical and Computer Engineering, University of Illinois at Urbana - Champaign
405 North Mathews Avenue, Urbana, IL 61801, USA

***Abstract*—This paper proposes a new method for comparing 3D facial shapes using facial level curves. The pair- and segment-wise distances between the level curves comprise the spatio-temporal features for expression recognition from 3D dynamic faces. The paper further introduces universal background modeling and maximum a posteriori adaptation for hidden Markov models, leading to a decision boundary focus classification algorithm. Both techniques, when combined, yield a high overall recognition accuracy of 92.22% on the BU-4DFE database in our preliminary experiments. Noticeably, our feature extraction method is very efficient, requiring simple preprocessing, and robust to variations of the input data quality.**

I. Introduction

Expression recognition from 3D facial data is a new and interesting problem. Several baseline methods have been introduced and gave very promising results [1] and [2]. In another direction, research has also been done on studying the dynamics of facial expressions in 2D images [3], proving that looking at sequences of face instances can help improve the recognition performance. These initiatives give us the awareness that facial expressions are highly dynamical processes in the 3D space, and therefore observing the state transitions of 3D faces could be a crucial clue to the investigation of the inner state of human subjects.

Recently, this direction is getting more attention with the introduction of appropriate databases such as the BU-4DFE database developed at Binghamton University [4]. It is also inspired by the revolution of inexpensive acquisition devices such as the consumer 3D cameras. One of the challenges for expression recognition using data from these low-end devices is that most of them produce noisy and low resolution depth images. In such kind of condition, expression recognition using traditional methods based on tracked facial feature points can be sensitive to the noise. Moreover, tracking facial feature points by itself would require more computations and thus make the systems harder to satisfy the real-time requirement. This situation shows the urgent need for a robust and efficient feature representation for 3D facial shapes together with a high performance classification algorithm to exploit the features for expression recognition.

In this work, we propose to use a facial level curves based representation of 3D faces for expression recognition. A similar representation was applied to face recognition by Samir and colleagues [5]. To the best of our knowledge, it has not been used for expression recognition. Besides being powerful and robust, one advantage of this representation is that the level curves can be extracted directly from depth images with some simple preprocessing steps.

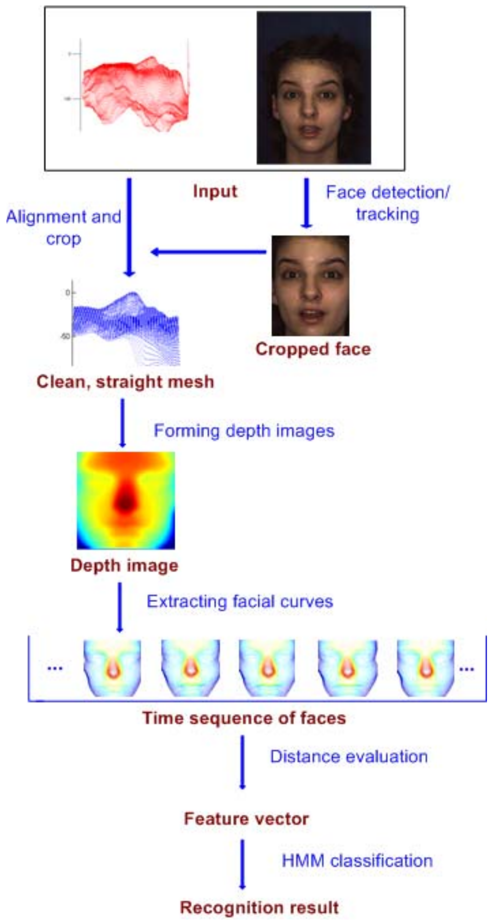


Fig. 1: Framework of expression recognition from 3D dynamic faces using facial level curves

On top of the representation, we introduce a novel method to measure the distances between the corresponding level curves of two 3D facial shapes, which are then used as spatio-temporal features for expression recognition. The method is based on the Chamfer distances of normalized segments partitioned from the level curves by an arclength parameterized function. This feature extraction method does not require feature point localization, and can deal with low resolution depth images. These characteristics make the method very friendly with low-end acquisition devices and the requirement of fast processing.

We further introduce universal background modeling and maximum a posteriori adaptation for hidden Markov models (HMMs), leading to an HMM-based decision boundary focus classification algorithm. Combined with the proposed feature extraction method, this classification algorithm yields a high overall recognition accuracy of 92.22% on the BU-4DFE database in our preliminary experiments. The overall pipeline of our expression recognition framework is depicted in Fig. 1.

This paper is organized as follows. Section II reviews the related work. The BU-4DFE database used in our experiments is introduced in Section III. Section IV describes the details of the proposed feature extraction method, and Section V the details of the proposed HMM-based decision boundary focus classification algorithm. In Section VI, the experiments and results are presented. Finally, Section VII draws the conclusion and discusses the future work.

II. Related Work

Shape representation and feature extraction for the analysis of 3D facial data have gained increased attention recently. In this section, we review some existing approaches that use 3D face models for facial expression recognition and some shape representation methods related to ours.

In an intuitive way of analyzing facial expressions, several works, such as [3], [6] and [7], follow the traditional approach of using 3D face models to estimate the movements of the facial feature points. These features are related to the action units (AUs) and their movements control the emotional states of the subject.

In a different way, a dense correspondence frame is built based on a set of predefined landmarks. The arrangement of the corresponded dense point set is used as the feature for classification. Mpiperis and colleagues, in [1], use a correspondence frame built by a subdivision surface model using a set of predefined feature points. The members of the 3D correspondence frame are then used as raw features in a bilinear model which helps separate the identity and expression factors in 3D static data. A tensor-like model is built as PCA subspaces, both among people and across the expressions. With such a relatively simple model, they can afford to utilize the dense features of thousands of dimensions and give impressive recognition results.

A dense correspondence can provide very informative features of the facial shape but will face the curse of dimensionality. Extracting the geometrical shape features at some important locations such as the convex parts, high curvatures areas, and saddle points may reduce the vector size while still keeping the representation robust. One example is the primitive label distribution used in a recent work by Yin et al. [2]. Whilst the feature definitions of the primitive label distribution are intuitively meaningful and shown to work well on studio data, the computation of curvatures in general involves numerical approximation of second derivatives and may be susceptible to observation noise.

Up to the present, most of the 3D face expression recognition works are based on static data. In a pioneer work that exploits dynamic data [8], Sun and Yin extract sophisticated features of geometric labeling and use 2D HMMs as classifiers for recognizing expressions from 3D face model sequences. Their method has led to very promising recognition performance.

In the aspect of 3D facial shape representations, Samir et al., in [9], develop an intrinsic mathematical and statistical framework for analyzing facial shapes for face recognition based on facial level curves. In that framework, the shape representation is similar to ours. However, the distance function used to extract and compare the level curves are significantly different. In Samir et al.'s work, they use the geodesic distances based on an angle function, which are shown to be relatively invariant to expressions [10] and therefore are more suitable for face recognition. Instead, in our work, we propose the localized Chamfer distances which are correlated with expression changes.

III. Database Description

In our experiments, we utilize the BU-4DFE database [4], which captures the dynamics of 3D expressive facial surfaces over a time period. The BU-4DFE database was created at the State University of New York at Binghamton by Yin et al. This database consists of 101 subjects, including 58 females and 43 males of different ethnicity: Asian, Black, Hispanic/Latino, and White. For each subject, there are six 3D face model sequences corresponding to the six fundamental facial expressions (namely anger, disgust, happiness, fear, sadness, and surprise). In each sequence, the face models are captured at the rate of 25 models per second. Each face model in a sequence contains the shape and texture of the subject during an expression period. All expression periods typically last about 4 seconds (approximately 100 frames).

In total, the database contains 606 face model sequences, that is more than 60600 face models. Each face model consists of a cloud of points containing around 35000 vertices and a texture image with a resolution of 1040×1329.

The dynamic characteristics of this database is crucial in describing the intermediate period between the peak of the emotions and the neutral state, which are apparently very important for expression recognition. Our algorithm takes advantage of this property, and the database turns out to be quite appropriate to verify our algorithm.

IV. Facial Level Curves Based Representation of 3D Shapes

Facial level curves are defined to be the planar curves constituted from a facial surface, which are created by extracting the points with the same values of a distance function to a center points [5]. In our approach, the distance function is chosen to be the height of the points in a normalized coordinate system. In this section, the method for extracting facial level curves

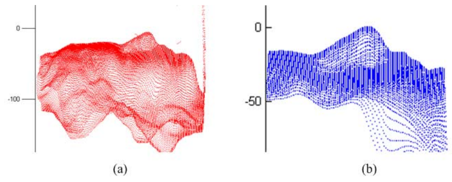


Fig. 2: Facial surface alignment. (a): Raw shape, (b): Aligned shape

and the method for comparing level curves and sets of level curves are described in detail.

A. Data preprocessing

Using the height information as the distance function has the advantages of fast processing time, being convenient with the input of range images and sensitive to expression changes. The most important challenge of using this function is the fact that it is not invariant to viewpoint changes. Therefore, in order to extract useful features with this distance, we need to perform several steps of preprocessing, namely pose normalization and face area extraction, to make sure the features extracted are unbiased and noise is not introduced by different viewpoints. Moreover, normalization will help us reduce the effect of identity variations among different subjects.

The face area and location of the nose and two eyes on a color image are located by the Pittpatt face detection and tracking library [11], [12]. This area is back projected to the 3D model to located the facial points in the 3D space. The outlier points are cut off from the mesh. The clean point cloud is then aligned so that it will be frontal when looking down from the z axis and the nose tip (the highest point in a frontal face) is at the coordinate center. The original and aligned faces are compared in Fig. 2.

In the next step of normalization, the face model is scaled down by suitable ratios for all of the three dimensions and stored in a range image. The purpose of this step is to squeeze a face model to make it fit in a fixed-size cube which will help reduce the variation in the geometry of the face. The range image for every face has now the same horizontal and vertical sizes. The depth values are scaled so that the depth of the nose tip has the value 1 and the depth of the eyes have the value 0.4. This will help provide the correspondence of the levels across different surfaces. The depth images are then ready to be fed into the main feature extraction process. A sample result of the preprocessing step is show in Fig. 3(a).

B. Shape representation

In our framework, a facial shape is represented as a collection of planar curves obtained by having a plane parallel to the xy plane moving down along the z axis to cut the aligned surface. At each level, the intersection of the plane and the surface is a planar curve. In the experiments, this process is done by going through various height levels. At the level of height h, we find all the points with a height in the range of $(h-\delta, h+\delta)$, with δ being some predefined constant. This set of points form a band with the thickness of 2δ. Projecting this

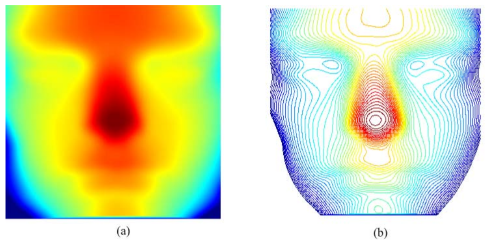


Fig. 3: Range image formation and facial level curves extraction in Matlab's jet colormap. (a): Normalized range image (b): Extracted facial level curves

band to the xy plane and finding a contour through the points, we have a planar facial curve. A sample of the extracted curves of a sample face is shown in Fig. 3(b). The curves are then stored as binary images which has value one at curve points and zero at others.

Based on the level curve representation of facial surfaces, the deformation of the face over time is extracted by comparing the faces in two consecutive frames in the sequence. The method of defining and extracting distances between curves and faces are described in the next section.

C. Localized Chamfer distances for curve and surface comparison

On top of the facial curves based representation method, several ways to estimate the deformation of 3D faces are introduced. In [13], Klassen et al. propose to use geodesics in the preshape space to compare 3D facial curves using direction functions or curvature functions. Using a similar shape representation as the one used in our work, Samir et al. compare two curves by finding the shortest geodesic path between the curves and use the L2 norm of the curves as the distance between the two faces [9].

In our work, with the objectives of building the features which can be extracted efficiently while at the same time exposing the deformation of the curves at local locations instead of the global deformation of the entire closed curves. This inspires us to segregate each facial curve into a set of small segments at corresponded locations and extract the deformation of each segment over time. With that intention, we propose to use the Chamfer distances of curve segments partitioned by a set of arclength boundaries which we call *localized Chamfer distance*. The detail of the partition process will be addressed next.

Given that a planar curve is simple (i.e. with no self crossings) and closed, it can be represented by an *arclength parameterized function* [13]:

$$\alpha(s) : [0, 2\pi] \rightarrow \mathbb{R}^2 \tag{1}$$

.

This function gives the 2D location of the curve points given the angle between the line connecting them to the coordinate center and the Ox axis (i.e. the corresponding arc length of the unit circle).

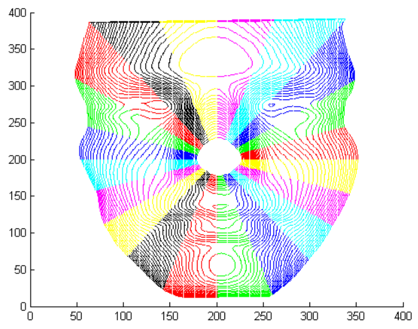


Fig. 4: The partitions of facial curves into bins by the arclength parameterized function. In this case, there are 20 segments made from a curve

The partition of the curve into n curve segments is done by breaking the function into n functions with the supports to be subsets of the main function's support. Each subset corresponds to an equal range of arclength s. The curve segment number i is defined by

$$\alpha_i(s) : [i\frac{2\pi}{n}, (i+1)\frac{2\pi}{n}) \rightarrow \mathbb{R}^2. \quad (2)$$

This segment partitioning process is equivalent to dividing a binary image of a curve into n sectors, with each sector covering an angle of $2\pi/n$ radians. The segments are the part of curves lying in the corresponding sectors. The illustration of this process in the case of 20 bins is depict in Fig. 4.

In our current method, because we have only one center point located at the nose tip, the curves are not always simple and closed, such as the ones drawn in Fig.5. Therefore, the arclength parameterized map is not stricly a function. However, we can still use the map in the same manner to obtain the curve partition with the similar intuitive meaning. The ultimate way to solve this problem is to use many center points and choose the suitable ones for different parts of curves so that they will satisfy the closed and simple requirements. This improvement will be implemented in the future work.

The comparison between two same-level curves are done by finding the Chamfer distances of the n pairs of curve segments and stack them up to form a distance vector. This vector represents the deformation intensity of the level curve over time in different directions. The distance vectors of the curves are further stacked to represent the deformation of the face. Therefore, at each frame transition we have a raw feature vector of $n \times l$ dimensions where l is the number of curves used. A sample of two curves being compared is depicted in Fig. 5

This representation indirectly implies a reference frame for correspondence. As the faces are now not represented as a set of points, this type of correspondence does not show the displacement of one particular point on the facial skin. Instead, comparing two corresponding curve segments of the same level and same sector from two consecutive frames shows how

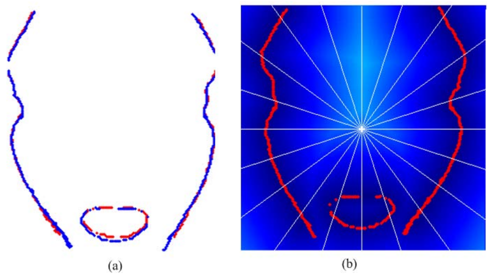


Fig. 5: Comparison between two sample curves. (a): two curves superimposed on each other(b): one curve superimposed on the other's distance transform image. The white lines indicate partition boundaries

the facial shape deforms at that relative location in space over time. Hence, this representation is able to express the shape deformation in the spatio-temporal realm.

In practical experiments, this representation may face the curse of dimensionality, due to the size of the distance vectors. Also, it only represents the deformation at one particular point in time whereas the current bigger context of the dynamics can make the feature more expressive. To alleviate those challenges, we utilize several steps of distance metric learning on the features which are detailed in the next subsection.

D. *Distance metric learning*

In our framework, we perform context expansion of the feature vectors. For each frame, we form an augmented vector by stacking up the feature vectors of the previous, current and future frames. This augmented vector is called the context expanded feature vector of current frame. Context expansion takes into account the temporal dynamics of the feature vectors, and is demonstrated to be an important stage for expression recognition from 3D dynamic faces. After context expansion, the feature vector has the size of $3 \times n \times l$ which span a space of thousands of dimensions. To reduce the dimensionality we use principal component analysis (PCA), followed by linear discriminant analysis (LDA). After this step, the feature vectors are ready to be used in the main classification algorithm which is described in the next section.

V. Decision Boundary Focus Classification with Hidden Markov Models

In this work, we adopt hidden Markov models (HMMs) [14] for facial expression classification. An HMM is a doubly stochastic process which consists of an underlying discrete random process possessing the Markov property (namely a Markov chain having a finite number of states) and an observed random process generated by a set of probabilistic functions of the underlying Markov chain, one of which is associated with each state of the Markov chain. At a discrete time instant, the HMM is assumed to be at a certain state, and an observation is generated by the probabilistic function associated with that particular state. The underlying Markov chain changes its state at every time instant according to some state transition probability distribution. Note that within an HMM, it is only the observations that are seen by an observer, who does not have any direct knowledge of the underlying state sequence that generated these observations. HMMs have been proven to be a natural and effective probabilistic models for sequential data such as audio, speech and video.

However, due to their generative nature, HMMs may not perform as well as discriminative models for the classification purpose when there is insufficient training data. To overcome this difficulty, we introduce a strategy based on the maximum a posteriori (MAP) adaptation of a universal background model (UBM) to generate the individual states of the class-dependent HMMs. For reasons that will be clear shortly, this strategy enhances the discriminatory power of the class-dependent HMMs by focusing on the decision boundaries when applied to classification tasks.

A. HMM-based facial expression recognition

Without loss of generality, we assume that a 3D facial expression model sequence is effectively represented by a sequence of observation vectors $O = \mathbf{o}_1\mathbf{o}_2 \cdots \mathbf{o}_T$. The joint probability distribution of O is a generative model which could have generated O. Let $p(O|c)$ denote the class-dependent model for facial expression c. The Bayesian or minimum-error-rate classification rule [15] is given by

$$c = \underset{c}{\operatorname{argmax}}\, p(c|O) = \underset{c}{\operatorname{argmax}}\, p(O|c)p(c) \quad (3)$$

where $p(c)$ is the prior probability of facial expression c. In this work, $p(O|c)$ is modeled by an HMM, namely

$$p(O|c) = \sum_{q_1 q_2 \cdots q_T} \left[\pi^c_{q_1} b^c_{q_1}(\mathbf{o}_1) \prod_{t=2}^{T} a^c_{q_{t-1}q_t} b^c_{q_t}(\mathbf{o}_t) \right] \quad (4)$$

The above formulas form the basis of HMM-based facial expression recognition.

B. The Baum-Welch learning algorithm

To perform HMM-based facial expression recognition, we need to learn a class-dependent HMM for every facial expression. An HMM is completely determined by its parameters $\lambda = \{A, B, \Pi\}$. Here, A is the state transition probability matrix whose entries, $a_{ij} = P(q_t = S_j | q_{t-1} = S_i)$, $1 \le i, j \le N$, specify the probabilities of transition from state S_i to state S_j at time t. B is the state emission probability matrix whose entries, $b_{jk} = P(o_t = v_k | q_t = S_j)$, $1 \le j \le N, 1 \le k \le M$, specify the probabilities of emitting an observation symbol v_k given that the model is in state S_j at time t. Π is the initial state probability matrix whose entries, $\pi_i = P(q_1 = S_i)$, $1 \le i \le N$, specify the probabilities of the model being initially in state S_i. For the case of continuous observations, the entries of the state emission probability matrix are given by continuous probability density functions, namely $b_j(o_t) = P(o_t | q_t = S_j)$, $1 \le j \le N$. One important class of continuous probability density functions widely used for the state emission densities of the continuous-observation HMM is the Gaussian mixture density functions of the form

$$\begin{aligned} b_j(o_t) &= \sum_{k=1}^{M} c_{jk} N(o_t | \mu_{jk}, \Sigma_{jk}) \\ & \quad 1 \le j \le N, 1 \le k \le M \end{aligned} \quad (5)$$

where M is the number of Gaussian components, c_{jk} is the k^{th} mixture weight, and $N(o_t|\mu_{jk}, \Sigma_{jk})$ is a multivariate Gaussian density function with mean vector μ_{jk} and covariance matrix Σ_{jk}. Since $b_j(o_t)$ is a valid probability density function, the following constraints must hold:

$$c_{jk} \ge 0, \quad 1 \le j \le N, \quad 1 \le k \le M \quad (6)$$

$$\sum_{k=1}^{M} c_{jk} = 1, \quad 1 \le j \le N \quad (7)$$

The Baum-Welch algorithm [14] for learning the parameters of an HMM, λ, given an observation sequence, $O = o_1 o_2 \cdots o_T$, is an iterative re-estimation procedure that increases the log likelihood $P(O|\lambda)$ monotonically. It is based on an efficient algorithm known as the forward-backward algorithm. In the forward algorithm, a "forward" variable, $\alpha_t(i)$, is defined as the probability of the partial observation sequence up to time t and state S_i being occupied at time t

$$\begin{aligned} \alpha_t(i) &= P(o_1 o_2 \cdots o_t, q_t = S_i | \lambda) \\ & \quad 1 \le i \le N, 1 \le t \le T \end{aligned} \quad (8)$$

The following iterative formulas are used to compute the α's efficiently:

$$\alpha_1(i) = \pi_i b_i(o_1), \quad i = 1, 2, \cdots, N \quad (9)$$

$$\begin{aligned} \alpha_t(j) &= \left[\sum_{i=1}^{N} \alpha_{t-1}(i) a_{ij} \right] b_j(o_t) \\ & \quad j = 1, 2, \cdots, N, t = 2, 3, \cdots, T \end{aligned} \quad (10)$$

In the backward algorithm, a "backward" variable, $\beta_t(i)$, is defined as the probability of the partial observation sequence from time $t+1$ to T given that the state S_i is occupied at time t

$$\begin{aligned} \beta_t(i) &= P(o_{t+1} o_{t+2} \cdots o_T | q_t = S_i, \lambda) \\ & \quad 1 \le i \le N, 1 \le t \le T \end{aligned} \quad (11)$$

Likewise, the following iterative formulas are used to efficiently compute the β's:

$$\beta_T(j) = 1, \quad j = 1, 2, \cdots, N \quad (12)$$

$$\begin{aligned} \beta_t(i) &= \sum_{j=1}^{N} a_{ij} b_j(o_{t+1}) \beta_{t+1}(j) \\ & \quad i = 1, 2, \cdots, N, t = T-1, T-2, \cdots, 1 \end{aligned} \quad (13)$$

Starting with random initialization (or initialization with a smarter scheme) of the model parameters λ, the Baum-Welch algorithm proceeds as follows:

(1) Re-estimation of the new model parameters $\hat{\lambda}$:

$$\hat{\pi}_i = \frac{\alpha_1(i)\beta_1(i)}{\sum_{j=1}^{N}\alpha_1(j)\beta_1(j)} \quad (14)$$

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T-1}\alpha_t(i)a_{ij}b_j(o_{t+1})\beta_{t+1}(j)}{\sum_{t=1}^{T-1}\alpha_t(i)\beta_t(i)} \quad (15)$$

$$\hat{c}_{jk} = \frac{\sum_{t=1}^{T}\gamma_t(j,k)}{\sum_{t=1}^{T}\sum_{k=1}^{M}\gamma_t(j,k)} \quad (16)$$

$$\hat{\mu}_{jk} = \frac{\sum_{t=1}^{T}\gamma_t(j,k)o_t}{\sum_{t=1}^{T}\gamma_t(j,k)} \quad (17)$$

$$\hat{\Sigma}_{jk} = \frac{\sum_{t=1}^{T}\gamma_t(j,k)(o_t-\hat{\mu}_{jk})(o_t-\hat{\mu}_{jk})^T}{\sum_{t=1}^{T}\gamma_t(j,k)} \quad (18)$$

where $1 \le i,j \le N, 1 \le k \le M$, and

$$\gamma_t(j,k) = \frac{\alpha_t(j)\beta_t(j)}{\sum_{i=1}^{N}\alpha_t(i)\beta_t(i)} \frac{c_{jk}N(o_t|\mu_{jk},\Sigma_{jk})}{\sum_{m=1}^{M}c_{jm}N(o_t|\mu_{jm},\Sigma_{jm})} \quad (19)$$

(2) Test of convergence: If $\|\hat{\lambda}-\lambda\|_2 < \theta$ (a threshold), the Baum-Welch algorithm converges. Otherwise, set $\lambda = \hat{\lambda}$ and go to step (1).

Note that in the above Baum-Welch re-estimation formulas, $\gamma_t(j,k)$ can be viewed as the posterior probability that the observation o_t was generated from state S_j and accounted for by the k^{th} component of the Gaussian mixture density of state S_j, given the current model parameter λ.

C. Universal background modeling and maximum a posteriori adaptation

The Baum-Welch algorithm is a maximum likelihood learning technique for learning the model parameters of an HMM given a set of training observation sequences. Given a sufficient amount of training data, the Baum-Welch algorithm can be used to learn high-quality HMMs. However, in situations where there is insufficient training data available for learning an HMM, the Baum-Welch algorithm is likely to lead to a poorly estimated model. This is known as the over-fitting problem, which is a general property of maximum likelihood estimation techniques. To overcome this difficulty, we introduce universal background modeling. Universal background modeling was originally proposed for biometric verification systems which use a universal background model (UBM) to represent the general and person-independent feature characteristics to be compared against a model of person-specific feature characteristics when making an accept or reject decision. For example, in a speaker verification system, the UBM is a speaker-independent Gaussian mixture model (GMM) trained with speech samples that come from a large set of speakers, which is used when training the speaker-specific model by acting as the prior model in MAP parameter estimation [16]. Under the context of HMMs, the UBM is obtained as follows. We first pool the separate training data for learning the individual HMMs together to form an aggregated training data set. This aggregated training data set is normally large enough that we can assume that it is likely to capture sufficient variations in the population of the data. We then use the Baum-Welch algorithm to learn a single global HMM based on the aggregated training data set. We call this single global HMM learned on the aggregated training data set the UBM, as this single global HMM is supposed to represent the probability distribution of the observations drawn from the population of the data. The UBM is in general a well-trained and robust model, from which we can derive particular individual HMMs specific to small amounts of training data using the MAP adaptation technique for HMMs, as described in detail next. An HMM that is adapted from the well-trained UBM with a small amount of training data is proven to be far more robust than an HMM that is learned directly with the same small amount of training data using the Baum-Welch algorithm.

The rational behind universal background modeling and adaptation is as follows: In the Bayesian or minimum error classification rule, an optimal decision boundary is formed by the posterior probability distributions of two classes, which may be computed from the class-dependent likelihood probability distributions of the two classes, respectively. There may be a lot of fine structures in the class-dependent likelihood probability distribution of either class, and normally we require a lot of training data to learn these fine structures. However, as far as the classification problem is concerned, only the regions of the class-dependent likelihood probability distributions near the decision boundary are important. The fine structures of the class-dependent likelihood probability distributions which are away from the decision boundary are of no use. Therefore, it is a waste of the precious training data to try to learn these fine structures all over, and more disastrously, the fine structures (both near and away from the decision boundary) will never be properly learned if the available training data is insufficient. The introduction of universal background modeling allows us to learn the fine structures irrelevant to the classification problem using a large aggregated training data set, and focus on the regions near the decision boundary by learning the fine structures within these regions using small amounts of training data.

The concept of universal background modeling is related to the more elegant Bayesian learning theory [17]. In Bayesian learning, a prior probability distribution is imposed on the model parameters, which will be adjusted as more and more evidence is present. The UBM may be considered as a prior model corresponding to the prior probability distribution of the model parameters. Bayesian learning is a powerful learning paradigm which has many advantages over maximum likelihood learning. However, it is computationally very expensive. Universal background modeling serves as a good trade-off point between full Bayesian learning and maximum likelihood learning.

In the Baum-Welch algorithm, when we re-estimate the parameters of the Gaussian mixture state emission densities, $\hat{c}_{jk}, \hat{\mu}_{jk}, \hat{\Sigma}_{jk}, 1 \le j \le N, 1 \le k \le M$, instead of starting with randomly initialized parameters, we start with a UBM with parameters $\bar{c}_{jk}, \bar{\mu}_{jk}, \bar{\Sigma}_{jk}, 1 \le j \le N, 1 \le k \le M$. In the following, we will drop the index ranges to avoid cluttering the equations by assuming that $1 \le j \le N, 1 \le k \le M, 1 \le t \le T$. For each observation vector o_t, we compute the posterior probability of o_t being generated by state S_j and Gaussian component k of the UBM

$$\gamma_t(j,k) = \frac{\alpha_t(j)\beta_t(j)}{\sum_{i=1}^{N}\alpha_t(i)\beta_t(i)} \frac{\bar{c}_{jk} N(o_t|\bar{\mu}_{jk}, \bar{\Sigma}_{jk})}{\sum_{m=1}^{M}\bar{c}_{jm} N(o_t|\bar{\mu}_{jm}, \bar{\Sigma}_{jm})} \tag{20}$$

Based on these posterior probabilities, we compute the data sufficient statistics

$$n(j,k) = \sum_{t=1}^{T}\gamma_t(j,k) \tag{21}$$

$$\mu(j,k) = \frac{1}{n(j,k)}\sum_{t=1}^{T}\gamma_t(j,k)o_t \tag{22}$$

$$S(j,k) = \frac{1}{n(j,k)}\sum_{t=1}^{T}\gamma_t(j,k)o_t o_t^T \tag{23}$$

where $n(j,k)$ can be interpreted as the (fractional) number of observation vectors for which the state S_j and Gaussian component k of the UBM are responsible. Notice that the model sufficient statistics given by the UBM are

$$\tilde{n}(j,k) = T\bar{c}_{jk} \tag{24}$$

$$\tilde{\mu}(j,k) = \bar{\mu}_{jk} \tag{25}$$

$$\tilde{S}(j,k) = \bar{\Sigma}_{jk} + \bar{\mu}_{jk}\bar{\mu}_{jk}^T \tag{26}$$

The MAP adaptation technique generates a new set of sufficient statistics by interpolating the data and model sufficient statistics, namely

$$\hat{n}(j,k) = \rho^{(1)}n(j,k) + (1-\rho^{(1)})\tilde{n}(j,k) \tag{27}$$

$$\hat{\mu}(j,k) = \rho^{(2)}\mu(j,k) + (1-\rho^{(2)})\tilde{\mu}(j,k) \tag{28}$$

$$\hat{S}(j,k) = \rho^{(3)}S(j,k) + (1-\rho^{(3)})\tilde{S}(j,k) \tag{29}$$

where the interpolation coefficients, $\rho^{(1)}, \rho^{(2)}, \rho^{(3)}$, are smartly adaptive to the amount of available training data according to the following empirical formula

$$\rho^{(l)} = \frac{n(j,k)}{n(j,k) + r^{(l)}}, \quad l = 1,2,3 \tag{30}$$

with $r^{(l)}$ being a tunable constant specified by the user.

The new set of sufficient statistics is now used for re-estimating the model parameters

$$\hat{c}_{jk} = \hat{n}(j,k)/T \tag{31}$$

$$\hat{\mu}_{jk} = \hat{\mu}(j,k) \tag{32}$$

$$\hat{\Sigma}_{jk} = \hat{S}(j,k) - \hat{\mu}_{jk}\hat{\mu}_{jk}^T \tag{33}$$

We call the above algorithm the UBM adapted Baum-Welch (UBM-BW) algorithm, in which the re-estimation formulas for the Gaussian mixture state emission densities are replaced by the above MAP adaptation formulas. Note that in the second iteration of the algorithm, the newly adapted Gaussian mixture state emission densities will replace the UBM in the re-estimation formulas. From then on, the estimated Gaussian mixture densities at a current iteration will serve as a prior model for the next iteration. The UBM-BW algorithm is summarized in Algorithm 1.

Algorithm 1 The UBM adapted Baum-Welch algorithm

```
1: Input: the UBM $\bar{b}_j(o_t)$, training data $O = o_1 o_2 \cdots o_T$.
2: Initialization: $\Pi$, $A$.
3: Set $b_j(o_t) = \bar{b}_j(o_t)$,   $1 \le j \le N$.
4: Repeatedly perform parameter re-estimation using Equations (14), (15), and (20)-(33) until convergence.
5: Output: final parameter estimates $\Pi$, $A$, $B$.
```

It is worth mentioning that in Equation (30), when the number of observation vectors for which state S_j and Gaussian component k are responsible is small, i.e., $n(j,k) \approx 0$, $\rho^{(l)}$ approaches $1/r^{(l)}$. In this case, the model sufficient statistics will dominate the new sufficient statistics, and hence the new model parameters will remain close to those of the prior model (e.g. the UBM). When the number of observation vectors for which state S_j and Gaussian component k are responsible is large, i.e., $n(j,k) \to \infty$, $\rho^{(l)}$ approaches 1. In this case, the model sufficient statistics will vanish from the interpolation formulas, and the new sufficient statistics consist of only the data sufficient statistics. This is exactly the case of the original Baum-Welch algorithm.

VI. Experiments

To demonstrate the effectiveness of our proposed methods, we perform expression recognition from 3D dynamic faces based on the data of 60 subjects from the BU-4DFE database. Due to the time constraint, we choose to conduct preliminary experiments on three most commonly seen expressions: happiness, sadness, and surprise.

All the 3D face models are preprocessed and stored as depth images of size 200×200 pixels, which are similar to the output of consumer depth cameras. The depth dimension is scaled to the range $[0, 1]$. As the lengths of the sequences are not always the same, we choose to keep 100 frames in each sequence. The feature vectors of sequences with lengths greater than 100 are trimmed at both sides whilst the shorter ones are zero padded.

In the facial level curve extraction phase, the step size between the curve levels is chosen to be 0.02 and the band size is 0.02. This means that all the points are used and distributed into one of 50 bands, ranging from 0 to 1 in depth. The bands are flattened out and stored as binary images. In the binary images, the number of points in the bands are different and sometimes the contours are discontinuous. We did some more morphological operations such as thinning and bridging to correct these issues.

On these binary images, we extract the distances between pairs of same-level curves at consecutive frames using the Chamfer distance of curve segments partitioned by arclength parameterized function. The number of sectors are chosen to be 20, and we compare the curves of 30 highest levels. After stacking up the distances, we have raw feature vectors of 600 dimensions. After context expansion, the feature vector dimension becomes 1800. We then perform PCA and LDA on the context expanded feature vectors. At the PCA step, we keep 100 first principal components. At the following LDA step, because of the small number of classes, we propose a special treatment: we divide every sequence into 5 subsequences of 20 frames each. The first subsequences of all sequences are labeled as 1, the last ones of all are labeled as 2, and all other subsequences are labeled based on the class labels of their parent sequence (e.g. 1-1,1-2,1-3). With this strategy, we expand the number of classes from 3 to 11. Using these finer artificial classes, LDA reduces the dimension of the feature vectors to 10. These 10D features vectors are now ready to be used for classification.

We randomly partition the 60 subjects into 10 sets, each containing 6 subjects. The experiment results are based on 10-fold cross validation, where at each round, 9 of the 10 folds (54 subjects) are used for training while the rest (6 subjects) are used for test. The recognition results of 10 rounds are then average to give a statistically significant performance measure of the algorithm. At each round, we first train a UBM, having 5 states and 16 Gaussian mixtures, with all the data for the 54 training subjects, regardless of the expressions. Once the UBM is trained, we adapt the UBM to the data for the 54 training subjects specific to the expressions, and generate three expression-dependent HMMs, one for each of the three expressions, namely happiness, sadness, and surprise. The Bayesian or minimum error rate classifier based on the adapted expression-dependent HMMs, as described in section V, is then used for expression recognition.

The results of our preliminary experiments show that the overall recognition accuracy is as high as 92.22%, with the highest performance obtained for the happiness expression: 95.00%. The confusion matrix is shown in table I. Note that our experiment results are very preliminary, as we have not had time to explore all the design choices. Nonetheless, these preliminary results clearly demonstrate the effectiveness of our proposed feature extraction and classification methods for expression recognition from 3D dynamic faces.

TABLE I: Confusion matrix.

	Happy	Sad	Surprise
Happy	0.95	0.0333	0.0167
Sad	0.0167	0.9167	0.0667
Surprise	0	0.1	0.9

VII. Conclusion and discussion

In this paper, we propose to use a facial level curves based representation of 3D facial shapes for expression recognition from 3D dynamic faces. We introduce a novel method for measuring the deformation of shapes in 3D face models. This method allows us to efficiently extract robust spatio-temporal local features. These features, when combined with an HMM-based decision boundary focus classification algorithm as a result of universal background modeling and maximum a posteriori adaptation, can yield very high performance on low resolution depth images.

With these promising preliminary results, our future work will be to quantitatively evaluate the robustness of the proposed features to noisy data captured from low-end consumer devices and to implement the benchmark test for the computation cost. These tasks will give us more clues to improve the algorithms. Besides, we will explore the idea of using many partition center points instead of a single one to assure that all facial level curves are simple in some coordinate system. We will investigate other distance functions such as the ones based on shape geodesics. The algorithms can also be parallelized and therefore are possible to be further sped up using multi-core processors to cater the needs of large-scale applications.

References

[1] I. Mpiperis, S. Malassiotis, and M. Strintzis, “Bilinear elastically deformable models with application to 3d face and facial expression recognition,” in *FG’08*, 2008, pp. 1–8.

[2] J. Wang, L. Yin, X. Wei, and Y. Sun, “3d facial expression recognition based on primitive surface feature distribution,” in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR2006)*, 2006, pp. 1399–1406.

[3] Z. Zeng, M. Pantic, G. I. Roisman, and T. S. Huang, “A survey of affect recognition methods: Audio, visual, and spontaneous expressions,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 31, January 2009.

[4] L. Yin, X. Chen, Y. Sun, T. Worm, and M. Reale, “A high-resolution 3d dynamic facial expression database,” in *The 8th International Conference on Automatic Face and Gesture Recognition (FGR08)*, Sep. 2008, pp. 17–19.

[5] C. Samir, A. Srivastava, and M. Daoudi, “Three-dimensional face recognition using shapes of facial curves,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 28, pp. 1858–1863, November 2006.

[6] H. Tang and T. Huang, “3d facial expression recognition based on automatically selected features,” in *3DFace08*, 2008, pp. 1–8.

[7] H. Tang and T. Huang, “3d facial expression recognition based on properties of line segments connecting facial feature points,” in *FG’08*, 2008, pp. 1–6.

[8] Y. Sun and L. Yin, “Facial expression recognition based on 3d dynamic range model sequences,” in *10th European Conference on Computer Vision (ECCV 2008)*, Oct. 2008, pp. 58–71.

[9] C. Samir, A. Srivastava, M. Daoudi, and E. Klassen, “An intrinsic framework for analysis of facial surfaces,” *Int. J. Comput. Vision*, vol. 82, pp. 80–95, April 2009

[10] E. M. Bronstein, M. M. Bronstein, and R. Kimmel, “Three-dimensional face recognition,” *International Journal of Computer Vision*, vol. 64, pp. 5–30, 2005.

[11] H. Schneiderman, “Feature-centric evaluation for efficient cascaded object detection.” in *CVPR (2)’04*, 2004, pp. 29–36.

[12] H. Schneiderman, “Learning a restricted bayesian network for object detection.” in *CVPR (2)’04*, 2004, pp. 639–646.

[13] E. Klassen, A. Srivastava, W. Mio, and S. H. Joshi, “Analysis of planar shapes using geodesic paths on shape spaces,” *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, vol. 26, no. 3, pp. 372–383, Mar. 2004.

[14] L. Rabiner, “A tutorial on hidden Markov models and selected applications in speech recognition,” in *Proceedings of the IEEE*, vol. 77, no. 2, 1989, pp. 257–286.

[15] R. Duda, P. Hart, , and D. Stork, “Pattern classification,” in *Wiley-Interscience*, 2001.

[16] D. A. Reynolds, “Universal background models,” in *Encyclopedia of Biometric Recognition Secaucus, NJ: Springer-Verlag*, 2008, pp. 31–36.

[17] C. Bishop, “Pattern recognition and machine learning,” in *Secaucus, NJ: Springer-Verlag*, 2006.

A Case for Parallelizing Web Pages

Haohui Mai, Shuo Tang, Samuel T. King
University of Illinois and *Valkyrie Computer Systems*

Calin Cascaval, Pablo Montesinos
Qualcomm Research

ABSTRACT

Mobile web browsing is slow. With advancement of networking techniques, future mobile web browsing is increasingly limited by serial CPU performance. Researchers have proposed techniques for improving browser CPU performance by parallelizing browser algorithms and subsystems. We propose an alternative approach where we parallelize web pages rather than browser algorithms and subsystems. We present a prototype, called Adrenaline, to perform a preliminary evaluation of our position. Adrenaline is a server and a web browser for parallelizing web workloads. The Adrenaline system parallelizes current web pages automatically and on the fly – it maintains identical abstractions for both end-users and web developers.

Our preliminary experience with Adrenaline is encouraging. We find that Adrenaline is a perfect fit for modern browser's plug-in architecture, requiring only minimal changes to implement in commodity browsers. We evaluate the performance of Adrenaline on a quad-core ARM system for 170 popular web sites. For one experiment, Adrenaline speeds up web browsing by 3.95x, reducing the page load latency time by 14.9 seconds. Among the 170 popular web sites we test, Adrenaline speeds up 151 out of 170 (89%) sites, and reduces the latency for 39 (23%) sites by two seconds or more.

1 INTRODUCTION

Web browsing on mobile devices is slow, yet recent reports from industry show that performance is critical [11, 19]. Google and Microsoft reported that a 200ms increase in page load latency times resulted in "strong negative impacts", and that delays of under 500ms seconds "impact business metrics" [16].

One source of overhead for web-based applications (web apps) is the network [18]. Engineers have attempted to mitigate this source of overhead with increased network bandwidth, prefetching, caching, content delivery networks, and by ordering network requests carefully.

A second and increasing source of overhead for web apps is the client CPU [6, 10]. Web browsers combine a parser (HTML), a layout engine, and a language environment (JavaScript), where the CPU sits squarely on the critical path [3, 7, 12]. Even though the serial performance of mobile CPUs continues to increase, the constraints on mobile device form factors and battery power imposes fundamental limitations on further improvement.

Component	% of CPU	4 cores	16 cores
V8	16%	1.13	1.17
X & Kernel	17%	1.14	1.19
Painting	10%	1.08	1.10
libc+Qt	25%	1.23	1.31
CSS	4%	1.03	1.04
Layout/Render	22%	1.20	1.27
Other	6%	1.05	1.06

Table 1: Breakdown of CPU time spent on web browsing. The last two columns predict the ideal speed ups with Amdahl's law, assuming that either 4 or 16 cores are available.

Recent work proposes exploiting parallelism to improve browser performance on multi-core mobile platforms [5, 15], including parallel layout algorithms [3, 12], and applying task-level parallelism to the browser [9]. These special cases, however, only speed up web apps that make heavy use of specific features, like cascading style sheets (CSS), or they are limited to the tasks that the browser developers identify ahead of time. Unfortunately, years of sequential optimizations, the sheer size of modern browsers, and the fundamentally single-threaded event-driven programming model of modern browsers make it challenging to generalize this approach to refactor today's browsers into parallel applications.

Our position is that *browser developers should focus on parallelizing web pages*. By taking a holistic approach, we anticipate an architecture that can work on a wide range of existing commodity browsers with only a few minor changes to their implementation, rather than a major refactoring of existing browsers or a re-implementation of these mature and feature-rich applications.

To back up our position, we present the design for Adrenaline, a prototype system that attempts to speed up web apps for multi-core mobile devices, like smart phones and tablets. Adrenaline consists of two components, a server-side preprocessor and a client (i.e., browser) that renders pages concurrently on the mobile device. The Adrenaline server decomposes existing web pages on the fly into loosely coupled sub pages, or *mini pages*. The Adrenaline browser processes mini pages in parallel. Each mini page is a "complete" web page that consists of HTML, JavaScript, CSS, and so on, running in a separate process. Therefore, the Adrenaline browser can download, parse, and render this web content in

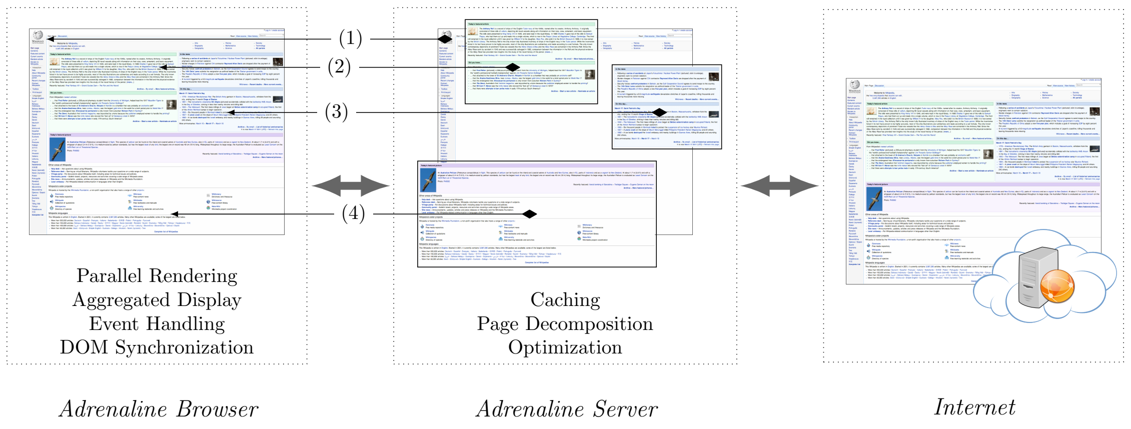


Figure 1: Workflow of Adrenaline when accessing `wikipedia.org`. In this figure, each of the numbers, 1-4, show the four mini pages Adrenaline uses for this web page. The Adrenaline server acts as a proxy between the Adrenaline browser and the Internet. It fetches the web page, optimizes and decomposes it into mini pages, then sends them back to the Adrenaline browser. The Adrenaline browser downloads and renders mini pages in parallel using multiple processes. To preserve the proper visual and programmatic semantics, the Adrenaline browser aggregates the displays for all mini pages, forwards DOM and UI events between mini pages, and synchronizes DOM interactions. Solid lines between the Adrenaline browser and the Adrenaline server show the mappings of mini pages.

parallel while still using a single-threaded and mature browser on the client.

2 Why Adrenaline?

To support our position, we present the performance characteristics of web browsing workloads to estimate potential performance improvements from parallelizing web browser subsystems.

We picked 170 web pages from the 250 most popular web sites according to Alexa [1], and mirrored them on our local network. We ran a QtWebkit-based browser and loaded each web page 20 times on a quad-core, 400 MHz ARM Cortex-A9 platform (refer to Section 6 for detailed set up), and instrumented its execution with OProfile [14] to derive the time spent on different components in the browser.

Table 1 categorizes the CPU time spent on web browsing into six components: (1) the V8 JavaScript engine [8] (*V8*), (2) The Linux kernel and X server (*X & Kernel*), (3) Qt Painting and rendering (*Painting*), (4) libc and other components in Qt (*libc+qt*), (5) CSS Selection (*CSS*), (5) WebKit layout and rendering (*Layout/Render*), (6) everything else (*Other*). Table 1 also shows the ideal speed-ups based on Amdahl's law when the platform has 4 or 16 cores, assuming each component can be parallelized completely.

Table 1 shows two findings: (i) no single component dominates the execution time, and (ii) potential gains from component-level task parallelism are moderate (up to 1.31x speed-ups for 16 cores).

These results suggest that browser developers should look at system-level ways to exploit parallelism in web browsing – parallelizing a single component results in limited speed-ups. For example, Meyerovich *et al.* reported a 80x speed up for their parallel layout algorithm [12], yet when other researchers implemented a similar scheme in Firefox their results showed a more modest speed up (1.6x), even on a layout-dominated web site [3]. It is challenging to estimate the overall speedup from parallelizing each component, in particular, because redesigning a component for task-level parallelism provides benefits beyond the exploiting the concurrency of the algorithm: the component becomes thread-safe, and therefore its execution may overlap with other components. The overlap is bounded by the web specifications and page structure, thus providing additional evidence that web page decomposition is required to achieve the full potential of browser parallelization.

3 The Adrenaline architecture

This section describes the overall Adrenaline architecture. Figure 1 shows the workflow when a user accesses `wikipedia.org` with the Adrenaline browser. First, the browser issues a request to the Adrenaline server. Second, the Adrenaline server fetches the contents of the web page, optimizes and decomposes it into mini pages. Third, the browser downloads, parses, and renders each of these mini pages in separate processes running in parallel. The browser is responsible for properly aggregating content into a single display, synchronizing global data structures, and propagating DOM and UI events to maintain correct web semantics. In this figure, the server decomposes the `wikipedia.org` page into four mini pages and the browser runs four processes in parallel to render the page.

This architecture offers four unique advantages compared to other techniques for parallelizing web browsers. First, Adrenaline is a data parallel system. It parallelizes web pages, rather than specific components in web browsers. Conceptually all components in a web browser can now be executed in parallel. Second, decomposition reduces the total amount of work from some tasks, particularly layout and rendering because of smaller working sets for each mini page. Third, careful decomposition could potentially remove serialization bottlenecks. Specifically, Adrenaline isolates JavaScript into a single mini page to allow tasks such as layout and rendering in other mini pages to run concurrently. Fourth, pre-processing the pages on the Adrenaline server creates opportunities to shift computation from the client to the server.

This architecture does also introduce two sources of overhead that the Adrenaline system must overcome. Fundamentally, the architecture places a proxy in between the Adrenaline browser and the Web. This additional component will add latency for individual network connections when compared to connecting to web sites directly. In addition, this architecture uses more resources on the mobile device through its use of multiple processes. Despite these inherent sources of overhead, the Adrenaline browser speeds up the overwhelming majority of sites we tested, as we demonstrate in Section 6.

4 Design challenges

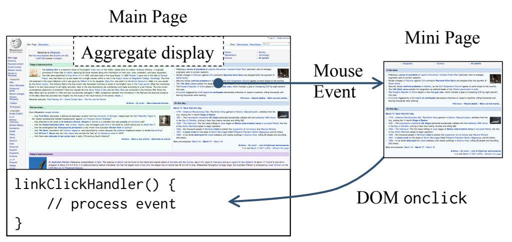
Designing the Adrenaline system presents three key challenges. First, Adrenaline has to generate web apps that look the same from the user's perspective. Second, Adrenaline has to ensure that the semantics of web apps remains the same, from the web developer's perspective. Third, Adrenaline has to minimize the overhead induced by this multi-process architecture.

In this section, we discuss our techniques for maintaining visual compatibility, JavaScript and DOM compatibility, and techniques to reduce synchronization overhead. In Section 5, we describe our server-side algorithm for decomposing web pages.

4.1 Visual compatibility

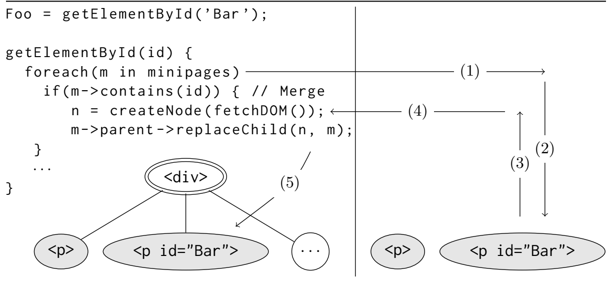
The Adrenaline browser is designed to be visually compatible with traditional mobile browsers, and to maintain identical side effects when the user interacts with a page. In the Adrenaline browser, a *main page* is responsible for this compatibility.

The main page assembles other mini pages in its display, and captures all external UI events. It is also responsible for rerouting events to mini pages. Figure 2



```
linkClickHandler() {
    // process event
}
```

Figure 2: Event routing. This figure shows how Adrenaline handles a mouse click on a link. All data is forwarded through Adrenaline's inter-process communication (IPC) channels.



```
Foo = getElementById('Bar');

getElementById(id) {
  foreach(m in minipages)
    if(m->contains(id)) { // Merge
      n = createNode(fetchDOM());
      m->parent->replaceChild(n, m);
  }
  ...
}
```

Figure 3: Merging a mini page. During merging, Adrenaline (1) issues a request to the remote mini page that (2) reads, (3) serializes, and (4) returns the results back to the main page. Then (5) the main page inserts the remote DOM into its own DOM before terminating the remote mini page.

shows an example of Adrenaline's event routing mechanisms. Consider the case where a user clicks on a link in a mini page. First, the main page routes the mouse event to the corresponding mini page based on the location of the mouse pointer. After the mini page processes the mouse event and determines that the click was on a link, the mini page wraps it into a `onclick` DOM event, like a traditional browser would. Then, the mini page forwards the DOM event to the main page where the JavaScript event handler runs.

4.2 JavaScript and DOM compatibility

Adrenaline has to preserve the semantics of JavaScript in order to run legacy web apps correctly. The problem becomes challenging if JavaScript is distributed across multiple mini pages. Therefore, the current implementation of Adrenaline chooses to put all JavaScript into one process (i.e., the main page) as a solution.

When JavaScript code accesses remote DOM states, it merges mini pages into the main page on demand. A common example is that the JavaScript code calls `getElementById()` to get a reference for a DOM element. In Figure 3, the first line of JavaScript runs in the main page and gets a reference to the element `Bar`. The Adrenaline browser runs this code, and once it finds out that the element Bar resides in a remote mini page, it asks the remote mini page to serialize its entire DOM and to send it back to the main page. The main page then inserts the remote mini page's DOM into its own DOM and terminates the remote mini page. After the main page merges a mini page, it can access the DOM states locally that used to reside in the mini page, and JavaScript execution can proceed.

Although Adrenaline runs all JavaScript in a single mini page, this architecture still has significant benefits for web pages where the JavaScript code accesses only a subset of the DOM. For these types of web pages the Adrenaline browser can process the DOM elements not accessed by JavaScript in separate mini pages, in parallel, *without* blocking on JavaScript execution like a traditional browser would.

The architecture of Adrenaline could introduce races when rendering web pages, but the Adrenaline browser handles these cases correctly. When traditional web browsers encounter JavaScript code, they execute the code with the current state of the DOM. In the Adrenaline browser each mini page builds up its own DOM structure in parallel, so when JavaScript code executes, the Adrenaline browser has to ensure that JavaScript accesses the correct DOM state. The Adrenaline browser inspects the program counter and call stack to ensure correctness. We omit the details here.

4.3 Minimizing synchronization overhead

JavaScript code calls getElementById() to get a reference for a specific DOM element, thus calls to getElementById() must check against each mini page for the requested element. The Adrenaline server computes a Bloom filter [4] for all elements in each mini page, and sends the filters along with the main page. The main page only sends inter-process requests to mini pages that can possibly contain the element (whose corresponding Bloom filters will have positive results), thus saving inter-process communication.

This is a safe optimization because a Bloom filter can only have false positives but not false negatives, meaning that an element is absent in the set if the testing result of Bloom filter is negative.

5 The Adrenaline server

From a high level, the Adrenaline server renders the web page, and extracts information about the rendered web page (e.g., element sizes, bounding boxes, and where elements are located visually on the page). It uses this information as inputs to a heuristic algorithm to decompose the page into mini pages. After the Adrenaline browser loads the page, it provides feedback to the server, such as any unanticipated DOM merges, to help the server adjust future decomposition of the same page.

In general, the algorithm tries to balance three main constraints. First, Adrenaline tries to keep JavaScript code and the DOM elements that the JavaScript code accesses in the same mini page to avoid merge operations. Second, Adrenaline uses only a continuous segment of the original DOM in mini pages to help simplify the implementation of merging. Third, Adrenaline ensures that mini pages occupy non-overlapping visual blocks (rectangles) to simplify mini page display and event handling.

In addition to decomposing pages, the Adrenaline server also optimizes mini pages and sends extra information to the browser. For example, the Adrenaline server customizes CSS rules for each individual mini page, and provides the Adrenaline browser with hints about resources that the page includes to enable prefetching.

Due to space limitations, we omit the full details of the Adrenaline decomposition algorithm and the full details of the server-side optimizations we perform on mini pages.

6 Experience with Adrenaline

We implemented the Adrenaline server as a HTTP proxy that fetches web pages and decomposes them on the fly automatically. This architecture mirrors closely the server-side architecture for other mobile browsers, like Opera mini, Skyfire, and Amazon Silk [2, 13, 17].

The Adrenaline browser uses the WebKit rendering engine and the V8 JavaScript engine. We use the Qt Toolkit to implement the platform specific portions of the browser. Mini pages are implemented as browser plugins in Adrenaline to reuse existing mechanisms to maintain visual compatibility. Our changes were rather minimal, and we believe that the same techniques are applicable to commodity browsers.

To test the performance of our prototype and to test the efficacy of the basic Adrenaline approach, we ran the Adrenaline browser on a CoreTile Express A9x4 ARM development board. The board has a quad-core Cortex-A9 CPU running at 400MHz and 768MB of DDR2 RAM. We tested Adrenaline on 170 of the most popular web sites (according to Alexa), and we compared against an unmodified version of a WebKit-based browser (which is called QtBrowser in later sections). To isolate the effects our our algorithms we mirror the web pages on our local network and connect to the server via a FastEthernet connection.

Our preliminary experience with Adrenaline is encouraging. Overall, Adrenaline reduces the page load latency by 1.75s on average, where industry considers a 0.5 second latency reduction as meaningful [11, 16, 19]. Adrenaline improves the page load latency time by 1.54x on average across the entire workload. For one experiment, Adrenaline speeds up web browsing by 3.95x, reducing the page load latency time by 14.9 seconds. Among the 170 popular web sites we tested, Adrenaline speeds up 151 out of 170 (89%) sites, and reduces the latency for 39 (23%) sites by two seconds or more.

7 Case study: wikipedia

This section describes a case study for the performance characterization of the Wikipedia entry for the Nokia page. We instrument the execution of both the Adrenaline browser and the QtBrowser with OProfile to collect run-time statistics.

Figure 4 describes the high-level performance characteristics of this case. Adrenaline reduces the page loading time by 11.7 seconds.

The case study contains a *timeline graph* and a *workload graph*. The timeline graph plots the total page loading time for QtBrowser and for Adrenaline. The top-most bar represents the total page load latency for QtBrowser. The shaded bars below represent the page load latency of the Adrenaline main page and mini pages. Thus, the total page load latency for the Adrenaline browser is determined by the shaded bar that completes last. For comparison, we load each mini page individually with QtBrowser and report its execution time with the corresponding white bar.

The workload graph classifies the workload of the two browsers into six disjoint categories: (1) the V8 JavaScript engine (V8), (2) The Linux kernel (Kernel), (3) Qt Painting and rendering (Painting), (4) CSS Selection (CSS), (5) WebKit sans CSS Selection (WK w/o CSS), and (6) libc and other components in Qt (libc+Qt). These six categories consume most of the CPU time. The execution time of each of these six components is normalized with respect to the total time spent by the QtBrowser to load the original page. For comparison, the workload graph stacks the execution of all Adrenaline processes into one bar even though their execution overlaps in the system.

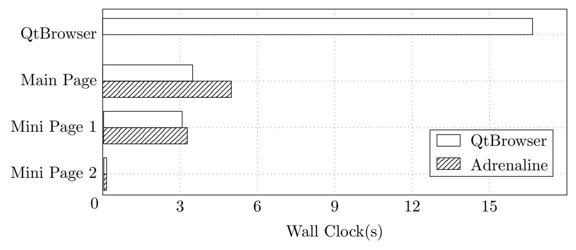
The timeline graph in Figure 4 shows that the Adrenaline server decomposes the page into three pages, and the Adrenaline browser is able to render them in parallel. This page is large enough for Adrenaline to harvest a sufficient amount of independent work for each mini page.

Parallelism by itself, however, does not fully explain why Adrenaline is so much faster than QtBrowser. The workload graph shows that there is almost a 3x reduction for both CSS and WK without CSS. For CSS, the decomposition brings in two benefits: (1) the Adrenaline server speeds up CSS for Mini Page 1 and 2 through inlining CSS rules. (2) CSS selection runs on fewer elements in the main page (30% of the original page), reducing the total amount of work.

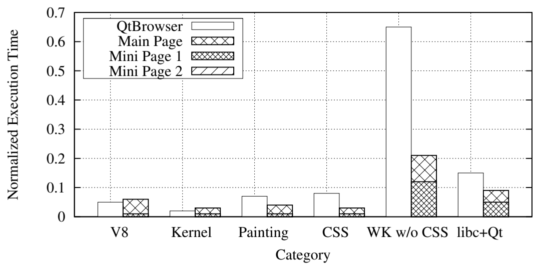
Case I: http://en.wikipedia.org/wiki/Nokia

Latency: QtBrowser: 16.7s, Adrenaline: 4.99s

Timeline graph



Workload Graph



Summary

Why faster?

+ Simplified layout and rendering
+ Simplified CSS
+ Layout and rendering are executed in parallel

Figure 4: Performance analysis for `http://en.wikipedia.org/wiki/Nokia`.

For WK without CSS, analysis reveals that the execution time reduction can be attributed to layout and rendering primarily. The decomposition enables the main page to treat both Mini Page 1 and Mini Page 2 as "blackboxes" during layout and rendering. The main page is no longer responsible for rendering elements inside mini pages, as the mini pages running are responsible for rendering them, which happens in parallel. For layout, the main page has fewer elements to layout, since it only needs to layout the remaining elements plus the containers of mini pages. Relayout, which the browser could trigger during loading in response to various events, is also simplified for the same reason: the rendering of individual elements in mini pages is deferred to the mini pages themselves and happens in parallel.

8 Summary and future work

In this paper, we advocated that browser developers should think about parallelizing web pages, rather than individual components of web browsers. Based on our initial experience with Adrenaline, we believe that Adrenaline can improve significantly the performance of web browsing on mobile devices.

We plan to further investigate the performance of Adrenaline under more realistic network conditions and hardware configurations. In addition, we plan to explore more heuristics on page decomposition, as well as providing APIs for web developers to express page-level parallelism. Finally, we plan to apply Adrenaline to a larger set of web sites to evaluate our techniques more comprehensively.

Acknowledgements

We would like to thank the anonymous reviewers who provided insigtful and detailed feedback on our paper. This research was funded in part by NSF grants CNS 0834738 and CNS 0831212, grant N0014-09-1-0743 from the Office of Naval Research, AFOSR MURI grant FA9550-09-01-0539, and by a grant from the Internet Services Research Center (ISRC) of Microsoft Research.

References

[1] Alexa. `http://www.alexa.com/topsites`.

[2] Amazon.com, Inc. Amazon silk browser. `http://amazonsilk.wordpress.com/`, Sept 2011.

[3] C. Badea et al. Towards parallelizing the layout engine of firefox. In *HotPar*, 2010.

[4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13:422–426, July 1970.

[5] Broadcom Inc. Bcm28150 - 1080p 4g hspa+ smartphone processor, 2011. `http://www.broadcom.com/products/Cellular/3G-Baseband-Processors/BCM28150`.

[6] S. Dubey. AJAX performance measurement methodology for internet explorer 8 beta 2. CODE Magazine, Vol. 5 Issue 3, 2008. `http://www.code-magazine.com/Article.aspx?quickid=0811102`.

[7] E. Fortuna, O. Anderson, L. Ceze, and S. Eggers. A limit study of javascript parallelism. In *IISWC*, 2010.

[8] Google V8 Team. `http://code.google.com/p/v8`.

[9] C. Grier, S. Tang, and S. T. King. Secure web browsing with the OP web browser. In *IEEE S&P*, 2008.

[10] C. G. Jones et al. Parallelizing the web browser. In *HotPar*, 2009.

[11] M. Mayer. Google I/O '08 keynote, 2008. `http://www.youtube.com/watch?v=6x0cAzQ7PVs`.

[12] L. A. Meyerovich and R. Bodík. Fast and parallel webpage layout. In *WWW*, 2010.

[13] Opera Software. `http://www.opera.com/mobile`.

[14] OProfile. `http://oprofile.sourceforge.net`.

[15] Samsung Inc. Samsung introduces high performance, low power dual cortex - a9 application processor for mobile devices, September 2010. `http://www.samsung.com/global/business/semiconductor/newsView.do?news_id=1195`.

[16] E. Schurman and J. Brutlag. Performance related changes and their user impact. `http://velocityconf.com/velocity2009`.

[17] SkyFire Labs, Inc. `http://www.skyfire.com`.

[18] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. Why are web browsers slow on smartphones? In *HotMobile'11*.

[19] Z. Yang. Every millisecond counts, 2009. `http://www.facebook.com/note.php?note_id=122869103919`.

How do Developers Use Parallel Libraries?

Semih Okur, Danny Dig
University of Illinois at Urbana-Champaign, IL, USA
okur2@illinois.edu, dig@illinois.edu

ABSTRACT

Parallel programming is hard. The industry leaders hope to convert the hard problem of *using parallelism* into the easier problem of *using a parallel library*. Yet, we know little about how programmers adopt these libraries in practice. Without such knowledge, other programmers cannot educate themselves about the state of the practice, library designers are unaware of API misusage, researchers make wrong assumptions, and tool vendors do not support common usage of library constructs.

We present the first study that analyzes the usage of parallel libraries in a large scale experiment. We analyzed 655 open-source applications that adopted Microsoft's new parallel libraries – Task Parallel Library (TPL) and Parallel Language Integrated Query (PLINQ) – comprising 17.6M lines of code written in C#. These applications are developed by 1609 programmers. Using this data, we answer 8 research questions and we uncover some interesting facts. For example, (i) for two of the fundamental parallel constructs, in at least 10% of the cases developers misuse them so that the code runs sequentially instead of concurrently, (ii) developers make their parallel code unnecessarily complex, (iii) applications of different size have different adoption trends. The library designers confirmed that our findings are useful and will influence the future development of the libraries.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Object-oriented constructs*

General Terms

Measurement, Experimentation

Keywords

Multi-core; empirical study; parallel libraries; C#.

1. INTRODUCTION

The computing hardware industry has resorted to multicore CPUs in order to keep up with the previous prediction of Moore's law. While the number of transistors will keep doubling, the multicore revolution puts pressure on software developers to use parallelism if they want to benefit from future hardware improvements. At the time, this seemed like a huge gamble: will software developers embrace parallelism in their applications? A few years after the irreversible conversion to multicore, we can finally answer such questions.

Parallel programming is hard. In the desktop computing, the dominant paradigm is thread-based parallelism on shared-memory systems. Under this paradigm, parallel programming is regarded as the art to balance conflicting forces: making code thread-safe requires protecting accesses to shared variables through synchronization, but this in turn reduces the scalability of parallel applications. Parallelism can also obfuscate the intent of the original sequential code [6]. Despite books on parallel programming and API documentation of parallel constructs [2, 10, 14–17], parallel programming education is lagging behind. Developers miss examples [32] of successful applications that use parallelism.

The industry leaders hope to convert the hard problem of using parallelism into the easier problem of using a parallel library. Microsoft provides Task Parallel Library (TPL) [15], Parallel Language Integrated Query (PLINQ) [23], Collections.Concurrent (CC) [3] and Threading [33] for .NET languages (e.g., C#). Java developers uses `java.util.concurrent` package. Intel provides Threading Building Blocks (TBB) [34] for C++. Despite syntactic differences, these libraries provide similar features such as scalable concurrent collections (e.g., `ConcurrentDictionary`), high-level parallel constructs (e.g., `Parallel.For`), and lightweight tasks. Their runtime systems also provides automatic load balancing [37]. Despite the recent surge in the number of these libraries, we know little about how practitioners adopt these libraries in practice.

We present the first empirical study that answers questions about parallel library usage in-depth and on a large scale. We analyzed 655 open-source applications that adopted Microsoft's new TPL and PLINQ libraries. In this corpus, we studied the usage of all four .NET parallel libraries (both old and new). These applications are hosted on Github [8] and Microsoft's CodePlex [4], and they comprise 17.6M non-blank, non-comment lines of code written in C# by 1609 programmers. We implemented a semantic analysis that uses type information to collect *precise* statistics about parallel constructs.

Using this data, we are able to answer several questions.

Q1: Are developers embracing multi-threading? Our data shows that 37% of all open-source C# applications in the most active code repositories use multi-threading. Out of these applications, 74% use multi-threading for concurrency and 39% use it for parallelism.

Q2: How quickly do developers start using the new TPL & PLINQ libraries? TPL and PLINQ have been released nearly 2 years ago (in April 2010). However, we found significant differences between the times when developers start using these libraries. We found that applications of different size have a different adoption tipping point. We also found that more applications are becoming parallel, and existing parallel applications are becoming more parallel.

Q3: Which parallel constructs do developers use most often? 10% of the API methods account for 90% of the library usage, thus newcomers can focus on learning a smaller subset of the parallel libraries.

Q4: How do developers protect accesses to shared variables? Locks are still the most used synchronization construct, but developers use a wide variety of alternatives.

Q5: Which parallel patterns do developers embrace? Out of the six widely-used parallel patterns that we analyzed, loop parallelism is the most common.

Q6: Which advanced features do developers use? We found that developers rarely use optional parameters such as customized task schedulers, aggregate exception handling, controlling the level of parallelism, etc.

Q7: Do developers make their parallel code unnecessarily complex? We found that developers sometimes use more powerful task constructs instead of the equivalent but simpler task constructs, even though they never use the extra power. Thus they make their code less readable and more verbose than it needs to be.

Q8: Are there constructs that developers commonly misuse? We found that for two of the fundamental parallel constructs, in at least 10% of the cases developers misuse them: the code runs sequentially instead of concurrently.

Our study has several practical implications. First, it is a tremendous resource for educating developers. The most common way to learn a new library is to study relevant examples of the API. Newcomers can start learning the APIs that are most widely used (see Q1 and Q3), and we can point them to the kinds of applications that are most likely to use the libraries (Q2). Newcomers should avoid common misuses (Q8) and constructs that unnecessarily increase the code complexity and the likelihood of errors (Q7). Our study also educates developers by showing real-world examples of parallel patterns (Q5).

Second, designers of these libraries can learn how to make the APIs easier to use (Q6). They can learn from observing which constructs do programmers embrace (Q3), and which ones are tedious to use or error-prone (Q8).

Third, researchers and tool vendors can focus their efforts on the constructs that are commonly used (Q3) or tedious or error-prone to use (Q8). For example, the refactoring community can decide which refactorings to automate. The testing and verification community can study the synchronization idioms that programmers use (Q4).

This paper makes the following contributions:

- To the best of our knowledge, this is the first empirical study to answer questions about parallel library usage on a large-scale, using semantic analysis.
- We present implications of our findings from the perspective of three different audiences: developers, library designers, and researchers.
- The tools and data are publicly available, as a tremendous education resource: `http://LearnParallelism.NET`

2. BACKGROUND

2.1 Parallel programming in .NET

We first give a brief introduction to parallel programming in .NET framework. The earlier versions provide the Threading library which contains many low-level constructs for building concurrent applications. `Thread` is the primary construct for encapsulating concurrent computation, and `ThreadPool` allows one to reuse threads. Synchronization constructs include three types: locks, signals, and non-blocking.

.NET 4.0 was enhanced with higher-level constructs. The new TPL library enables programmers to introduce task parallelism in their applications. `Parallel`, `Task`, and `TaskFactory` classes are the most important constructs in TPL.

`Task` is a lightweight thread-like entity that encapsulates an asynchronous operation. Using tasks instead of threads has many benefits [15] - not only are tasks more efficient, they also abstract away from the underlying hardware and the OS specific thread scheduler. `Task<>` is a generic class where the associated action returns a result; it essentially encapsulates the concept of a "Future" computation. `TaskFactory` creates and schedules tasks. Here is a fork/join task example from the *passwordgenerator* [28] application:

```
for (uint i = 0; i < tasks.Length; i++)
  tasks[i] = tf.StartNew(() => GeneratePassword(
       length, forceNumbers, ...), _cancellation.Token
       );
try{ Task.WaitAll(tasks, _cancellation.Token); } ...
```

The code creates and spawns several tasks stored in an array of tasks (the fork step), and then waits for all tasks to complete (the join step).

`Parallel` class supports parallel loops with `For` and `ForEach` methods, and structured fork-join tasks with `Invoke` method. The most basic parallel loop requires invoking `Parallel.For` with three arguments. Here is a usage example from the *ravendb* [30] application:

```
Parallel.For(0, 10, counter => {... ProcessTask(
    counter, database, table)} )
```

The first two arguments specify the iteration domain, and the third argument is a C# lambda function called for each iteration. TPL also provides more advanced variations of `Parallel.For`, useful in map/reduce computations.

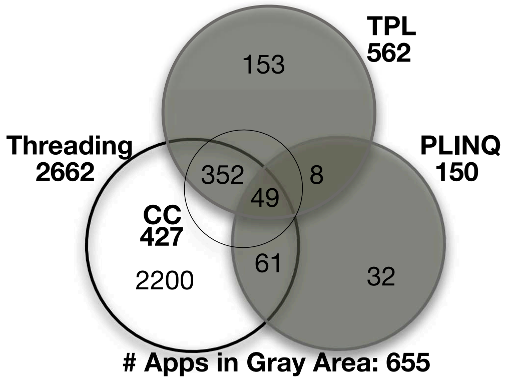
.NET also provides the CC library, which supports several thread-safe, scalable collections such as `ConcurrentDictionary`.

.NET 4.0 provides a fourth parallel library, the Parallel Language-Integrated Query (PLINQ) library, which supports a declarative programming style. PLINQ queries operate on `IEnumarable` objects by calling `AsParallel()`. Here is an example from the *AppVisum* [24] application:

```
assembly.GetTypes().AsParallel()
  .Where(t => t.IsSubclassOf(typeof(ControllerBase)))
  .Select(t => new ...)
  .ForAll(t => controllersCache.Add(t.Name, t.Type));
```

After the `AsParallel`, the data is partitioned to worker threads, and each worker thread executes in parallel the following `Where`, `Select`, and `ForAll`.

Figure 1: Number of applications that use Threading, TPL, PLINQ or CC libraries.



2.2 Roslyn

The Microsoft Visual Studio team has recently released Roslyn [31], as a community technology preview, with the goal to expose the compiler-as-a-service through APIs to other tools like code generation, analysis, and refactoring. Roslyn has components such as Syntax, Symbol Table, and Binding and Flow Analysis APIs.

The Syntax API allows one to parse the structure of a program. While a C# file can be syntactically analyzed in isolation, we cannot ask questions such as "what is the type of this variable". The type may be dependent on assembly references, namespace imports, or other code files. To further improve the analysis, we use the Symbol and Binding APIs to get semantic information such as type information, compiler options (e.g., targeting .NET 4.0). We used Syntax, Symbol and Binding APIs to parse our corpus data and statically analyze the usage of concurrent constructs.

3. METHODOLOGY

In this section we briefly describe the set of applications, the experimental setup, and the analysis infrastructure.

3.1 Corpus of Data

We analyze all open-source C# applications from two repositories, CodePlex [4] and Github [8]. We chose these two repositories because according to a recent study [19], most C# applications reside in these two repositories. Codeplex is Microsoft's code repository, and Github is now the most popular open source software repository, surpassing Google Code and SourceForge.

From these repositories, we want to filter those applications that use TPL, PLINQ, CC, and Threading libraries. For this, we implemented a tool, Collector. Next we explain how Collector works.

Collector downloaded all C# applications that contain at least one commit after April 2010, the release date of TPL and PLINQ. In the Git community, developers often fork an application and start making changes in their own copies. Sometimes, the main application might merge changes from the forked applications, but many times the forked applications start evolving independently. Collector ignores all forked applications. It also ignores the "toy applications", i.e., the ones that have less than 1000 non-comment, non-blank lines of code (SLOC). We discard such applications because many are just experimentally written by developers who learn a new construct, and they do not represent realistic usage of production code.

After eliminating applications that do not compile due to the missing libraries, incorrect configurations, etc, we had 7778 applications targetting .NET 4.0. From these, we want to select the applications that truly use the parallel libraries. For example, 648 applications imported the TPL library, but only 562 actually invoke functions from the TPL libraries. Thus, Collector removed the applications that import but never invoke any parallel library construct. Table 1 shows 2855 applications that truly use the parallel libraries.

Figure 1 shows that some applications use only one library, while other applications use these four libraries together. The TPL or PLINQ applications that also use Threading does not imply that these applications use threads. Threading library also provides synchronization constructs, and they are used in conjunct with TPL and PLINQ. The 2200 applications that only use the Threading library use multi-threading with explicit threads and thread pools. We excluded applications that use the Threading library to only insert delays and timers.

In the rest of the paper, we will focus on the applications that adopted the new parallel libraries, TPL and PLINQ. In this corpus, we also study the usage of Threading and CC. After all the filters, Collector retained 655 applications (shown within the gray area inside Fig. 1), comprising 17.6M SLOC, produced by 1609 developers. The only exception is our research question Q1 (the adoption of multi-threading), where we take into account all applications in Fig. 1.

We analyze all these 655 applications, without sampling, and these applications are from the most widely used C# repositories. This makes our findings representative.

3.2 Analysis Infrastructure

We implemented another tool, Analyzer, that performs the static analysis and gathers statistical usage data. We run Analyzer over each application from our corpus data. For each of these applications, Analyzer inspects the version from the main development trunk as of Jan 31st, 2012. The only exception is Q2 (the trends in adoption), where we analyze monthly code snapshots.

We implemented a specific analysis for each question using Roslyn's API. Since two projects in an application can share the same source file, Analyzer ensures that each source file is counted only once. Also, a .NET project can import system libraries in source format, so Analyzer ignores any classes that reside in the `System` namespace. This ensures that we are not studying the usage patterns in Microsoft's library code, but we study the usage only in the applications' code. When we discuss each empirical question, we present the static analysis that we used in order to collect the results.

4. RESULTS

Q1: Are developers embracing multi-threading?

As seen in Table 1, 37% of the 7778 applications use at least one of the four parallel libraries, which means they use some form of multi-threading. When we take into account only the category of large projects, 87% use multi-threading.

Why do programmers use multi-threading? Sometimes, multi-threaded code is a *functional* requirement. For example, an operating system with a graphical user interface must support concurrency in order to display more than one window at a time. Sometimes it is more convenient to write multi-threaded code even when it runs on a uniprocessor machine. For example, online transaction processing, reactive,

Table 1: Corpus Data

Type	Small (1K-10K)	Medium (10K-100K)	Large (>100K)	Total
# Applications compilable and targetting .NET 4.0	6020	1553	205	7778
# Multi-threaded Applications	1761	916	178	2855
# Applications adopted new libraries (TPL, PLINQ)	412	203	40	655

event-driven code is easier to express with threads. In such scenarios developers use multi-threading for *concurrency.*

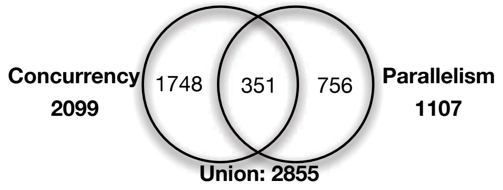
However, other times developers use multi-threading to improve a *non-functional* requirement such as performance. For this, they use multiple threads that run on multicore machine, thus they use multi-threading for *parallelism.*

Out of the applications that use multi-threading, 74% use it for concurrency and 39% use it for parallelism. Figure 2 shows the distribution. Some applications using multi-threading for both concurrency and parallelism

Next we manually analyzed the top 50 applications that highly use parallelism. We aim to find the killer applications for parallelism. We list their domain and how many applications we found from each domain: developer tools (7), data mining (7), multimedia (6), graphics (6), games (5), cloud computing (5), finance (3), database (3), networking (3), social media (2), office productivity (2), web server (1).

Program Analysis: To find whether an application uses multi-threading for concurrency or for parallelism, Analyzer first tabulates the usage of the multi-threading constructs (e.g., `Thread`, `Task`, `Parallel.For`, etc.) from each library in each application. Some constructs are clearly intended for concurrency (e.g., `FromAsync`, `TaskCompletionSource`, UI event dispatching thread) or for parallelism (e.g., `Parallel.For`, all PLINQ constructs). Other constructs (e.g., `Thread`, `Task`) can be used for either concurrency or parallelism. A typical usage scenario is to spawn threads in the iterations of a `for` loop. If the main thread waits for the child threads to finish, it means that the intent of the programmer is to have the threads execute *at the same time*, thus it is an example of parallelism. If the main thread does not wait for the child threads, it means that the intent is to have the threads *be in progress*, which is an example of concurrency. Thus, Analyzer checks whether the spawned constructs are waited or joined in the calling context.

Figure 2: Concurrency vs. Parallelism



Many applications have embraced multi-threading, however many of them use it for concurrency rather than parallelism.

Q2: How quickly do developers start using the new TPL & PLINQ libraries?

In the rest of the paper we move away from the applications that only use the Threading library and will focus on the 655 applications that adopted the new libraries (in the gray area in Fig. 1). Microsoft released the new libraries along with .NET 4.0 in April 2010. We want to find out how long it takes for developers to start using such libraries.

To analyze such adoption trends, from the set of 655 applications that eventually use TPL/PLINQ we select the subset of applications that exist in the repository as of April 2010. This subset comprises of 54 applications. If we had analyzed all TPL/PLINQ applications, regardless of their starting date, then as time goes by, we would see an increased number of constructs due to adding more applications.

For each of these 54 applications, we analyze monthly snapshots. In total, we analyze 31.9MLOC, comprising 694 different versions.

Figure 3 shows the number of applications that use at least one construct in each month. We split the 54 applications according to the size of their source code (small, medium, large). This prevents the trends in the small applications to obscure the trends in the larger applications (notice the different vertical scale in Fig 3). The results show that more applications are using the libraries as time goes by.

Figure 4 shows the average number of constructs per application. Here is an example of how we compute this number for the month of June 2010 for small applications. There are 24 constructs and 9 applications that use TPL at this time, so the average usage per application is 24/9 = 2.6. In April 2010 the average usage for small and medium applications is not zero because these applications were using the "developer preview release" of the libraries.

Looking at both Fig. 3 and 4, we can notice a very different adoption rate among the three sizes of applications. If we look for the "tipping point" [9], i.e., the point in time when there is a major increase in the adoption rate (noticeable by a steep gradient of the slope), we can notice very different trends. The small applications are the early adopters of new libraries (2-3 months after the release), medium applications adopt around 4-5 months, and large applications are late adopters (8-9 months after the release).

Figures 3 and 4 show complementary data: the former shows that more applications are becoming parallel, whereas the latter shows that each application is becoming more parallel, i.e., it uses more parallel constructs.

Figure 5 shows the average number of Threading constructs per application does not decrease over time. This makes sense because most of the synchronization constructs are in the Threading library. Also, one can notice that compared with the TPL/PLINQ average density, Threading density is higher; this makes sense because the latter library has lower-level constructs.

Program Analysis: To find whether an application exists in April 2010, Collector looks at the creation date of each application, as listed in Github or Codeplex. After determining the set of 54 applications, our script checks out the source code snapshot for each month from April 2010 to February 2012. Then, for each snapshot, Analyzer collects usage details of TPL/PLINQ libraries. In the next question (Q3) we provide more information on how Analyzer collects usage details for one single snapshot.

Applications of different size adopt the new parallel libraries differently.

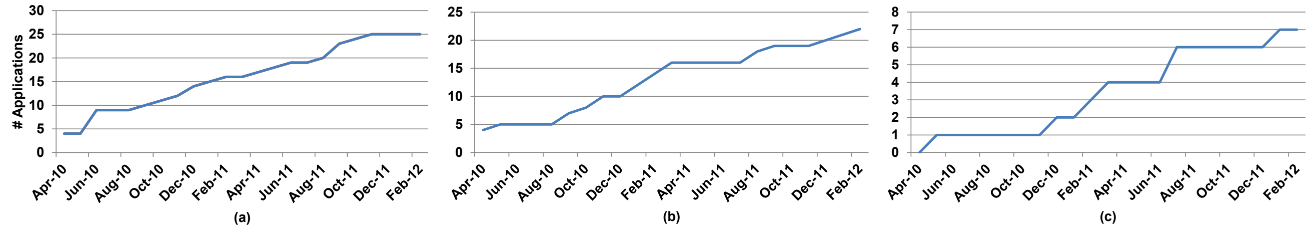


Figure 3: Number of (a) small-, (b) medium-, (c) large-size applications that use TPL/PLINQ

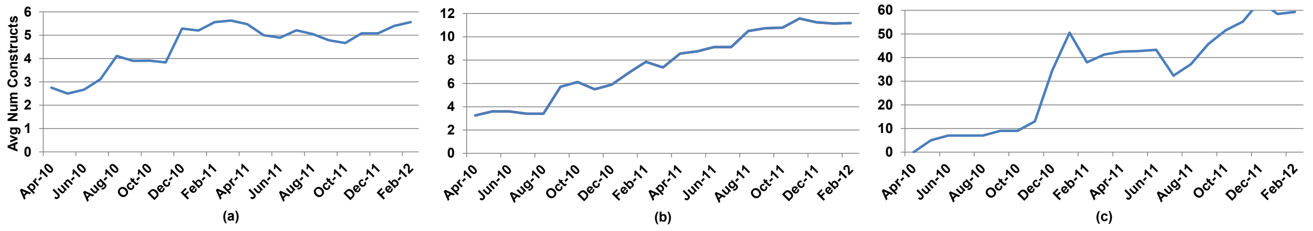


Figure 4: Average number of TPL/PLINQ constructs per application for (a) small-, (b) medium-, (c) large-size applications.

Q3: Which parallel constructs do developers use most often?

Table 2 tabulates the constructs that developers used most often from the TPL, Threading, PLINQ, and CC libraries. For example, lets drill down inside the TPL library and see the usage of class `Task`. Its methods account for 23% of all method call sites for the TPL library. One particular method, `Start`, has 243 call sites in 92 different applications. These call sites account for 18% of all call sites for methods from `Task` class.

Among these 4 libraries, they define 138 classes containing 1651 methods (counting constructors and overloaded methods). In table 2 we show combined usage for overloaded methods (e.g., we combine all 17 overloaded `StartNew` methods into one single method). ANALYZER collects usage details for each of these methods. Due to the space limitations, we only tabulate the most used classes and methods for each library. The companion website [36] presents a complete fine-grained view.

The data shows that among the 1651 methods, some methods are used much more frequently than othes. For example, 10% of the methods are used 90% of the times. 1114 methods are never used. While similar trends are expected for any rich library APIs, it is important that we find the widely used APIs so that developers can focus on these.

We now discuss some of the findings for each library.

TPL: As shown in Table 2, `Parallel`, `Task`, and `TaskFactory` are the TPL classes most commonly used. When it comes to creating tasks, developers prefer to use the factory method `TaskFactory.StartNew` rather than invoking the task constructor. `Task<>` (i.e., the "Future" construct) is used nearly half as many times as `Task`.

Threading: `WaitHandle` is an abstract class for synchronization primitives, e.g., semaphore, mutex, so it is the second most popular class after `Thread`, the main class of the library.

Concurrent Collection: `ConcurrentDictionary`, a thread-safe implementation of `HashMap` is the most widely used.

Program Analysis: To accurately detect usage of a particular method, ANALYZER needs type and binding information. ANALYZER needs to know not only the name of the method, but also the type of the receiver object and the type of the arguments, and where does a method bind. This lets the analysis differentiate between `t.start()` when `t` is an instance of `Thread`, and the cases when `t` is an instance of a business class defined by the application. Because ANALYZER uses the Symbol and Binding services of Roslyn, our reported usage numbers are 100% precise. Other empirical studies of library usage [1, 11, 35] have only used syntactical analysis, which can limit the accuracy of the results.

Parallel library usage follows a power-law distribution: 10% of the API methods account for 90% of the total usage.

Q4: How do developers protect accesses to shared variables?

Table 3 shows the type of synchronization, the name of the library constructs, how many times each construct was employed, and what is the usage frequency in comparison with other constructs within the same type of synchronization. Table 3 list *all* five kinds of synchronization constructs. `lock` and volatile accesses are language features, `Task.Wait` is a method of TPL, implicit synchronization constructs are from CC, and the rest of all is from Threading. To compute the number of implicit synchronization constructs, we sum the number of call sites for each API method that has implicit synchronization in its implementation. Notice that `lock` is by far the most dominant construct followed by `Volatile` accesses.

Program Analysis: To count one usage of a lock, ANALYZER tries to match a pair of lock acquire and release operations. When one of the acquire or release operations is used more often than the other, we take the minimum number of these operations. Similarly, a pair of signal and wait operations count as one occurrence.

Finding accesses to volatile variables takes most of the analysis running time. Using the binding information, ANALYZER looks up the definition of each accessed variable and field and checks whether it is `volatile` variable or field.

While locks are still very popular, developers use a wide variety of other synchronization constructs.

Figure 5: Average number of Threading constructs per application for (a) small-, (b) medium-, (c) large-size applications.

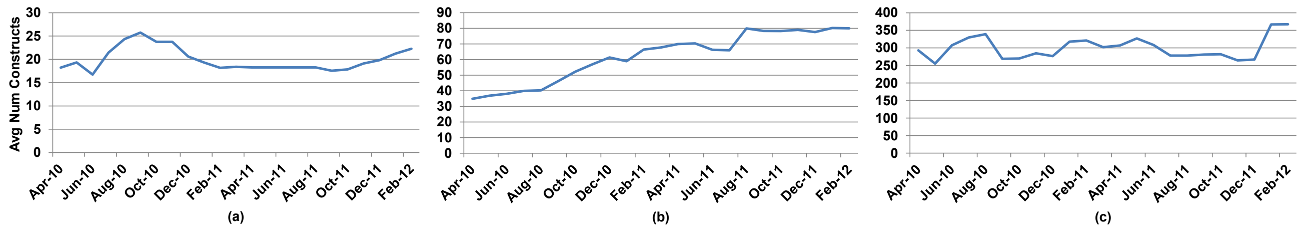
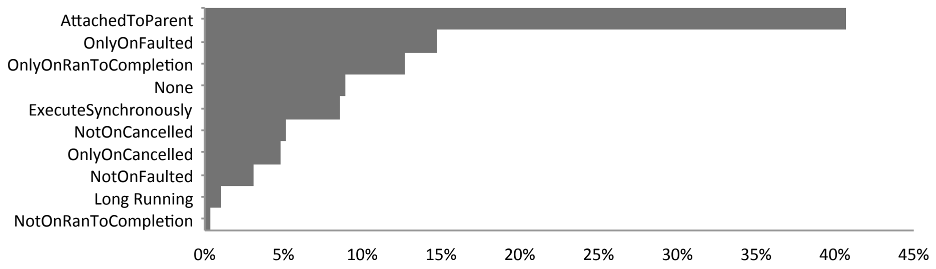


Figure 6: Distribution of Task Continuation Options



Q5: Which parallel patterns do developers embrace? Using the classification from the .NET Parallel Programming book [2], we analyzed the usage of six parallel patterns. Table 4 tabulates the usage of these patterns. The second column reports the popularity of task vs. data parallelism. The third column provides the names of patterns within each category, and the fourth column gives a brief explanation of the pattern. Last two columns show the number of individual instances of patterns, and the popularity percentage within its category.

Program Analysis: To automatically detect these patterns, we developed heuristics. We also randomly sampled from the inferred patterns to ensure that the reported patterns are inferred correctly. Because these patterns have several syntactical variations, it is very hard to detect all instances of patterns. Thus, the numbers that we report may be under-estimated, but not over-estimated.

For instance, to detect fork/join tasks pattern, ANALYZER tries to match pairs of statements that create tasks and statements that wait for tasks completion. Our heuristic is to match such pairs intra-procedurally, not inter-procedurally. Although this heuristic correctly labels many cases, it fails to label a pattern that creates tasks in one method and waits for completion in another method.

Second, to detect data parallelism, ANALYZER collects `Parallel.For`, `Parallel.ForEach` and `AsParallel` method calls. Since these method calls are perfect examples of data parallelism, we do not need to use heuristics. Loops that iterate over collections and launch a task to process each element are also counted by ANALYZER as data parallelism.

Next we describe how ANALYZER finds aggregation patterns. In a parallel aggregation pattern, the parallel loop uses unshared, local variables, that are combined at the end to compute the final result. ANALYZER searches for `Parallel.ForEach` and `Parallel.For` method calls that use a `ThreadLocal` object as a parameter. This is the parameter that encapsulates the unshared variable. As for PLINQ's code, ANALYZER checks whether the `AsParallel` method calls are followed by `Sum`, `Aggregate`, `Min`, etc. methods.

Finally, we illustrate how ANALYZER detects tasks that dynamically spawn other tasks, e.g., in a recursive divide-and-conquer algorithm. Starting from a task's body, it analyzes the method invocations inside. If one of these invocations calls recursively the method which encapsulates the starting task, ANALYZER labels it a dynamic task pattern.

Regular data parallelism is the most used parallel pattern in practice.

Q6: Which advanced features do developers use?

Now we focus on the most important parallel classes, `Parallel` and `Task`. Their methods take optional arguments related to performance and exception handling. Since these optional arguments distinguish TPL from other parallel libraries (e.g., TBB or Java's ForkJoinTask), we wonder if developers use them.

Parallel Class: `Parallel` class has `Invoke`, `For`, and `ForEach` methods. These methods can take an optional argument, `ParallelOptions`. With `ParallelOptions`, one can insert a cancellation token, limit the maximum concurrency, and specify a custom task scheduler. Of 852 method calls of `Parallel` class, only 3% use `ParallelOptions`.

Similarly, `For` and `ForEach` methods calls can take an optional `ParallelLoopState` which enables iterations to signal events (e.g., interrupt) to other iterations. Of 852 calls, only 3% use `ParallelLoopState`.

Task Class: When creating tasks, a developer can specify the execution order or the granularity of the task with an optional argument `TaskCreationOptions`. However, only 12% of task creation method calls use `TaskCreationOptions`.

Another advanced feature, `TaskContinuationOptions`, specifies the behavior for a task that is created as a continuation of another task. 28% of the continuation tasks use `TaskContinuationOptions`. Figure 6 tabulates the distribution of various continuation options.

Program Analysis: Because `TaskCreationOptions` and `TaskContinuosOperations` are enums, ANALYZER also visits field accesses.

The advanced features and optional arguments are rarely used in practice.

Table 2: Usage of TPL, Threading, PLINQ, and CC classes and their methods. The third column shows the percentage of usages of a class in comparison with usages of all classes from the library. The fourth column lists the main parallel methods in the parallel class. The fifth column shows the number of call sites for each method. The sixth column shows the percentage of usage of a method from one parallel class. The last column shows how many applications use this method.

Library	Class Name	% in Library	Method Name	# Call Sites	% in Class	# Apps
TPL	TaskFactory	30	StartNew	1256	72	286
			FromAsync	121	7	32
	Task	23	ContinueWith	372	28	122
			Wait	273	20	110
			Start	243	18	92
			Constructor	225	17	82
			WaitAll	172	13	91
	Parallel	14	For	450	53	102
			ForEach	365	43	133
			Invoke	37	4	23
	Task<TResult>	11	ContinueWith	536	86	113
			Constructor	85	14	40
Threading	Thread	17	Start	985	32	212
			Constructor	937	30	206
			Join	382	12	101
			Abort	294	10	82
	WaitHandle	11	WaitOne	1585	81	206
			Close	176	9	46
	Interlocked	10	CompareExchange	580	34	95
			CompareExchange	518	31	126
	ThreadPool	5	QueueUserWorkItem	814	90	125
PLINQ	ParallelEnumerable	100	AsParallel	221	24	150
			Select	136	15	46
			Where	62	7	30
			ForAll	61	7	29
CC	ConcurrentDictionary	72	Constructor	883	32	140
			TryGetValue	458	17	83
	ConcurrentQueue	13	Enqueue	194	38	63
			Constructor	178	35	70
	BlockingCollection	7	Add	85	30	25
			Constructor	78	28	25

Q7: Do developers make their parallel code unnecessarily complex? TPL provides some high-level constructs that allow developers to implement parallel code more concisely. These constructs decrease the number of lines of code and makes the parallel code easier to read, thus improving code quality.

Consider the example below, taken from *backgrounded* [25] application. It illustrates fork-join task parallelism.

The code on the bottom is the equivalent of the code on the top. It is much simpler to read because it uses `Parallel.Invoke`, a higher-level construct.

```
var runDaemons = new Task(RunDaemonJobs, ..token);
.....
var runScheduledJobs = new Task(RunScheduledJobs, ..
    token);
var tasks = new[] {runDaemons, ..., runScheduledJobs
    };
Array.ForEach(tasks, x => x.Start());
Task.WaitAll(tasks);
```

$$\rightleftharpoons$$

```
Parallel.Invoke(new ParallelOptions(CancellationToken
    =..token),
      RunDaemonJobs, ..., RunScheduledJobs);
```

Analyzer found that in 63 out of 268 regular fork/join task parallelism, the programmers could have used `Parallel.Invoke`, which would have reduced the complexity of the parallel code.

```
for (int i = 1; i <= threadCount; i++)
{
  var copy = i;
  var taskHandle = Task.Factory.StartNew(() =>
       DoInefficientInsert(server.Database.
       Configuration.ServerUrl, copy));
  tasks.Add(taskHandle);
}
Task.WaitAll(tasks);
```

$$\rightleftharpoons$$

```
Parallel.For(1,threadCount, (i)=> DoInefficientInsert
    (server.Database.Configuration.ServerUrl, i));
```

Analyzer found 189 `for/foreach` loops that launch tasks inside. Launching tasks inside a `for` loop is not only increasing the number of lines of code, but is also error-prone. In the code example above from *ravendb* [30], the programmer needs to make sure the iteration variable `i` is local to each task, otherwise the reading/writing accesses would exhibit data-races. 55 out of 189 cases could have used `Parallel.For` or `Parallel.ForEach`.

Table 3: Usage of Synchronization Constructs

Type	% in Types	Name	#	% in Type	# Apps
Locking	39	lock (language feature)	6643	89	361
		ReaderWriterLockSlim	258	3	68
		Monitor - Enter/Exit	245	3	66
		Mutex	94	1	46
		Semaphore	75	1	23
		ReaderWriterLock	65	1	24
		SpinLock	31	0.4	11
		SemaphoreSlim	20	0.3	10
Non-Blocking	26	Volatile Accesses	3212	65	152
		Interlocked Methods	1696	34	126
		Thread.MemoryBarrier	50	1	15
Implicit	21	CC Operations	4021	100	283
Signaling	9	ManualResetEvent	671	38	150
		AutoResetEvent	647	37	102
		Monitor - Wait/Pulse	168	10	31
		ManualResetEventSlim	167	10	37
		CountdownEvent	58	3	9
		Barrier	33	2	6
Blocking	5	Thread.Join	382	38	101
		Thread.Sleep	350	35	132
		Task.Wait	273	27	110

Table 4: Usage of Parallelism Patterns.

Main Pattern	%	Pattern Name	Brief explaination	#	%
Data Parallelism	68	Regular	parallel loops with `For, ForEach,` and PLINQ	954	92
		Aggregation	parallel dependent loops (map reduce algorithms)	82	8
Task Parallelism	32	Regular	regular fork&join tasks	268	56
		Futures	task dependency on results	155	32
		Pipeline	assembly line parallelism with `BlockingCollection`	41	8
		Dynamic	dynamically created tasks	18	4

There might be many other patterns of accidental complexity. We focused on two of them based on our own observations and discussions with the library designers.

Program Analysis: To detect tasks that could have used the `Parallel.Invoke`, Analyzer filters those tasks that are created and are also waited upon immediately.c More precisely, Analyzer checks that the main thread does not execute other statements between the statements that create and wait for tasks. It also checks that there are no dependencies among the created tasks, e.g., tasks are not linked with continuations like `ContinueWith`. In addition, Analyzer also discards the fork-join tasks that use `TaskCreationOptions` since `Parallel.Invoke` does not provide such a feature.

Despite the fact that parallel programs are already complex, developers make them even more complex than they need to be.

Q8: Are there constructs that developers commonly misuse?

`Parallel.Invoke(params action)` is a construct that executes in parallel the actions passed as arguments. It is a fork-join with blocking semantics: the main thread will wait until all actions specified as arguments have finished. Our analysis found that 11% of all usages of `Parallel.Invoke` take one action parameter in different applications. Consider the example from the *gpxviewer* [27] application:

```
Parallel.Invoke(() => i.ImportGPX(null, GPXFile));
```

Notice that in this case there is only one single action to be performed, and the main thread will block until this action has finished. In this case, the parallelism has no effect: the code executes sequentially, `ImportGPX` followed by the main thread. Developers might erroneously believe that `ImportGPX` will execute in parallel with the main thread, when in fact it doesn't.

When we look at PLINQ code, the `AsParallel` method converts an `Enumerable` into an `ParallelEnumerable` collection. Any method called on such a parallel enumeration will execute in parallel. We found 27 cases in 19 applications (representing 12% of all `AsParallel` usages) where developers misuse a parallel enumeration as the iteration source of a sequential `for` or `foreach` loop. Consider the example from the *profit* [29] application:

```
foreach (var module in Modules.AsParallel())
    module.Refresh();
```

Notice that despite `AsParallel` being placed at the end of the `Modules` collection, there is no operation performed on the "parallel" `Modules`. The `foreach` proceeds sequentially. Developers might erroneously believe that the code runs in parallel, when in fact it runs sequentially.

Program Analysis: To answer misusage questions, Analyzer encodes the erroneous usage patterns. For example, it searches for calls to `Parallel.Invoke` with one single argument, where the argument is an `Action` object (e.g., a method name or a lambda expression). For the PLINQ misusage, Analyzer searches for expressions where `AsParallel` is the last subexpression. We then manually analyze whether it is present in `for` or `foreach` loop whose iteration does not create any threads.

Misuse of parallel constructs can lead to code with parallel syntax but sequential execution.

5. IMPLICATIONS

There are several implications of our study. We organize them based on the community for which they are relevant.

5.1 Developers

Q1 (adoption): Becoming proficient with a new programming model requires a long-term commitment. Developers without parallel programming experience might ask themselves: should we learn how to use parallel libraries, or should we avoid them because they are a passing fad. Our data shows that 37% of all applications use the multi-threaded paradigm, so many developers will not be able to completely avoid multi-threaded programming. Sooner or later, most programmers will have to become familiar with this model.

Q2 (trends in adoption): Learning how to use effectively a library requires studying examples of the library API in real code. Where can developers find such examples? Our data shows that smaller applications are the early adopters of the parallel libraries. In addition, these applications have a much higher density of parallel constructs per thousand of SLOC. Looking in Fig 4, we can divide the average number of parallel constructs by 1K, 10K, 100K for small, medium, and large applications respectively. The average density is 5‰, 1.2‰, and .6‰ respectively. When taking into account the effort to understand unknown code, developers are better off looking for examples in small applications.

Q3 (usage): We notice a power-law distribution: 10% of the API methods are responsible for 90% of all usages. If we look at the classes, 15% of classes are responsible for 85% of all usages. This is good news for developers who are just learning parallel libraries: they can focus on learning a relatively small subset of the library APIs and still be able to master a large number of parallelism scenarios.

5.2 Library Designers

Q3 (usage): Surprisingly lower usage numbers like the ones for PLINQ can highlight the APIs that need better documentation and more advertisement on mailing lists, developer forums, etc.

Q4 (synchronization): Designers of concurrent data structures and synchronization constructs are always asking themselves on what to focus. Table 3 shows that developers are more likely to use the faster synchronization constructs. For example, `ReaderWriterLockSlim` is used four times more often than the slower `ReaderWriterLock`.

Q6 (advanced features): Library designers pay special attention to making the APIs easier to use. This involves making the syntax for the common case more concise. We observed in Figure 6 that programmers prefer to create new tasks attached to the parent task (40% are `AttachedToParent`). So, library designers could make this the default behavior for nested tasks. Similarly, 80% of times when developers used `ParallelOptions` they only specify one single option, `MaxDegreeOfParallelism`. Library designers may make this an argument to `Parallel` class methods instead of encapsulating it in `ParallelOptions`.

Additionally, 60% of the times developers overwrite `MaxDegreeOfParallelism`; they make it equal with the number of processors found at runtime. This means that developers are not happy about the degree of parallelism chosen by .NET. TPL architects should consider making the number of processors the default value for the max degree of parallelism. Stephen Toub, who is one of the main architects of TPL, confirmed our suggestion.

Q8 (misusage): Library designers can also remove the constructs that are error-prone. We found that developers are not aware that `Parallel.Invoke` is a blocking operation, so they invoke it with one single action parameter (which results in executing the code sequentially). Library designers may consider removing `Parallel.Invoke` version that takes only one action parameter.

5.3 Researchers

Q1 (adoption): Since we list the domains and the applications that use parallelism most heavily, the researchers can use them to create benchmarks for parallel programming.

Q4 (synchronization): Researchers that work on ensuring correctness (e.g., data-race detection) should notice from Table 3 that developers use a wide variety of synchronization constructs. Thus, data-race detectors should also model these other synchronization constructs.

.NET parallel libraries provide more than 20 synchronization constructs divided into 5 different categories. It is difficult for developers to select the most appropriate one. Each construct has tradeoffs, depending on the context where it is used. This is an opportunity for developing intelligent tools that suggest which constructs developers should use in a particular context.

Q7 (complexity): Researchers in the refactoring community can get a wealth of information from the usage patterns. For example, developers should use higher-level constructs to manage the complexity of the parallel code: 24% of fork-join tasks can be converted to `Parallel.Invoke`, which reduces many lines of code. Refactorings that allow programmers to improve the readability of their parallel code have never been automated before, but are invaluable.

6. THREATS TO VALIDITY

Construct: Are we asking the right questions? We are interested to asses the state of the practice w.r.t. usage of parallel libraries, so we think our questions provide a unique insight and value for different stakeholders: potential users of the library, designers of the library, researchers.

Internal: Is there something inherent to how we collect and analyze the usage that could skew the accuracy of our results? Microsoft's Roslyn, on which we built our program analysis, is now in the Community Technology Preview and has known issues (we also discovered and reported new bugs). For some AST nodes, we did not get semantic information. We printed these nodes, and they are not parallel constructs, thus they do not affect the accuracy.

Second, the study is only focusing on static usage of parallel constructs, but one use of a construct (i.e., a call site) could correspond to a large percentage of execution time, making it a very parallel program. Likewise, the opposite could be true. However, we are interested in the developer's view of writing, understanding, maintaining, evolving the code, not on the performance tools' view of the code (i.e., how much of the total running time is spent in multi-threaded code). For our purpose, a static usage is much more appropriate.

Third, do the large applications shadow the usage of constructs in the smaller applications? Tables 2 and 3 provide the total tally of constructs across all applications and there is a possibility that most usages come from a few large applications. To eliminate this concern, the last column in the two tables list the number of applications that use each kind of construct. Due to lack of space, we do not present the mean, max, min, standard deviation in the paper, but they are available on the companion website [36].

Fourth, static analysis offers limited insight in the performance of parallel applications. While the real purpose of using parallel libraries is to improve performance, we can not estimate this based solely on static analysis.

External: Are the results generalizable to other programming languages, libraries, and applications? First, despite the fact that our corpus contains only open-source applications, the 655 applications span a wide range from tools, IDEs, games, databases, image processing, video encoding/decoding, search engines, web systems, etc., to third party libraries. They are developed by different teams with 1609 contributors from a large and varied community. Still, we cannot be sure whether this usage is representative for proprietary applications.

While we answer the questions for the C# ecosystem, we expect they can cross the boundary from C# to Java and C++. For example, we expect such empirical studies that reveal pain-points and common errors in using parallel library APIs to be useful to the TBB/C++ and `j.u.c.`/Java designers since these libraries provide very similar abstractions. Furthermore, C# with .NET is used on wide range of platforms – desktop, server, mobile, and web applications.

Reliability: Can others replicate our study? A detailed description of our results with fine-grained reports and analysis tools are available online [36].

7. RELATED WORK

There are several empirical studies [1, 11, 13, 35] on the usage of libraries or programing language features. These studies rely only on syntactic analysis. To best of our knowledge, ours is the first large-scale study that uses both syntactic and semantic analysis, thus increasing the accuracy of the usage statistics.

Robillard and DeLine [32] study what makes large APIs hard to learn and conclude that one of the important factors is the lack of usage examples. Our current study provides lots of usage examples from real code which can hopefully educate newcomers to the parallel library.

Monperrus et al. [18] study the API documentation of several libraries and propose a set of 23 guidelines for writing effective API documentation.

Dig et al. [7] and Pankratius et al. [21] analyzed concurrency-related transformations in a few Java applications. Our current study does not look at the evolution of concurrent applications, but at how developers use parallel libraries.

Pankratius [20] proposes to evaluate the usability of parallel language constructs by extending the Eclipse IDE to record usage patterns and then infer correlations using data mining techniques.

Other empirical studies on the practice of multicore programming [5] focused on identifying the contented resources (e.g., shared cache) that adversely impact the parallel performance. Our fourth research question identifies a wide variety of synchronization constructs that impact performance.

In the same spirit like our paper, Parnin et al. [22] study the adoption patterns of Java generics in open-source applications. While some of our research questions specifically address adoption patterns (Q1 and Q2), the remaining questions provide an extensive exploration into the practice of using parallel libraries.

Others [12] have studied the correlation between usage of the MPI parallel library and productivity of the developers.

The closest work to ours is done by Weslley et al. [35] on the usage of concurrent programming constructs in Java. They study around 2,000 applications and give some coarse-grain usage results like the number of synchronized blocks and the number of classes extending `Thread`. In contrast, our study looks at every parallel construct in the parallel libraries, and we also look at how these constructs form patterns and structures. Although they analyze the usage of very few constructs, their results are not accurate due to missing type information because they only perform lexical analysis. Also, their count of the constructs' usage can be misleading. For example, they measure the usage of `java.util.concurrent` by counting statements that import the library. In our study, there are many applications that import TPL but never invoke any construct. For example, there is an application, *DotNetWebToolkit* [26], that imports TPL 111 times but invokes TPL just once.

8. CONCLUSION

Parallelism is not a passing fad; it is here for the foreseeable future. To encourage more programmers to embrace parallelism, we must understand how parallel libraries are currently used. Our empirical study on the usage of modern parallel libraries reveals that programmers are already embracing the new programming models. Our study provides tremendous education value for developers who can educate themselves on how to correctly use the new parallel constructs. It also provides insights into the state of the practice in using these constructs, i.e., which constructs developers find tedious and error-prone. Armed with this information, library designers and researchers can develop effective tools and techniques to better match the current practice and transform it.

More studies are needed if we want to fully understand the state of the practice, and we hope that our study inspires follow-up studies.

Acknowledgements: This research was partially funded by Microsoft through an SEIF award, and by Intel through a gift grant. The authors would like to thank Stephen Toub, Kevin Pilch-Bisson, Shan Lu, Mark Grechanik, Michael Hind, Stas Negara, Cosmin Radoi, Yu Lin, Phil Miller, Milos Gligoric, Samira Tasharofi, Mohsen Vakilian, Yun Young Lee, Darko Marinov, and anonymous reviewers for providing helpful feedback on earlier drafts of this paper.

9. REFERENCES

[1] O. Callaú, R. Robbes, É. Tanter, and D. Röthlisberger. How developers use the dynamic features of programming languages: the case of smalltalk. In *MSR '11: Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 23–32, 2011.

[2] C. Campbell, R. Johnson, A. Miller, and S. Toub. *Parallel Programming with Microsoft .NET: Design Patterns for Decomposition and Coordination on Multicore Architectures.* Microsoft Press, 2010.

[3] Collections.Concurrent (CC). July'12, `http://msdn.microsoft.com/en-us/library/dd997305.aspx/`.

[4] CodePlex. July'12, `http://codeplex.com`.

[5] T. Dey, Wei Wang, J.W. Davidson, and M.L. Soffa. Characterizing multi-threaded applications based on shared-resource contention. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 76–86, 2011.

[6] D. Dig. A refactoring approach to parallelism. *Software, IEEE*, 28(1):17–22, 2011.

[7] D. Dig, J. Marrero, and M. D. Ernst. How do programs become more concurrent? a story of program transformations. In *IWMSE '11: Proceedings of the 4th International Workshop on Multicore Software Engineering*, pages 43–50, 2011.

[8] Github. July'12, `https://github.com`.

[9] M. Gladwell. *The Tipping Point: How Little Things Can Make a Big Difference.* Back Bay Books, 2002.

[10] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice.* Addison-Wesley Professional, 2006.

[11] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi, D. Poshyvanyk, C. Fu, Q. Xie, and C. Ghezzi. An empirical investigation into a large-scale java open source code repository. In *ESEM '10: Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–10, 2010.

[12] L. Hochstein, F. Shull, and L. B. Reid. The role of mpi in development time: a case study. In *SC Conference*, pages 1–10, 2008.

[13] S. Karus and H. Gall. A study of language usage evolution in open source software. In *MSR '11: Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 13–22, 2011.

[14] D. Lea. *Concurrent Programming in Javał: Design Principles and Pattern.* Prentice Hall, 1999.

[15] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. *ACM SIGPLAN Not.*, 44(10):227–242, 2009.

[16] B. P. Lester. *The Art of Parallel Programming.* 1st World Publishing, Inc., 2006.

[17] T. G. Mattson, B. A. Sanders, and B. L. Massingill. *Patterns for Parallel Programming.* Addison-Wesley Professional, 2005.

[18] M. Monperrus, M. Eichberg, E. Tekes, and M. Mezini. What should developers be aware of? an empirical study on the directives of api documentation. *Empirical Software Engineering*, Online Edition, 2011.

[19] Survival of the Forgest. July'12, `http://redmonk.com/sogrady/2011/06/02/blackduck-webinar/`.

[20] V. Pankratius. Automated usability evaluation of parallel programming constructs. In *ICSE '11 (NIER track): Proceedings of the 33rd International Conference on Software Engineering*, pages 936–939, 2011.

[21] V. Pankratius, C. Schaefer, A. Jannesari, and W. F. Tichy. Software engineering for multicore systems: an experience report. In *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*, pages 53–60, 2008.

[22] C. Parnin, C. Bird, and E. Murphy-Hill. Java generics adoption: how new features are introduced, championed, or ignored. In *MSR '11: Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 3–12, 2011.

[23] Parallel Language Integrated Query (PLINQ). July'12, `http://msdn.microsoft.com/en-us/library/dd460688.aspx/`.

[24] AppVisum Project. July'12, `https://github.com/Alxandr/AppVisum.Sys`.

[25] Backgrounded Project. July'12, `http://www.github.com/swedishkid/backgrounded`.

[26] DotNetWebToolkit Project. July'12 `https://github.com/chrisdunelm/DotNetWebToolkit`.

[27] Gpxviewer Project. July'12, `https://github.com/andrewgee/gpxviewer`.

[28] PasswordGenerator Project. July'12, `https://github.com/PanosSakkos/PasswordGenerator`.

[29] Profit Project. July'12, `http://profit.codeplex.com/`.

[30] Ravendb Project. July'12, `https://github.com/ravendb/ravendb`.

[31] The Roslyn Project. July'12, `http://msdn.microsoft.com/en-us/hh500769`.

[32] Martin P. Robillard and Robert Deline. A field study of api learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011.

[33] System.Threading. July'12, `http://msdn.microsoft.com/en-us/library/system.threading`.

[34] Threading Building Block (TBB). July'12, `http://threadingbuildingblocks.org/`.

[35] W. Torres, G. Pinto, B. Fernandes, J. P. Oliveira, F. A. Ximenes, and F. Castor. Are java programmers transitioning to multicore?: a large scale study of java floss. In *SPLASH '11 Workshops*, pages 123–128, 2011.

[36] Companion TPL usage data. July'12, `http://learnparallelism.net`.

[37] ForkJoinTask Doug Lea's Workstation. July'12, `http://gee.cs.oswego.edu/dl/jsr166/dist/jsr166ydocs/jsr166y/ForkJoinTask.html`.

Transformation for Class Immutability

Fredrik Kjolstad Danny Dig Gabriel Acevedo Marc Snir
University of Illinois at Urbana-Champaign
{kjolsta1, dig, acevedo7, snir}@illinois.edu

ABSTRACT

It is common for object-oriented programs to have both mutable and immutable classes. Immutable classes simplify programing because the programmer does not have to reason about side-effects. Sometimes programmers write immutable classes from scratch, other times they transform mutable into immutable classes. To transform a mutable class, programmers must find all methods that mutate its transitive state and all objects that can enter or escape the state of the class. The analyses are non-trivial and the rewriting is tedious. Fortunately, this can be automated.

We present an algorithm and a tool, Immutator, that enables the programmer to safely transform a mutable class into an immutable class. Two case studies and one controlled experiment show that Immutator is useful. It (i) reduces the burden of making classes immutable, (ii) is fast enough to be used interactively, and (iii) is much safer than manual transformations.

Categories and Subject Descriptors: D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement

General Terms: Design, Management

Keywords: Program transformation, immutability

1. INTRODUCTION

An immutable object is one whose state can not be mutated after the object has been initialized and returned to a client. By object state we mean the *transitively reachable state*. That is, the state of the object and all state reachable from that object by following references.

Immutability makes sequential programs simpler. An immutable object, sometimes known as a *value object* [17], is easier to reason about because there are no side-effects [7]. Applications that use immutable objects are therefore simpler to debug. Immutable objects facilitate persistent storage [2], they are good hash-table keys [12], they can be compared very efficiently by comparing identities [2], they can reduce memory footprint (through interning/memoization [12,14] or flyweight [8]). They also enable compiler optimizations such as reducing the number of dynamic reads [16]. In fact, some argue that we should always use immutable classes unless we explicitly need mutability [3].

In addition, immutability makes distributed programming simpler [2]. With current middleware technologies like Java RMI, EJB, and Corba, a client can send messages to a distributed object via a local proxy. The proxy implements an update protocol, so if the distributed object is immutable then there is no need for the proxy.

Moreover, as parallel programming becomes ubiquitous in the multicore era, immutability makes parallel programming simpler [9,13]. Since threads can not change the state of an immutable object, they can share it without synchronization. An immutable object is *embarrassingly thread-safe*.

However, mainstream languages like Java, C#, and C++ do not support deep, transitive immutability. Instead, they only support shallow immutability through the `final`, `readonly`, and `const` keywords. This is not enough, as these keywords only make *references* immutable, not the objects referenced by them. Thus, the transitive state of the object can still be mutated.

To get the full benefits of immutability, deep immutability must therefore be *built into* the class. If a class is *class immutable*, none of its instances can be transitively mutated. Examples in Java include `String` and the classes in the `Number` class hierarchy.

It is common for OO programs to contain both mutable and immutable classes. For example, the JDigraph open-source library contains `MapBag` and `ImmutableBag`. `MapBag` is intended for cases where mutation is frequent, and `ImmutableBag` where mutations are rare.

Sometimes programmers write an immutable class from scratch, other times they refactor a mutable class into an immutable class. The refactoring can be viewed as two related technical problems:

1. The *conversion problem* consists of generating an immutable class from an existing mutable class.

2. The *usage problem* consists of modifying client code to use the new immutable class in an immutable fashion.

This paper solves the conversion problem. To create an immutable class from a mutable class (from here on referred as the *target class*), the programmer needs to perform several tasks. The programmer must search through the methods of the target class and find all the places where the transitive state is mutated. This task is further complicated by polymorphic methods and mutations nested deep inside call chains that may extend into third party code.

Moreover, the programmer must ensure that objects in the transitive state of the target class do not escape from it, otherwise they can mutated by client code. Such escapes can happen through return statements, parameters, or static fields. Finding objects that escape is non-trivial. For example, an object can be added to a `List` that is returned from a method, causing the object to escape along with the `List`.

Furthermore, once the programmer found all mutations, she must rewrite mutator methods, for example by converting them to factory methods. She must also handle objects that enter or escape the class, for example by cloning them.

In 346 cases we studied these code transformations required changing 45 lines of code per target class, which is tedious. Furthermore, it required analyzing 57 methods in the call graph of each target class to find mutators and entering/escaping objects. Because this analysis is inter-procedural and requires reasoning about the heap, it is non-trivial and error-prone. In a controlled experiment where 6 experienced programmers converted JHotDraw classes to immutable counterparts, they took an average of 27 minutes, and introduced 6.37 bugs per class.

To alleviate the programmer's burden when creating an immutable class from a mutable class, we designed an algorithm and implemented a tool, Immutator, that works on Java classes. We developed Immutator on top of Eclipse's refactoring engine. Thus, it offers all the convenience of a modern refactoring tool: it enables the user to preview and undo changes and it preserves formating and comments. To use it the programmer selects a target class and chooses Generate Immutable Class from the refactoring menu. Immutator then verifies that the transformation is safe, and rewrites the code if the preconditions are met. However, if a precondition fails, it warns the programmer and provides useful information that helps the programmer fix the problem.

At the heart of Immutator are two inter-procedural analyses that determine the safety of the transformation. The first analysis determines which methods mutate the transitive state of the target class. The second analysis is a class escape analysis that detects whether objects in the transitive state of the target class state *may* escape. Although Immutator transforms the source code, the analyses work on bytecode and correctly account for the behavior of third-party Java libraries.

There is a large body of work [1,18–20] on detecting whether methods have side effects on program state. Previous analyses were designed to detect *any* side effect, including changes to objects reachable through method arguments and static variables. In contrast, our analysis intersects the mutated state with the objects reachable through the `this` reference. Therefore, it only reports methods that have a side effect on the current target object's state.

Similarly, previous escape analyses [4, 24] report any object that escapes a method, including locally created objects. Our analysis only reports those escaping objects that are also a part of the transitive state of the target class.

This paper makes the following contributions:

Problem Description While there are many approaches to specifying and checking immutability this is, to the best of our knowledge, the first paper that describes the problems and challenges of transforming a mutable class into an immutable class.

Transformations We present the transformations that convert a Java class to an immutable Java class.

Algorithm We have developed an algorithm to automatically convert a mutable class to an immutable class. The algorithm performs two inter-procedural analyses; one that determines the mutating methods, and one that detects objects that enter or escape the target class. Based on information retrieved from these and other analyses our algorithm checks preconditions and performs the mechanical transformations necessary to enforce immutability.

Implementation We have implemented the analyses and code transformations in a tool, Immutator, that is integrated with the Eclipse IDE.

Evaluation We ran Immutator on 346 classes from known open-source projects. We also studied how open-source developers create immutable classes manually. Additionally, we designed a controlled experiment with 6 programmers transforming JHotDraw classes manually. The results show that Immutator is useful. First, the transformation is widely applicable: in 33% of the cases Immutator was able to transform classes with no human intervention. Second, several of the manually-performed transformations are not correct: open-source developers introduced an average of 2.1 errors/class, while participants introduced 6.37 errors/class; in contrast, Immutator is safe. Third, on average, Immutator runs in 2.33 seconds and saves the programmer from analyzing 57 methods and changing 45 lines per transformed class. In contrast, participants took an average of 27 minutes per class. Thus, Immutator dramatically improves programmer productivity.

Immutator as well as the experimental data can be downloaded from: `http://refactoring.info/tools/Immutator`

2. MOTIVATING EXAMPLE

We describe the problems and challenges of transforming a mutable class into an immutable class using a running example. Class `Circle`, shown on the left-hand side of Fig. 1, has a center, stored in field `c`, and a radius, stored in field `r`. There are several methods to modify or retrieve the state. The programmer decides to transform this class into an immutable class, since it makes sense to treat mathematical objects as value objects.

Transforming even a simple class like `Circle` into an immutable class, as shown on the right-hand side of Fig. 1, is non-trivial. First, the programmer must find all the mutating methods. Method `setRadius` on line 19 is a direct mutator, and is easy to spot because it assigns directly to a field. Method `moveTo(int, int)` on line 27 is a mutator too. However, the code on line 30 does not change the value of `c` directly, but instead changes the object that `c` references. Therefore, this method mutates the transitive state of `Circle`. Method `moveBy` on line 34 is another mutator that does not mutate the object directly. Instead, it mutates state indirectly by calling `moveTo(Point)`. Finding all mutators (transitive and indirect) is complicated by long call chains, polymorphic methods, aliases, and third-party library code.

Furthermore, the programmer must locate all the places where an object enters or escapes the target class. Consider a client that creates a `Point` object and passes it to

```java
1  public class Circle {
2    private Point c = new Point(0, 0);
3    private int   r = 1;
4
5
6
7
8
9
10
11
12
13
14
15   public int getRadius() {
16     return r;
17   }
18
19   public void setRadius(int r) {
20     this.r = r;
21   }
22
23   public void moveTo(Point p) {
24     this.c = p;
25   }
26
27   public void moveTo(int x, int y) {
28
29
30     c.setLocation(x, y);
31
32   }
33
34   public void moveBy(int dx, int dy) {
35     Point center = new Point(c.x+dx, c.y+dy);
36     moveTo(center);
37
38   }
39
40   public Point getLocation() {
41     return c;
42   }
43 }
```

```java
1  public final class ImmutableCircle {
2    private final Point c;
3    private final int   r;
4
5    public ImmutableCircle() {
6      this.c = new Point(0, 0);
7      this.r = 1;
8    }
9
10   private ImmutableCircle(Point c, int r) {
11     this.c = c;
12     this.r = r;
13   }
14
15   public int getRadius() {
16     return r;
17   }
18
19   public ImmutableCircle setRadius(int r) {
20     return new ImmutableCircle(this.c, r);
21   }
22
23   public ImmutableCircle moveTo(Point p) {
24     return new ImmutableCircle(p.clone(), this.r);
25   }
26
27   public ImmutableCircle moveTo(int x, int y) {
28     ImmutableCircle _this =
29       new ImmutableCircle(this.c.clone(), this.r);
30     _this.c.setLocation(x, y);
31     return _this;
32   }
33
34   public ImmutableCircle moveBy(int dx, int dy) {
35     Point center = new Point(c.x+dx, c.y+dy);
36     ImmutableCircle _this = moveTo(center);
37     return _this;
38   }
39
40   public Point getLocation() {
41     return c.clone();
42   }
43 }
```

Figure 1: Immutator converts a mutable `Circle` (left pane) into an immutable class (right pane).

`moveTo(Point)`. Since the client holds a reference to the point, it can still mutate the object through the retained reference. The programmer may not have access to all existing and future client code so she must conservatively assume that the target class can be mutated through entering and escaping objects. Therefore, to enforce deep immutability, the programmer must find all the places where objects enter the target class (line 23–24) or escape (line 41), and clone them. However, the programmer should avoid excessive cloning and only clone where absolutely required.

Even for this simple example, the transformation requires inter-procedural analysis (line 30 and 36), which must take pointers into account (line 30). Our approach combines the strength of the programmer (the higher-level understanding of where immutability should be employed) with the strengths of a tool (analyzing many methods and making mechanical transformations).

Immutator automatically handles the rewriting (Section 4) and analysis (Section 5) required to make a class immutable.

3. IMMUTATOR

We implemented our algorithm for Generate Immutable Class as a plugin in the Eclipse IDE. To use Immutator, the programmer selects a class and then chooses the Generate Immutable Class option from the refactoring menu. Before applying the changes, Immutator gives the programmer the option to preview them in a before-and-after pane. Then Immutator makes the class *deeply* immutable.

Our algorithm transforms the target class in-place. However, the tool makes a copy of the target class and then transforms this copy. This provides the programmer with two variants of the same class: a mutable and an immutable one. The programmer decides where it makes sense to use one over the other.

However, the programmer can not use the deeply immutable version if the class is to be used in client code that relies on structural sharing of mutable state. Consider a `Graph` that contains mutable `Node` objects. The semantics of the `Graph` class ensure that several nodes can share the same successor node. If the programmer made `Graph` immutable, Immutator would change mutator methods like `addEdge(n1,n2)` to clone the entering nodes, thus transforming the graph into a tree. On the other hand, structural sharing of immutable objects does not contradict with deep-copy immutable semantics. If the `Graph` contained immutable `Node` objects, then Immutator would not clone `Node` objects, thus preserving the sharing semantics of the original class.

Before transforming the target class, Immutator checks that it meets four preconditions, and reports failed preconditions to the programmer. The programmer can decide to ignore the warnings and proceed, or cancel the operation, fix the root cause of the warnings and then re-run Immutator.

3.1 Transformation Preconditions

Immutator checks the following preconditions:

Precondition #1 The target class can only have superclasses that do not have any *mutable* state.

Precondition #2 The target class can not have subclasses as these can add mutable state to the target objects.

Precondition #3 Mutator methods in the target class must have a `void` return type and must not override methods in superclasses. This is because Immutator rewrites mutator methods to return new instances of the target class and must use the return type for this. Methods in Java can only return one value and it is not allowed to change the return type when overriding a method.

Precondition #4 Objects that enter or escape the transitive state of the target class must either already implement `clone`, or the source code of their classes must be available so that a `clone` method can be added.

While these preconditions may seem restrictive, we believe that value classes are likely to meet them. For example, software that follows the command-query separation principle (methods either perform an operation, or return a value) will not have mutators with non-void return types, thus meeting precondition 3. Furthermore, preconditions 1 and 2 are limitations of the current implementation, and not inherent to the approach. We leave for future work to refactor a whole class hierarchy.

4. TRANSFORMATIONS

This section describes the transformations that Immutator applies to the target class. We will use the motivating example introduced in Fig. 1 to illustrate the transformations.

Make fields and class final First, Immutator makes all the fields of the class `final`. Final fields in Java can only be initialized once, in constructors or field initializers. Immutator also makes the target class final. This prevents it from being extended with subclasses that add mutable state.

Generate constructors Immutator adds two new constructors (line 5 and 10). The first constructor is the default constructor and it does not take any arguments. This constructor initializes each field to their initializer value in the original class or to the default value if they had none. The second constructor is a *full* constructor. It takes one initialization argument for each field, and is private as it is only used internally to create instances.

4.1 Convert Mutators into Factory Methods

Since the fields are `final`, methods can not assign to them. Immutator converts mutator methods into factory methods that create and return new objects with updated state.

We call a method a *mutator* if it (i) assigns to a field in the transitive state of a target class instance, or (ii) invokes a method that is a mutator method.

Convert direct mutators Setters are a common type of mutator in object-oriented programs. Lines 19–21 on the right-hand side of Fig. 1 show the transformation of `setRadius` to a factory method. Immutator changes (i) the return type to the type of the target class, and (ii) the method body to construct and return a new object, created using the full constructor. The constructor argument that is assigned to the `r` field is set to the right-hand-side of the assignment expression. The arguments for the other fields (e.g., `c`) are copied from the current object. Thus, the factory method returns a new object where the `r` field has the new value, while the other fields remain unchanged.

However, not all mutators are simple setters. Some contain multiple statements, while others mutate fields indirectly by calling other mutators. `moveBy`, on line 34–38, demonstrates both of these traits. It contains two statements, and it mutates `c` indirectly by calling `moveTo`.

The right-hand side shows how Immutator transforms `moveBy` into a factory method. It introduces a new local reference, called `_this`, to act as a placeholder for Java's built-in `this` reference. After `_this` is defined at the first mutation, Immutator replaces every explicit and implicit `this` with `_this`.

Furthermore, for every statement that calls a mutator, Immutator assigns the return value of the method (which is now a factory method) back to `_this`. Thus, the rest of the method sees and operates on the object constructed by the factory method. Finally, the `_this` reference is returned.

An interesting property of this technique is that it shifts the mutations from the target object to the `_this` reference. That is, instead of mutating the object pointed to by `this`, it mutates the state of `_this`. Ideally, Immutator would reassign back to `this`, but in Java the built-in `this` reference can not be assigned to. Therefore, Immutator replaces it with the mutable place-holder `_this`.

Convert transitive mutators Consider the `moveTo(int, int)` method on line 27–32. Although this method never assigns to the `c` field, it still mutates `c`'s transitive state through the `setLocation` method. Immutator notices that the method `setLocation` does not belong to the target class, but to `java.awt.Point` in the GUI library. Therefore, Immutator can not rewrite `setLocation` into a factory method.

As before, Immutator creates the `_this` reference, and returns it at the end of the method. Furthermore, Immutator clones `c`, so that the mutation does not affect the original object referenced by `this`. The cloned `c` is passed as an argument to the new `Circle`, which is assigned to `_this`. Since `_this.c` now refers to a clone of the original `this.c`, we can allow the mutation through `setLocation`.

4.2 Clone the Entering and Escaping state

Another way the transitive state of the target object can be mutated is if client code gets hold of a reference to an object in its internal state, and then mutates it outside of the target class. This can happen in two ways: (i) through objects that are entering the target class (e.g., `Point p` on line 24), or (ii) through objects that are escaping the target class (e.g., `c` on line 41).

An object *enters the target class* if it is visible from client code, and is assigned to a field in the transitive state of the target class. For example, the client code could call `moveTo(Point)` and then mutate the point through the retained reference.

We define a *target class escape* as an escape from any of its methods, including constructors. An escape from a method means that an object that is transitively reachable through a field of the target class is visible to the client code after the method returns. For example, on line 41, the object pointed to by `c` escapes through the return statement, and can then be mutated by client code. Escapes can also occur through parameters and static fields

If an object enters or escapes then current or future client code may perform any operations on it, and Immutator must conservatively assume that it will be mutated.

Immutator handles entering and escaping objects by inserting a call to the `clone` method to perform a deep copy of the object in question, as seen on the right-hand side of line 24 and 41. However, if the entering or escaping object is itself immutable, Immutator does not clone it. The current implementation considers the following classes to be immutable: the target class, `String`, Java primitive wrapper classes (e.g., Integer), and classes annotated with `@Immutable`.

When Immutator needs to use a `clone` method that does not exist, it generates a `clone` stub and reports this to the user, who must implement the stub.

Immutator avoids excessive cloning. For example, it could have inserted a `clone` call in the private constructor on line 11, but this would have caused unnecessary cloning. Instead, Immutator calls `clone` sparsely, at the location where objects enter or escape, or where the target class is transitively mutated (e.g., on line 30).

Moreover, Immutator ensures some structural sharing [11], by not adding calls to clone objects that enter from another instance of the same class. For example, when the transformed `setRadius` method is called, a new instance of `ImmutableCircle` is created (line 20 on the right-hand side). However, only the `r` field is mutated, while the `c` field (the center) remains the same. Since the old circle will not mutate the center, and since the center is not visible from the outside, the new circle does not have to clone it. The result is that the two circles share a part of their state.

5. PROGRAM ANALYSIS

In the previous section we discussed the transformations to make an existing class immutable. In order to perform these transformations Immutator first analyzes the source code to establish preconditions and to collect information for the transformation phase.

Immutator does not perform a whole-program analysis, but only analyses the target class and methods invoked from it. Thus, the analysis is fast and can be used interactively.

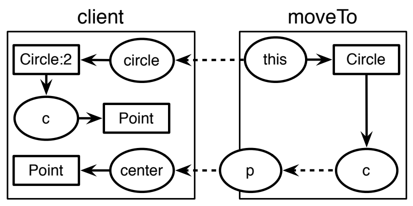
At the heart of Immutator are two analyses. The first detects mutating methods so that these can be converted to factory methods. The second detects objects that enter or escape the target class so that they can be cloned. Both analyses work on a representation generated from byte code, and can therefore analyze third-party library code.

5.1 Analysis Data Structures

Immutator creates several data structures that are necessary for the program analyses. It constructs both of these data structures using the WALA analysis library [23] as a starting point.

The first data structure is a *call graph* (CG) starting from every non-private method of the target class. The call graph is used to find mutators as well as entering/escaping objects. For each node in the callgraph Immutator also constructs a *control flow graph* (CFG) that is used later to find transitive mutations and to build a points-to graph.

In addition to the control-flow structures, Immutator builds a *points-to graph* (PTG). Points-to analysis establishes which pointers (or *references* in Java terminology) point to which *storage locations*. We model the heap storage locations as object allocation sites.



```
 1 public void client() {
 2   Circle circle = new Circle();
 3   Point center = new Point(5, 5);
 4   circle.moveTo(center);
 5 }

 6 public class Circle {
 7   // ...
 8   public void moveTo(Point p) {
 9     this.c = p;
10   }
11 }
```

Figure 2: An example points-to graph

An example of the points-to graphs that Immutator create is illustrated using a simple client program in Fig. 2. The graph contains two types of nodes: references, depicted graphically as ellipses, and heap-allocated objects depicted as rectangles. The explicit formal arguments of a method are placed on the border of its bounding box. Directed edges connect references to the objects they point to. For example, the object created on line 2 is represented by the rectangle `Circle:2`, and the reference it is assigned to on the same line is represented by the `circle` ellipse. This object has a field `c`, which is constructed in the field initializer of class `Circle`. References are connected to their objects by directed edges. The points-to graph only captures relations between references and objects, and does not include scalar primitives.

Notice that the assignment on line 9 creates an alias between the references `c` and `p`. This is represented in the points-to graph as a dashed arrow, and is called a *deferred edge*. A deferred edge means that `c` can point to any objects that `p` can point to. We also use deferred edges to represent the relations between formal and actual arguments since Java is a pass-by-value language where actuals are copied into the formals.

Immutator constructs this points-to graph using an inclusion-based (Andersen-style) points-to analysis. The analysis is partly *flow-sensitive* with respect to local variables as it is computed from an SSA representation of the source code. It is also *context-insensitive* since it does not take the calling context into account.

Note that Immutator constructs additional nodes that do not exist in the program when they are needed to complete a method summary. One such example is the `Circle` allocation site and its `c` field in the `moveTo` method. When Immutator creates the summary for `moveTo`, the `this` reference is not connected to any allocation sites. Therefore, Immutator constructs additional object and field nodes in order to add the deferred edge that represents the assignment of `p` to `c`.

5.2 Detecting Transitive Mutators

The goal of this analysis is to find the methods that are mutating the transitive state of the target object, either directly or indirectly by calling another mutator method.

Fig. 3 shows the pseudocode of the algorithm for detecting mutator methods. The algorithm takes as input the set M of

```
Input: $M \leftarrow$ Set of Methods in CG,
       $MTC \leftarrow$ Methods in Target Class,
       $PTG \leftarrow$ Points-to Graph
Output: $MUT$ // Set of mutator methods

  // Step 1: Find the transitive state of the target class
  $TARG \leftarrow \cup_{m \in MTC}(transitiveClosure(this, PTG))$

  // Step 2: Find transitive mutators
  for each $m$ in $M$ do
    for each $fieldAssignments$ $<o.f = expr>$ do
      if $o$ can reach $TARG$ through deferred edges in PTG then
        $MUT \leftarrow MUT \cup m$

  // Step 3: Find indirect mutators
  for each $m$ in $M$, in reverse topological order do
    for each $m'$ in $callees(m)$ do
      if $m' \in MUT$ then
        $MUT \leftarrow MUT \cup m$
```

Figure 3: Detecting transitive and indirect mutators

methods in the call graph, the set MTC of methods declared in the target class, and the points-to graph presented in Section 5.1. The output of the algorithm is a set MUT of mutator methods.

In Step 1, the algorithm finds the objects and fields that represent the transitive state of the target class. To do so, the algorithm computes the transitive closure of the `this` references of the target class, i.e, all nodes in the points-to graph reachable from `this`. These nodes, called $TARG$, are the set union of all nodes reachable from the `this` reference in target class methods.

In Step 2, the algorithm finds all transitive mutating methods. These include mutators inside and outside (e.g., `setLocation()`, called on line 30) the target class. The algorithm visits every field assignment instruction in all the target class methods, as well as methods invoked from the target class. For each assignment it checks whether the left-hand side of the assignment is a reference node that may point to one of the objects in the transitive state of the target class. If it can, this means that the instruction assigns to the transitive state of the target class, and the algorithm marks the method as a direct mutator.

In Step 3, the algorithm propagates mutation summaries from direct mutators backwards through the call graph. If method m calls m' and m' is a mutator, then m becomes a mutator too. To do this, the analysis visits, in reverse topological order (post-order), the methods in the call graph and merges the mutation summaries of the callees with the summaries of the callers.

5.3 Detecting Escaping and Entering Objects

The goal of this analysis is to find mutable objects that enter or escape the target class. These objects can be mutated by a client, thus mutating the transitive state of the target class. Therefore, the analysis finds and clones them.

The algorithm detects entering/escaping objects that are mutable *and* assigned/fetched to/from the transitive state of the target class.

Fig. 4 shows the pseudocode of the algorithm for detecting entering or escaping objects. The algorithm takes as input

```
Input: $API \leftarrow$ Set of non-private methods in Target Class
       $MTC \leftarrow$ Methods in Target Class,
       $PTG \leftarrow$ Points-to Graph,
Output: $ESC$ // Set of Escaping Objects
        $ENT$ // Set of Entering Objects

  // Step 1: Find the transitive state of the target class
  $TARG \leftarrow \cup_{m \in MTC}(transitiveClosure(this, PTG))$

  // Step 2: Find the transitive closure of the outside nodes
  $OUT \leftarrow \cup_{m \in API}(transitiveClosure(actuals, PTG)$
                $\cup\ transitiveClosure(returns, PTG)$
                $\cup\ transitiveClosure(statics, PTG))$

  // Step 3: Find the escaping objects
  for each deferred edge $e \in PTG$ do
    if $(e.source \in OUT)$ && $(e.sink \in TARG)$ then
      $ESC \leftarrow ESC \cup e.sink$

  // Step 4: Find the entering objects
  for each deferred edge $e \in PTG$ do
    if $(e.source \in TARG)$ && $(e.sink \in OUT)$ then
      $ENT \leftarrow ENT \cup e.source$
```

Figure 4: Detecting entering and escaping objects

the points-to graph presented in Section 5.1. The output of the algorithm are two sets: ENT containing objects that enter the target class, and ESC containing objects that escape the target class.

In Step 1, the algorithm finds the nodes that form the transitive state of the target class. The transitive state, denoted by the $TARG$ set, is the transitive closure of the *this* reference of every method in the target class.

In Step 2, the algorithm finds the nodes that are outside of the target class, but that interface with it. Since these are the nodes through which client code interacts with the target class, they are also the nodes that objects can enter or escape through. We call these nodes the *boundary nodes*, as they are at the boundary of the target class.

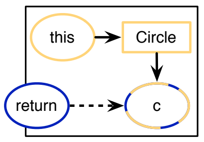
The boundary nodes are:

- actual arguments passed to non-private (API) methods
- references returned from non-private methods
- static reference fields.

The algorithm computes the transitive closure of boundary nodes, and labels the resulting set OUT.

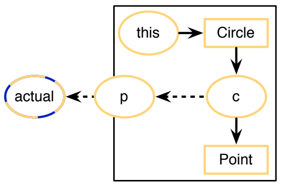
In Step 3, the algorithm finds the escaping objects. Escaping objects are the objects in the transitive state of the target class that can be *seen from methods outside the target class.* To find these objects, the algorithm visits all the deferred edges that start in OUT and end in $TARG$. We are only interested in the edges that end in $TARG$, because we only care about escaping objects in the transitive state of the target class. For such edges, the algorithm adds the sink target node to the ESC set.

Fig. 5(a) shows an example of an escaping object. It shows the points-to graph for the `getLocation` method, with an additional node representing the return statement. We color the transitive state of the target class (which is the transitive closure of `this`) with orange. We then color the outside nodes with blue. In this example, the only boundary node is



```
public Point getLocation() {
  return c;
}
```

(a) Example of an object escaping from `getLocation`



```
public void moveTo(Point p) {
  this.c = p;
}
```

(b) Example of an object entering through `moveTo`

Figure 5: Detecting escaping and entering objects

the return node, and its transitive closure includes `c`. Notice that `c` is colored with both blue and orange. This means `c` escapes because it can be seen from the outside (it is blue), and it is part of the transitive state of the class (it is orange).

In Step 4, the algorithm finds the entering objects. Entering objects are objects that are visible outside the target class methods, and that can be *seen from the target class*. To find these objects, the algorithm visits all the deferred edges that start in $TARG$ and end in OUT. We only visit edges that start in $TARG$, because we only care about the entering objects that are assigned to a field in the transitive state of the target class. For such edges, the algorithm adds the sink outside node to the ENT set.

Fig. 5(b) shows an example of an entering object. It shows the points-to graph for the `moveTo` method. As before, the transitive state of the target class is colored orange, and the transitive closure of the boundary nodes (i.e., the actual argument) are blue. The actual parameter is a part of the transitive state of the target class (it is orange), and it can be seen from the outside (it is blue). Therefore, objects may enter through it.

For pedagogical reasons, we chose simple examples to illustrate escaping and entering objects. In the codes illustrated in Fig. 5(a) and 5(b), it is very easy to spot the entering/escaping objects. However, in many cases they are more difficult to find, especially if objects enter or escape through containers, or escape through parameters. Section 7 shows an example of a state object escaping through an iterator container. The open-source developer overlooked this escaping object, but IMMUTATOR correctly finds it.

6. DISCUSSION

There are cases when the programmer wants only partial immutability. For example, the programmer wants some fields to be excluded from the immutable state of the class (e.g., a `Logger` field), or some fields to be shallowly immutable. Or the programmer does not want to clone the entering/escaping objects (e.g., for performance reasons), but rather to document contracts. These are trivial extensions to IMMUTATOR and require no additional analysis.

Currently, IMMUTATOR handles most of the complexities of an object-oriented language like Java: arrays, aliases, polymorphic methods, and generics. It models arrays as an allocation site with just one field, which represents all the array elements. Although this abstraction does not allow IMMUTATOR to distinguish between array elements, it allows IMMUTATOR to detect objects that enter or escape through arrays. IMMUTATOR disambiguates polymorphic method calls by computing the dynamic type of the receiver object using the results of the points-to analysis described in Section 5.1. IMMUTATOR also preserves the generic types during the rewriting.

Limitations Since IMMUTATOR analyzes bytecode, it correctly handles calls to third-party libraries. However, if the program invokes native code, IMMUTATOR can not analyze it. Also, like any practical refactoring tool, IMMUTATOR does not handle uses of dynamic class loaders or reflection.

Future work We plan to solve the *usage problem*, i.e., updating the client code to use the transformed class in an immutable fashion.

Additionally, we will relax some of the constrains imposed by the current preconditions, to allow IMMUTATOR to transform more classes. For example, we could completely eliminate the requirement that the target class has no superclass/subclass (P1/P2), by allowing IMMUTATOR to transform a whole class inheritance hierarchy at once. Similarly, we could eliminate the requirement that mutators have a void return type (P3). IMMUTATOR could, for example, return a `Pair` object which encapsulates both the old return type, and the newly created object. IMMUTATOR would then have to change the callers of such methods to fetch the appropriate fields.

7. EVALUATION

To evaluate the usefulness of IMMUTATOR we answer the following research questions:

Q1: How applicable is IMMUTATOR?

Q2: Is IMMUTATOR safer than manual transformations?

Q3: Does it make the programmer more productive?

All these questions address the higher level question "Is IMMUTATOR useful?" from different angles. Applicability measures how many classes in real-world programs can be directly transformed, i.e., they meet the preconditions. Correctness ensures that the runtime behavior is not modified by the transformation. Productivity measures whether automation saves programmer time.

7.1 Methodology

We use a combination of three empirical methods, one controlled experiment and two case studies, that complement each other. The experiment allows us to quantify the programmer time and programmer errors, while the case studies give more confidence that the proposed algorithm and experiment findings generalize to real-world situations.

Case Study #1 (CS1) We ran IMMUTATOR on *all* classes in 3 open-source projects, a total of 346 concrete classes. Table 1 shows the projects that we used: Jutil Coal 0.3, jpaul 2.5.1 and Apache Commons Collections 3.2.1.

We do not suggest that every class in a project should be immutable. That is not for a tool to decide. Rather, we evaluate how well the transformation works over all classes

proj.	SLOC	tests	classes	analyzed methods	edits/ class[2]	time/ class	passed preconditons				failed preconditons				
							classes	mutator	enter	escape	classes	P1	P2	P3	P4
jutil	4,605	70	70	3,397	43	2.09 s	32	39	12	19	34	3	16	10	24
jpaul	5,661	42	54	2,471	33	2.16 s	21	25	11	2	26	4	11	9	9
apache	26,323	13,009	222	12,857	50	2.44 s	57	29	16	13	156	24	90	64	63
Total	36,589	13,122	346	18,725	45	2.33 s	110	93	39	34	216	31	117	83	96

Table 1: Results of applying Immutator to 3 open-source projects

without imposing a selection criteria that could limit the generalization of the findings.

Case Study #2 (CS2) We also conducted case studies of how open-source programmers implement immutability. To find existing immutable classes in real-world projects we used two code search engines: krugle (`www.krugle.org`), and Google (`www.google.com/codesearch`). We searched for Java classes whose name contains the word 'Immutable' and classes whose documentation contained the word 'Immutable'. These are classes that are likely to be immutable, and the documentation of these classes confirmed that the developers intended them to be immutable. We also searched for classes implementing an `Immutable` interface, a convention used in some open-source projects. In cases when we found errors in their immutable classes, we contacted the developers to ask for clarification.

Controlled Experiment We asked 6 experienced programmers (with an average of 7 years of Java programming) to manually transform for immutability 8 classes from the JHotDraw 5.3 framework. JHotDraw is an open-source 2D graphics framework for structured drawing editors.

We gave each programmer a 1-hour tutorial on making classes immutable, and then we asked them to transform one or two JHotDraw classes and report the time. We used classes from the `Figure` class hierarchy that made sense to become immutable. Since the `Figure` classes are part of a deep class inheritance hierarchy, we told the participants to treat the target class as if it was the only class in the hierarchy, i.e., to change only the target class. No programmer got a class larger than 400 LOC. We also used Immutator to transform the same classes (we relaxed the first two preconditions), and we compared the results against a golden-standard.

To answer the applicability question, we wrote a statistics tool that applied the transformation to all classes in each project from CS1. For classes that did not pass all preconditions, the tool collected the failed preconditions. Since we ran Immutator in automatic mode, it only applied the transformation to classes that passed all preconditions. In interactive mode, Immutator could have transformed more classes, after the programmer addressed failed preconditions.

To answer the correctness question, we ran extensive test suites before and after all transformations from CS1. We only used projects that had extensive tests to help us confirm that the transformation did not break the systems. We also carefully inspected a few classes that we chose randomly.

To be able to run existing test suites, we wrote a tool that generates a mutable adapter between the immutable classes and the tests. The adapter has the same interface as the original class, but contains a reference to an instance of the immutable class. When a test calls a mutator, the adapter invokes the corresponding factory method of the immutable instance, and assigns the returned object to the reference. Our generated adapters were not adequate for 9% of the case study classes, due to not supporting static instance fields. Additionally, due to exceptions raised by our current implementation, we failed to analyze 20 of the classes in CS1. These were excluded from the reported data.

Furthermore, to compare correctness of manual versus tool-assisted transformation, we carefully analyzed the immutable classes that were produced manually in the second case study (CS2) and in the controlled experiment.

To answer the productivity question, we used Immutator to transform all the classes in Table 1 that met the preconditions. For each class, we report the number of methods that Immutator analyzed, as well as the number of source changes. We further broke this down into the total number of lines that had edits, the number of mutators that had to be converted to factory methods, and the number of entering or escaping objects that had to be cloned. We also report the time Immutator spent analyzing and transforming the code. For the controlled experiment, we asked each programmer to report the time spent to analyze and transform a class.

7.2 Results

To be useful, Immutator must be applicable, correct, and must increase programmer productivity.

7.2.1 Applicability

Table 1 shows that 33.74% of the classes in CS1 meet the preconditions without requiring any modification from the programmer. Out of the classes that failed preconditions, most are due to superclasses containing mutable state (P2), entering/escaping objects (P4), and mutators with non-void return values (P3).

However, keep in mind that a programer would not select all classes, but rather the ones that provide benefit. We hypothesize that such classes are more likely to meet the preconditions. Even in cases when classes do not meet all preconditions, Immutator enables the programmer to identify issues with the push of a button.

7.2.2 Correctness

For each project in CS1, we ran the full test suite before and after the transformations. The transformations did not cause any new failures.

Table 2 shows that even expert programmers make errors when creating immutable classes. The last set of three columns show how many entering or escaping objects the open-source programmers forgot to clone, and how many mutating methods they still left in the immutable class.

We confirmed with the open-source developers that our findings indicate genuine immutability errors in their code, and that developers meant those classes to be deeply immutable. Most agreed that their implementation choice was an incorrect design decision or was made for the sake of performance. Furthermore, the JDigraph developers took our patch and fixed the errors.

Table 3 shows the data for the controlled experiment. Pro-

[2]Does not include the adapter class

project	immutable class	programmer errors		
		mutator	enter	escape
JDigraph	ImmutableBag	-	1	-
	FastNodeDigraph	-	2	-
	HashDigraph	-	2	-
	ArrayGrid2D	-	2	-
	MapGrid2D	-	2	-
WALA	ImmutableByteArray	-	1	-
	ImmutableStack	-	2	3
j.u.c.[3]	ImmutableEntry	-	2	2
Guava	ImmutableEntry	-	-	2
peaberry	ImmutableAttribute	-	-	1
Spring	ImmutableFlow-AttributeMapper	2	2	-

Table 2: Immutability errors in open-source projects

grammers made errors similar with the ones in CS2. However, the density of errors was higher: 6.37 errors/class. The manual inspection of the immutable classes generated by our prototype implementation revealed 4 bugs. None of these were inherent to the algorithm.

7.2.3 *Productivity*

Table 1 shows that Immutator saved the programmer from editing 45 lines of code per target class on average. More important, many of these changes are non-trivial: they require analyzing 57 methods in context to find transitive mutations, entering and escaping objects. In contrast, when using Immutator, the programmer only has to initiate the transformation. On average, Immutator analyzes and transforms a class in 2.33 seconds using a Macbook Pro 4.1 with a 2.4 GHz Core 2 Duo CPU. Compared to the time taken to manually transform a class in the controlled experiment, 27 minutes, this is an improvement of almost 700x.

8. RELATED WORK

Specifying and checking immutability There is a large body of work in the area of *specifying* or *checking* immutability [16, 22, 26].

Pechtchanski and Sarkar [16] present a framework for specifying immutability constraints along three dimensions: lifetime (e.g., the whole lifetime of an object, or only during a method call), reachability (e.g., shallow or deep immutability), and context. Immutator enforces deep immutability for the whole lifetime of an object, on all method contexts.

Tschantz and Ernst [22] present Javari, a type-system extension to Java for specifying *reference immutability.* Reference immutability means that an object can not be mutated through a particular reference, though the object could be mutated through other references. In contrast, *object immutability* specifies that an object can not be mutated through any reference, even if other instances of the same class can be. Zibin et al. [26] build upon the Javari work and present IGJ that allows both reference and object immutability to be specified. *Class immutability* specifies that no instance of an immutable class may be mutated. Reference immutability is more flexible, but weaker than object immutability, which in turn is weaker than class immutability. Immutator enforces class immutability.

[3]java.util.collections

JHotDraw class	SLOC	time [min]	programmer errors		
			mutator	escape	enter
EllipseFigure	104	17	1	-	5
ArrowTip	145	15	-	-	-
ColorEntry	97	16	-	-	1
ImageFigure	154	20	2	-	4
LineConnection	344	53	2	1	2
FigureAttributes	204	24	1	1	1
TextFigure	381	45	7	2	6
PertFigure	311	30	10	-	5
Total	1740	220	23	4	24

Table 3: Results of the controlled experiment

These systems are very useful to document the intended usage and to detect violations of the immutability constraints. But they leave to the programmer the tedious task of removing the mutable access. In contrast, Immutator performs the tedious task of getting rid of mutable access, by converting mutators into factory method, and cloning the state that would otherwise escape.

Supporting program analyses Components of our program analyses have previously been published: detecting side-effect free methods [1, 18–20] and escape analysis [4, 24]. Our analyses detect side effects and escapes *only* on state that is reachable from the target class.

Side-effect analysis [1, 18–20] uses inter-procedural alias analysis and dataflow propagation algorithms to compute the side effects of functions. There are two major differences between these algorithms and Immutator's analysis for detecting mutators. First, the search scope is different. Our algorithm detects side-effects to variables that are part of the transitive state of the target class, whereas previous work determines all side-effects (including side effects to method arguments that do not belong to the transitive state). Consider the method `drawFrame` from `TextFigure` in JHotDraw:

```
public void drawFrame(Graphics g) {
  g.setFont(fFont);
  g.setColor((Color)getAttribute("TextColor"));
  g.drawString(fText, ...);
}
```

The previous algorithms would determine that `drawFrame` is a mutator method, because it has side effects on the graphics device argument, `g`.

However, if Immutator transforms `TextFigure` then `drawFrame` will not mutate the transitive state of the target class, thus eliminating the need to clone the graphics device.

Second, our algorithm distinguishes between (i) methods in the target class that directly or indirectly assign to the fields of the target class and (ii) methods outside the target class (potentially in libraries) that do not assign to target class' fields, but mutate these fields transitively. Immutator converts the former mutators into factory methods, and rewrites the calls to the latter methods into calls dispatched to a copy of `this` (e.g., see the `_this` receiver in Fig. 1, lines 28–29). This enables Immutator to correctly transform code that invokes library methods.

Escape analysis [4, 24] determines if an object escapes the current context. So far, the primary applications of this analysis has been to determine whether (i) an object allocated inside a function does not escape and thus can be allocated on the stack, and (ii) an object is only accessed by a single thread, thus any synchronizations on that object can be removed. There are three major differences between these algorithms and Immutator's escape analysis. First, our algorithm detects escaped objects that belong to the transitive state of the target class. Second, our algorithm is designed to be used in an interactive environment. Thus, it does not perform an expensive whole program analysis, but only analyzes the boundary methods of the target class. Third, in addition to escaping objects, our algorithm also detects entering objects.

Refactoring The earliest refactoring research focused on achieving behavior-preservation through the use of pre- and post-conditions [15] and program dependence graphs [10]. Traditionally, refactoring tools have been used to improve the design of sequential programs. The more recent work has expanded the area with new usages. We have used refactoring [5, 6] to retrofit parallelism into sequential applications via concurrent libraries. In the same spirit, Wloka et al. [25] present a refactoring for replacing global state with thread local state. Schäfer et al. [21] present Relocker, a refactoring tool that lets programmers replace usages of Java built-in locks with more flexible locks. Our transformations for class immutability makes code easier to reason about and enables parallelism by prohibiting changes to shared state.

9. CONCLUSIONS

Programmers use immutability to simplify sequential, parallel, and distributed programming. Although some classes are designed from the beginning to be immutable, other classes are retrofitted with immutability. Transforming mutable to immutable classes is tedious and error-prone.

Our tool, Immutator, automates the analysis and transformations required to make a class immutable. Experiments and case studies of manual transformations, as well as running Immutator on 346 open-source classes, show that Immutator is useful. It is applicable in more than 33% of the studied classes. It is safer than manual transformations which introduced between 2 and 6 errors/class. It can save the programmer significant work (analyzing 57 methods and editing 45 lines) and time (27 minutes) per transformed class.

Acknowledgments

This research was partially funded by Intel and Microsoft through the UPCRC Center at Illinois, and partially supported by NSF grant 0833128. The authors would like to thank Vikram Adve, John Brant, Nick Chen, Ralph Johnson, Darko Marinov, Edgar Pek, Cosmin Radoi, Manu Sridharan, and anonymous reviewers for providing helpful feedback. Danny thanks Monika Dig, his greatest supporter.

10. REFERENCES

[1] J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *POPL*, pages 29–41, 1979.

[2] D. Bäumer, D. Riehle, W. Siberski, C. Lilienthal, D. Megert, K.-H. Sylla, and H. Züllighoven. Want value objects in java? Technical report, 1998.

[3] J. Bloch. *Effective Java: Programming Language Guide.* Addison-Wesley, 2001.

[4] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for java. In *OOPSLA*, pages 1–19, 1999.

[5] D. Dig, J. Marrero, and M. D. Ernst. Refactoring sequential java code for concurrency via concurrent libraries. In *ICSE*, pages 397–407, 2009.

[6] D. Dig, M. Tarce, C. Radoi, M. Minea, and R. Johnson. Relooper: refactoring for loop parallelism in Java. In *OOPSLA*, pages 793–794, 2009.

[7] J. J. Dolado, M. Harman, M. C. Otero, and L. Hu. An empirical investigation of the influence of a type of side effects on program comprehension. *IEEE TSE*, 29(7):665–670, 2003.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[9] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice.* Addison-Wesley, 2006.

[10] W. G. Griswold and D. Notkin. Automated assistance for program restructuring. *ACM TOSEM*, 2(3):228–269, 1993.

[11] R. Hickey. The clojure programming language. In *DLS*, 2008.

[12] Java SE 6 API Specification. `http://java.sun.com/javase/6/docs/api`.

[13] D. Lea. *Concurrent Programming In Java.* Addison-Wesley, second edition, 2000.

[14] D. Marinov and R. O'Callahan. Object equality profiling. In *OOPSLA*, pages 313–325, 2003.

[15] W. F. Opdyke and R. E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *SOOPPA*, pages 145–160, 1990.

[16] I. Pechtchanski and V. Sarkar. Immutability specification and its applications. In *Java Grande*, pages 202–211, 2002.

[17] D. Riehle. Value object. In *PLOP*, 2006.

[18] A. Rountev. Precise identification of side-effect-free methods in Java. In *ICSM*, pages 82–91, 2004.

[19] B. G. Ryder, W. Landi, P. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM TOPLAS*, 23(2):105–186, 2001.

[20] A. Salcianu and M. C. Rinard. Purity and side effect analysis for java programs. In *VMCAI*, pages 199–215, 2005.

[21] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Refactoring java programs for flexible locking. *To appear in ICSE*, 2011.

[22] M. S. Tschantz and M. D. Ernst. Javari: Adding reference immutability to Java. In *OOPSLA*, pages 211–230, 2005.

[23] T.J. Watson Libraries for Analysis (WALA). `http://wala.sourceforge.net/wiki/index.php`.

[24] J. Whaley and M. C. Rinard. Compositional pointer and escape analysis for java programs. In *OOPSLA*, pages 187–206, 1999.

[25] J. Wloka, M. Sridharan, and F. Tip. Refactoring for Reentrancy. In *FSE*, pages 173–182, 2009.

[26] Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kiezun, and M. D. Ernst. Object and Reference Immutability using Java Generics. In *FSE*, pages 75–84, 2007.

Hydra : Automatic Algorithm Exploration from Linear Algebra Equations

Alexandre X. Duchâteau
Department of Computer Science
University of Illinois
axdn@illinois.edu

David Padua
Department of Computer Science
University of Illinois
padua@illinois.edu

Denis Barthou
Labri
Université de Bordeaux
denis.barthou@inria.fr

Abstract

Hydra accepts an equation written in terms of operations on matrices and automatically produces highly efficient code to solve these equations. Processing of the equation starts by tiling the matrices. This transforms the equation into either a single new equation containing terms involving tiles or into multiple equations some of which can be solved in parallel with each other.

Hydra continues transforming the equations using tiling and seeking terms that Hydra knows how to compute or equations it knows how to solve. The end result is that by transforming the equations Hydra can produce multiple solvers with different locality behavior and/or different parallel execution profiles. Next, Hydra applies empirical search over this space of possible solvers to identify the most efficient version. In this way, Hydra enables the automatic production of efficient solvers requiring very little or no coding at all and delivering performance approximating that of the highly tuned library routines such as Intel's MKL.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers, Optimization

General Terms Performance, Algorithms, Parallelism

Keywords Automatic Derivation, Linear Algebra

1. Introduction

Years of research have led to very powerful algorithms to solve linear algebra on all classes of machines. The algorithms and implementation strategies used for sequential systems differ from those used for parallel systems. For this reason, implementations that were developed for sequential machines may not be the ideal place to start when looking towards parallel solutions to a problem since some of the selections made to obtain efficient sequential codes may have to be changed in order to obtain as good parallel version.

Loss of information. Typically, a program is developed starting with an examination of the problem. Then, an algorithm to solve it is devised and refined with certain objectives in mind. We can assume that the goal was to minimize complexity while ensuring good numerical behavior. And while minimizing the complexity of an algorithm usually translates into less computation and thus faster sequential programs, this is not always the most important consideration for modern machines where locality and parallelism are of crucial importance. For parallel systems in particular, one should focus on minimizing execution time, reducing power consumption or a combination of these. Thus finding and exposing the independence of the computation becomes an important factor which is sometimes more important than minimizing the quantity of computation.

The next step is the implementation of the algorithm. Given that compilers often fail to generate optimal programs, programmers that aim at maximum performance will often apply transformations on their code to help the compiler in its optimization process to the point where it can become difficult to recognize what the code is doing. For example, Figure 1 presents a simple triply nested loop that performs a matrix multiplication. Figure 2 is the same code, after application of a set of source optimization: tiling, scalar promotion, loop unrolling and loop interchange. The code now has 3 additional loops with larger strides, the innermost loop has two statements operating on scalars and single dimension arrays instead of three double dimension arrays. While these transformations may optimize sequential performance on a specific machine, they may hide parallelism to a parallelizing or vectorizing compiler or even from the programmer.

We thus make the argument that, when possible, parallel programs should be written starting at the problem specification rather than with a sequential implementation or algorithm.

Tuned parallel code generation. In this paper, we describe a system that automatically derive parallel codes from

CGO'13 February 2013, Shenshen, China.
Copyright © 2013 ACM [to be supplied]. . . $10.00

```
for(int i = 0 ; i < N ; i++)
 for(int j = 0 ; j < N ; j++)
  for(int k = 0 ; k < N ; k++)
   c[i][j] += a[i][k] * b[k][j];
```

Figure 1. Matrix multiplication baseline

```
for(int ii = 0 ; ii < N ; ii+=B){
 for(int jj = 0 ; jj < N ; jj+=B){
  for(int kk = 0 ; kk < N ; kk+=B){
   for(int i = ii ; i < ii + B ; i++){
    for(int k = kk ; k < kk + B ; k++){
     c_i = c[i];
     a_ik = a[i][k];
     b_k = b[k];
     for(int j = jj ; j < jj+B ; j+=2){
      c_i[j] += a_ik * b_k[j];
      c_i[j+1] += a_ik * b_k[j+1];
     }
    }
   }
  }
 }
}
```

Figure 2. Optimized Matrix Multiplication

high level descriptions of linear equations. This description includes the expressions in the mathematical equation, and information on the operands. Working from this equation, the system defines parameters to characterize a class of parallel solutions using a divide and conquer approach, then explores this space of solutions to determine the best. Our system's output is a collection of equations connected by a dependence graph that describes a solution to the original equation.

Outline. The rest of this paper is organized as follows. Section 2 describes the system. and Section 3 elaborates on the generator component that is at the core of our contribution. Section 4 presents results obtained on some matrix problems. Section 5 discusses related work and finally Section 6 describes our conclusions.

2. Overview of Hydra

Hydra is a code generator that starts from a mathematical description of matrix linear equations to solve, and generates parallel codes for multi-core architectures. The steps involved in this generation are described in Figure 3. The input is the description of the equation to solve and a collection of routines with their associated signatures. The signature identifies the form of the equation to be solved and the nature of the terms. For example, the signature

$$LT \cdot UNK = MT$$

represents an equation of the form $L \cdot X = M$ with X an unknown (denoted UNK) and L a lower triangular matrix (denoted LT). Relying on algebraic properties and on the shape of the matrices, Hydra applies a divide-and-conquer strategy to automatically generate different recurrent formulations of the initial problem. This crucial step is presented in the following section. As a result, multiple formulations are generated, they differ in terms of parallelism, computation grain and data locality.

To identify the fastest version produced by the generator, each one is executed on the actual target multi-core machine. This requires the target system to be available and some input data sets for the problem to solve. An alternative to this auto-tuning approach would be to rely on performance prediction [10] and keep for testing only the versions with the best performance prediction, or even remove the need for any execution. This possibility is not addressed in this paper. For input data sets, we assume that the user provides sample data sets or data generators. For dense linear algebra, the determining factor of performance is the size of the input data. It is thus important that the data provided matches the size or size range of the data with which the program would be used afterwards. The rest of this section describes the different components and their roles in more details.

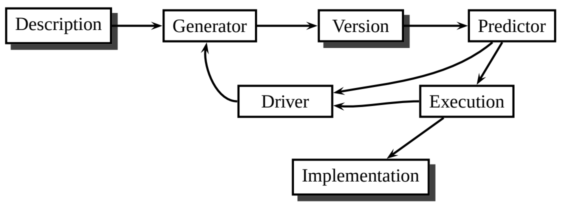


Figure 3. System graph

Our mathematical **description language** can represent matrix equations to solve. The only required information beyond the actual equation is the shapes of the matrices involved (e.g. triangular, symmetric matrices).

For a class of problems which can be said are *natively supported*, no additional information is required. In other words, it is not necessary to provide an algorithm to solve these problem. Figure 4 presents a full example with the description of a discrete triangular Sylvester equation (DTSY).

The input consists in:

- An equation on matrices. Basic matrix operations can be handled. So far, only addition, substraction, multiplication are supported in our current implementation. The matrices used in the equation are each described by their shapes, using the keywords `Square`, `Upper Triangular`, `Lower Triangular` and by their nature with the keyword `Unknown`. Matrices are assumed to be known by default, i.e. part of the input.
- The optional description of a library function, a kernel, that can be executed to solve this problem. Hydra can use this function for the base case of the recursion.
- An optional list of equations corresponding to other problems with the kernels to solve them; This list can be used by Hydra to solve subproblems of the initial problem.

```
%% Operands
X: Unknown Square Matrix
A: Upper Triangular Square Matrix
B: Lower Triangular Square Matrix
C: Square Matrix

%% Equation
A·X·B−X = C

%% Parameters
@name sylvester
@codelet sylsolv
@operands A B C
```

Figure 4. Discrete Triangular Sylvester Equation description

The main component is the **Generator**. The generator has a set of native transformation rules that it applies on equations. In particular, it transforms a single equation into a set of equations, in order to generate divide-and-conquer solutions to the problem. The results are task graphs that represent different possible implementations. The generator is further described in section 3.

In the future, a **Predictor** filter could be inserted between the generator and the empirical evaluation step. This component could increase the number of valid algorithms and implementation that can be evaluated in a fixed amount of time. Performance prediction is a difficult problem, but analysis of the generated task dependence graph can be performed to build bounds on achievable performance. The graphs width and the length of the critical path are examples of metrics that can be used to such end.

The **Code Generator / Execution** component performs empirical evaluation of the different versions. It first converts the task graph into code, using the StarPU[1] runtime scheduler API, compiles it and runs it on sample data to measure appropriate metrics (e.g. execution time, memory usage, power consumption, ...). Performance data are sent forth to the driver component. Sample data or data generators must be provided. Although in the current version, the output is selected based on a single data set, it is easy to extend the system so that it could generate input dependent libraries, that select a version as a function of characteristics of the input data, which in the case of dense linear algebra would be the size of the matrices and vectors.

If a codelet implementation is missing to translate a graph, a message is generated, presenting both the equation and its signature. Hydra can be used as an interactive development tool.

Finally, the **Driver** manages the process. In the simplest case, which is the one currently implemented, it restricts the search to a subset of all possible tilings of the equation, keeping track of the fastest version, applying successive recursive decompositions and stopping the generator once all cases have been tested. But it can also implement machine learning techniques to reduce the search space and improve filtering poor versions out.

3. Generator

The generator is Hydra's main component. It accepts the high level description of an equation, breaks it into multiple equations operating on smaller matrices, and possibly repeats this process by further breaking the generated equations. Along the way, the generators builds a task dependence graph specifying the necessary order in which these equations must be solved.

The generator operates on one equation at a time and on the dependence graph. At the beginning, the dependence graph has a single node representing the initial equation. At each step, the generator expands an equation by tiling its operands and then adjusting the dependence graph to incorporate the new equations and remove the one that was expanded.

Example 1: Consider the equation $M = L \cdot X$ with M a known matrix, L a known lower triangular matrix and X an unknown matrix. A way to expand this equation, is to convert each operand into a tiled array with two tiles along each dimension. Therefore, from

$$\begin{pmatrix} M_{00} & M_{01} \\ M_{10} & M_{11} \end{pmatrix} = \begin{pmatrix} L_{00} & 0 \\ L_{10} & L_{11} \end{pmatrix} \cdot \begin{pmatrix} X_{00} & X_{01} \\ X_{10} & X_{11} \end{pmatrix}$$

we obtain the following four equations.

$$M_{00} = L_{00} \cdot X_{00} \quad (1)$$
$$M_{01} = L_{00} \cdot X_{01} \quad (2)$$
$$M_{10} = L_{10} \cdot X_{00} + L_{11} \cdot X_{10} \quad (3)$$
$$M_{11} = L_{10} \cdot X_{01} + L_{11} \cdot X_{11} \quad (4)$$

The dependence graph associated with the initial $M = L \cdot X$ equation is a single node. For the equations resulting from the expansion the dependence graph has four nodes, one per equation. We label the nodes with the numbers given to each equation above. The graph has two arcs. One from node (1) to node (3) because X_{00} must be computed by solving equation (1) before equation (3) can be solved for X_{10} and another from node (2) to node (4) because X_{01} is needed to solve equation (4).

Example 2: Figure 5 illustrates the behavior of the generator for equation $L \cdot X \cdot U - X = C$. The first column of the the flow diagram, illustrates the initial steps followed by the generator. At the beginning, the original equation is available and the associated dependence graph is empty. Next, in a process we call *derivation*, the equation is expanded into four new equations while the dependence graph stays unchanged (i.e. empty). Finally, a process that we call *identification* generates a dependence graph linking the newly generated equations.

The last column of the figure, illustrates a step further down the road. An equation, the one labeled (4), has been selected for expansion. It is expanded into four new equations, and the identification step generates a dependence graph for

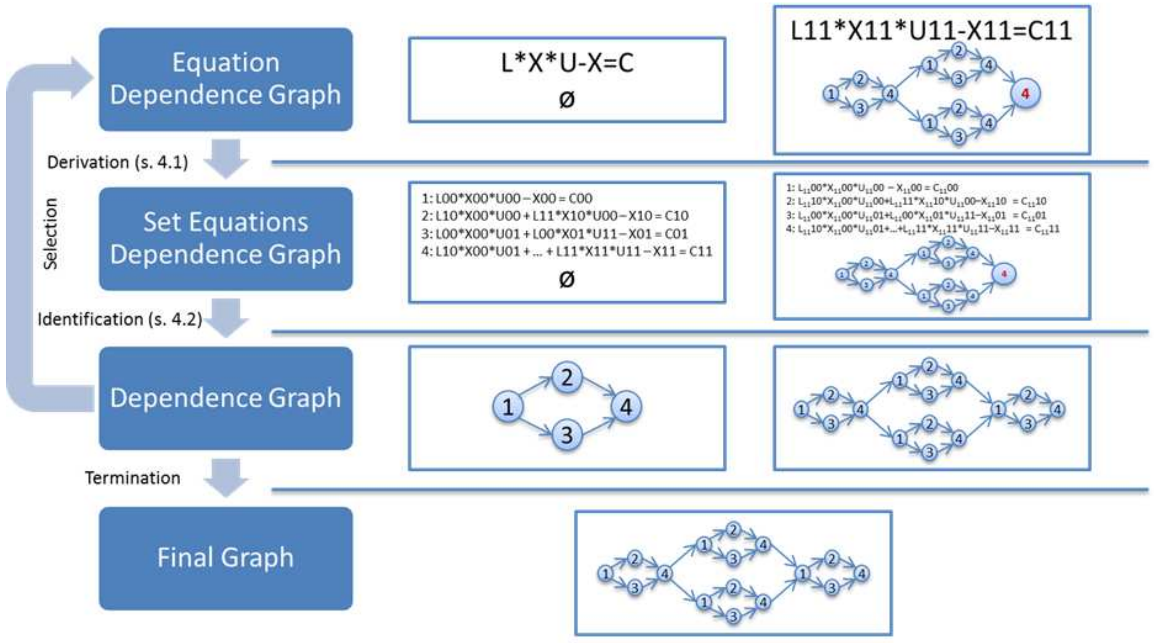

Equation Dependence Graph
Derivation (s. 4.1)
Set Equations Dependence Graph
Selection
Identification (s. 4.2)
Dependence Graph
Termination
Final Graph
L*X*U-X=C
Ø
L11*X11*U11-X11=C11
1: L00*X00*U00 − X00 = C00
2: L10*X00*U00 + L11*X10*U00 − X10 = C10
3: L00*X00*U01 + L00*X01*U11 − X01 = C01
4: L10*X00*U01 + ... + L11*X11*U11 − X11 = C11
Ø
1: L11 00*X11 00*U11 00 − X11 00 = C11 00
2: L11 10*X11 00*U11 00+L11 11*X11 10*U11 00−X11 10 = C11 10
3: L11 00*X11 00*U11 01+L11 00*X11 01*U11 11−X11 01 = C11 01
4: L11 10*X11 00*U11 01+...+L11 11*X11 11*U11 11−X11 11 = C11 11

Figure 5. Generator overview with example

those new equations as well as integrates it into the larger dependence graph that represents the problem.

Termination of the generator can be decided by characteristics of the dependence graph, or by a recursion depth when all the operators have been tiled to the same granularity. In Example 2 we assume that termination happens after all equations have been expanded twice (recursion depth of two).

3.1 Equation Expansion or Derivation

Equation expansion is the process by which different algorithms are generated by the system. Different ways of tiling and different depths of recursion produce different algorithms. The first step of expansion, described in section 3.1.1, is to make sure that tiling is done in the right way. There would typically be numerous ways of tiling the operands and this defines the exploration space from where the final version of the solver will be selected. Section 3.1.2 describes how a solution is generated for one point in the exploration space.

3.1.1 Validity of tiling

The first step in the process of deriving an equation is to partition the operands of this equation into tiles. When considering matrix operations, one cannot arbitrarily partition the operands. Since we partition the operands in order to perform symbolic execution of the operation, a few basic rules must hold. e.g. when multiplying two matrices A and B, the number of columns of A must be equal to the number of rows of B. A blocking that does not conserve those rules is considered invalid and shouldn't be considered.

Instead of generating all possible tilings and then checking their validity, we ensure that only useful tilings are generated. To do so, we use the matrix operation properties to build a set of relations between the operand dimensions and only generate tilings satisfying those relations. Block and matrix sizes are at this point completely symbolic.

The shapes of the operands may also guide how blocking is applied to generate new equations with recognizable shapes. For example, we may want to block a triangular matrix so that it contains triangular matrices on the diagonal.

Figures **??** to **??** illustrate the first step of the process. Where we propagate the operands' dimensions. For this example, we look at equation $A \cdot X \cdot B - X = C$.

First (figure **??**) the system creates the operation tree assigning a tuple (x,y) to each operand where x and y are the number of blocks per column and row respectively (see figure 7). Real operands correspond to the leaves and the root of the tree, they are named in the original equation. Virtual operands are inner nodes of the tree and have no name in the original equation.

We then look at the virtual operands to assign them a tuple of dimensions. Knowing the dimensions of two operands of a matrix operation, we can deduce the dimensions of the result. i.e. the product of a $m \times n$-matrix by a $n \times l$-matrix will produce a $m \times l$-matrix. For example (figure **??**) we can assign the tuple (x_A, y_X) to the node that is the result of the product of A by X.

Now that all operands (real and virtual) have been assigned a set of dimensions, the system will examine each operational node in the tree and create the set of equations (1). For example, when looking at the node that represents the

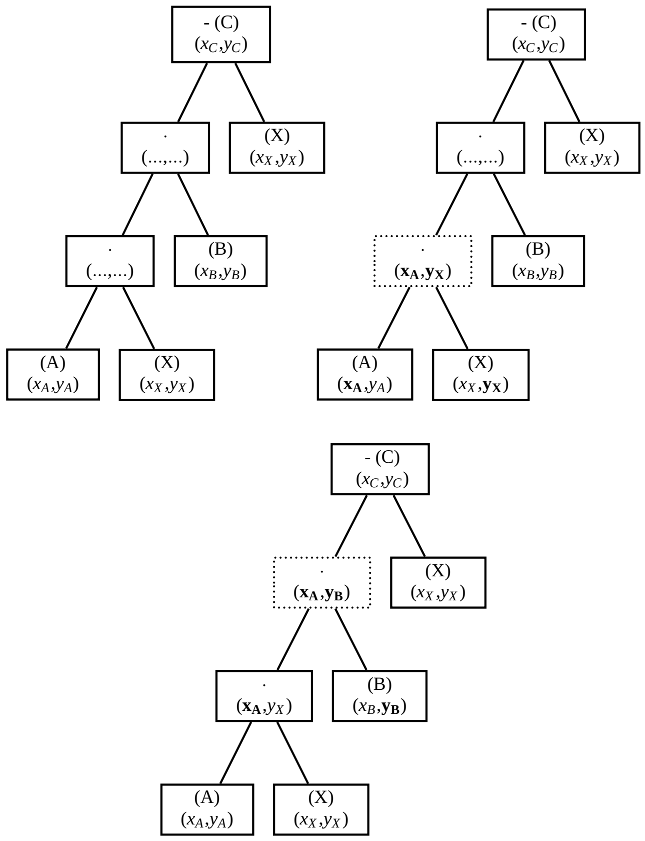


Figure 6. Operation Tree

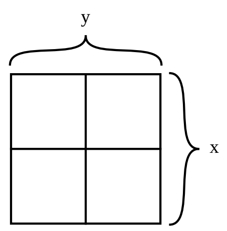


Figure 7. An operand's tiling dimensions

product of A.X by B. The number of columns of the left hand operand has to be equal to the number of lines of the right operand. That leads to equation $y_A = x_X$.

From equations $y_A = x_X$, $y_X = x_B$, $x_A = x_X$, $y_B = y_X$, $x_C = x_X$, and $y_C = y_X$ we get two sets of constraints on the number of tiles per dimension.

$$\begin{aligned} \{x_A = y_A \quad &= x_X = x_C\} \\ \{x_B = y_B \quad &= y_X = y_C\} \end{aligned} \tag{1}$$

Exploring all possible tilings of the problem is now reduced to finding two values and assigning them to their respective sets of variables.

3.1.2 Tiling

To derive the equation the system first partitions (tiles) its operands. It relies on the properties of matrix operations. Once the operands have been partitioned, we use symbolic execution of tiled operations to generate new sets of equations.

An important aspect of our system is how the operands' shapes are used to identify the unnecessary computation. 0-blocks (a matrix block that only contains 0 values) are absorbing elements for the matrix multiplication (i.e. $0 \cdot X = 0$) and identity elements for matrix addition (i.e. $0 + X = X$), thus computation involving 0-blocks can be simplified.

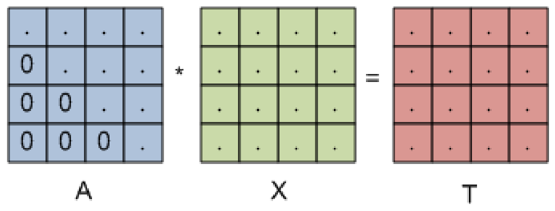


Figure 8. Original equation with operand shapes

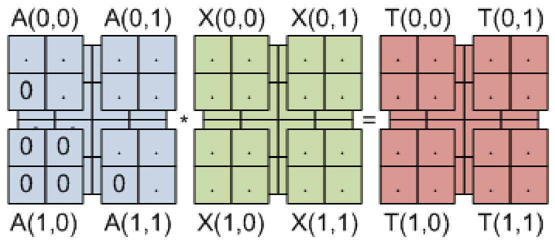


Figure 9. Tiling operands

```
T(0,0) = A(0,0)*X(0,0) + A(0,1)*X(1,0)
T(0,1) = A(0,0)*X(0,1) + A(0,1)*X(1,1)
T(1,0) = A(1,0)*X(0,0) + A(1,1)*X(1,0)
T(1,1) = A(1,0)*X(0,1) + A(1,1)*X(1,1)
```

Figure 10. Expansion generates new equations

```
T(0,0) = A(0,0)*X(0,0) + A(0,1)*X(1,0)
T(0,1) = A(0,0)*X(0,1) + A(0,1)*X(1,1)
T(1,0) =                 A(1,1)*X(1,0)
T(1,1) =                 A(1,1)*X(1,1)
```

Figure 11. Removing 0 operands

Figures 8 to 11 illustrates an example of such block partitioning. In 11 the bottom two equations have been simplified since it is known from the shape information that block A(1,0) is all zeros.

3.2 Identification and Dependence Graph Computation

Once a collection of new equations is created by tiling the operands (section 3.1), the generator proceeds to identify the tasks described by those equations, and generate the dependence graph between those tasks. In this section, we describe how to achieve this.

3.2.1 Building Dependences

The crucial step in building the dependence graph is determining what is the input and output of each one of the equations. The input to Hydra identifies matrices whose values that are known at the outset. As discussed in Section 2, these are the matrices not annotated with the `Unknown` keyword in the input to the system. These matrices are placed by Hydra in the *input set* to the equations where they appear. Also all tiles of these known matrices and vectors are assumed to be input to the equations where they appear. All other operands are initially placed in the *output set* of the equations.

Example 3: Let us consider again the equation $M = L \cdot X$ from Example 1.

Because matrices M and L are known, we have that matrices L_{ij} and M_{ij} for $i, j \in \{0,1\}$ are also known, and because matrix X is unknown we have that matrices X_{ij} for $i, j \in \{0,1\}$ are unknown.

And here are a couple of input/output set examples :

Equation	Input set	Output set
$M_{00} = L_{00} \cdot X_{00}$	$\{M_{00}, L_{00}\}$	$\{X_{00}\}$
$M_{10} = L_{10} \cdot X_{00} + L_{11} \cdot X_{10}$	$\{M_{10}, L_{10}, L_{11}\}$	$\{X_{00}, X_{10}\}$

Algorithm 1 describes the process used to build the dependence graph for a newly created set of equations. The algorithm first (line 2) selects an *unidentified equation* from E. Then (line 3) it removes e from E converting in this way e into an *identified equation*. An equation e is selected if the system contains a kernel capable of solving the equation. This means that there is a kernel that accepts as input all matrices in the input set of the equation, solves the equation, and returns values for each of the matrices in the output set of the equation.

Algorithm 1 Building the dependence tree

```
1: while $E \neq \emptyset$ do
2:     Select $e$ in $E$ % See section 3.2.2
3:     $E \leftarrow E \setminus \{e\}$
4:     for all $o \in e.output$ do
5:         for all $d \in E$ do
6:             if $o \in d.output$ then
7:                 $T \leftarrow T \cup \{(e; d)\}$
8:                 $d.output \leftarrow d.output \setminus \{o\}$
9:                 $d.input \leftarrow d.input \cup \{o\}$
```

All the matrices in the output set of e can, after identification, be considered as inputs to any of the equations in E. To reflect that fact, the loop on line 4 finds every equation in E that has e's output matrices in their own output set (until this point, those variables were unknown to every equation using them) and transfers it to their input set. In addition, a dependence edge is added between equation e and every equation that uses matrices from its output set. Section 3.2.2 discusses the details of this process of selection.

Once E is empty, every equation has been identified and added to the dependence graph. The process is then over.

Example 4: Let us consider one more time the equation

$$M = L \cdot X,$$

where M is a known matrix, L is a known lower triangular matrix and X is an unknown matrix. Each matrix is partitioned once along each dimension. The following equations are generated and added to set E with their associated input and output sets.

	Equation	Input set	Output set
(1)	$M_{00} = L_{00} \cdot X_{00}$	$\{M_{00}, L_{00}\}$	$\{X_{00}\}$
(2)	$M_{01} = L_{00} \cdot X_{01}$	$\{M_{01}, L_{00}\}$	$\{X_{01}\}$
(3)	$M_{10} = L_{10} \cdot X_{00} + L_{11} \cdot X_{10}$	$\{M_{10}, L_{10}, L_{11}\}$	$\{X_{00}, X_{10}\}$
(4)	$M_{11} = L_{10} \cdot X_{01} + L_{11} \cdot X_{11}$	$\{M_{11}, (4)_{10}, L_{11}\}$	$\{X_{01}, X_{11}\}$

Equation (1) can clearly be solved using a triangular solver, but equation (3) cannot be solved until (1) has been solved because the value of $X_{0,0}$ is needed for its solution. Also, equation (2) can be solved, but equation (4) must wait for the solution to equation (2). When equation (1) is selected, the matrix X_{00} becomes a known variable. Since equation (3) has X_{00} in its output set, a dependence edge is created between (1) and (3) $T = T \cup \{((1) \rightarrow (3))\}$. And the input and output sets are updated. In addition, (1) is removed from E.

Other arcs in the dependence graph (those corresponding to incoming and outgoing arcs from the equation before expansion) can be trivially added since all that is needed is to connect elements in the output set in one equation to elements in the input set of other equations. The reason is that, except for newly expanded equations (which as mentioned above may have an incorrect number of matrices in the output set), the input and output set of all equations are properly defined.

	Equation	Input set	Output set
(2)	$M_{01} = L_{00} \cdot X_{01}$	$\{M_{01}, L_{00}\}$	$\{X_{01}\}$
(3)	$M_{10} = L_{10} \cdot X_{00} + L_{11} \cdot X_{10}$	$\{M_{10}, L_{10}, L_{11}, X_{00}\}$	$\{X_{10}\}$
(4)	$M_{11} = L_{10} \cdot X_{01} + L_{11} \cdot X_{11}$	$\{M_{11}, L_{10}, L_{11}\}$	$\{X_{01}, X_{11}\}$

3.2.2 Selection

The selection of an equation to add to the dependence tree is performed following algorithm 2.

This process consists in identifying which equations are solvable and thus can be added to the dependence graph.

First (line 1), we look for an equation that matches the original problem. This is done by direct comparison of the equation's signature to the signature of the original problem. Signatures are explained in detail in section 3.2.3.

If no such equation is found (line 4), we examine the equations looking for one that can be massaged into a match of the original problem. This is achieved by simplification of the equations signature, if an equation's signature can be made to match the main equation's signature then they

Algorithm 2 Equation Selection

```
Require: Set $E$ of equations
 1: for all $e \in E$ do
 2:     if $e.signature = main.signature$ then
 3:         return e
 4: for all $e \in E$ do
 5:     if $|e.output| = |main.output|$ then
 6:         if simplification(e.signature,main.signature) then
 7:             $e \leftarrow expand(e)$
 8:             return e
 9: for all $e \in E$ do
10:     if $|e.output| = 1$ and $solvable(e)$ then
11:         return e
12: print Error
```

are equivalent if some pre-processing computation is performed. For example, the equation $L \cdot X + B = M$ with L lower triangular and X unknown does not match the signature $LT \cdot UNK = MT$, but if we expand that into two equations: (1) $R = M - B$ and (2) $L \cdot X = M$ where (1) is a pre-processing step, we would get a match.

On line 7, the equation is replaced by a subgraph that contains the pre-processing steps and the new equation that matches the original. Simplification rules are explained in section 3.2.3 and the expansion step is explained in section 3.2.4.

Finally, if no equation is found that matches the original or can be made to match the original, we look for simple equations that produce a single output and are directly solvable (e.g. a matrix multiplication of the form Unknown = Known * Known)

Example 5: In the first selection step in Example 4, both $M_{00} = L_{00} \cdot X_{00}$ and $M_{01} = L_{00} \cdot X_{01}$ are possible candidates. Both equations match the original problem : i.e. the product of a lower triangular matrix by an unknown equaled to a known matrix.

3.2.3 Signatures and Simplification

We define an equation's signature as the combination of the operations it contains and the shapes of its operands.

Let the following abbreviations stand:

- LT : Known Lower Triangular Matrix
- UT : Known Upper Triangular Matrix
- MT : Known Matrix of Unspecified shape
- UNK : Unknown matrix
- UNK_LT : Unknown Lower Triangular matrix
- UNK_UT : Unknown Upper Triangular matrix

For example, for LU decomposition ($L \cdot U = A$) the signature is :

$$UNK\_LT \cdot UNK\_UT = MT$$

The signature is used to identify the nodes. In particular to identify when the new generated equations are instances of the original problem on smaller data sets.

For the purpose of identification, simplification rules are defined on an equation's signature.

A few examples are :

- $MT + MT \Rightarrow MT$
- $MT \cdot MT \Rightarrow MT$
- $\ldots MT = MT \Rightarrow \ldots = MT - MT$

The rules presented have variants for each combination of shapes for the matrices. e.g.

- $LT \cdot LT \Rightarrow MT$
- $LT + LT \Rightarrow LT$

Example: Consider the equation $L \cdot X + X \cdot U = M$ with M a known matrix, L a known lower triangular matrix, U a known upper triangular matrix and X and unknown matrix. Each operand is blocked twice in each dimension.

The signature of the original problem is $LT \cdot UNK + UNK \cdot UT = MT$

Consider the derivated equation

$$L_{00} \cdot X_{01} + X_{00} \cdot U_{01} + X_{01} \cdot U_{11} = M_{01}$$

at a stage where X_{01} is the only output. The equation signature is thus

$$LT \cdot UNK + MT \cdot MT + UNK \cdot UT = MT$$

and does not match the signature of the original problem.

$$\begin{aligned} & LT \cdot UNK + MT \cdot MT + UNK \cdot UT = MT \\ \Leftrightarrow \quad & LT \cdot UNK + MT + UNK \cdot UT = MT \\ \Leftrightarrow \quad & LT \cdot UNK + UNK \cdot UT = MT - MT \\ \Leftrightarrow \quad & LT \cdot UNK + UNK \cdot UT = MT \end{aligned}$$

After simplification, the signature matches the original problem.

3.2.4 Expansion

The expansion function allows to translate the simplifications applied on an equation's signature into tasks. Every simplification step applied on the signature is applied on the actual operands of the equation, generating a graph of simple solvable equations that lead to the new equation matching the original problem.

Example: Consider the equation and the simplification process described in the example in section 3.2.3.

	$LT \cdot UNK + MT \cdot MT$ $+ UNK \cdot UT = MT$	$L_{00} \cdot X_{01} + X_{00} \cdot U_{01}$ $+ X_{01} \cdot U_{11} = M_{01}$
$\Leftrightarrow$	$LT \cdot UNK + MT + UNK \cdot UT = MT$	$T_0 = X_{00} \cdot U_{01}$ $L_{00} \cdot X_{01} + T_0 + X_{01} \cdot U_{11} = M_{01}$
$\Leftrightarrow$	$LT \cdot UNK + UNK \cdot UT = MT - MT$	
$\Leftrightarrow$	$LT \cdot UNK + UNK \cdot UT = MT$	$T_0 = X_{00} \cdot U_{01}$ $T_1 = M_{01} - T_0$ $L_{00} \cdot X_{01} + X_{01} \cdot U_{11} = T_1$

For each simplification step that reduces the number of operands, the corresponding operation is added to the nodes equation set.

The set of operations corresponding to BLAS functions are built-in Hydra, that is able to match them automatically. In the previous example, $T_0 = X_{00} \cdot U_{01}$ thus matches a matrix multiply kernel.

4. Results

Starting from different linear problems on matrices and sequential kernels, Hydra generates automatically parallel codes solving these problems and resorting to these kernels. The task graph generated by Hydra is scheduled dynamically with StarPU runtime system [1].

In the following experiments, all sequential kernels used are from Intel MKL library [6]. In order to evaluate the capabilities of Hydra in terms of parallel code generation, we compare performance between the sequential MKL version with the best parallel version generated by Hydra for any given problem size. Besides, we compare the performance with the parallel MKL version. All our experiments were conducted on a 32-core (64 threads) platform composed of four 8-core Intel L7555 CPUs with 64GB of memory.

Figure 12 presents the results for the matrix multiplication. Here the decomposition obtained through Hydra corresponds to a block matrix multiplication. Note that these block matrix multiplications are not performed in-place, Hydra generates copies for each tile, improving here locality. We observe that the best parallel code generated by Hydra consistently outperforms the MKL parallel version of the matrix multiplication, for matrix sizes over 4000.

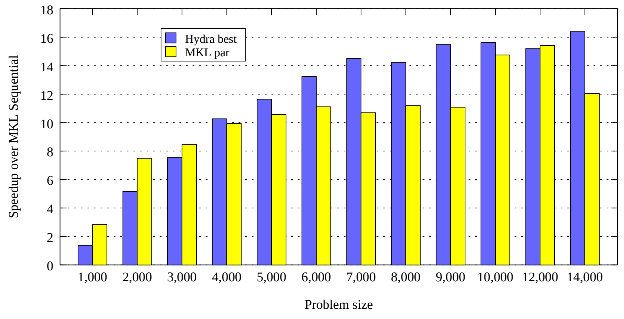


Figure 12. Matrix Matrix Multiplication: X = A*B

For the triangular solver, Figure 13 shows performance speed-ups compared to the sequential MKL and comparison with the parallel version. Performance of Hydra remains within roughly 10% of the parallel MKL performance. A more detailed analysis in Figure 14 shows the influence of the number of blocks on performance: tiling matrices in 10 by 10 blocks brings the best speed-up or large matrices.

Table 1 shows that for a blocking factor of 10, there are 1000 tasks created, the maximum number of tasks executable at the same time during the course of the execution is 90 (this is the width of the graph) and 255 copies of blocks are performed. The high number of copies compared to the number of computational tasks may account for some performance loss.

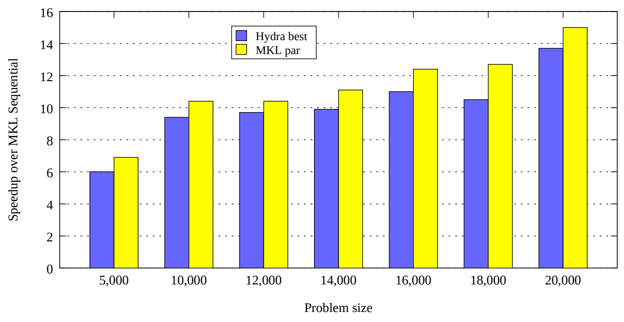


Figure 13. Triangular Solver : L*X = C

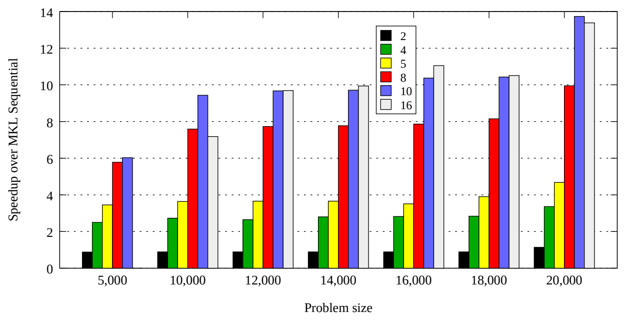


Figure 14. Exploring blocking factors for the triangular solver : L*X = C.

Blocking factor	Tasks	Max Parallelism	Copies
2	8	2	11
4	64	12	42
5	125	20	65
8	512	56	164
10	1000	90	255
16	4096	240	648

Table 1. Triangular Solver: Characteristics of the different versions generated by Hydra, according to the blocking factor.

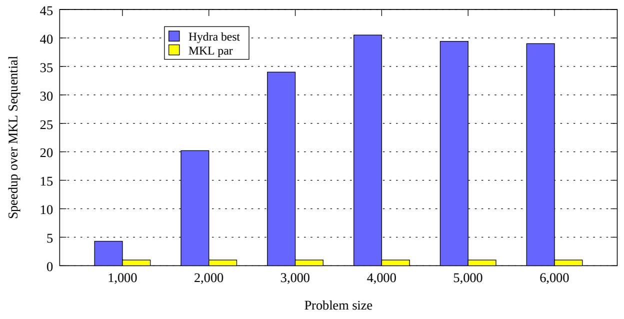


Figure 15. Triangular Sylvester: $AXB - X = C$

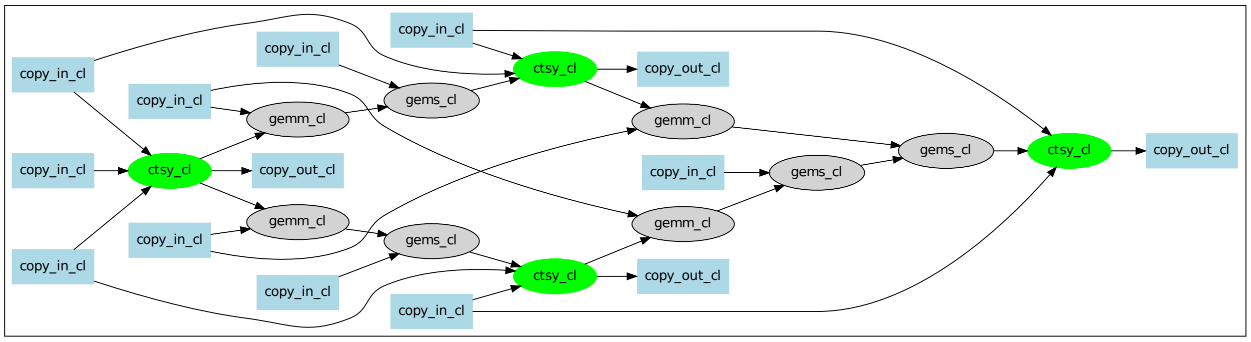


Figure 16. CTSY task graph for 2 by 2 blocking

Figure 15 show the speed-up of Hydra best code compared to the sequential MKL version. On a 32-core machine, the speed-up over 40 can be explained by the fact that Hydra decomposes the triangular Sylvester problem into subproblems that have a higher sequential efficiency than MKL CTSY (such as matrix multiplication). Thus, the speed-up results from both the parallelization of the computation and from the use of efficient kernels. The task graph obtained for a 2 by 2 blocking of CTSY is shown in Figure 16. Square tasks are copy tasks, darker rounded tasks are smalled instances of CTSY and the others are various BLAS-3 operations. The method generated by Hydra to solve CTSY corresponds to the one described by Jonsson *et al.* [7].

Moreover, we observe that the parallel MKL version of CTSY has the same performance as the sequential one. This shows here all the benefits of Hydra: from sequential kernels and the initial formulation of the problem, we are able to generate automatically, with no efforts in manual code tuning, a parallel version of CTSY.

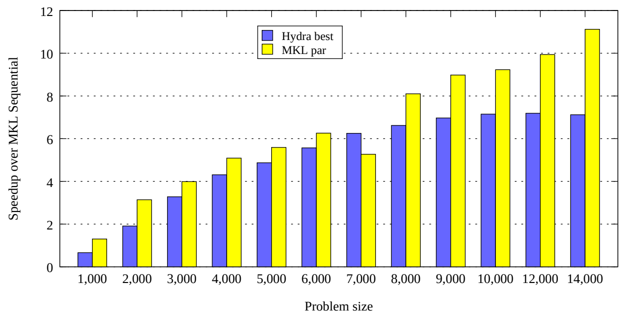


Figure 17. LU Factorization: L*U = A

Finally, Figure 17 shows performance for LU factorization. While the parallel MKL LU outperforms the code generated by Hydra, we note that the performance of Hydra consistently grows with the problem size.

5. Related Works

The field of autotuning software generation tries to answer the problem of generating high performance libraries that are portable across platforms. The necessity comes from the fact that compilers often fail to produce the best possible executable from a normal source code, forcing programmers to manually develop codes that are only optimized for the specific machine it was developed for. Many projects have tackled this problem in different fields, proving the validity of exhaustive search to produce high performance library generators.

ATLAS [9] is a system that exhaustively searches a space of code transformations to find optimal implementations of matrix multiplications and other BLAS operations on a target machine.

The Spiral [8] project is the closest to what is proposed in this document. Spiral is a system to automatically generate high performance libraries for Digital Signal Processing (DSP). It offers a language and set of operators to specify linear transforms for DSP, from which their automatic generation system can derive different algorithms and in the end implementations. They also use exhaustive search to evaluate performance and select the best implementation among all the versions generated by the system.

Our proposed system, differs from Spiral by its targeted domain and from ATLAS in that its main focus on exposing task parallelism. However, both projects offer insights in the different techniques that can be applied to guide the process of exhaustive search through empirical execution of different implementations.

The Flame [3] project advocates goal-oriented programming. It offers a platform to develop algorithms in a systematical way to formally prove they achieve their goal. Flame offers a framework to write iterative algorithms, while we try to start from a problem and automatically derive algorithms recursively using a divide-and-conquer approach. Besides, the code generation approach presented here, relying on the dynamic scheduling of a parallel task graph, differs from the path chosen by Flame. Recent work from Fabregat-Traver and Bientinesi [4] proposes an approach close to ours for finding algorithmic solutions to matrix equations from their mathematical expression. However, they do not explain how the code is generated nor present any performance figures.

Finally, work by Barthou *et al.* [2] on auto-tuning at source code level produce good results on matrix multiplication, but suffered on more complex problems. The exploration space for source to source transformation has to be defined by the user through pragmas. For complex transformations, such as the ones leading to the task graphs produced by Hydra, the sequence of pragmas required would by difficult to identify, even by an expert. Besides, multiple implementations of a same algorithm can become radically different, advocating for looking at problems at a higher level.

6. Conclusion

Hydra is a parallel code generator for a class of linear algebra problems. It starts from the high-level expression of the equation to solve and generates multiple versions of parallel task graphs solving the problem, for multi-core architectures. The essential idea of Hydra is to use a divide-and-conquer approach to find an algorithmic solution to the initial description of the problem. While the recursive decomposition could lead to scalar problems, we choose to rely on existing highly optimized sequential libraries for the resolution of small enough problems. Moreover, we resort to dynamic scheduling techniques in order to avoid load balancing issues.

We have shown that this approach is able to generate parallel codes with no development effort: the user only needs to specify the equation to solve and provide sequential kernels. Moreover, following an auto-tuning approach, the multiple versions generated by Hydra are combined into a code with performance comparable to those of Intel parallel MKL functions, even outperforming for matrix multiplication and Sylvester triangular system resolution the parallel functions of Intel MKL library.

For future works, we plan to generalize Hydra for the generation of parallel codes for heterogeneous architectures. Indeed, one advantage of using a dynamic scheduler such as StarPU [1] is its capacity to handle systems with both CPUs and GPUs. The decision of whether to run the kernel on a CPU or an accelerator is made by the runtime system. The runtime also handles all necessary data transfers. It only requires to provide CPU and GPU versions for all kernels (for instance MKL [6] and PLASMA [5] libraries). Moreover, Hydra offers the opportunity to generate parallel task graphs with non-uniform granularity, through different blocking sizes. Such graphs would then have coarser grain execution paths biased towards GPU execution and finer grain paths, better suited for multicore execution.

Acknowledgments

This work was funded by the Illinois-Intel Parallelism Center at the University of Illinois at Urbana-Champaign. The Center is sponsored by the Intel Corporation.

References

[1] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, Feb. 2011. doi: 10.1002/cpe.1631. URL `http://hal.inria.fr/inria-00550877`.

[2] D. Barthou, S. Donadio, P. Carribault, A. X. Duchâteau, and W. Jalby. Loop optimization using hierarchical compilation and kernel decomposition. In *CGO*, pages 170–184, 2007.

[3] P. Bientinesi, J. A. Gunnels, M. E. Myers, E. S. Quintana-Ortí, and R. A. van de Geijn. The Science of Deriving Dense Linear Algebra Algorithms. *ACM Trans. Math. Softw.*, 31(1):1–26, Mar. 2005.

[4] D. Fabregat-Traver and P. Bientinesi. Knowledge-based automatic generation of partitioned matrix expressions. In *Proceedings of the 13th international conference on Computer algebra in scientific computing*, CASC'11, pages 144–157, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-23567-2. URL `http://dl.acm.org/citation.cfm?id=2040148.2040160`.

[5] U. o. T. Innovative Computing Laboratory. Plasma, 2012. http://icl.cs.utk.edu/plasma/pubs/index.html.

[6] Intel®. Intel Math Kernel Library, 2011. http://software.intel.com/en-us/articles/intel-mkl/.

[7] I. Jonsson and B. Kågström. Recursive blocked algorithms for solving triangular systems - Part I: one-sided and coupled Sylvester-type matrix equations. *ACM Trans. Math. Software*, 28(4):392–415, Dec. 2002. ISSN 0098-3500. doi: 10.1145/592843.592845. URL `http://doi.acm.org/10.1145/592843.592845`.

[8] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[9] R. Whaley, A. Petitet, and J. Dongarra. Automated Empirical Optimizations of Sofware and the ATLAS Project. *Parallel Computing*, 27(1-2):3–35, 2001.

[10] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance blas? *Proceedings of the IEEE*, 93(2):358 –386, feb. 2005. ISSN 0018-9219. doi: 10.1109/JPROC.2004.840444.

Hierarchical Overlapped Tiling

Xing Zhou, Jean-Pierre Giacalone†, María Jesús Garzarán,
Robert H Kuhn†, Yang Ni†, David Padua

University of Illinois at Urbana-Champaign
{zhou53,garzaran,padua}@illinois.edu

†Intel Corporation
{jean-pierre.giacalone,bob.kuhn,yang.ni}@intel.com

ABSTRACT

This paper introduces hierarchical overlapped tiling, a transformation that applies loop tiling and fusion to conventional loops. Overlapped tiling is a useful transformation to reduce communication overhead, but it may also generate a significant amount of redundant computation. Hierarchical overlapped tiling performs overlapped tiling hierarchically to balance communication overhead and redundant computation, and thus has the potential to provide better performance.

In this paper, we describe the hierarchical overlapped tiling optimization and its implementation in an OpenCL compiler. We also evaluate the effectiveness of this optimization using 8 programs that implement different forms of stencil computation. Our results show that hierarchical overlapped tiling achieves an average 37% speedup over traditional tiling on a 32-core workstation.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

General Terms

Algorithms, Languages, Performance

Keywords

Loop tiling and fusion, compiler optimization, stencil computation

1. INTRODUCTION

1.1 Overlapped Tiling

Numerous techniques for the tiling of iteration spaces have been proposed. The goal of tiling is to for improve data locality [27, 28, 1, 2, 24, 9, 22], or contribute to the scheduling of parallel computation [26, 29, 6, 10, 30]. A complementary transformation, loop fusion can be used to decrease loop overhead and enhance locality [17, 20].

In this paper, we discuss transformation by tiling and fusion of stencil computation. Consider the code in Figure 1-(a). Although this figure shows a natural representation of the computation, the pair of loops may cause unnecessary cache misses, depending on how they are scheduled. If the loops are scheduled in the order specified by the code, the second loop will incur frequent cache misses. To increase locality, and also coarsen the granularity of the parallel tasks, the programmer can tile and fuse the loops, as shown in Figure 1-(b). The resulting code requires an explicit barrier to guarantee correctness, because of the data dependences between neighboring tiles of iterations (during iteration t of the outer loop, j consumes data produced by adjacent tiles of loop i, namely tiles $t-1$ and $t+1$ or just one of them at the boundaries). Notice that locality would improve if the same task executes the corresponding i and j tiles in the code of Figure 1-(b). However, good locality is only possible if array A can be kept in cache memory when the execution moves from the first to the second loop. If, however, the array A is larger than the total cache of the processors executing the loops, the traditional loop fusion and tiling transformation applied in Figure 1-(b) will not benefit from locality, because all the iterations of the i loop must complete before the j loop executes. Besides the difficulties for achieving locality of naive tiling, the parallelization transformation may do a suboptimal job because of the barrier introduced. On some architectures, barriers are expensive synchronization operations, and could additionally cause load imbalance. Furthermore, the transformation from Figure 1-(a) to Figure 1-(b) is not possible in some languages, such as OpenMP and OpenCL [14], which do not allow global barriers inside data parallel constructs.

To remove the synchronization and enhance locality, the code can be transformed into the form shown in Figure 1-(c). In this case, each iteration of the outer loop t produces all the data it needs so that its iterations (which correspond to tiles) are independent from each other. This is achieved because each iteration performs redundant computation. The result is a code without the BARRIER and with increased locality.

In the example in Figure 1-(c) loop i produces $A[max(0, t * T - 1) : min(N, (t + 1) * T)]$ in each iteration of the outer

```
parallel for(int i = 0 : N-1)
    A[i] = ...;
parallel for(int j = 0 : N-1)
    ... = A[j-1] + A[j] + A[j+1];
```

(a) Original loops

```
parallel for(int t = 0 : N/T)
    for(int i = t*T; i < min(N, (t+1)*T; i++)
        A[i] = ...;
    BARRIER;
    for(int j = t*T; j < min(N, (t+1)*T; j++)
        ... = A[j-1] + A[j] + A[j+1];
}
```

(b) Traditional tiling and fusion

```
parallel for(int t = 0 : N/T) {
    for(int i = max(0,t*T-1);
            i < min(N,(t+1)*T+1); i++)
        A[i] = ...;
    for(int j = t*T; j<min(N,(t+1)*T); j++)
        ... = A[j-1] + A[j] + A[j+1];
}
```

(c) Overlapped tiling

Figure 1: A simple tiling example for parallel loops

loop, that is, $T+2$ elements, 2 more than the number of elements of A computed by loop i in Figure 1-(b). In total the N/T executions of loop i in Figure 1-(c) produce $\frac{T+2}{T}*N$ elements, so this loop performs $\frac{T+2}{T}*N-N=\frac{2*N}{T}$ more iterations than the corresponding loop of Figure 1-(b). We call the transformation leading to a loop of the form of Figure 1-(c) *overlapped tiling*.

Figure 2-(a) shows a code snippet which represents a typical stencil computation. Pairs of consecutive executions of the inner loop form a pattern similar to that of the two inner loops in Figure 1-(a). If we apply overlapped tiling repetitively and fuse all K executions of the inner loop, it is possible to execute the outer loop without using any barrier. The number of consecutive loops fused is the *depth* of the transformation. In this example, the fusion *depth* is K. The total amount of redundant computation usually grows with the value of *depth*. Figure 2-(b) shows the area of overlap. The triangles that bracket each tile represent the redundant computation.

1.2 Hierarchical Overlapped Tiling

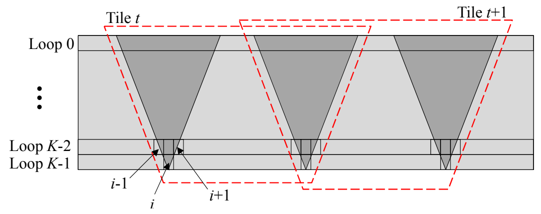
While overlapped tiling removes synchronization and enhances locality, it usually suffers from substantial amount of redundant computation. As the fusion *depth* increases, more synchronizations can be removed, but there is also an increase in the total amount of redundant computation (the shadowed triangle areas in Figure 2-(b)). Hence, we propose the use of *hierarchical overlapped tiling* to balance communication overhead and redundant computation. Figure 3 contrasts overlapped tiling with hierarchical overlapped tiling.

The example assumes 8 consecutive loops executing on a 4-way/8-core multicore system where the two cores on each processor share the last level cache. Figure 3-(a) shows the result of overlapped tiling across the eight cores while Figure 3-(b) shows the result of applying overlapped tiling hierarchically. In Figure 3-(b), overlapped tiling is first applied across the four processors. Within each processor, pairs of consecutive loops are fused. This forces a local barrier between each pair of loops ($loop_0$ and $loop_1$, $loop_2$ and $loop_3$, and so on). This barrier, however, only synchronizes the two cores on each processor and therefore its cost should be relatively low. Within each processor, overlapped tiling is applied to enable the parallel execution of pairs of loops across the two cores without the need for a barrier. Compared to overlapped tiling, the total amount of redundant computation (the shadowed triangle areas between different processors and the smaller shadowed triangles between neighboring cores) caused by the 2 levels of tiling is much smaller. This reduction of the redundant computation is the main source of the performance benefit of hierarchical overlapped tiling over plain overlapped tiling.

In this paper we describe the compiler implementation of the hierarchical overlapped tiling optimization. Automating

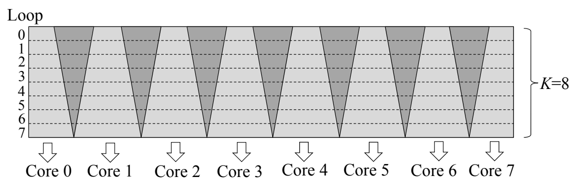
```
for(int k = 0 ; k < K; k++) {
    parallel for(int i = 0 : N-1)
        B[i] = A[i-1] + A[i] + A[i+1];
    swap(A, B);
}
```

(a) Code snippet of K consecutive loops in a stencil code

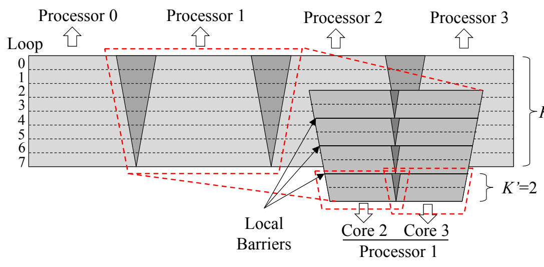


(b) Overlapped tiling

Figure 2: Overlapped tiling of K loops



(a) Overlapped tiling



(b) 2-level hierarchical overlapped tiling

Figure 3: Comparison of overlapped tiling and hierarchical overlapped tiling on a 4-way/8-core multicore system, where each processor contains 2 cores on chip that share the last level cache.

the transformation in a compiler simplifies the task of the programmer during code development and porting. This paper makes the following contributions:

- The introduction of a new tiling transformation called *hierarchical overlapped tiling.*
- The description of its implementation in an OpenCL compiler with the support of Cetus [12] and the Omega library [13] for OpenCL programs which implements the proposed transformation.
- An evaluation of the effect of the techniques on stencil computation. Our experimental results show that hierarchical overlapped tiling is effective and achieves significant speedups when compared to the traditional loop fusion and tiling transformation.

The rest of the paper is organized as follows: Section 2 gives a quantitative analysis of overlapped and hierarchical overlapped tiling; Section 3 describes our compiler implementation; Section 4 discusses the environmental setup for the experimental evaluation; Section 5 shows our experimental results; Section 6 discusses our related work; Section 7 presents the conclusions.

2. ANALYTICAL MODELING

This section gives a quantitative analysis of overlapped tiling and hierarchical overlapped tiling.

2.1 Integer Tuple Set and Relation

We use the notion of iteration space to describe and analyze the transformation introduced here. A d-dimensional *integer tuple* $\vec{x} = [x_0, x_1, ..., x_{d-1}]$ is a vector of d integers. Constraints in the form of equations and inequalities can be used to describe sets of integer tuples. For example, the set $S_0 = \{[1,1],[1,2],...,[1,N]\}$ can be represented as $\{[i,j] : i = 1 \wedge 1 \leq j \leq N\}$.

We assume that the arithmetic expressions in the equations and inequalities are affine and the terms are integers. Logical operators $\neg$, $\wedge$ and $\vee$, and the existential and universal quantifiers $\exists$ and $\forall$ are also needed to describe the set of integer tuples. The representations we use are known as Presburger formulas [15]. In our implementation, these expressions are manipulated using the Omega Library [13].

We also represent *integer tuple relations* with rules described by Presburger formulas. For example the relation $T = \{[i,j] \rightarrow [x,y] : i-1 \leq x \leq i \wedge j-1 \leq y \leq j+1\}$ when applied to the previous S_0 yields the following set:

$$T(S_0) = \{[x,y] : 0 \leq x \leq 1 \wedge 0 \leq y \leq N+1\}$$

Note that given a relation R the size and shape of the original set S and that of $R(S)$ for a relation R can be different. The union $\cup$ of two relations T_1 and T_2 is defined as:

$$T_1 \cup T_2 = T, if\ \forall S, T_1(S) \cup T_2(S) = T(S)$$

2.2 Terminology

Consider K consecutive parallel loops $loop_0$, $loop_1$, ..., $loop_{K-1}$. Assume that the k-th loop $loop_k$ $(0 \leq k < K)$ has the form shown Figure 4.

```
parallel for(int $\vec{i} = [i_0, i_1, ..., i_{D-1}] \in I_0$) { // $loop_0$
        ...
}
parallel for(int $\vec{i} = [i_0, i_1, ..., i_{D-1}] \in I_1$) { // $loop_1$
        ...
}
        ...
parallel for(int $\vec{i} = [i_0, i_1, ..., i_{D-1}] \in I_k$) { // $loop_k$
        ... = $A_k^0[\vec{f}_k^0(\vec{i})]$;
        ... = $A_k^1[\vec{f}_k^1(\vec{i})]$;
                ...
        ... = $A_k^{L_k-1}[\vec{f}_k^{L_k-1}(\vec{i})]$;
        $B_k^0[\vec{g}_k^0(\vec{i})]$ = ...;
        $B_k^1[\vec{g}_k^1(\vec{i})]$ = ...;
                ...
        $B_k^{M_k-1}[\vec{g}_k^{M_k-1}(\vec{i})]$ = ...;
}
        ...
parallel for(int $\vec{i} = [i_0, i_1, ..., i_{D-1}] \in I_{K-1}$){// $loop_{K-1}$
        ...
}
```

Figure 4: A sequence of K parallel loops

Without loss of generality, we assume the body of $loop_k$ contains read access to L_k D'-dimensional arrays A_k^0, A_k^1, ..., $A_k^{L_k-1}$ and write access to M arrays B_k^0, B_k^1,...,$B_k^{M_k-1}$ which are also D' dimensional. We assume that no A array overlaps with a B array. To simplify discussion we assume $A_k^0 = A_k^1 = ... = A_k^{L_k-1} = A_k$ and $B_k^0 = B_k^1 = ... = B_k^{M_k-1} = B_k$. There are L_k references to A_k on the RHS of the first L_k assignment statements in the body of the loop: $A_k[\vec{f}_k^0(\vec{i})]$, $A_k[\vec{f}_k^1(\vec{i})]$, ..., $A_k[\vec{f}_k^{L_k-1}(\vec{i})]$, with $\vec{f}_k^l : \mathcal{Z}^D \rightarrow \mathcal{Z}^{D'}, 0 \leq l < L_k$. There are M_k references to elements of B_k on the LHS of the last M_k statements: $B_k[\vec{g}_k^0(\vec{i})]$, $B_k[\vec{g}_k^m(\vec{i})]$, ..., $B_k[\vec{g}_k^{M_k-1}(\vec{i})]$, with $\vec{g}_k^m : \mathcal{Z}^D \rightarrow \mathcal{Z}^{D'}, 0 \leq m < M_k$.

We compute 3 sets for $loop_k$: I_k, the iteration space; R_k, the set of subscripts of A_k; and W_k the set of subscripts of B_k:

$$R_k = \{\vec{r}\} = \bigcup_{l=0}^{L_k-1} \{[r_0, r_1, ..., r_{D'-1}] : \vec{r} = \vec{f}_k^l(\vec{i}) \wedge \vec{i} \in I_k\}$$

$$W_k = \{\vec{w}\} = \bigcup_{m=0}^{M_k-1} \{[w_0, w_1, ..., w_{D'-1}] : \vec{w} = \vec{g}_k^m(\vec{i}) \wedge \vec{i} \in I_k\}$$

We define C_k for the *consuming relation* as the relation from I_k to R_k, $C_k(I_k) = R_k$, and P_k for the *producing relation*, as the relation from W_k to I_k, $P_k(W_k) = I_k$. C_k and P_k represent the access pattern of the loop body. We use C_k and P_k to describe the different tiling transformations.

2.3 Overlapped Tiling

In this subsection we describe the overlapped tiling loop transformation.

Performing loop fusion and tiling for a sequence of loops of the form shown in Figure 4 is equivalent to finding Q partitions (tiles) I_k^0, I_k^1, ..., I_k^{Q-1} of the iteration space I_k of each $loop_k$. Similar to the definition of R_k and W_k discussed in the last subsection, we define R_k^q as the set of array element indices of A for tile I_k^q, and W_k^q to denote the set of array elements indices of B for tile I_k^q. So, we have:

$$R_k^q = C_k(I_k^q), \qquad I_k^q = P_k(W_k^q) \;\; or \;\; W_k^q = P_k^{-1}(I_k^q)$$

Traditional loop fusion and tiling, such as the code shown in Figure 1-(b), produce orthometric tiles. Since the tiles are a partition of the iterations space, we have:

$$I_k^{q1} \cap I_k^{q2} = \phi, \quad q1 \neq q2$$

However, with overlapped tiling the sum of the size of the tiles I_k^q can be larger than the size I_k. We define the difference as the amount of redundant computation RC_k performed by $loop_k$:

$$RC_k = |I_k^0| + |I_k^1| + ... + |I_k^{Q-1}| - |I_k| \geq 0$$

In traditional loop fusion and tiling, the tiles of the different loops can be unrelated thanks to the barriers. However, in overlapped tiling the data read in an iteration tile must be produced within the same tile to eliminate the need of synchronization. To simplify the discussion, we assume that the data flow in the sequence of fused loops $loop_0$, $loop_1$, ... $loop_{K-1}$ forms a linear chain, which means that $A_{k+1} = B_k$ for all k. Under this assumption, it is necessary that $R_{k+1}^q \subseteq W_k^q$ in overlapped tiling and to avoid unnecessary work we set $R_{k+1}^q = W_k^q$. Hence the corresponding tiles I_{k+1}^q and I_k^q of neighboring loops are related as follows:

$$R_{k+1}^q = C_{k+1}(I_{k+1}^q) = P_k^{-1}(I_k^q) = W_k^q \quad \Rightarrow$$

$$I_k^q = P_k(W_k^q) = P_k(R_{k+1}^q) = P_k(C_{k+1}(I_{k+1}^q)) \qquad (1)$$

Equation 1 shows that the tiles of $loop_k$ are determined by the tiles of $loop_{k+1}$. Equation 1 provides the procedure to perform overlapped tiling: given an arbitrary partition of tiles for the last loop $loop_{K-1}$, the tiles of previous loops $loop_k$ $(0 \leq k < K)$ can be determined iteratively; then all the corresponding tiles from the different loops are fused to build the new loop body.

Consider the code in Figure 2-(a) as an example to perform overlapped tiling. After unrolling, there will be a sequence of K loops. Since every $loop_k$ in Figure 3-(a) is the same, we have:

$$\begin{aligned} I_0 = I_1 = ... = I_{K-1} = I = \{[i] : 0 \leq i < N\} \\ C_0 = C_1 = ... = C_{K-1} = C = \{[i \to x] : i-1 \leq x \leq i+1\} \\ P_0 = P_1 = ... = P_{K-1} = P = \{[x \to i] : i = x\} \end{aligned}$$

Initially, we assign the following tiling partition for the last $loop_{K-1}$:

$$I_{K-1}^q = \{[i] : q \times N/Q \leq i < (q+1) \times N/Q\}$$

which simply evenly partitions the iteration space, I_{K-1}, where the size of each tile is $|I_{K-1}^q| = N/Q$.

Next, the previous loops can be tiled using Equation 1 (to simplify the discussion, the boundaries are ignored):

$$\begin{aligned} |I_{K-2}^q| &= |P_{K-2}(C_{K-1}(I_{K-1}^q))| = |P(C(I_{K-1}^q))| \\ &= |\{[i] : q \times N/Q - 1 \leq i < (q+1) \times N/Q + 1\}| \\ &= N/Q + 2 = |I_{K-1}^q| + 2 \\ |I_{K-3}^q| &= |P(C(I_{K-2}^q))| = N/Q + 4 = |I_{K-2}^q| + 2 \\ &... \\ |I_0^q| &= |P(C(I_1^q))| = N/Q + 2 \times (K-1) = |I_1^q| + 2 \end{aligned}$$

Therefore:

$$|I_k^q| = |P(C(I_{k+1}^q))| = N/Q + 2 \times (K-1-k)$$

$$RC_k = (\sum_{q=0}^{Q-1} |I_k^q|) - |I_k| = 2 \times Q \times (K-1-k)$$

The total amount of redundant computation RC is defined as the sum of the redundant computation of each $loop_k$:

$$RC = \sum_{k=0}^{K-1} RC_k = Q \times K \times (K-1) \qquad (2)$$

RC is a monotonic function of the number of tiles Q and the number of loops to fuse K. According to Equation 2, for the example shown in Figure 2-(a), we can see the trend: the amount of redundant computation RC increases with both Q and K. Although this observation is derived from the specific example, it is easy to see that the trend is true in general. Compared with traditional loop fusion and tiling, where synchronization is necessary, overlapped tiling saves the overhead of $K-1$ barriers. Suppose the average overhead of each barrier is t_s, and the average computation time for each iteration in the original code is t_c, the overhead difference between overlapped tiling over traditional tiling is:

$$\Delta Overhead = t_s \times (K-1) - t_c \times RC/Q \qquad (3)$$

2.4 Hierarchical Overlapped Tiling

According to the analysis in the last subsection, coarse grain tiles (and thus small number of tiles) reduce the amount of redundant computation in overlapped tiling. However, too few tiles would reduce the amount of parallelism. Similarly, reducing the number of fused loops also reduces the amount of redundant computation, but at the expense of the additional synchronization required between loops.

The goal of hierarchical overlapped tiling is to adapt to the memory hierarchy of the target machine. Consider a multicore system with N_p processors, where each processor has N_c cores sharing the last level cache. It is expected that the average overhead of synchronization of processors sharing a cache (t'_s) will be significantly smaller than that of synchronizing cores on different processors (t_s) that need to communicate through main memory and/or bus.

$$t_s/t'_s \gg 1 \qquad (4)$$

Based on the above observation, we proceed as follows: first, we perform coarse-grain tiling for processors; then perform overlapped tiling within the tile that is assigned to each processor, and generate sub-tiles for each core of the processor. During this 2-level tiling, the parameters for overlapped tiling are determined separately for each level.

For simplicity, we assume that the total number of tiles, Q, generated by the plain overlapped tiling discussed in the previous subsection is equal to the number of cores: $Q = N_p \times N_c$. During the first level tiling of the hierarchical overlapped tiling, only N_p tiles are generated, so the total amount of redundant computation introduced in this level is:

$$RC_1 = RC(N_p, K) = N_p \times K \times (K-1)$$

After the first level tiling, each processor q is assigned a coarse-grain tile: I_0^q, I_1^q, ..., I_{K-1}^q $(0 \leq q < N_p)$. The tile for each processor is implemented as a sequence of inner-loops, whose boundaries are defined by the constraints discussed at the beginning of Section 2.1. Then, overlapped tiling can be applied within each tile. However, since the sub-tiles are going to be mapped to cores of the same processor, it is expected that the overhead of synchronization of cores within the same processor, t'_s will be significantly smaller than the synchronization between cores across processors, t_s. As a result, for each tile in the second level of tiling the number of fused loops (fusion *depth*) can be smaller. Although this increases the amount of synchronization, it also reduces the amount of redundant computation. Suppose the second level of tiling only fuses K' consecutive loops each time and $K' < K$, then the amount of redundant computation for each processor in the second level tiling would be:

$$RC_2 = RC(N_c, K') = N_c \times K' \times (K'-1) \qquad (5)$$

Therefore, the total amount of redundant computation of 2-level overlapped tiling is:

$$\begin{aligned} RC' &= RC_1 + N_p \times RC_2 \\ &= N_p \times K \times (K-1) + N_p \times N_c \times K' \times (K'-1) \\ &= \frac{Q}{N_c} \times K \times (K-1) + Q \times K' \times (K'-1) \\ &= Q \times K \times (K-1) \times (\frac{1}{N_c} + \frac{K' \times (K'-1)}{K \times (K-1)}) \\ &\approx RC \times (1/N_c + (K'/K)^2) \end{aligned}$$

Furthermore, additional K/K' local synchronization operations are introduced during the second level of tiling. This adds an extra latency of $t'_s \times K/K'$. Hence the overhead difference between 2-level overlapped tiling and traditional tiling is:

$$\begin{aligned} \Delta Overhead' &= t_s \times (K-1) - t'_s \times K/K' - t_c \times RC'/Q \\ &\approx t_s \times (K-1) - t'_s \times \frac{K}{K'} - \frac{t_c}{Q} \times RC \times (\frac{1}{N_c} + (\frac{K'}{K})^2) \\ &= t_s \times (K - 1 - \frac{t'_s}{t_s} \times \frac{K}{K'}) - \frac{t_c}{Q} \times RC \times (\frac{1}{N_c} + (\frac{K'}{K})^2) \end{aligned}$$

As mentioned before, t'_s is much smaller than t_s (Equation 4). Thus, if K' is appropriately chosen, $\Delta Overhead'$ can be made larger than $\Delta Overhead$ as defined in 3, which shows the potential benefit of hierarchical overlapped tiling.

```
__kernel void kernel(__global float *A,
                     __global float *B) {
    int i = get_global_id(0);
    B[i] = A[i-1]+A[i]+A[i+1];
}
```

(a) OpenCL kernel code

```
cl_mem mem_A, mem_B, *p1=&mem_A, *p2=&mem_B;
...;
for(int k = 0 ; k < K; k++) {
    clSetKernelArg(kernel,0,sizeof(cl_mem),p1);
    clSetKernelArg(kernel,1,sizeof(cl_mem),p2);
    size_t global_work_size[] = {N};
    new_evt=(cl_event*)malloc(sizeof(cl_event));
    clEnqueueNDRangeKernel(queue,kernel,1,NULL,
        global_work_size,NULL,1,event,new_evt);
    event = new_event;
    swap(p1, p2);
}
```

(b) OpenCL host code

Figure 5: OpenCL code example

3. IMPLEMENTATION

3.1 OpenCL

An OpenCL program consists of two classes of components: host code and kernel code. The host code contains the control logic and usually runs on a general purpose processor, while the kernel code contains most of the computation and executes on the target device, such as an accelerator. The host code is a C/C++ program with OpenCL API invocations. The kernel code is written in OpenCL C, which is a subset of C99 with extensions. OpenCL *kernel*s resemble C procedures. During execution, the host code compiles the kernel code. This dynamic runtime compilation makes OpenCL programs portable across different devices.

OpenCL kernels follow the SPMD execution model. The host code specifies the work item organization of each kernel, where a *work item* is the unit of scheduling. Work items are grouped into *work groups*. The work groups and the work items within each work group have N dimensions $(N \leq 3)$. Each work group is represented by an N-tuple work group ID (which can be accessed during execution using `get_group_id()`), and each work item also has an N-tuple work item IDs. The functions (`get_global_id()`) and (`get_local_id()`) can be used to access the global or local work item ID respectively.. The work item ID defines an N-dimensional index space. Thus, we can conceive the execution of a sequence of OpenCL kernels as a sequence of parallel loops. For example, the OpenCL code in Figure 5 is equivalent to the code in Figure 2-(a).

3.2 Implementation Overview

We implemented the hierarchical overlapped tiling optimization to evaluate its effectiveness. A diagram representing our experimental system is shown in Figure 6. The dashed arrows represent offline data flow while solid arrows represent runtime data flow. The system contains three major components: a delayed compilation mechanism, an offline analyzer and the optimizer. The offline analyzer is implemented as a pass of Cetus [12]. The optimizer is a source-to-source translator which reads the original kernel code and generates OpenCL code, which is fed to the OpenCL runtime.

Both, the offline analyzer and the optimizer use the Omega library [13] to perform the integer tuple space computations. In addition, the analyzer uses the Omega Library to generate loops from the integer tuple sets.

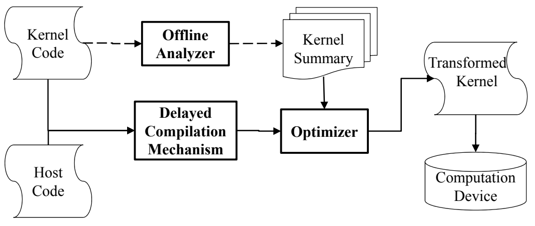


Figure 6: Framework of the automatic transformation tool.

3.3 Delayed Compilation Mechanism

In our framework, each OpenCL kernel execution (or instance) is seen as a parallel loop. This view combines information from the host code (the iteration space) and the OpenCL kernel code (the body of the loop). The framework applies overlapped tiling or hierarchical overlapped tiling to a sequence of kernel invocations. To fully automate the process, two problems must be solved: 1) The dynamic compilation unit in OpenCL is kernel, but overlapped tiling fuses across kernel boundaries. Therefore, some form of global analysis is needed. 2) The optimal value of the transformation parameters usually depend on the iteration space, which is the work size of an OpenCL kernel. However, in OpenCL the work size is specified by the host code, and sometimes the values of the work size can only be resolved after compiling the kernel code. In order to solve these two problems, we designed and implemented a delayed compilation mechanism. It postpones the compilation process (which should be done in `clBuildProgram()` in the standard OpenCL APIs) of the kernel code until a sufficient number of kernels have been created and enqueued by the host code. In order to enable the reuse of the standard compilation process, our delayed compilation mechanism is implemented as wrapper functions.

Our delayed compilation mechanism works as follows: when the host code invokes `clEnqueueNDRangeKernel()` to push a kernel instance into the command queue, the kernel instance is actually held within a `pending_kernel` object in the pending queue. Each pending kernel carries with it the work size and arguments specified by the host code. The kernel will not be compiled until:

1. a synchronization point is found during execution of the host code; this usually means that the results of the pending kernels are needed to continue execution, or
2. the number of pending pending kernels exceeds a given threshold.

Each time the compilation process is triggered, the optimizer will be invoked to select some kernels from the pending queue to apply the appropriate transformation. After compilation, the transformed kernel generated by the optimizer is executed. This way, the transformation is transparent to the programmer.

3.4 Offline Analyzer

As mentioned before, the consuming relation C_k and producing relation P_k represent the access pattern of $loop_k$. Since we view each OpenCL kernel as a parallel loop, we can also use consuming and producing relations to represent the access pattern of the kernel code (the body of the loop). Computing consuming and producing relations requires traversing the syntax tree of the kernel code to collect every global memory reference, which might be an expensive operation. If symbolic variables are allowed in the constraints of integer tuple relations, it is possible to compute the consuming and producing relations of each kernel offline to reduce the overhead of the online compilation.

The algorithm to compute consuming and producing relations makes use of symbolic range propagation [7]. This produces a conservative approximation to the range of possible values of every variable used to compute array ranges. The output of the offline analyzer is the summary for each $Kernel_k$, which contains the consuming relation C_k^A or producing relation P_k^A of every global array A accessed by the kernel body.

3.5 Optimizer

When the compilation process is triggered by the delayed compilation mechanism, the optimizer tries to apply the overlapped tiling transformation on the pending kernels. OpenCL programs can be configured for in-order execution or out-of-order execution. In in-order execution mode, kernels are executed in the same order that they are enqueued, while in out-of-order execution mode, the scheduler of OpenCL runtime is free to reorder kernels as long as the partial order specified by the programmer is respected. Fusion can be applied to a sequence of loops if the runtime is configured for in-order execution or to the topological sort of a partial order if the runtime is configured for out-of-order execution.

The optimizer uses the producing/consuming relations discussed in Section 2 to represent the dependence of kernel instances. The discussion in Section 2 only considers the situation when all the data consumed by a loop are produced by its predecessor. However, many programs do not conform to this simplification. We say that there is a *producer-consumer relation* between two kernels X and Y: if and only if kernel X is an ancestor of kernel Y (in terms of the partial execution order enforced by event objects) and the elements produced by kernel X are consumed by kernel Y. We say that X is Y's producer and Y is X's consumer. The goal is to guarantee that for each tile, the producer kernel produces all the data (array elements) that the consumer kernel needs. According to the code in Figure 5, kernel instances $Kernel_0$, $Kernel_2$, ..., $Kernel_{2n}$, ... read `mem_A` and write `mem_B`, while $Kernel_1$, $Kernel_3$, ..., $Kernel_{2n+1}$, ... read `mem_B` and write `mem_A`. Hence the producers of $Kernel_k$ include $Kernel_{k-2n} \forall n \leq k/2$. This is denoted as $Producer(k) = \{k - 2n | \forall n = 1, 2, ..., \lfloor k/2 \rfloor\}$.

We can represent producing/consuming relations in terms of data dependences. If we view the OpenCL code in Figure 5 to be equivalent to a doubly nested loop like in Figure 2-

(a), there are read-after-write dependences from iterations $(k-2n+1, i)$ and $(k-2n+1, i \pm 1)$ to iterations (k, i), $n = 1, 2, ..., \lfloor k/2 \rfloor$. So the dependence vectors are $(2n-1, 0)$ and $(2n-1, \pm 1)$, which correspond to be direction vectors $(<, =)$ and $(<, *)$. With the distance vectors or direction vectors, loop transformations can be performed as described in [3] On the other hand, the dependence vectors can be computed through producing/consuming relations of kernel instances.

Although we could use the dependence information, what we need for the transformation is only the data flow between kernels, which can be naturally represented by producing/-consuming relations. Thus, the analysis in the optimizer uses the producing/consuming relations instead of data dependences.

3.5.1 Tiling Algorithm

To apply the overlapped tiling optimization to the sequence of kernels, we first compute the symbolic tiles for each kernel, and then the actual size of the tiles is determined.

```
Input: Consuming and producing relations $C_k$ and $P_k$
        for each $Kernel_k$
Output: $Tile_k$ for each $Kernel_k$
Initialization: $Tile_k = \emptyset$, for each $Kernel_k$

$Kernel_{K-1}$ = the last kernel in a topological order;
$Kernel_0$ = the first kernel in the same topological order;
$Tile_{K-1} = \{[id_0, ..., id_{D-1}] : s_0 \leq id_0 < s_0 + len_0 \wedge$
        $... \wedge s_{D-1} \leq id_{D-1} < s_{D-1} + len_{D-1}\}$;
for(each $Kernel_k$ in $Kernel_{K-1}, .., Kernel_0$
                          in reversed topological order)
  for(each array $A$ that is an argument read by $Kernel_k$)
    for(each $Kernel_{k'}$ which is an ancestor of $Kernel_k$)
      $Tile_{k'} = Tile_{k'} \cup P^A_{k'}(C^A_k(Tile_k))$;
```

Figure 7: Algorithm for symbolic tiling

Symbolic Tiling. Figure 7 shows a simplified algorithm for tiling. We assume a collection of K kernels ($kernel_0$, $kernel_1$, ..., $kernel_{K-1}$). The basic idea of this algorithm is, starting from the last kernel in topological order ($kernel_{K-1}$), traverse backwards the consumer-producer chain to determine all the data that must be computed inside each tile, so that no inter-tile communication or synchronization is needed.

$Tile_k$ is a set of *integer tuple*s where each element represents a work item ID in the resulting tile for $Kernel_k$. At the beginning, the tile for the start kernel $Tile_{K-1}$ is initialized with $\{[id_0, ..., id_{D-1}] : s_0 \leq id_0 < s_0 + len_0 \wedge ... \wedge s_{D-1} \leq id_{D-1} < s_{D-1} + len_{D-1}\}$ in which s_i and len_i are variables whose values will be determined later. We assume without loss of generality that the work items for $Kernel_{K-1}$ are organized into D-dimensional objects. The output of this algorithm is the symbolic tile $Tile_k$ for each kernel, which contains symbol variable s_i and len_i in the constraints. The symbolic tiling algorithm can be used by the plain overlapped tiling and the hierarchical overlapped tiling, since it requires neither the work size nor the tile boundaries as input.

Determining Tile Size. With the symbolic tile for each kernel, we can estimate the memory footprint of each tile. For each global array A, a superset of the elements read (R_A) or written (W_A) within a tile can be computed as follows:

$$R_A = \bigcup_{k=0}^{K-1} C^A_k(Tile_k), \quad W_A = \bigcup_{k=0}^{K-1} {P^A_k}^{-1}(Tile_k)$$

If we only count global array elements [1], the size of memory footprint is $FP = \sum_A |R_A \cup W_A|$. FP is a function of the tile length (len), the number of kernels to fuse (K) and the work size of $Kernel_{K-1}$ (I_{K-1}). Given K and I_{K-1}, the tile size can be determined by finding a value of len that satisfies the constraint below, to guarantee that the footprint of each tile fits in cache:

$$FP = FP(len, K, I_{K-1}) < Eff\_Cache$$

Eff_Cache is the effective size of the target level of cache. For plain overlapped tiling, Eff_Cache is set to be a fraction of the shared last level cache on each processor; for the second level of hierarchical overlapped tiling, I_{K-1} should be the tile size generated by the higher level tiling and Eff_Cache is determined by the size of the private cache (L1 or L2). For other architecture such as GPUs, Eff_Cache is determined by the size of the local storage.

Determining the Loop Fusion Depth K. The discussion in the last subsection assumes that the number of kernels to fuse (K) is known. As discussed in Subsection 2.3, the amount of redundant computation introduced by plain overlapped tiling (Equation 5) and hierarchical overlapped tiling is a function of the loop fusion depth K. We provide two ways to specify the value of K: (1) the programmer specifies the value of K as an input parameter; (2) the value of K can be determined based on the performance feedback obtained using training inputs.

When using the second method for plain overlapped tiling, we define $\Delta RC(k) = RC(k) - RC(k-1)$, which is the redundant computation increment when adding $Kernel_k$ for fusion. We find that usually $\Delta RC(k)$ is a monotonically nondecreasing function, e.g., for the code in Figure 2-(a) discussed in Subsection 2.3, $\Delta RC(k) = 2 \times Q \times (k-1)$. Thus, in order to obtain the maximum $\Delta Overhead$ (Equation 3) of overlapped tiling over traditional tiling, we should stop adding new kernels when the following become true:

$$t_c \times \Delta RC(k)/Q \geq t_s \quad or \quad \Delta RC(k)/Q \geq t_s/t_c \qquad (6)$$

In the above inequality, t_s is platform dependent, while t_c is application dependent. We assume that it is possible to profile the value of t_s/t_c with a representative input data. Then, for adaptive tuning, the optimizer keeps adding new kernels from the pending queue to be transformed until inequality 6 becomes true; at that point, the number of kernels added is the desired value of K.

For hierarchical overlapped tiling, tuning must be done for

[1]Our current implementation only considers the global memory space, since the main data structures of the programs we evaluated are in global memory space. However, to be accurate, objects in the local memory space such as local memory objects and stack variables should also be taken into account.

each level of tiling to determine K and K' separately, with different average cost of synchronization (t_s or t'_s).

Create Thread-local Buffers. For some architectures such as GPGPUs, different OpenCL address spaces `__global`, `__local` and `__private` are mapped to physical storages with different speeds. Thus, it is beneficial for performance to create thread-local buffers and promote working data into faster storages. For multicore platforms with transparent caches, the different OpenCL address spaces are all mapped to main memory and therefore the use of these address spaces does not impact performance. However, since different tiles compute redundantly some of the array elements, the data for each tile must be privatized for correctness.

For each work item we create a local buffer for each array A of size $|R_A \cup W_A|$, so that the thread-local buffers are large enough to hold every element of the array base A accessed by each tile. In our implementation, we only create buffers of regular shape, e.g., for 2D data we always create a rectangular area for the thread local buffer which covers every element accessed by the work item. This strategy simplifies the loop boundary control and array element index translation of the generated code, but can introduce more redundant computation when the dependences are irregular.

3.6 Other Implementation Issues

In order to avoid recompilation by our delayed compilation mechanism, we implemented a cache to hold previous compilation results. Each compilation result (compiled code of transformed kernel) is indexed by the sequence of kernel names, and the value of the tiling-related arguments of each kernel. An argument is tiling-related only if it is involved in any constraints of the consuming or producing relation of the kernel of which it is an argument. For instance, the argument `A` in Figure 5 is not tiling-related. Thanks to the compilation cache, if the same sequence of kernels occurs more than once during execution, previous compilation results can be accessed to avoid recompilation. For OpenCL programs with stable repetition units, such as stencil code, the total number of compilation passes can be reduced to 1 or 2: one for the steady state, the other for the epilog if it exists. If the OpenCL program does not have a stable repetition pattern, the total compilation time could increase. But if the OpenCL programs run on a heterogeneous platform with multiple computation devices, it is possible to overlap the transformation and compilation with the kernel execution.

For stencil programs which iteratively execute the same kernel, delayed compilation has an effect similar to loop unrolling, which may results in code explosion and hurt instruction locality (and hence dramatically increase instruction cache misses) when the fusion depth is large. To avoid code explosion, our implementation includes a pattern matching-based ”re-rolling” pass where the generated code only contains two loops: an outer loop, whose iteration count is the number of loops being fused, and an inner loop, with a variable number of iterations that depends on the value of the induction variable of the outer loop.

4. EVALUATION ENVIRONMENT

4.1 Target platform

We evaluate the efficiency of the proposed transformation on an SMP workstation with 4 Intel Xeon L7555 processors running at 1.87GHz. Each processor has 8 cores, sharing a 24MB unified L3 cache on chip. Each core contains a 256KB private L2 cache and 32KB L1 D-cache. SMT is disabled for each core, so there is one hardware thread per core. After transformation, the OpenCL code is compiled using an experimental OpenCL compiler from Intel Labs.

4.2 Benchmarks

For the evaluation we use 8 benchmarks: 1D/2D/3D-Jacobi, PathFinder, Poisson, Biharmonic, HotSpot and Cell. The first 3 benchmarks (1D/2D/3D-Jacobi) are Jacobi iterations for synthetic linear systems; PathFinder uses dynamic programming to find a minimum weighted path; Poisson is a numerical solver of the poisson equation, calculating the Laplace operator [5] over a 2D grid with the 5-point stencil. Biharmonic is the numerical PDE solver calculating the Biharmonic operator [5] over a 2D grid with a 13-point stencil. HotSpot implements a chip temperature estimation model[11]. Cell [4] is a 3D game of life. For each application, the operation in the body of the stencil loop is implemented as an OpenCL kernel. The inputs of the benchmarks are listed in Table 1.

	Data Dimension	**Problem Size**	**Points of Stencil**
1D-Jacobi	1	64K	3
2D-Jacobi	2	256x256	9
3D-Jacobi	3	64x64x64	27
PathFinder	1	100K	3
Poisson	2	256x256	5
Biharmonic	2	256x256	13
HotSpot	2	512x512	9
Cell	3	60x60x60	27

Table 1: Benchmarks

For some stencil programs, such as Jacobi, the number of steps of the outer loop depends on a convergence test. This requires a synchronization between kernel code and host code that prevents loop fusion. To enable the overlapped and hierarchical overlapped tiling optimizations we modified the code so that the convergence test (and as result the synchronization) only occurs every 1024 iterations. The total iterations number for the benchmarks without convergence test is 16,384.

We use `pthread_setaffinity_np()` to set the appropriate affinity for each worker thread to guarantee that the threads executing the sub-tiles within the same tile communicate through a shared cache for hierarchical overlapped tiling.

5. EXPERIMENTAL EVALUATION

In this Section, we present our experimental results. Section 5.1 presents the main results, Section 5.2 discusses parameter sensitivity, and Section 5.3 discusses compilation overhead.

5.1 Performance Overview

Figure 8 shows the performance speedup of traditional tiling, overlapped tiling and hierarchical overlapped tiling relative to the original OpenCL code. Since OpenCL only supports 2 levels of work item organization, a 2-level hierarchical overlapped tiling is used for each benchmark. For overlapped tiling and hierarchical overlapped tiling, the figure shows the speedup for the best value of the loop fusion depth K, as shown in Table 2 (and discussed in the next Section). In general, the performance of hierarchical overlapped tiling is always better than that of plain overlapped tiling, and overlapped tiling is always better than that of traditional tiling, with the exception of Cell and 3D-Jacobi. For Cell and 3D-Jacobi, the performance curves of the three tiling transformations are very similar. The reason is that their main data structure is 3-dimensional and the amount of redundant computation grows quartically with the fusion depth K. Therefore, there are not many opportunities for overlapped tiling and hierarchical overlapped tiling. In our experiments, overlapped tiling and the 2-level hierarchical overlapped tiling achieves an average speedup of 18% and 37% over traditional tiling, respectively.

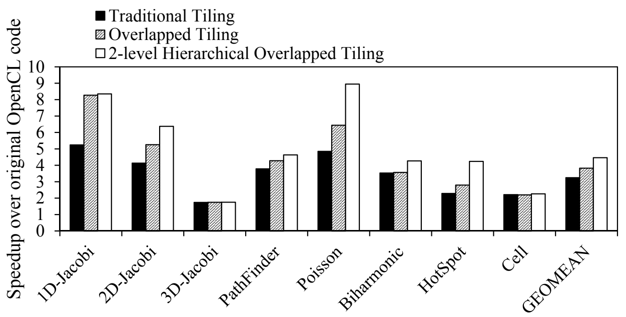


Figure 8: Speedup of traditional tiling, overlapped tiling and hierarchical overlapped tiling over the original openCL code.

5.2 Parameter Sensitivity

Loop Fusion Depth for the First Level of Tiling. Figure 9 shows the performance of overlapped and hierarchical overlapped tiling as the value of the loop fusion depth K changes. For overlapped tiling, there is only one value of K; for 2-level hierarchical overlapped tiling, K is the loop fusion depth for the first level of tiling, while K' is the value of loop fusion depth for the second level of tiling (K' is kept constant at 2 for the experiments in Figure 9). As discussed in Subsection 2.3 and 2.4, K, determines the amount of redundant computation introduced by overlapped and hierarchical overlapped tiling, and thus the overall speedup over traditional tiling.

In Figure 9, lines a, b and c show the speedup of traditional tiling, overlapped tiling, and 2-level hierarchical overlapped tiling over the original OpenCL code, respectively. In addition, we manually modified the overlapped tiling code to remove the redundant computation (hence, there are race conditions and the results of the executed code are not guaranteed to be correct), and its performance is shown by Line d. OpenCL standard does not support a global barrier across work item groups, which is required for traditional tiling (See Figure 1-(b)); thus, the barriers used for traditional tiling (Line a in Figure 9) are barriers that we implemented with low level primitives. However, overlapped tiling and 2-level hierarchical overlapped tiling only require synchronization within the work item groups, which is supported by the OpenCL standard. The discrepancy between overlapped tiling and Line d shows the overhead of the redundant computation introduced by overlapped tiling; the difference between traditional tiling and Line d shows the synchronization overhead saved by fusing the kernels. In Figure 9 we can see that Line d typically grows as the loop fusion depth

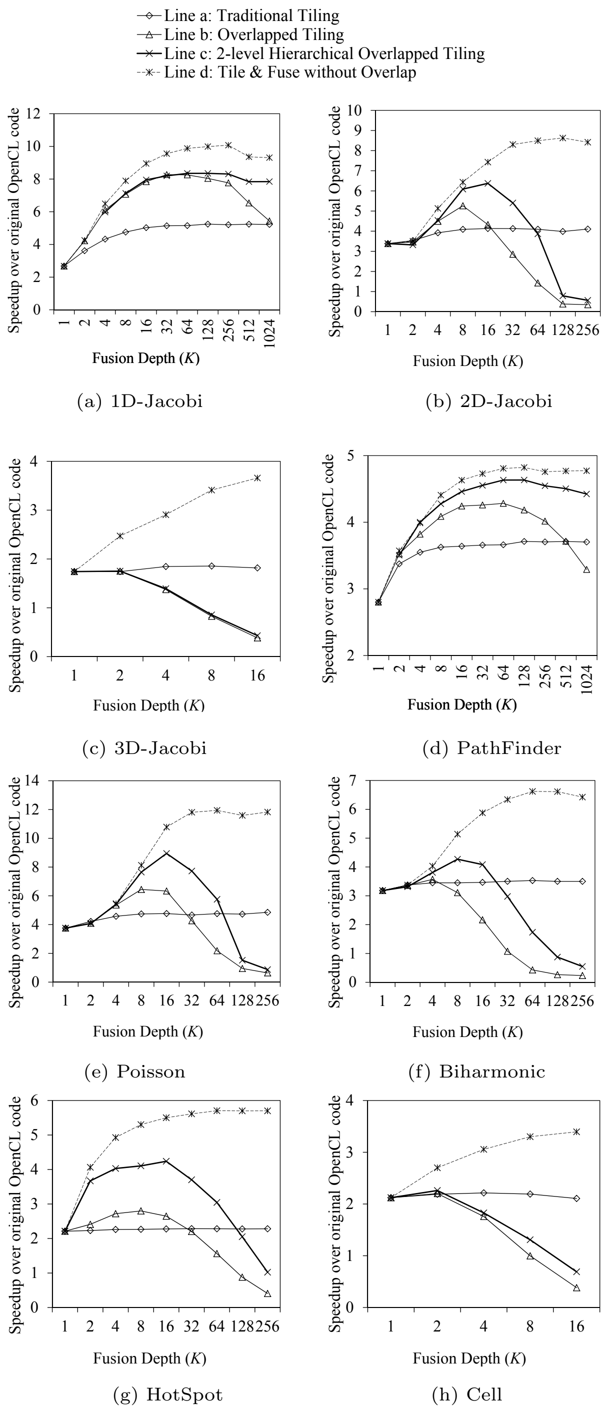


Figure 9: Evaluating the performance overlapped tiling and hierarchical overlapped tiling by scaling fusion depth K.

K increases. This is because the number of syncrhonization operations removed grows with the depth. If we do not count the cost of the redundant computation introduced, the performance benefit is always positive (if $RC = 0$, $\Delta Overhead$ in Equation 3 always increases when K increases). In fact, Line d defines the upper bound of performance for overlapped and hierarchical overlapped tiling.

For the two 1D benchmarks (1D-Jacobi and PathFinder), both plain overlapped tiling and hierarchical overlapped tiling can achieve significant speedup over traditional tiling, because the growth rate of redundant computation for overlapped tiling is low. For the 2D benchmarks (2D-Jacobi, Poisson, Biharmonic and HotSpot) the growth rate of the redundant computation is higher than for the 1D benchmarks. Thus, the interval of benefit (the values of fusion depth K for which lines b or c are above line a) of the 2D benchmarks is smaller than that of the 1D benchmarks for both, overlapped and hierarchical overlapped tiling. The figure also shows that the interval of benefit of hierarchical overlapped tiling is always bigger than that of plain overlapped tiling. Within the 2D benchmarks, Biharmonic shows the least speedup with overlapped tiling over traditional tiling. This is because the stencil of Biharmonic depends on 13 neighboring points versus 9 for 2D-Jacobi, 5 for Poisson, and 5 for Hotspot (see Table 1). As a result, the redundant computation of Biharmonic grows faster than that of the other 2D benchmarks. However, hierarchical overlapped tiling still achieves speedup over traditional tiling. Since the input data of Cell and 3D-Jacobi are 3-dimensional, the amount of redundant computation increases so fast that there is no opportunity for overlapped tiling.

When comparing hierarchical overlapped tiling versus overlapped tiling the plots in Figure 9 show that hierarchical overlapped tiling performs better as the value of loop fusion depth K increases (the only exception occurs with the 3D benchmarks where all 3 tiling mechanisms behave the same), because the growth rate of redundant computation is lower for hierarchical overlapped tiling than for plain overlapped tiling. Hierarchical overlapped tiling is less sensitive to the value of K than overlapped tiling because, as mentioned above, the beneficial region of hierarchical tiling is larger than that of overlapped tiling.

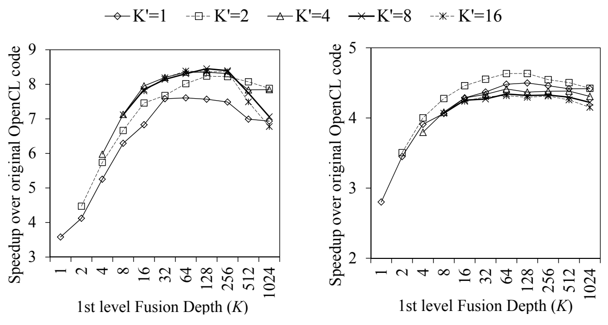
The adaptive fusion mechanism described in Subsection 3.5 uses inequality 6 and the value of t_s/t_c profiled with a smaller input set to determine the choice of the loop fusion depth K value. Our results show that the optimal values for K found using the adaptive mechanism are the same as the ones found using the empirical search in Figure 9, and shown in Table 2.

Loop Fusion Depth for the Second Level of Tiling. Compared to the loop fusion depth K for the first level tiling of hierarchical overlapped tiling, the tuning of loop fusion depth K' for the second level tiling is more involved. This is partially because the performance of the second level tiling depends on the first level of tiling. Figure 10 shows the performance of hierarchical overlapped tiling for 1D-Jacobi and PathFinder with different pairs of K and K'. We can see that $K' = 2$ is the best choice for PathFinder; but there is no obvious optimal value for 1D-Jacobi. Thus, manual tuning

	Overlapped Tiling	**Hierarchical Overlapped Tiling**
1D-Jacobi	$K = 32$	$K = 64$
2D-Jacobi	$K = 8$	$K = 16$
3D-Jacobi	$K = 2$	$K = 2$
PathFinder	$K = 32$	$K = 64$
Poisson	$K = 8$	$K = 16$
Biharmonic	$K = 4$	$K = 8$
HotSpot	$K = 4$	$K = 16$
Cell	$K = 1$	$K = 2$

Table 2: Loop Fusion depth K determined by the adaptive mechanism.

may be required to find the optimal fusion depth K' for the second level of hierarchical overlapped tiling. However, the performance impact of the second level tiling is significantly smaller than that of the first level tiling. For instance, when K' is set to 2, the average performance loss is less than 5% compared to the best K'.



(a) 1D-Jacobi (b) PathFinder

Figure 10: Fusion depth K' for the second level of hierarchical overlapped tiling.

Input Size. Figure 11 shows the speedup of hierarchical overlapped tiling over plain overlapped tiling for the different benchmarks, as the input size increases. The speedups are computed using the best value of K. The figure shows that hierarchical overlapped tiling performs better than overlapped tiling for the 2D-benchmarks. It also shows, that the performance difference between hierarchical overlapped tiling and plain overlapped tiling becomes smaller as the input data size increases. This is because since the total number of worker threads remains constant, the tile size is determined by the input data size. With smaller tiles, the redundant computation has a higher impact; thus, overlapped tiling is less efficient, while hierarchical overlapped has more opportunities to reduce the overheads introduced by the redundant computation. With larger tiles, the redundant computation has less impact, and as a result, there is less difference between overlapped and hierarchical overlapped tiling. For the 3D benchmarks 3D-Jacobi and Cell, since the amount of redundant computation grows so fast, the optimal K for both plain overlapped tiling and hierarchical overlapped tiling is less than 2, so there is no obvious performance discrepancies between the two schemes.

5.3 Compilation Overhead

Since our tool transforms OpenCL kernel code at runtime, the overheads introduced need to be considered. The over-

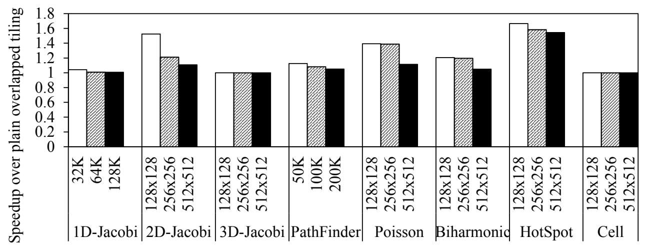


Figure 11: Speedup of hierarchical overlapped tiling over plain overlapped tiling with different input sizes. The horizontal axis is the input size for each benchmark.

head consists of two parts: The OpenCL runtime that transforms the kernel code and the execution of the Omega Library. The compilation cache described in Subsection 3.6 can help reduce both parts of the runtime overhead: if the sequence of kernels selected for optimization are the same as a previous sequence, no compilation needs to be done. For the benchmarks used in this paper, the OpenCL runtime compilation needs to be invoked at most twice, one for the steady state and another for the epilog. Figure 12 shows the compilation times of overlapped tiling and hierarchical overlapped tiling, normalized to the compilation time of the original program. On average, overlapped tiling requires 37% more compilation time, while hierarchical overlapped tiling costs about 60% more.

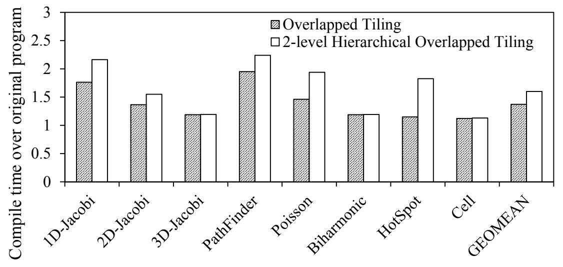


Figure 12: Compile time for overlapped tiling and hierarchical overlapped tiling, normalized to the compile time of the original OpenCL code.

6. RELATED WORK

Loop tiling is a traditional but effective optimization for performance. Numerous optimizations based on the tiling of iteration spaces have been proposed for improving data locality [27, 28, 1, 2, 24, 9, 22], or exploiting parallelism [26, 29, 6, 10, 30].

The closest work to our overlapped tiling is that of Krishnamoorthy et al.[16]. Their approach uses the polyhedral model of computation and manipulates regular data dependencies. Other works such as Ripeanu et al. [23] and Meng et al. [21] describe performance models to predict the optimal amount of redundant computation for stencil computation in a grid environment with message passing or for GPUs, respectively. However, no fully automated tool for overlapped tiling is discussed in these papers. Our work uses producer/consumer relations to represent dependence of kernel instances, and the transformation tool designed and implemented in this paper is fully automated and transparent to OpenCL programs. The most important difference between our work and existing work is the hierarchical overlapped tiling transformation proposed in this paper. The overhead of redundant computation is the main drawback of the overlapped tiling approach; by applying overlapped tiling hierarchically, we can decrease this overhead.

Bondhugula et al. design and implement Pluto [8], which can automatically transform loops for parallelism and locality based on the polyhedral model. Their transformation techniques includes only traditional tiling; overlaps between tiles are not considered. Tang et al. implemented the Pochoir compiler [25], which automates a trapezoidal decomposition for stencil code. However, their decomposition algorithm does not consider overlap, either.

Other publications that discuss how to take advantage of the hierarchy of the hardware include Liu et al. [19], that proposed a cache hierarchy-aware tile scheduling technique to maximize data reuse; and Leung et al. [18] that implement a C-to-CUDA compiler which performs hierarchical decomposition for multiple GPUs. The techniques presented in these papers are orthogonal to our proposed transformation. In fact, their techniques and ours could be combined.

7. CONCLUSION

In this paper, we propose a new transformation, hierarchical overlapped tiling. By creating hierarchical overlapping tiles, we reduce communication overhead among tiles while introducing smaller amount of redundant computation compared to plain overlapped tiling. We implemented the proposed transformations for OpenCL programs. Experimental results show that on average overlapped tiling and hierarchical overlapped tiling achieves 18% and 37% speedup over traditional tiling, respectively.

8. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Awards CNS 1111407 and CCF 0702260, and by the Illinois-Intel Parallelism Center at the University of Illinois at Urbana-Champaign. The Center is sponsored by the Intel Corporation.

9. REFERENCES

[1] W. Abu-Sufah, D. Kuck, and D. Lawrie. On the performance enhancement of paging systems through program analysis and transformations. *Computers, IEEE Transactions on*, C-30(5):341 –356, may 1981.

[2] N. Ahmed, N. Mateev, and K. Pingali. Tiling imperfectly-nested loop nests. In *Supercomputing, ACM/IEEE 2000 Conference*, page 31, nov. 2000.

[3] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kaufmann, 2001.

[4] M. Alpert. Not just Fun and Games. *Scientific American*, 4, 1999.

[5] W. F. Ames. *Numerical Methods for Partial Differential Equations.* Academic, San Diego, CA, sencond edition, 1977.

[6] R. Andonov, S. Balev, S. Rajopadhye, and N. Yanev. Optimal semi-oblique tiling. In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '01, pages 153–162, New York, NY, USA, 2001. ACM.

[7] W. Blume and R. Eigenmann. Symbolic range propagation. In *Proceedings. of 9th International Parallel Processing Symposium, 1995.*, pages 357 –363, apr 1995.

[8] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[9] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 279–290, New York, NY, USA, 1995. ACM.

[10] K. Högstedt, L. Carter, and J. Ferrante. Selecting tile shape for minimal execution time. In *Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures*, SPAA '99, pages 201–211, New York, NY, USA, 1999. ACM.

[11] W. Huang, M. R. Stan, K. Skadron, K. Sankaranarayanan, S. Ghosh, and S. Velusam. Compact thermal modeling for temperature-aware design. In *Proceedings of the 41st annual Design Automation Conference*, DAC '04, pages 878–883, New York, NY, USA, 2004. ACM.

[12] T. Johnson, S.-I. Lee, L. Fei, A. Basumallik, G. Upadhyaya, R. Eigenmann, and S. Midkiff. Experiences in using cetus for source-to-source transformations. In *Languages and Compilers for High Performance Computing*, volume 3602 of *Lecture Notes in Computer Science*, pages 922–922. 2005.

[13] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The omega library interface guide. Technical report, 1995.

[14] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008.

[15] G. Kreisel and J.-L. Krivine. *Elements of mathematical logic*. North-Holland Pub. Co., 1967.

[16] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 235–244, New York, NY, USA, 2007. ACM.

[17] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '81, pages 207–218, New York, NY, USA, 1981. ACM.

[18] A. Leung, N. Vasilache, B. Meister, M. Baskaran, D. Wohlford, C. Bastoul, and R. Lethin. A mapping path for multi-gpgpu accelerated computers from a portable high level programming abstraction. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 51–61, New York, NY, USA, 2010. ACM.

[19] J. Liu, Y. Zhang, W. Ding, and M. Kandemir. On-chip cache hierarchy-aware tile scheduling for multicore machines. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, pages 161 –170, april 2011.

[20] D. B. Loveman. Program improvement by source-to-source transformation. *Journal of the ACM*, 24:121–145, 1977.

[21] J. Meng and K. Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 256–265, New York, NY, USA, 2009. ACM.

[22] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for nonshared memory machines. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, pages 111–120, New York, NY, USA, 1991. ACM.

[23] M. Ripeanu, A. Iamnitchi, and I. Foster. Cactus application: Performance predictions in grid environments. In *Euro-Par 2001 Parallel Processing*, volume 2150 of *Lecture Notes in Computer Science*, pages 807–816, 2001.

[24] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, pages 215–228, New York, NY, USA, 1999. ACM.

[25] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The pochoir stencil compiler. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 117–128, New York, NY, USA, 2011. ACM.

[26] M. Wolf and M. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452 –471, oct 1991.

[27] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, pages 30–44, New York, NY, USA, 1991. ACM.

[28] M. Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, Philadelphia, PA, USA, 1989. Society for Industrial and Applied Mathematics.

[29] M. Wolfe. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, Supercomputing '89, pages 655–664, New York, NY, USA, 1989. ACM.

[30] D. Wonnacott. Time skewing for parallel computers. In *Languages and Compilers for Parallel Computing*, volume 1863, pages 477–480, 2000.

A Type and Effect System for Deterministic Parallel Java [*]

Robert L. Bocchino Jr. Vikram S. Adve Danny Dig
Sarita V. Adve Stephen Heumann Rakesh Komuravelli Jeffrey Overbey
Patrick Simmons Hyojin Sung Mohsen Vakilian

Department of Computer Science
University of Illinois at Urbana-Champaign
dpj@cs.uiuc.edu

Abstract

Today's shared-memory parallel programming models are complex and error-prone. While many parallel programs are intended to be deterministic, unanticipated thread interleavings can lead to subtle bugs and nondeterministic semantics. In this paper, we demonstrate that a practical *type and effect system* can simplify parallel programming by *guaranteeing deterministic semantics* with modular, compile-time type checking even in a rich, concurrent object-oriented language such as Java. We describe an object-oriented type and effect system that provides several new capabilities over previous systems for expressing deterministic parallel algorithms. We also describe a language called Deterministic Parallel Java (DPJ) that incorporates the new type system features, and we show that a core subset of DPJ is sound. We describe an experimental validation showing that DPJ can express a wide range of realistic parallel programs; that the new type system features are useful for such programs; and that the parallel programs exhibit good performance gains (coming close to or beating equivalent, nondeterministic multithreaded programs where those are available).

Categories and Subject Descriptors D.1.3 [*Software*]: Concurrent Programming—Parallel Programming; D.3.1 [*Software*]: Formal Definitions and Theory; D.3.2 [*Software*]: Language Classifications—Concurrent, distributed, and parallel languages; D.3.2 [*Software*]: Language Classifications—Object-oriented languages; D.3.3 [*Software*]: Language Constructs and Features—Concurrent Programming Structures

General Terms Languages, Verification, Performance

Keywords Determinism, deterministic parallelism, effects, effect systems, commutativity

* This work was supported by the National Science Foundation under grants CCF 07-02724 and CNS 07-20772, and by Intel, Microsoft and the University of Illinois through UPCRC Illinois.

1. Introduction

The advent of multicore processors demands parallel programming by mainstream programmers. The dominant model of concurrency today, multithreaded shared memory programming, is inherently complex due to the number of possible thread interleavings that can cause nondeterministic program behaviors. This nondeterminism causes subtle bugs: data races, atomicity violations, and deadlocks. The parallel programmer today prunes away the nondeterminism using constructs such as locks and semaphores, then *debugs* the program to eliminate the symptoms. This task is tedious, error prone, and extremely challenging even with good debugging tools.

The irony is that a vast number of computational algorithms (though not all) are in fact *deterministic*: a given input is always expected to produce the same output. Almost all scientific computing, encryption/decryption, sorting, compiler and program analysis, and processor simulation algorithms exhibit deterministic behavior. Today's parallel programming models force programmers to implement such algorithms in a nondeterministic notation and then convince themselves that the behavior will be deterministic.

By contrast, a *deterministic-by-default programming model* [9] can *guarantee* that any legal program produces the same externally visible results in all executions with a particular input *unless* nondeterministic behavior is explicitly requested by the programmer in disciplined ways. Such a model can make parallel application development and maintenance easier for several reasons. Programmers do not have to reason about notoriously subtle and difficult issues such as data races, deadlocks, and memory models. They can start with a sequential implementation and incrementally add parallelism, secure in the knowledge that the program behavior will remain unchanged. They can use familiar sequential tools for debugging and testing. Importantly, they can test an application only once for each input [19].

Unfortunately, while guaranteed determinism is available for some restricted styles of parallel programming (e.g., data parallel, or pure functional), it remains a challenging research problem to guarantee determinism for imperative, object-oriented languages such as Java, C++, and C#. In such languages, object references, aliasing, and updates to mutable state obscure the data dependences between parts of a program, making it hard to prove that those dependences are respected by the program's synchronization. This is a very important problem as many applications that need to be ported to multicore platforms are written in these languages.

We believe that a *type and effect system* [27, 26, 12, 30] is an important part of the solution to providing guaranteed deterministic semantics for imperative, object-oriented languages. A type and effect system (or effect system for short) allows the programmer to give names to distinct parts of the heap (we call them *regions*) and specify the kind of accesses to parts of the heap (e.g., *read or write effects*) in different parts of the program. The compiler can then check, using simple modular analysis, that all pairs of memory accesses either commute with each other (e.g., they are both reads, or they access disjoint parts of the heap) or are properly synchronized to ensure determinism. A robust type and effect system with minimal runtime checks is valuable because it enables checking at compile time rather than runtime, eliminates unnecessary runtime checks (thus leading to less overhead and/or less implementation complexity), and contributes to program understanding by showing *where* in the code parallelism is expressed – and where code must be rewritten to make parallelism available. Effect annotations can also provide an enforceable contract at interface boundaries, leading to greater modularity and composability of program components. An effect system can be supplemented with runtime speculation [23, 51, 38, 31, 50] or other runtime checks [43, 20, 47, 6] to enable greater expressivity.

In this paper, we develop a new type and effect system for expressing important patterns of deterministic parallelism in imperative, object-oriented programs. FX [33, 27] showed how to use regions and effects in limited ways for deterministic parallelism in a mostly functional language. Later work on object-oriented effects [26, 12, 30] and object ownership [16, 32, 14] introduced more sophisticated mechanisms for specifying effects. However, studying a wide range of *realistic* parallel algorithms has shown us that some significantly more powerful capabilities are needed for such algorithms. In particular, all of the existing work lacks general support for fundamental parallel patterns such as parallel updates on distinct fields of nested data structures, parallel array updates, in-place divide and conquer algorithms, and commutative parallel operations.

Our effect system can support all of the above capabilities, using several novel features. We introduce *region path lists*, or RPLs, which enable more flexible effect summaries, including effects on nested structures. RPLs also allow more flexible subtyping than previous work. We introduce an *index-parameterized array type* that allows references to provably distinct objects to be stored in an array *while still permitting arbitrary aliasing of the objects through references outside the array*. We are not aware of any statically checked type system that provides this capability. We define the notions of *subarrays* (i.e., one array that shares storage with another) and *partition operations*, that together enable in-place parallel divide and conquer operations on arrays. Subarrays and partitioning provide a natural object-oriented way to encode disjoint segments of arrays, in contrast to lower-level mechanisms like separation logic [35] that specify array index ranges directly. We also introduce an *invocation effect*, together with simple *commutativity annotations*, to permit the parallel invocation of operations that may actually interfere at the level of reads and writes, but still commute logically, i.e., produce the same final (logical) behavior. This mechanism supports concurrent data structures such as concurrent sets, hash maps, atomic counters, etc.

We have designed a language called *Deterministic Parallel Java* (DPJ) incorporating these features. DPJ is an extension to Java that enforces deterministic semantics via compile-time type checking. Because of the guaranteed deterministic semantics, existing Java code can be ported to DPJ *incrementally*. Furthermore, porting to DPJ will have minimal impact on program testing: developers can use the same tests and testing methodology for the ported parallel code as they had previously used for their sequential code.

The choice of Java for our work is not essential; similar extensions could be applied to other object-oriented languages, and we are currently developing a version of the language and compiler for C++. We are also exploring how to extend our type system and language to provide disciplined support for explicitly nondeterministic computations.

This paper makes the following contributions:

1. **Novel features.** We introduce a new region-based type and effect system with several novel features (RPLs, index-parameterized arrays, subarrays, and invocation effects) for expressing core parallel programming patterns in imperative languages. These features guarantee determinism at compile-time.
2. **Formal definition.** For a core subset of the type system, we have developed a formal definition of the static and dynamic semantics, and a detailed proof that our system allows sound static inference about noninterference of effects. We present an outline of the formal definition and proof in this paper. The full details are in an accompanying technical report [10] available via the Web [1].

3. **Language Definition.** We have designed a language called DPJ that incorporates the type and effect system into a modern O-O language (Java) in such a way that it supports the full flexibility of the sequential subset of Java, enables incremental porting of Java code to DPJ, and guarantees semantic equivalence between a DPJ program and its obvious sequential Java version. We have implemented a prototype compiler for DPJ that performs the necessary type checking and then maps parallelism down to the ForkJoinTask dynamic scheduling framework.

4. **Empirical evaluation.** We study six real-world parallel programs written in DPJ. This experience shows that DPJ can express a range of parallel programming patterns; that all the novel type system features are useful in real programs; and that the language is effective at achieving significant speedups on these codes on a commodity 24-core shared-memory processor. In fact, in 3 out of 6 codes, equivalent, manually parallelized versions written to use Java threads are available for comparison, and the DPJ versions come close to or beat the performance of the Java threads versions.

The rest of this paper proceeds as follows. Section 2 provides an overview of some basic features of DPJ, and Sections 3–5 explain the new features in the type system (RPLs, arrays, and commutativity annotations). Section 6 summarizes the formal results for a core subset of the language. Section 7 discusses our prototype implementation and evaluation of DPJ. Section 8 discusses related work, and Section 9 concludes.

```
 1 class TreeNode<region P> {
 2     region Links, L, R;
 3     double mass in P;
 4     TreeNode<L> left in Links;
 5     TreeNode<R> right in Links;
 6     void setMass(double mass) writes P { this.mass = mass; }
 7     void initTree(double mass) {
 8         cobegin {
 9             /* reads Links writes L */
10             left.mass = mass;
11             /* reads Links writes R */
12             right.mass = mass;
13         }
14     }
15 }
```

Figure 1. Basic features of DPJ. Type and effect annotations are italicized. Note that method `initTree` (line 7) has no effect annotation, so it gets the default effect summary of "reads and writes the entire heap."

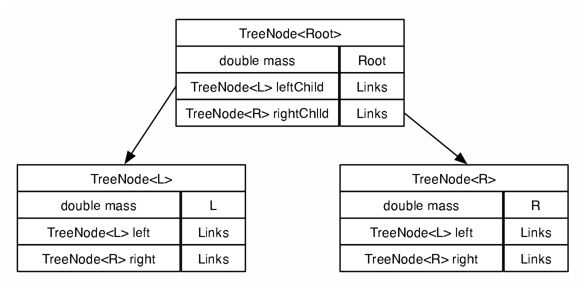


Figure 2. Runtime heap typing from Figure 1

2. Basic Capabilities

We begin by summarizing some basic capabilities of DPJ that are similar to previous work [33, 30, 26, 14, 15]. We refer to the example in Figure 1, which shows a simple binary tree with three nodes and a method `initTree` that writes into the `mass` fields of the left and right child nodes. As we describe more capabilities of DPJ, we will also expand upon this example to make it more realistic, e.g., supporting trees of arbitrary depth.

Region names. In DPJ, the programmer uses named regions to partition the heap, and writes method effect summaries stating what regions are read and written by each method. A *field region declaration* declares a new name r (called a *field region name*) that can be used as a region name. For example, line 2 declares names `Links`, `L`, and `R`, and these names are used as regions in lines 4 and 5.[1] A field region name is associated with the static class in which it is declared; this fact allows us to reason soundly about effects without alias restrictions or interprocedural alias analysis. A field region name functions like an ordinary class member: it is inherited by subclasses, and outside the scope of its defining class, it must be appropriately qualified (e.g., `TreeNode.L`). A *local region declaration* is similar and declares a region name at local scope.

Region parameters. DPJ provides class and method region parameters that operate similarly to Java generic parameters. We declare region parameters with the keyword `region`, as shown in line 1, so that we can distinguish them from Java generic type parameters (which DPJ fully supports). When a region-parameterized class or method is used, region arguments must be provided to the parameters, as shown in lines 4–5. Region parameters enable us to create multiple instances of the same class with their data in different regions.

Disjointness constraints. To control aliasing of region parameters, the programmer may write a disjointness constraint [14] of the form P_1 `#` P_2, where P_1 and P_2 are parameters (or regions written with parameters; see Section 3) that are required to be disjoint. Disjointness of regions is fully explained in Section 3; in the case of simple names, it means the names must be different. The constraints are checked when instantiating the class or calling the method. If the disjointness constraints are violated, the compiler issues a warning.

Partitioning the heap. The programmer may place the keyword `in` after a field declaration, followed by the region,

[1] As explained in Section 3, in general a DPJ region is represented as a *region path list* (RPL), which is a colon-separated list of elements such as `Root:L:L:R` that expresses the nested structure of regions. When a simple name r functions as a region, as shown in this section, it is short for `Root:`r.

as shown in lines 3–5. An operation on the field is treated as an operation on the region when specifying and checking effects. This effectively partitions the heap into regions. See Figure 2 for an illustration of the runtime heap typing, assuming the root node has been instantiated with `Root`.

Method effect summaries. Every method (including all constructors) must conservatively summarize its heap effects with an annotation of the form `reads` *region-list* `writes` *region-list*, as shown in line 6. Writes imply reads. When one method overrides another, the effects of the superclass method must contain the effects of the subclass method. For example, if a method specifies a `writes` effect, then all methods it overrides must specify that same `writes` effect. This constraint ensures that we can check effects soundly in the presence of polymorphic method invocation [30, 26]. The full DPJ language also includes *effect variables* [33], to support writing a subclass whose effects are unknown at the time of writing the superclass (e.g., in instantiating a library or framework class); however, we leave the discussion of effect variables to future work.

Effects on local variables need not be declared, because these effects are masked from the calling context. Nor must initialization effects inside a constructor body be declared, because the DPJ type and effect system ensures that no other task can access `this` until after the constructor returns. Read effects on `final` variables are also ignored, because those reads can never cause a conflict. A method or constructor with no externally visible heap effects may be declared `pure`.

To simplify programming and provide interoperability with legacy code, we adopt the rule that no annotation means "reads and writes the entire heap," as shown in Figure 1. This scheme allows ordinary sequential Java to work correctly, but it requires the programmer to add the annotations in order to introduce safe parallelism.

Expressing parallelism. DPJ provides two constructs for expressing parallelism, the `cobegin` block and the `foreach` loop. The `cobegin` block executes each statement in its body as a parallel task, as shown in lines 8–13. The `foreach` loop is used in conjunction with arrays and is described in Section 4.1.

Proving determinism. To type check the program in Figure 1, the compiler does the following. First, check that the summary `writes P` of method `setMass` (line 6) is correct (i.e., it covers all effect of the method). It is, because field `mass` is declared in region `P` (line 3), and there are no other effects. Second, check that the parallelism in lines 8–13 is safe. It is, because the effect of line 10 is `reads Links writes L`; the effect of line 12 is `reads Links writes R`; and `Links`, `L`, and `R` are distinct names. Notice that this analysis is entirely intraprocedural.

3. Region Path Lists (RPLs)

An important concept in effect systems is *region nesting*, which lets us partition the heap hierarchically so we can express that different computations are occurring on different parts of the heap. For example, to extend the code in Figure 1 to a tree of arbitrary depth, we need a tree of nested regions. As discussed in Section 4, we can also use nesting to express that two aggregate data structures (like arrays) are in distinct regions, and the components of those structures (like the cells of the arrays) are in distinct regions, each nested under the region containing the whole structure.

Effect systems that support nested regions are generally based on object ownership [16, 14] or use explicit declarations that one region is under another [30, 26]. As discussed below, we use a novel approach based on chains of elements called *region path lists*, or RPLs, that provides new capabilities for effect specification and subtyping.

3.1 Specifying Single Regions

The region path list (RPL) generalizes the notion of a simple region name r. Each RPL names a single *region*, or set of memory locations, on the heap. The set of all regions partitions the heap, i.e., each memory location lies in exactly one region. The regions are arranged in a tree with a special region `Root` as the root node. We say that one region is *nested under* (or simply under) another if the first is a descendant of the second in the tree. The tree structure guarantees that for any two distinct names r and r', the set of regions under r and the set of regions under r' have empty intersection, and we can use this guarantee to prove disjointness of memory accesses.

Syntactically, an RPL is a colon-separated list of names, called *RPL elements*, beginning with `Root`. Each element after `Root` is a declared region name r,[2] for example, `Root:A:B`. As a shorthand, we can omit the leading `Root`. In particular, a bare name can be used as an RPL, as illustrated in Figure 1. The syntax of the RPL represents the nesting of region names: one RPL is under another if the second is a prefix of the first. For example, `L:R` is under `L`. We write $R_1 \leq R_2$ if R_1 is under R_2.

We may also write a region parameter, instead of `Root`, at the head of an RPL, for example `P:A`, where `P` is a parameter. When a class with a region parameter is instantiated at runtime, the parameter is resolved to an RPL beginning with `Root`. Method region parameters are resolved similarly at method invocation time. Because a parameter `P` is always bound to the same RPL in a particular scope, we can make sound static inferences about parametric RPLs. For example, for all `P`, $\texttt{P:A} \leq \texttt{P}$, and $\texttt{P:A} \neq \texttt{P:B}$ if and only if $\texttt{A} \neq \texttt{B}$.

Figure 3 illustrates the use of region nesting and class region parameters to distinguish different fields as well as different objects. It extends the example from Figure 1 by

[2] As noted in Section 2, this can be a package- or class-qualified name such as `C.r`; for simplicity, we use r throughout.

```
 1 class TreeNode<region P> {
 2     region Links, L, R, M, F;
 3     double mass in P:M;
 4     double force in P:F;
 5     TreeNode<L> left in Links;
 6     TreeNode<R> right in Links;
 7     void initTree(double mass, double force) {
 8         cobegin {
 9             /* reads Links writes L:M */
10             left.mass = mass;
11             /* reads Links writes L:F */
12             left.force = force;
13             /* reads Links writes R:M */
14             right.mass = mass;
15             /* reads Links writes R:F */
16             right.force = force;
17         }
18     }
19 }
```

Figure 3. Extension of Figure 1 showing the use of region nesting and region parameters.

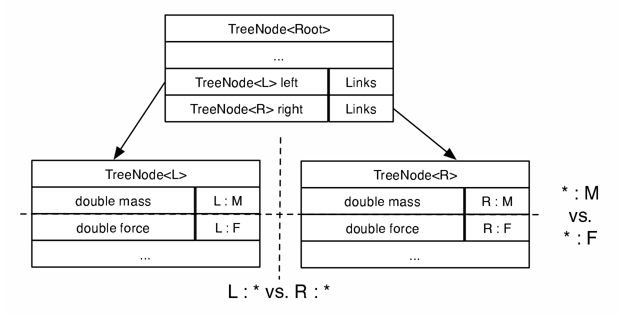


Figure 4. Graphical depiction of the distinctions shown in Figure 3. The * denotes any sequence of RPL elements; this notation is explained further in Section 3.2.

adding a `force` field to the `TreeNode` class, and by making the `initTree` method (line 7) set the `mass` and `force` fields of the left and right child in four parallel statements in a `cobegin` block (lines 9–16).

To establish that the parallelism is safe (i.e., that lines 9–16 access disjoint locations), we place fields `mass` and `force` in distinct regions `P:M` and `P:F`, and the links `left` and `right` in a separate region `Links` (since they are only read). The parameter `P` appears in both regions and `P` is bound to different regions (`L` and `R`) for the left and right subtrees, because of the different instantiations of the parametric type `TreeNode` for the fields `left` and `right`. Because the names `L` and `R` used in the types are distinct, we can distinguish the effects on `left` (lines 10–12) from the effects on `right` (lines 14–16). And because the names `M` and `F` are distinct, we can distinguish the effects on the different fields within an object i.e., lines 10 vs. 14 and lines 12 vs. 16, from each other. Figure 4 shows this situation graphically.

3.2 Specifying Sets of Regions

Partially specified RPLs. To express recursive parallel algorithms, we must specify effects on *sets of regions* (e.g., "all regions under R"). To do this, we introduce *partially specified RPLs*. A partially specified RPL contains the symbol *

```
 1 class TreeNode<region P> {
 2     region Links, L, R, M, F;
 3     double mass in P:M;
 4     double force in P:F;
 5     TreeNode<P:L> left in Links;
 6     TreeNode<P:R> right in Links;
 7     TreeNode<*> link in Links;
 8     void computeForces() reads Links, *:M writes P:*:F {
 9         cobegin {
10             /* reads *:M writes P:F */
11             this.force = (this.mass * link.mass) * R_GRAV;
12             /* reads Links, *:M writes P:L:*:F */
13             if (left != null) left.computeForces();
14             /* reads Links, *:M writes P:R:*:F */
15             if (right != null) right.computeForces();
16         }
17     }
18 }
```

Figure 5. Recursive computation showing the use of partially specified RPLs for effects and subtyping.

("star") as an RPL element, standing in for some unknown sequence of names. An RPL that contains no * is *fully specified*.

For example, consider the code shown in Figure 5. Here we are operating on the same `TreeNode` shown in Figs. 1 and 3, except that we have added (1) a `link` field (line 7) that points to some other node in the tree and (2) a `computeForces` method (line 8) that recursively descends the tree. At each node, `computeForces` follows `link` to another node, reads the `mass` field of that node, computes the force between that node and this one, and stores the result in the `force` field of this node. This computation can safely be done in parallel on the subtrees at each level, because each call writes only the `force` field of `this`, and the operations on other nodes (through `link`) are all reads of the `mass`, which is distinct from `force`. To write this computation, we need to be able to say, for example, that line 13 writes only the left subtree, and does not touch the right subtree.

Distinctions from the left. In lines 11–15 of Figure 5, we need to distinguish the write to `this.force` (line 11) from the writes to the `force` fields in the subtrees (lines 13 and 15). We can use partially specified RPLs to do this. For example, line 8 says that `computeForces` may read all regions under `Links` and write all regions under `P` that end with `F`.

If RPLs R_1 and R_2 are the same in the first n places, they differ in place $n+1$, and neither contains a * in the first $n+1$ places, then (because the regions form a tree) the set of regions under R_1 and the set of regions under R_2 have empty intersection. In this case we say that R_1`:*` and R_2`:*` are *disjoint*, and we know that effects on these two RPLs are noninterfering. We call this a "distinction from the left," because we are using the distinctness of the names to the left of any star to infer that the region sets are non-intersecting. For example, a distinction from the left establishes that the region sets `P:F`, `P:L:*:F`, and `P:R:*:F` (shown in lines 10-15) are disjoint, because the RPLs all start with `P` and differ in the second place.

Distinctions from the right. Sometimes it is important to specify "all fields x in any node of a tree." For example, in lines 10–15, we need to show that the reads of the `mass` fields are distinct from the writes to the `force` fields. We can make this kind of distinction by using different names *after* the star: if R_1 and R_2 differ in the nth place from the right, and neither contains a `*` in the first n places from the right, then a simple syntactic argument shows that their region sets are disjoint. We call this pattern a "distinction from the right," because the names that ensure distinctness appear to the right of any star. For example, in lines 10–15, we can distinguish the reads of `*:M` from the writes to `P:L:*:F` and `P:R:*:F`.

More complicated patterns. More complicated RPL patterns like `Root:*:A:*:B` are supported by the type system. Although we do not expect that programmers will need to write such patterns, they sometimes arise via parameter substitution when the compiler is checking effects.

3.3 Subtyping and Type Casts

Subtyping. Partially specified RPLs are also useful for subtyping. For example, in Figure 5, we needed to write the type of a reference that could point to a `TreeNode<P>`, for any binding to `P`. With fully specified RPLs we cannot do this, because we cannot write a type to which we can assign both `TreeNode<L>` and `TreeNode<R>`. The solution is to use a partially specified RPL in the type, e.g., `TreeNode<*>`, as shown in line 7 of Figure 5. Now we have a type that is flexible enough to allow the assignment, but retains soundness by explicitly saying that we do not know the actual region.

The subtyping rule is simple: C<R_1> is a subtype of C<R_2> if the set of regions denoted by R_1 is included in the set of regions denoted by R_2. We write $R \subseteq R_2$ to denote set inclusion for the corresponding sets of regions. If R_1 and R_2 are fully specified, then $R_1 \subseteq R_2$ implies $R = R_2$. Note that nesting and inclusion are related: $R_1 \leq R_2$ implies $R_1 \subseteq R_2$`:*`. However, nesting alone does *not* imply inclusion of the corresponding sets. For example, `A:B` $\leq$ `A`, but `A:B` $\not\subseteq$ `A`, because `A:B` and `A` denote distinct regions. In Section 6 we discuss the rules for nesting, inclusion, and disjointness of RPLs more formally.

Figure 6 illustrates one possible heap typing resulting from the code in Figure 5. The DPJ typing discipline ensures the object graph restricted to the `left` and `right` references is a tree. However, the full object graph including the `link` references is more general and can even include cycles, as illustrated in Figure 6. Note how our effect system is able to prove that the updates to different subtrees are distinct, even though (1) non-tree edges exist in the graph; and (2) those edges are followed to do possibly overlapping reads.

Type casts. DPJ allows any type cast that would be legal for the types obtained by erasing the region variables. This approach is sound if the region arguments are consistent. For example, given `class B<region R> extends A<R>`, a cast from `A<r>` to `B<r>` is sound, because either the reference is `B<r>`, or it is not any sort of `B`, which will cause

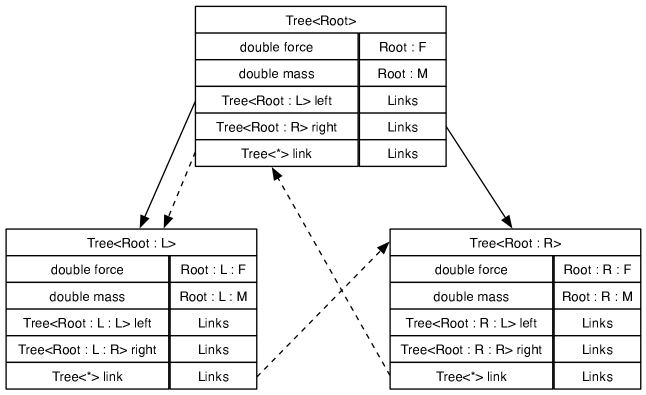


Figure 6. Heap typing from Figure 5. Reference values are shown by arrows; tree arrows are solid, and non-tree arrows are dashed. Notice that all arrows obey the subtyping rules.

a `ClassCastException` at runtime. However, a cast from `Object` to `B<r1>` is unsound and could violate the determinism guarantee, because the `Object` could be a `B<r2>`, which would not cause a runtime exception. The compiler allows this cast, but it issues a warning.

4. Arrays

DPJ provides two novel capabilities for computing with arrays: *index-parameterized arrays* and *subarrays*. Index-parameterized arrays allow us to traverse an array of object references and safely update the objects in parallel, while subarrays allow us to dynamically partition an array into disjoint pieces, and give each piece to a parallel subtask.

4.1 Index-Parameterized Arrays

A basic capability of any language for deterministic parallelism is to operate on elements of an array in parallel. For a loop over an array of values, it is sufficient to prove that each iteration accesses a distinct array element (we call this a *unique traversal*). For a loop over an array of references to mutable objects, however, a unique traversal is not enough: we must also prove that any memory locations updated by following references in distinct array cells (possibly through a chain of references) are distinct. Proving this property is very hard in general, if assignments are allowed into reference cells of arrays. No previous effect system that we are aware of is able to ensure disjointness of updates by following references stored in arrays, and this seriously limits the ability of those systems to express parallel algorithms.

In DPJ, we make use of the following insight:

Insight 1. *We can define a special array type with the restriction that an object reference value o assigned to cell n (where n is a natural number constant) of such an array has a runtime type that is parameterized by n. If accesses through cell n touch only region n (even by following a chain*

```
1  class Body<region P> {
2      region Link, M, F;
3      double mass in P:M;
4      double force in P:F;
5      Body<*> link in Link;
6      void computeForce() reads Link, *:M writes P:F {
7          force = (mass * link.mass) * R_GRAV;
8      }
9  }
10
11 final Body<[_]>[]<[_]> bodies = new Body<[_]>[N]<[_]>;
12 foreach (int i in 0, N) {
13     /* writes [i] */
14     bodies[i] = new Body<[i]>();
15 }
16 foreach (int i in 0, N) {
17     /* reads [i], Link, *:M writes [i]:F */
18     bodies[i].computeForce();
19 }
```

Figure 7. Example using an index-parameterized array.

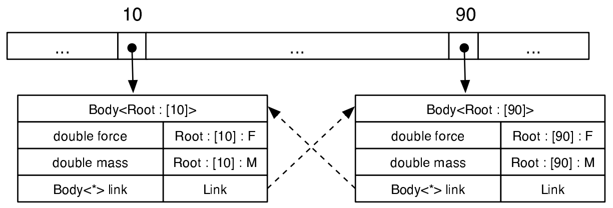


Figure 8. Heap typing from Figure 7. The type of array cell i is parameterized by i. Cross-links are possible, but if any links are followed to access other array cells, the effects are visible.

of references), then the accesses through different cells are guaranteed to be disjoint.

We call such an array type an *index-parameterized array*. To represent such arrays, we introduce two language constructs:

1. An *array RPL element* written `[`e`]`, where e is an integer expression.
2. An *index-parameterized array type* that allows us to write the region and type of array cell e using the array RPL element `[`e`]`. For example, we can specify that cell e resides in region `Root:[`e`]` and has type `C<Root:[`e`]>`.

At runtime, if e evaluates to a natural number n, then the static array RPL element `[`e`]` evaluates to the *dynamic array RPL element* `[`n`]`.

The key point here is that we can distinguish `C<[`e_1`]>` from `C<[`e_2`]>` if e_1 and e_2 always evaluate to unequal values at runtime, just as we can distinguish `C<`r_1`>` from `C<`r_2`>`, where r_1 and r_2 are declared names, as discussed in Section 3.1. Obviously, the compiler's capability to distinguish such types will be determined by its ability to prove the inequality of the symbolic expressions e_1 and e_2. This is possible in many common cases, for the same reason that array dependence analysis is effective in many, though not all, cases [24]. The key benefit is that *the type checker has then proved the uniqueness of the target objects, which would not follow from dependence analysis alone*.

In DPJ, the notation we use for index-parameterized arrays is T`[]<`R`>#`i, where T is a type, R is an RPL, `#`i declares a fresh integer variable i in scope over the type, and `[`i`]` may appear as an array RPL element in T or R (or both). This notation says that array cell e (where e is an integer expression) has type $T[i \leftarrow e]$ and is located in region $R[i \leftarrow e]$. For example, `C<r1:[i]>[]<r2:[i]>#i` specifies an array such that cell e has type `C<r1:[`e`]>` and resides in region `r2:[`e`]`. If T itself is an array type, then nested index variable declarations can appear in the type. However, the most common case is a single-dimensional array, which needs only one declaration. For that case, we provide a simplified notation: the user may omit the `#`i and use an underscore (`_`) as an implicitly declared variable. For example, `C<[_]>[]<[_]>` is equivalent to `C<[i]>[]<[i]>#i`.

Figure 7 shows an example, which is similar in spirit to the Barnes-Hut force computation discussed in Section 7. Lines 1–9 declare a class `Body`. Line 11 declares and creates an index-parameterized array `bodies` with `N` cells, such that cell i resides in region `[`i`]` and points to an object of type `Body<[`i`]>`. Figure 8 shows a sample heap typing, for some particular value n of `N`.

Lines 12–15 show a `foreach` loop that traverses the indices $i \in [0, n-1]$ in parallel and initializes cell i with a new object of type `Body<[`i`]>`. The loop is noninterfering because the type of `bodies` says that cell `bodies[`i`]` resides in region `[`i`]`, so distinct iterations i and j write disjoint regions `[`i`]` and `[`j`]`. Lines 16–19 are similar, except that the loop calls `computeForce` on each of the objects. In iteration i of this loop, the effect of line 16 is `reads [`i`]`, because it reads `bodies[`i`]`, together with `reads Link, *:M writes [`i`]:F`, which is the declared effect of method `computeForce` (line 6), after substituting `[`i`]` for `P`. Again, the effects are noninterfering for $i \neq j$.

To maintain soundness, we just need to enforce the invariant that, at runtime, cell `A[`i`]` never points to an object of type `C<[`j`]>`, if $i \neq j$. The compiler can enforce this invariant through symbolic analysis, by requiring that if type `C<[`e_1`]>` is assigned to type `C<[`e_2`]>`, then e_1 and e_2 must always evaluate to the same value at runtime; if it cannot prove this fact, then it must conservatively disallow the assignment. In many cases (as in the example above) the check is straightforward.

Note that because of the typing rules, no two distinct cells of an index-parameterized array can point to the same object. However, it is perfectly legal to reach the same object by following chains of references from distinct array cells, as shown in Figure 8. In that case, in a parallel traversal over the array, either the common object is not updated, in which case the parallelism is safe; or a write effect on the same region appears in two distinct iterations of a parallel loop, in which case the compiler can catch the error.

Note also that while no two cells in an index-parameterized array can alias, references may be freely shared with other

```
 1 class QSort<region P> {
 2     DPJArrayInt<P> A in P;
 3     QSort(DPJArray<P> A) pure { this.A = A; }
 4     void sort() writes P:* {
 5         if (A.length <= SEQ_LENGTH) {
 6             seqSort();
 7         } else {
 8             /* Shuffle A and return pivot index */
 9             int p = partition(A);
10             /* Divide A into two disjoint subarrays at p */
11             final DPJPartitionInt<P> segs =
12                 new DPJPartitionInt<P>(A, p, OPEN);
13             cobegin {
14                 /* writes segs:[0]:* */
15                 new QSort<segs:[0]:*>(segs.get(0)).sort();
16                 /* writes segs:[1]:* */
17                 new QSort<segs:[1]:*>(segs.get(1)).sort();
18             }
19         }
20     }
21 }
```

Figure 9. Writing quicksort with the partition operation. `DPJArrayInt` and `DPJPartitionInt` are specializations to `int` values. In line 12, the argument `OPEN` indicates that we are omitting the partition index from the subarrays, i.e., they are open intervals.

variables (including cells in other index-parameterized arrays), unlike linear types [26, 12, 13]. For example, if cell i of a particular array has type `C<[`i`]>`, the object it points to could be referred to by cell i of any number of other arrays (with the same type), or by any reference of type `C<*>`. Thus, when we are traversing the array, we get the benefit of the alias restriction imposed by the typing, but we can still have as many other outstanding references to the objects as we like.

The pattern does have some limitations: for example, we cannot move an element from position i to position j in the array C`<[i]>[]#i`. However, we can copy the references into a different array C`<*>[]` and shuffle those references as much as we like, though we cannot use those references to update the objects in parallel. We can also make a new copy of element i with type C`<[`j`]>` and store the new copy into position j. This effectively gives a kind of reshuffling, although the copying adds performance overhead. Another limitation is that our `foreach` currently only allows regular array traversals (including strided traversals), though it could be extended to other unique traversals.

4.2 Subarrays

A familiar pattern for writing divide and conquer recursion is to partition an array into two or more disjoint pieces and give each array to a subtask. For example, Figure 9 shows a standard implementation of quicksort, which divides the array in two at each recursive step, then works in parallel on the halves. DPJ supports this pattern with three novel features, which we illustrate with the quicksort example.

First, DPJ provides a class `DPJArray` that wraps an ordinary Java array and provides a view into a contiguous segment of it, parameterized by start position S and length L. In Figure 9, the `QSort` constructor (line 3) takes a `DPJArray` object that represents a contiguous subrange of the caller's array. We call this subrange a *subarray*. Notice that the `DPJArray` object does *not* replicate the underlying array; it stores only a reference to the underlying array, and the values of S and L. The `DPJArray` object translates access to element i into access to element $S + i$ of the underlying array. If $i < 0$ or $i \geq L$, then an array bounds exception is thrown, i.e., access through the subarray must stay within the specified segment of the original array.

Second, DPJ provides a class `DPJPartition`, representing an indexed collection of `DPJArray` objects, all of which point into mutually disjoint segments of the original array. To create a `DPJPartition`, the programmer passes a `DPJArray` object into the `DPJPartition` constructor, along with some arguments that say how to do the splitting. Lines 11–12 of Figure 9 show how to split the `DPJArray A` at index `p`, and indicate that position `p` is to be left out of the resulting disjoint segments. The programmer can access segment i of the partition `segs` by saying `segs.get(`i`)`, as shown in lines 15 and 17.

Third, to support recursive computations, we need a slight extension to the syntax of RPLs (Section 3). Notice that we cannot use a simple region name, like r, for the type of a partition segment, because different partitions can divide the same array in different ways. Instead, we allow a `final` local variable z (including `this`) of class type to appear at the head of an RPL, for example z`:r`. The variable z stands in for the object reference o stored into the variable at runtime, which is the actual region. Using the object reference as a region insures that different partitions get different regions, and making the variable `final` ensures that it always refers to the same region.

We make these "z regions" into a tree as follows. If z's type is `C<`$R, \ldots$`>`, then z is nested under R; the first region parameter of a class functions like the *owner parameter* in an object ownership system [18, 16]. In the particular case of `DPJPartition`, if the type of z is `DPJPartition<`R`>`, then the type of z`.get(`i`)` is z`:[`i`]:*`, where $z \leq R$. Internally, the `get` method uses a type cast to generate a `DPJArray` of type `this:[`i`]:*` that points into the underlying array. The type cast is not sound within the type system, but it is hidden from the user code in such a way that all well-typed uses of `DPJPartition` are noninterfering.

In Figure 9, the sequence of recursive `sort` calls creates a tree of `QSort` objects, each in its own region. The `cobegin` in lines 13–17 is safe because `DPJPartition` guarantees that the segments `segs.get(0)` and `segs.get(1)` passed into the recursive parallel `sort` calls are disjoint. In the user code, the compiler uses the type and effect annotations to prove noninterference as follows. First, from the type of `QSort` and the declared effect of `sort` (line 4), the compiler determines that the effects of lines 15 and 17 are `writes segs:[0]:*` and `writes segs:[1]:*`, as shown. Second, the regions `segs:[0]:*` and `segs:[1]:*` are disjoint, by a distinction from the left (Section 3.2). Finally, the effect `writes P:*` in line 4 correctly summarizes the effects of `sort`, because lines 6 and 9 write `P`, lines 15 and 17 write under `segs`, and `segs` is under `P`, as explained above.

Notice that `DPJPartition` can create multiple references to overlapping data with different regions in the types. Thus, there is potential for unsoundness here if we are not careful. To make this work, we must do two things. First, if z_1 and z_2 represent different partitions of the same array, then z_1`.get(0)` and z_2`.get(1)` could overlap. Therefore, we must not treat them as disjoint. This is why we put `*` at the end of the type z`:[`i`]:*` of z`.get(`i`)`; otherwise we could incorrectly distinguish z_1`:[0]` from z_2`:[1]`, using a distinction from the right. Second, if z has type `DPJPartition<`R`>`, then z`.get(`i`)` has type `DPJArray<`z`:[`i`]:*>` and points into a `DPJArray<`R`>`. Therefore, we must not treat z`:[`i`]:*` as disjoint from R. Here, we simply do not include this distinction in our type system. All we say is that z`:[`i`]:*` $\leq R$. See Section 6.3 and Appendix C.2 for further discussion of the disjointness rules in our type system.

```
1  class IntSet<region P> {
2      void add(int x) writes P { ... }
3      add commuteswith add;
4  }
```

(a) Declaration of `IntSet` class with commutative method `add`

```
1  IntSet<R> set = new IntSet<R>();
2  foreach (int i in 0, N)
3      /* invokes IntSet.add with writes R */
4      set.add(A[i]);
```

(b) Using `commuteswith` for parallelism

```
1  class Adder<region P> {
2      void add(IntSet<P> set, int i)
3        invokes IntSet.add with writes P {
4          set.add(i);
5      }
6  }
7  IntSet<R> set = new IntSet<R>();
8  Adder<R> adder = new Adder<R>();
9  foreach (int i in 0, N)
10     /* invokes IntSet.add with writes R */
11     adder.add(set, A[i]);
```

(c) Using `invokes` to summarize effects

Figure 10. Illustration of `commuteswith` and `invokes`.

5. Commutativity Annotations

Sometimes to express parallelism we need to look at interference in terms of higher-level operations than read and write [29]. For example, insertions into a concurrent `Set` can go in parallel and preserve determinism even though the order of interfering reads and writes inside the `Set` implementation is nondeterministic. Another such example is computing connected components of a graph in parallel.

In DPJ, we address this problem by adding two features. First, classes may contain declarations of the form m `commuteswith` m', where m and m' are method names, indicating that any pair of invocations of the named methods may be safely done in parallel, *regardless of the read and write effects of the methods*. See Figure 10(a). In effect, the `commuteswith` annotation says that (1) the two invocations are *atomic* with respect to each other, i.e., the result will be as if one occurred and then the other; and (2) either order of invocation produces the same result.

The commutativity property itself is not checked by the compiler; we must rely on other forms of checking (e.g., more complex program logic [52] or static analysis [42, 4]) to ensure that methods declared to be commutative are really commutative. In practice, we anticipate that `commuteswith` will be used mostly by library and framework code that is written by experienced programmers and extensively tested. Our effect system does guarantee deterministic results for an application using a commutative operation, assuming that the operation declared commutative is indeed commutative.

Second, our effect system provides a novel *invocation effect* of the form `invokes` m `with` E. This effect records that an invocation of method m occurred with underlying effects E. The type system needs this information to represent and check effects soundly in the presence of commutativity annotations: for example, in line 4 of Fig. 10(b), the compiler needs to record that `add` was invoked there (so it can disregard the effects of other `add` invocations) *and* that the underlying effect of the method was `writes R` (so it can verify that there are no other interfering effects, e.g., reads or writes of `R`, in the invoking code).

When there are one or more intervening method calls between a `foreach` loop and a commutative operation, it may also be necessary for a method effect summary in the *program text* to specify that an invocation occurred inside the method. For example, in Figure 10(c), the `add` method is called through a wrapper object. We could have correctly specified the effect of `Adder.add` as `writes P`, but this would hide from the compiler the fact that `Adder.add` commutes with itself. Of course we could use `commuteswith` for `Adder.add`, but this is highly unsatisfactory: it just propagates the unchecked commutativity annotation out through the call chain in the application code. The solution is to specify the invocation effect `invokes IntSet.add with writes P`, as shown.

Notice that the programmer-specified invocation effect exposes an internal implementation detail (i.e., that a particular method was invoked) at a method interface. However, we believe that such exposure will be rare. In most cases, the effect `invokes` $C.m$ `with` E will be conservatively summarized as E (Section 6.1 gives the formal rules for covering effects). The invocation effect will *only* be used for cases where a commutative method is invoked, and the commutativity information needs to be exposed to the caller. We believe these cases will generally be confined to high-level public API methods, such as `Set.add` in the example given in Figure 10.

Meaning	Symbol	Definition
Programs	*program*	*region** *class** e
Regions	*region*	$\texttt{region}\ r$
Classes	*class*	$\texttt{class}\ C\texttt{<}P\texttt{>}\ \{$ *field** *method** *comm** $\}$
RPLs	R	$\texttt{Root} \mid P \mid z \mid R:r \mid R:[i] \mid R:*$
Fields	*field*	$T\ f\ \texttt{in}\ R_f$
Types	T	$C\texttt{<}R\texttt{>} \mid T[\,]\texttt{<}R\texttt{>}\#i$
Methods	*method*	$T\ m(T\ x)\ E\ \{\ e\ \}$
Effects	E	$\emptyset \mid \texttt{reads}\ R \mid \texttt{writes}\ R \mid$ $\texttt{invokes}\ C.m\ \texttt{with}\ E \mid E \cup E$
Expressions	e	$\texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{this}.f = z \mid \texttt{this}.f \mid$ $z[n] = z \mid z[n] \mid z.m(z) \mid z \mid \texttt{new}\ C\texttt{<}R\texttt{>} \mid$ $\texttt{new}\ T[n]\texttt{<}R\texttt{>}\#i$
Variables	z	$\texttt{this} \mid x$
Commutativity	*comm*	$m\ \texttt{commuteswith}\ m$

Figure 11. Core DPJ syntax. C, P, f, m, x, r, and i are identifiers, and n is a natural number. R_f denotes a fully specified RPL (i.e., containing no $*$).

6. The Core DPJ Type System

We have formalized a subset of DPJ, called *Core DPJ*. To make the presentation more tractable and to focus attention on the important aspects of the language, we make the following simplifications:

1. We present a simple expression-based language, omitting more complicated aspects of the real language such as statements and control flow.
2. Our language has classes and objects, but no inheritance.
3. Region names r are declared at global scope, instead of at class scope. Every class has one region parameter, and every method has one formal parameter.
4. To avoid dealing with integer variables and expressions, we require that array indices are natural number literals.

Removing the first simplification adds complexity but raises no significant technical issues. Adding inheritance raises standard issues for formalizing an object-oriented language. We omit those here in order to focus on the novel aspects of our system, but we describe them in [10]. Removing simplifications 3 and 4 is mostly a matter of bookkeeping. To handle arrays in the full language, we need to prove equivalence and non-equivalence of array index expressions, but this is a standard compiler capability.

We have chosen to make Core DPJ a sequential language, in order to focus on our mechanisms for expressing effects and noninterference. In Section 6.4, we discuss how to extend the formalism to model the `cobegin` and `foreach` constructs of DPJ.

6.1 Syntax and Static Semantics

Figure 11 defines the syntax of Core DPJ. The syntax consists of the key elements described in the previous sections (RPLs, effects, and commutativity annotations) hung upon a toy language that is sufficient to illustrate the features yet reasonable to formalize. A program consists of a number of region declarations, a number of class declarations, and an expression to evaluate. Class definitions are similar to Java's, with the restrictions noted above.

(a) Programs

$\triangleright$ *program*	Valid program	$\triangleright$ *class*	Valid class definition
$\triangleright\Gamma$	Valid environment	$\Gamma \triangleright$ *field*	Valid field
$\Gamma \triangleright$ *method*	Valid method	$\Gamma \triangleright$ *comm*	Valid commutativity annotation

(b) RPLs

$\Gamma \triangleright R$	Valid RPL	$\Gamma \triangleright R \leq R'$	R under R'
$\Gamma \triangleright R \subseteq R'$	R included in R'		

(c) Types

$\Gamma \triangleright T$	Valid type	$\Gamma \triangleright T \leq T'$	T a subtype of T'

(d) Effects

$\Gamma \triangleright E$	Valid effect	$\Gamma \triangleright E \subseteq E'$	E a subeffect of E'

(e) Expressions

$\Gamma \triangleright e : T, E$	e has type T and effect E in Γ

Figure 12. Core DPJ type judgments. We extend the judgments to groups of things (e.g., $\Gamma \triangleright$ *field**) in the obvious way.

We define the static semantics of Core DPJ with the judgments stated in Figure 12. The judgments are defined with respect to an environment Γ, where each element of Γ is one of the following:

- A binding $z \mapsto T$ stating that variable z has type T. These elements come into scope when a new variable (`let` variable or formal parameter) is introduced.
- A constraint $P \subseteq R$ stating that region parameter P is in scope and included in region R. These elements come into scope when we capture the type of a variable used for an invocation (see the discussion of expression typing judgments below).
- An integer variable i. These elements come into scope when we are evaluating an array type or new array expression.

The formal rules for making the judgments are stated in full in Appendix A. Below we briefly discuss each of the five groups of judgments.

Programs. These judgments state that a program and its top-level components (classes, methods, etc.) are valid. Most rules just require that the component's components are valid in the surrounding environment. The rule for valid method definitions (METHOD) requires that the method body's type and effect are a subtype and subeffect of the return type and declared effect. These constraints ensure that we can use the method declaration to reason soundly about a method's return type and effect when we are typing method invocation expressions.

RPLs. These judgments define validity, nesting, and inclusion of RPLs. Most rules are a straightforward formal translation of the relations that we described informally in Section 3.2. The key rule states that if R is under R' in some environment, then R is included in R'`:*` in that environment:

$$\text{(INCLUDE-STAR)} \quad \frac{\Gamma \triangleright R \leq R'}{\Gamma \triangleright R \subseteq R' : *}$$

Types. These define when one type is a subtype of another. The class subtyping rule is just the formal statement of the rule we described informally in Section 3.3:

$$\text{(SUBTYPE-CLASS)} \quad \frac{\Gamma \triangleright R \subseteq R'}{\Gamma \triangleright C\texttt{<}R\texttt{>} \leq C\texttt{<}R'\texttt{>}}$$

The array subtyping rule is similar:

$$\text{(SUBTYPE-ARRAY)} \quad \frac{\Gamma \cup \{i\} \triangleright R \subseteq R'[i' \leftarrow i] \quad T \equiv T'}{\Gamma \triangleright T[]\texttt{<}R\texttt{>}\#i \leq T'[]\texttt{<}R'\texttt{>}\#i'}$$

Here $\equiv$ means identity of element types up to the names of integer variables i. More flexible element subtyping is not possible without sacrificing soundness. We could allow unsound assignments and check for them at runtime (as Java does for class subtyping of array elements), but this would require that we retain the class region binding information at runtime.

Effects. These judgments define when an effect is valid, and when one effect is a subeffect of another. Intuitively, "E is a subeffect of E'" means that E' contains all the effects of E, i.e., we can use E' as a (possibly conservative) summary of E. The rules for reads, writes, and effect unions are standard [16, 33], but there are two new rules for invocation effects. First, if E' covers E, then an invocation of some method with E' covers an invocation of the same method with E:

$$\text{(SE-INVOKES-1)} \quad \frac{\Gamma \triangleright E \subseteq E'}{\Gamma \triangleright \texttt{invokes}\ C.m\ \texttt{with}\ E \subseteq \texttt{invokes}\ C.m\ \texttt{with}\ E'}$$

Second, we can conservatively summarize the effect `invokes` $C.m$ `with` E as just E:

$$\text{(SE-INVOKES-2)} \quad \frac{}{\Gamma \triangleright \texttt{invokes}\ C.m\ \texttt{with}\ E \subseteq E}$$

Expressions. These judgments tell us how to compute the type and effect of an expression. They also ensure that the types of component expressions (for example at assignments and method parameter bindings) match in a way that guarantees soundness. The rules for field and array access and assignment, variable lookup, and new classes and arrays are straightforward. In the rule for `let` $x = e$ `in` e', we type e, bind x to the type of e, and type e'. If x appears in the type or effect of e', we replace it with R`:*` to generate a type and effect for the whole expression that is valid in the outer scope.

In the rule for method invocation (INVOKE), we translate the type T_x of the method formal parameter to the current context by creating a fresh region parameter P included in the region R of z's type. This technique is similar to how Java handles the capture of a generic wildcard. Note that simply substituting R for param(C) in translating T_x would not be sound; see [10] for an explanation and an example.

Meaning	Symbol	Definition
RPLs	dR	`Root` $\mid o \mid dR : r \mid dR : [i] \mid dR : [n] \mid dR : *$
Types	dT	$C\texttt{<}dR\texttt{>}$
Effects	dE	$\emptyset \mid$ `reads` $dR \mid$ `writes` $dR \mid$ `invokes` $C.m$ `with` $dE \mid dE \cup dE$

Figure 13. Dynamic syntax of Core DPJ. dR_f denotes a fully-specified dynamic RPL (i.e., containing no $*$).

We also check that the actual argument type is a subtype of the declared formal parameter type, and we report the invocation of the method with its declared effect.

6.2 Dynamic Semantics

The syntax for entities appearing in the dynamic semantics is shown in Figure 13. At runtime, we have dynamic regions (dR), dynamic types (dT) and dynamic effects (dE), corresponding to static regions (R), types (T) and effects (E) respectively. Dynamic regions and effects are not recorded in a real execution, but here we thread them through the execution state so we can formulate and prove soundness results [16]. We also have object references o, which are the actual values computed during the execution.

The dynamic execution state consists of (1) a heap H, which is a function taking values to objects; and (2) a dynamic environment $d\Gamma$, which is a set of elements of the form $z \mapsto o$ (variable z is bound to value o) or $P \mapsto dR$ (region parameter P is bound to region dR). $d\Gamma$ defines a natural substitution on RPLs, where we replace the variables with values and the region parameters with regions as specified in the environment. We denote this substitution on RPL R as $d\Gamma(R)$, and we extend this notation to types and effects in the obvious way. Notice that we get the syntax of Figure 13 by applying the substitution $d\Gamma$ to the syntax of Figure 11.

An object is a partial function taking field names to object references. If the function is undefined on all field names, then we say it is a *null object*. We use null objects because we need to track the actual types of null references to establish soundness. Since the actual implementation does not need to do this tracking, it can just use the single value `null`. Every object reference $o \in \text{Dom}(H)$ has a type, determined when the object is created, and we write $H \triangleright o : dT$ to mean that the reference o has type dT with respect to heap H.

We write the evaluation rules in large-step semantics notation, using the following evaluation function:

$$(e, d\Gamma, H) \rightarrow (o, H', dE),$$

where e is an expression to evaluate, $d\Gamma$ and H give the dynamic context for evaluation, o is the result of the evaluation, H' is the updated heap, and dE represents the effects of the evaluation. A program evaluates to reference o with heap H and effect dE if its main expression is e and $(e, \emptyset, \emptyset) \rightarrow (o, H, dE)$.

Section B of the Appendix states the rules for program evaluation. The rules are standard for an imperative language, except that we record read effects in DYN-FIELD-

ACCESS and DYN-ARRAY-ACCESS and write effects in DYN-FIELD-ASSIGN and DYN-ARRAY-ASSIGN. Rules DYN-LET and DYN-INVOKE accumulate the effects of the component expressions. Note that when we evaluate `new` T we eliminate any $*$ from T in the dynamic type of the new reference, e.g., `new` C`<Root:*>` is the same as `new` C`<Root>`; this rule ensures that all object fields are allocated in fully specified regions. This rule is sound for the same reason that assigning C`<Root>` to a variable of type C`<Root:*>` is sound.

6.3 Soundness

Our key soundness result is that we can define and check a static property of noninterference of effect between expressions in the language, such that static noninterference implies dynamic noninterference. Appendix C states the major steps of the proof in formal terms. We divide the steps into three groups: type and effect preservation (Section C.1), disjointness (Section C.2), and noninterference of effect (Section C.3).We provide further explanation and a full proof in our technical report [10].

Type and effect preservation. In Section C.1, we assert some preliminary definitions and the preservation result. A dynamic environment $d\Gamma$ is valid (**Definition 1**) if the types and RPLs appearing on the right of its bindings are valid, and it is internally consistent. A heap H is valid (**Definition 2**) if the reference stored in every object field or array cell of H is consistent with the declared type of the field or cell, translated to $d\Gamma$. A dynamic environment $d\Gamma$ *instantiates* a static environment Γ (**Definition 3**) if the bindings to variables in $d\Gamma$ are consistent with the bindings to the corresponding variables in Γ, after translation to $d\Gamma$.

Theorem 1 establishes that we can use the static types and effects (Section 6.1) to reason soundly about dynamic types and effects (Section 6.2). It states that if we type an expression e in environment Γ, and we evaluate e in dynamic environment $d\Gamma$, where $d\Gamma$ instantiates Γ, then (a) the evaluation takes a valid heap to a valid heap; (b) the static type of e bounds the dynamic type of the value o that results from the evaluation; and (c) the static effect of e bounds the dynamic effect that results from the evaluation.

Disjoint RPLs. In Section C.2, we formally define a disjointness relation on pairs of RPLs ($\Gamma \triangleright R \# R'$). The relation formalizes distinctions from the left and right, as discussed informally in Section 3.2. **Definition 4** formally expresses how to interpret a dynamic RPL as a set of fully-specified RPLs (i.e., regions). **Definition 5** shows how to associate every object field and array cell with a region of the heap. **Proposition 1** states that disjoint RPLs imply disjoint sets of fully specified regions, i.e., disjoint sets of locations. **Proposition 2** states that at runtime, disjoint fully-specified regions imply disjoint locations.

Noninterference. In Section C.3, we formally define a noninterference relation on pairs of static effects ($\Gamma \triangleright E \# E'$). The rules express four basic facts: (1) reads commute with reads; (2) writes commute with reads or writes if the regions are disjoint; (3) invocations commute with other effects if the underlying effects are disjoint; and (4) two invocations commute if the methods are declared to commute, regardless of interference between the underlying effects.

Theorem 2 expresses the main soundness property of Core DPJ, which is that the execution order of noninterfering expressions does not matter. It states that in a well-typed program, if e and e' are expressions with types T and T' and effects E and E', and E and E' are noninterfering, then either order of evaluating e and e' produces the same values o and o', the same effects dE and dE', and the same final heap H.

The claim is true for dynamic effects from the commutativity of reads, the disjointness results of Section C.2, and the assumed correctness of the commutativity specifications for methods. The claim is true for static effects by the type and effect preservation property above. See [10] for the formal proof.

6.4 Deterministic Parallelism

As discussed in Sections 2 and 4, the actual DPJ language includes `foreach` for parallel loops and `cobegin` for a block of parallel statements. We briefly discuss how to extend the formalism to model these constructs.

We can easily simulate `cobegin` by adding a parallel composition operator $e|e'$, which says to execute e and e' in the same environment, in an unspecified order, with an implicit join at the end of the execution. We can simulate `foreach` by allowing an induction variable i to appear in expressions inside the scope of a `foreach`, mapping i to n over the index range of the `foreach`, and evaluating all e_n in unspecified order. In both cases we can extend the static typing rules to say that for any pair of expressions e and e' as to which the order of execution is unspecified, then the effects of e and e' must be noninterfering.

It follows directly from Theorem 2 that parallel composition of noninterfering expressions produces the same result as sequential composition of those expressions. This guarantees determinism of execution regardless of the order of parallel execution. The formalization of this property is straightforward, and we omit it from our technical report.

7. Evaluation

We have carried out a preliminary evaluation of the language and type system features presented in this paper. Our evaluation addressed the following questions:

- **Expressiveness.** Can the type system express important parallel algorithms on object-oriented data structures? When does it fail to capture parallelism and why?
- **Coverage.** Are each of the *new* features in the DPJ type system important to express one or more of these algorithms?

- **Performance.** For each of the algorithms, what increase in performance is realized in practice? This is a quantitative measure of how much parallelism the type system can express for each algorithm (note that the runtime overheads introduced by DPJ are negligible).

To do the evaluation, we extended Sun's *javac* compiler so that it compiles DPJ into ordinary Java source. We built a runtime system for DPJ using the *ForkJoinTask* framework that will be added to the `java.util.concurrent` standard library in Java 1.7 [2]. *ForkJoinTask* supports dynamic scheduling of lightweight parallel tasks, using a work-stealing scheduler similar to that in Cilk [8]. The DPJ compiler automatically translates `foreach` to a recursive computation that successively divides the iteration space, to a depth that is tunable by the programmer, and it translates a `cobegin` block into one task for every statement. Code using *ForkJoinTask* is compatible with Java threads so an existing multithreaded Java program can be incrementally ported to DPJ. Such code may still have some guarantees, e.g., the DPJ portions will be guaranteed deterministic if the explicitly threaded and DPJ portions are separate phases that do not run concurrently.

Using the DPJ compiler, we studied the following programs: Parallel merge sort, two codes from the Java Grande parallel benchmark suite (a Monte Carlo financial simulation and IDEA encryption), the force computation from the Barnes-Hut n-body simulation [45], k-means clustering from the STAMP benchmarks [34], and a tree-based collision detection algorithm from a large, real-world open source game engine called JMonkey (we refer to this algorithm as Collision Tree). For all the codes, we began with a sequential version and modified it to add the DPJ type annotations. The Java Grande benchmarks are explicitly parallel versions using Java threads (along with equivalent sequential versions), and we compared DPJ's performance against those. We also wrote and carefully tuned the Barnes-Hut force computation using Java threads as part of understanding performance issues in the code, so we could compare Java and DPJ for that one as well.

7.1 A Realistic Example

We use the Barnes-Hut force computation to show how to write a realistic parallel program in DPJ. Figure 14 shows a simplified version of this code. The main simplification is that the `Vector` objects are immutable, with `final` fields (so there are no effects on these objects), whereas our actual implementation uses mutable objects. The class `Node` represents an abstract tree node containing a mass and position. The mass and position represent the actual mass and position of a body (at a leaf) or the center of mass of a subtree (at an inner node). The `Node` class has two subclasses: `InnerNode`, representing an inner node of the tree, and storing an array of children; and `Body`, representing the body data stored at the leaves, and storing a force. The `Tree` class stores the tree,

```
 1 /* Abstract class for tree nodes */
 2 abstract class Node<region R> {
 3     region MP;           /* Region for mass and position */
 4     double mass in R:MP; /* Mass */
 5     Vector pos in R:MP;  /* Position */
 6 }
 7
 8 /* Inner node of the tree */
 9 class InnerNode<region R> extends Node<R> {
10     region Children;
11     Node<R:*>[]<R:Children> children in R:Children;
12 }
13
14 /* Leaf node of the tree */
15 class Body<region R> extends Node<R> {
16     region Force;            /* Region for force */
17     Vector force in R:Force; /* Force on this body */
18
19     /* Compute force of entire subtree on this body */
20     Vector computeForce(Node<R:*> subtree)
21         reads R:*:Children, R:*:MP { ... }
22 }
23
24 /* Barnes-Hut tree */
25 class Tree<region R> {
26     region Tree;                        /* Region for tree */
27     Node<R> root in R:Tree;                      /* Root */
28     Body<R:[i]>[]<R:[i]>#i bodies in R:Tree;  /* Leaves */
29
30     /* Compute forces on all bodies */
31     void computeForces() writes R:* {
32         foreach (int i in 0, bodies.length) {
33             /* reads R:Tree, R:*:Node.Children, R:[i],
34                R:*:Node.MP writes R:[i]:Node.Force */
35             bodies[i].force = bodies[i].computeForce(root);
36         }
37     }
38 }
```

Figure 14. Using DPJ to write the Barnes-Hut force computation.

together with an array of `Body` objects pointing to the leaves of the tree.

The method `Tree.computeForces` does the force computation by traversing the array of bodies and calling the method `Body.computeForce` on each one, to compute the force between the body `this` and `subtree`. If `subtree` is a body, or is sufficiently far away that it can be approximated as a point mass, then `Body.computeForce` computes and returns the pairwise interaction between the nodes. Otherwise, it recursively calls `computeForce` on the children of `subtree`, and accumulates the result.

We use a region parameter on the node classes to distinguish instances of these nodes. Class `Tree` uses the parameters to create an index-parameterized array of references to distinct body objects; the parallel loop in `computeForces` iterates over this array. This allows distinctions from the left for operations on `bodies[i]` (Section 3). We also use distinct region names within each class (in particular, for the force, masses and positions, and the children array) to enable distinctions from the right.

The key fact is that, from the effect summary in line 21 and the code in line 35, the compiler infers the effects shown in lines 33–34. Using distinctions from the left and right, the compiler can now prove that (1) the updates are distinct for distinct iterations of the `foreach`; and (2) all the updates are distinct from the reads. Notice also how the nested RPLs allow us to describe the entire effect of `computeForces` as `writes R:*`. That is, to the outside world, `computeForces` just writes under the region parameter of `Tree`. Thus with careful use of RPLs, we can enforce a kind of encapsulation of effects, which is important for modular software design.

7.2 Expressiveness and Coverage

We used DPJ to express *all* available parallelism (except for vector parallelism, which we do not consider here) for Merge Sort, Monte Carlo, IDEA, K-Means, and Collision Tree. For Barnes-Hut, the overall program includes four major phases in each time step: tree building; center-of-mass computation; force calculations; and position calculations. Expressing the force, center of mass, and position calculations is straightforward, but we studied only the force computation (the dominant part of the overall computation) for this work. DPJ can also express the tree-building phase, but we would have to use a divide-and-conquer approach, instead of inserting bodies from the root via "hand-over-hand locking," as in in [45].

Briefly, we parallelized each of the codes as follows. MergeSort uses subarrays (Section 4.2) to perform in-place parallel divide and conquer operations for both merge and sort, switching to sequential merge and sort for subproblems below a certain size. Monte Carlo uses index-parameterized arrays (Section 4.1) to generate an array of tasks and compute an array of results, followed by commutativity annotations (Section 5) to update to globally shared data inside a reduction loop. IDEA uses subarrays to divide the input array into disjoint pieces, then uses `foreach` to operate on each of the pieces. Section 7.1 describes our parallel Barnes-Hut force computation. Collision Tree recursively walks two trees, reading the trees and collecting a list of intersecting triangles. At each node, a separate triangle list is computed in parallel for each subtree, and then the lists are merged. Our implementation uses method-local regions to distinguish the writes to the left and right subtree lists. K-Means uses commutativity annotations to perform simultaneous reductions, one for each cluster. Table 1 summarizes the novel DPJ capabilities used for each code.

Table 1. Capabilities Used In The Benchmarks
1. Index-parameterized array; 2. Distinctions from the left; 3. Distinctions from the right; 4. Recursive subranges; 5. Commutativity annotations.

Benchmark	1	2	3	4	5
Merge Sort	-	Y	-	Y	-
Monte Carlo	Y	Y	-	-	Y
IDEA	-	Y	-	Y	-
Barnes-Hut	Y	Y	Y	-	-
Collision Tree	-	Y	-	-	-
K Means	-	-	-	-	Y

Our evaluation and experience showed some interesting limitations of the current language design. To achieve good cache performance in Barnes-Hut, the bodies must be reordered according to their proximity in space on each time step [45]. As discussed in Section 7.1, we use an index-parameterized array to update the bodies in parallel. As discussed in Section 4.1, this requires that we copy each body with the new destination regions at the point of re-insertion. As future work, we believe we can ease this restriction by adding a mechanism for disjointness checking at runtime.

Num	**Monte Carlo**		**IDEA**		**Barnes Hut**	
Cores	DPJ	Java	DPJ	Java	DPJ	Java
2	2.00	1.80	1.95	1.99	1.98	1.99
3	2.82	2.50	2.88	2.97	2.96	2.94
4	3.56	3.09	3.80	3.91	4.94	3.88
7	5.53	4.65	6.40	6.70	6.79	7.56
12	8.01	6.46	9.99	11.04	11.4	13.65
17	10.02	7.18	12.70	14.90	15.3	19.04
22	11.50	7.98	18.70	17.79	23.9	23.33

Table 2. Comparison of DPJ vs. Java threads performance for Monte Carlo, IDEA encryption, and Barnes Hut.

7.3 Performance

We measured the performance of each of the benchmarks on a Dell R900 multiprocessor running Red Hat Linux with 24 cores, comprising four six-core Xeon processors, and a total of 48GB of main memory. For each data point, we took the minimum of five runs on an idle machine.

We studied multiple inputs for each of the benchmarks and also experimented with different limits for recursive codes. We present results for the inputs and parameter values that show the best performance, since our main aim is to evaluate how well DPJ can express the parallelism in these codes. The sensitivity of the parallelism to input size and/or recursive limit parameters is a property of the algorithm and not a consequence of using DPJ.

Figure 15 presents the speedups of the six programs for $p \in \{1, 2, 3, 4, 7, 12, 17, 22\}$ processors. All speedups are relative to an equivalent sequential version of the program, *with no DPJ or other multithreaded runtime overheads*. All six codes showed moderate to good scalability for all values of p. Barnes-Hut and Merge Sort showed near-ideal performance scalability, with Barnes-Hut showing a superlinear increase for $p = 22$ due to cache effects.

Notably, as shown in Table 2, for the three codes where we have manually parallelized Java threads versions available, the DPJ versions achieved speedups close to (IDEA and Barnes Hut), or better than (Monte Carlo), the Java versions, for the same inputs on the same machines. We believe the Java threads codes are all reasonably well tuned; the two Java Grande benchmarks were tuned by the original authors and the Barnes Hut code was tuned by us. The manually parallelized Monte Carlo code exhibited a similar leveling off in speedup as the DPJ version did beyond about 7 cores because both have a significant sequential component that makes copies of a large array for each parallel task. Overall, in all three programs, DPJ is able to express the available parallelism as efficiently as a lower-level hand coded parallel programming model that provides no guarantees of determinism or even race-freedom.

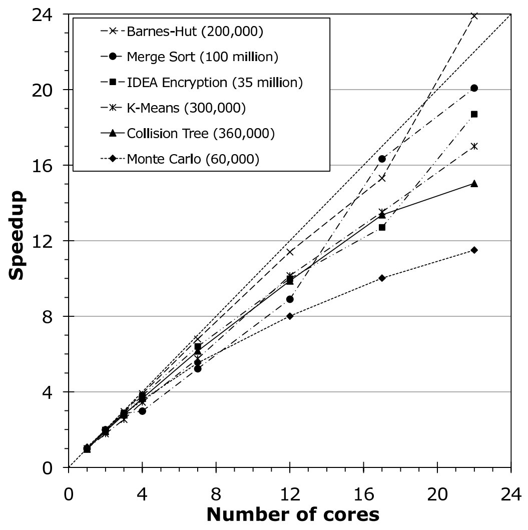


Figure 15. Speedups. Numbers in legend are input sizes.

Program	Total SLOC	Annotated SLOC	Region Decls	RPLs	Effect Summ.	Comm.
MergeSort	295	38 (12.9%)	15	41	7	0
Monte Carlo	2877	220 (7.6%)	13	301	161	1
IDEA	228	24 (10.5%)	8	22	2	0
Barnes-Hut	682	80 (11.7%)	25	123	38	0
CollisionTree	1032	233 (22.6%)	82	408	58	0
K-means	501	5 (1.0%)	0	3	3	1
Total	**5615**	**600 (10.7%)**	143	**898**	**269**	**2**

Table 3. Annotation counts for the case studies.

Our experience so far has shown us that DPJ itself can be very efficient, even though both the compiler and runtime are preliminary. In particular (except for very small runtime costs for the dynamic partitioning mechanism for subarrays), our type system requires no runtime checks or speculation and therefore *adds negligible runtime overhead for achieving determinism*. On the other hand, it is possible that the type system may constrain algorithmic design choices. The limitation on reordering the array of bodies in Barnes-Hut, explained in Section 7.2, is one such example.

7.4 Porting Effort

Table 3 shows the number of source lines changed and the number of annotations, relative to the program size. Program size is given in non-blank, non-comment lines of source code, counted by *sloccount*. The next column shows how many LOC were changed when annotating. The last four columns show (1) the number of declarations using the `region` keyword (i.e., field regions, local regions, and region parameters); (2) the number of RPLs appearing as arguments to `in`, types, methods, and effect summaries; (3) the number of method effect summaries, counting `reads` and `writes` separately; and (4) the number of commutativity annotations. As the table shows, the fraction of lines of code changed was not large, averaging 10.7% of the original lines. Most of the changed lines were due to writing RPL arguments when instantiating types (represented in column four), followed by writing method effect summaries (column five).

More importantly, we believe that the overall effort of writing, testing, and debugging a program with *any* parallel programming model is dominated by the time required to understand the parallelism and sharing patterns (including aliases), and to debug the parallel code. The regions and effects in DPJ provide *concrete guidance to the programmer on how to reason about parallelism and sharing*. Once the programmer understands the sharing patterns, he or she explicitly documents them in the code through region and effect annotations, so other programmers can gain the benefit of his or her understanding.

Further, programming tools can alleviate the burden of writing annotations. We have developed an interactive porting tool, DPJIZER [49], that infers many of these annotations, using iterative constraint solving over the whole program. DPJIZER is implemented as an Eclipse plugin and correctly infers method effect summaries for a program that is already annotated with region information. We are currently extending DPJIZER to infer RPLs, assuming that the programmer declares the regions.

In addition, a good set of defaults can further reduce the amount of manually written annotations. For example, if the programmer does not annotate a class field, its default region could be the RPL *default-parameter*:*field-name*. This default distinguishes both instances of the same class and fields within a class. The programmer can override the defaults if she needs further refinements.

8. Related Work

We group the related work into five broad categories: effect systems (not including ownership-based systems); ownership types (including ownership with effects); unique references; separation logic; and runtime checks.

Effect Systems: The seminal work on types and effects for concurrency is FX [33, 27], which adds a region-based type and effect system to a Scheme-like, implicitly parallel language. Leino et al. [30] and Greenhouse and Boyland [26] first added effects to an object-oriented language. None of these systems can represent arbitrarily nested structures or array partitioning, and they cannot specify arbitrarily large sets of regions. Also, the latter two systems rely on alias restrictions and/or supplementary alias analysis for soundness of effect, whereas DPJ does not.

Ownership Types: Some ownership-based type systems have been combined with effects to enable reasoning about noninterference. Both JOE [16, 46] and MOJO [14] have sophisticated effect systems that allow nested regions and effects. However, neither has the capabilities of DPJ's array partitioning and partially specified RPLs, which are crucial to expressing the patterns addressed in this paper. JOE's `under` effect shape is similar to DPJ's $*$, but it cannot do the equivalent of our distinctions from the right. JOE allows slightly more precision than our rule LET when a type or effect uses a local variable that goes out of scope, but we have found that this precision is not necessary for expressing deterministic parallelism. MOJO has a wildcard region specifier $?$, but it pertains to the orthogonal capability of *multiple ownership*, which allows objects to be placed in multiple regions. Leino's system also has this capability, but without arbitrary nesting.

Lu and Potter [32] show how to use effect constraints to break the owner dominates rule in limited ways while still retaining meaningful guarantees. The `any` context of [32] is identical to `Root:*` in our system, but we can make more fine-grained distinctions. For example, we can conclude that a pair of references stored in variables of type `C<`R_1`:*>` and `C<`R_2`:*>` can never alias, if R_1`:*` and R_2`:*` are disjoint.

Several researchers [11, 3, 28] have described effect systems for enforcing a locking discipline in nondeterministic programs, to prevent data races and deadlocks. Because they have different goals, these effect systems are very different from ours, e.g., they cannot express arrays or nested effects.

Finally, an important difference between DPJ and most ownership systems is that we allow *explicit region declarations*, like [33, 30, 26], whereas ownership systems generally couple region creation with object creation. We have found many cases where a new region is needed but a new object is not, so the ownership paradigm becomes awkward. Supporting field granularity effects also is difficult with ownership.

Unique References: Boyland [13] shows how to use alias restrictions to guarantee determinism for a simple language with pointers. Terauchi and Aiken [48] have extended this work with a type inference algorithm that simplifies the type annotations and elegantly expresses some simple patterns of determinism. Alias restrictions are a well-known alternative to effect annotations for reasoning about heap access, and in some cases they can complement effect annotations [26, 12]. However, alias restrictions severely limit the expressivity of an object-oriented language. It is not clear whether the techniques in [13, 48] could be applied to a robust object-oriented language. Clarke and Wrigstad's external uniqueness [17] is better suited to an object-oriented style, but it is not clear whether external uniqueness is useful for deterministic parallelism.

Separation Logic: Separation logic [40] (SL) is a potential alternative to effect systems for reasoning about shared resources. O'Hearn [35] and Gotsman et al. [25] use SL to check race freedom, though O'Hearn includes some simple proofs of noninterference. Parkinson [37] has extended C# with SL predicates to allow sound inference in the presence of inheritance. Raza et al. [39] show how to use separation logic together with shape analysis for automatic parallelization of a sequential program.

While SL is a promising approach, applying it to realistic programs poses two key issues. First, SL is a *low-level* specification language: it generally treats memory as a single array of words, on which notions of objects and linked data structures must be defined using SL predicates [40, 35]. Second, SL approaches generally *either* require heavyweight theorem proving and/or a relatively heavy programmer annotation burden [37] *or* are fully automated, and thereby limited by what the compiler can infer [25, 39].

In contrast, we chose to start from the extensive prior work on regions and effects, which is more mature than SL for OO languages. As noted in [40], type systems and SL systems have many common goals but have developed largely in parallel; as future research it would be useful to understand better the relationship between the two.

Runtime Checks: A number of systems enforce some form of disciplined parallelism via runtime checks. Jade [43] and Prometheus [5] use runtime checks to guarantee deterministic parallelism for programs that do not fail their checks. Jade also supports a simple form of commutativity annotation [41]. Multiphase Shared Arrays [20] and PPL1 [47] are similar in that they rely on runtime checks that may fail if determinism is violated. None of these systems checks nontrivial sharing patterns at compile time.

Speculative parallelism [7, 23, 51] can achieve determinism with minimal programmer annotations, compared to DPJ. However, speculation generally either incurs significant software overheads or requires special hardware [38, 31, 50]. Grace [7] reduces the overhead of software-only speculation by running threads as separate processes and using commodity memory protection hardware to detect conflicts at page granularity. However, Grace does not efficiently support essential sharing patterns such as (1) fine-grain access distinctions (e.g., distinguishing different fields of an object, as in Barnes-Hut); (2) dynamically scheduled fine-grain tasks (e.g., *ForkJoinTask*); or (3) concurrent data structures, which are usually finely interleaved in memory. Further, unlike DPJ, a speculative solution does not document the parallelization strategy or show how the code must be rewritten to expose parallelism.

Kendo [36] and DMP [21] use runtime mechanisms to guarantee equivalence to some (arbitrary) serial interleaving of tasks; however, that interleaving is not necessarily obvious from the program text, as it is in DPJ. Further, Kendo's guarantee fails if the program contains data races, and DMP requires special hardware support. SharC [6] uses a combination of static and dynamic checks to enforce race freedom, but not necessarily deterministic semantics, in C programs.

Finally, a determinism checker [44, 22] instruments code to detect determinism violations at runtime. This approach is not viable for production runs because of the slowdowns caused by the instrumentation, and it is limited by the coverage of the inputs used for the dynamic analysis. However, it is sound for the observed traces.

9. Conclusion

We have described a novel type and effect system, together with a language called DPJ that uses the system to enforce deterministic semantics. Our experience shows that the new type system features are useful for writing a range of programs, achieving moderate to excellent speedups on a 24-processor system with guaranteed determinism.

Our future goals are to exploit region and effect annotations for optimizing memory hierarchy performance; to add runtime support for more flexible operationson index-parameterized arrays; to add support for object-oriented parallel frameworks; and to add support for explicitly nondeterministic algorithms.

Acknowledgments

The following persons provided invaluable insight at various stages of this work: Ralph Johnson, Christopher Rodrigues, Marc Snir, Dan Grossman, Brad Chamberlain, John Brant, Rajesh Karmani, and Maurice Rabb.

References

[1] `http://dpj.cs.uiuc.edu`.

[2] `http://gee.cs.oswego.edu/dl/concurrency-interest`.

[3] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *TOPLAS*, 2006.

[4] F. Aleen and N. Clark. Commutativity analysis for software parallelization: letting program transformations see the big picture. *ASPLOS*, 2009.

[5] M. D. Allen, S. Sridharan, and G. S. Sohi. Serialization sets: A dynamic dependence-based parallel execution model. *PPOPP*, 2009.

[6] Z. Anderson, D. Gay, R. Ennals, and E. Brewer. SharC: Checking data sharing strategies for multithreaded C. *PLDI*, 2008.

[7] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe Multithreaded Programming for C/C++. *OOPSLA*, 2009.

[8] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *PPOPP*, 1995.

[9] R. Bocchino, V. Adve, S. Adve, and M. Snir. Parallel programming must be deterministic by default. *First USENIX Workshop on Hot Topics in Parallelism (HotPar)*, 2009.

[10] R. L. Bocchino and V. S. Adve. Formal definition and proof of soundness for Core DPJ. Technical Report UIUCDCS-R-2008-2980, U. Illinois, 2008.

[11] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. *OOPSLA*, 2002.

[12] J. Boyland. The interdependence of effects and uniqueness. *Workshop on Formal Techs. for Java Programs*, 2001.

[13] J. Boyland. Checking interference with fractional permissions. *SAS*, 2003.

[14] N. R. Cameron, S. Drossopoulou, J. Noble, and M. J. Smith. Multiple ownership. *OOPSLA*, 2007.

[15] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *OOPSLA*, 2005.

[16] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. *OOPSLA*, 2002.

[17] D. Clarke and T. Wrigstad. External uniqueness is unique enough. *ECOOP*, 2003.

[18] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. *OOPSLA*, 1998.

[19] J. Dennis. Keynote address. *PPOPP*, 2009.

[20] J. DeSouza and L. V. Kalé. MSA: Multiphase specifically shared arrays. *LCPC*, 2004.

[21] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic Shared Memory Multiprocessing. *ASPLOS*, 2009.

[22] M. Feng and C. E. Leiserson. Efficient detection of determinacy races in Cilk programs. *SPAA*, 1997.

[23] C. Flanagan and M. Felleisen. The semantics of future and its use in program optimization. *POPL*, 1995.

[24] R. Ghiya, D. Lavery, and D. Sehr. On the importance of points-to analysis and other memory disambiguation methods for C programs. *PLDI*, 2001.

[25] A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular shape analysis. *PLDI*, 2007.

[26] A. Greenhouse and J. Boyland. An object-oriented effects system. *ECOOP*, 1999.

[27] R. T. Hammel and D. K. Gifford. FX-87 performance measurements: Dataflow implementation. Technical Report MIT/LCS/TR-421, 1988.

[28] B. Jacobs, F. Piessens, J. Smans, K. R. M. Leino, and W. Schulte. A programming model for concurrent object-oriented programs. *TOPLAS*, 2008.

[29] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. *PLDI*, 2007.

[30] K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. *PLDI*, 2002.

[31] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. POSH: a TLS compiler that exploits program structure. *PPOPP*, 2006.

[32] Y. Lu and J. Potter. Protecting representation with effect encapsulation. *POPL*, 2006.

[33] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. *POPL*, 1988.

[34] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. *IISWC*, 2008.

[35] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comp. Sci.*, 2007.

[36] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. *ASPLOS*, 2009.

[37] M. J. Parkinson and G. M. Bierman. Separation logic, abstraction and inheritance. *POPL*, 2008.

[38] M. K. Prabhu and K. Olukotun. Using thread-level speculation to simplify manual parallelization. *PPOPP*, 2003.

[39] M. Raza, C. Calcagno, and P. Gardner. Automatic parallelization with separation logic. *ESOP*, 2009.

[40] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. *Symp. on Logic in Comp. Sci.*, 2002.

[41] M. C. Rinard. *The design, implementation and evaluation of Jade: A portable, implicitly parallel programming language*. PhD thesis, Stanford University, 1994.

[42] M. C. Rinard and P. C. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *TOPLAS*, 1997.

[43] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of Jade. *TOPLAS*, 1998.

[44] C. Sadowski, S. N. Freund, and C. Flanagan. SingleTrack: A dynamic determinism checker for multithreaded programs. *ESOP*, 2009.

[45] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. Technical report, Stanford University, 1992.

[46] M. Smith. Towards an effects system for ownership domains. *ECOOP*, 2005.

[47] M. Snir. Parallel Programming Language 1 (PPL1), V0.9 — Draft. Technical Report UIUCDCS-R-2006-2969, U. Illinois, 2006.

[48] T. Terauchi and A. Aiken. A capability calculus for concurrency and determinism. *TOPLAS*, 2008.

[49] M. Vakilian, D. Dig, R. Bocchino, J. Overbey, V. Adve, and R. Johnson. Inferring Method Effect Summaries for Nested Heap Regions. *ASE*, 2009. To appear.

[50] C. von Praun, L. Ceze, and C. Caşcaval. Implicit parallelism with ordered transactions. *PPOPP*, 2007.

[51] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. *OOPSLA*, 2005.

[52] K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. *PLDI*, 2008.

A. Static Semantics Rules

We divide the static semantics in to five parts: rules for valid program elements (**Figure 16**), rules for validity, nesting, and inclusion of RPLs (**Figure 17**), rules for valid types and subtypes (**Figure 18**), rules for valid effects and subeffects (**Figure 19**), and rules for typing expressions (**Figure 20**).

B. Dynamic Semantics Rules

Figure 21 gives the rules for evaluating programs. If $f : A \to B$ is a function, then $f \cup \{x \mapsto y\}$ is the function $f' : A \cup \{x\} \to B \cup \{y\}$ defined by $f'(a) = f(a)$ if $a \neq x$ and $f'(x) = y$. new(C) is the function taking each field of class C with type T to a null reference of type $d\Gamma(T)$, and new$(T[n])$ is the function taking each index $n' \in [0, n-1]$ to a null reference of type $d\Gamma(T)[i \leftarrow n']$.

The rules for dynamic RPLs, types, and effects are nearly identical to their static counterparts. Instead of writing out all the rules, we describe how to generate them via simple substitution from the rules given in Section A. For every rule given there except RPL-VAR, RPL-PARAM, UNDER-VAR, INCLUDE-PARAM, and INCLUDE-FULL, do the following: (1) append DYN- to the front of the name; (2) replace Γ with H and $[i]$ with $[n]$; and (3) replace R with dR, T with dT, and E with dE. For example, here are the rules for dynamic class subtyping, generated by the substitution above from the rule SUBTYPE-CLASS:

$$\text{(DYN-SUBTYPE-CLASS)} \quad \frac{H \rhd dR \subseteq dR'}{H \rhd C\texttt{<}dR\texttt{>} \leq C\texttt{<}dR'\texttt{>}}$$

Then add the following rules:

$$\text{(DYN-RPL-REF)} \quad \frac{H \rhd o : dT}{H \rhd o} \qquad \text{(DYN-UNDER-REF)} \quad \frac{H \rhd o : C\texttt{<}dR\texttt{>}}{H \rhd o \leq dR}$$

$$\text{(DYN-TYPE-ARRAY)} \quad \frac{H \rhd dT[i \leftarrow n] \quad H \rhd dR[i \leftarrow n]}{H \rhd dT[]\texttt{<}dR\texttt{>}\#i}$$

C. Soundness

C.1 Type and Effect Preservation

Definition 1 (Valid dynamic environments)**.** *A dynamic environment $d\Gamma$ is valid with respect to heap H ($H \rhd d\Gamma$) if the following hold: (1) for every binding $z \mapsto o \in d\Gamma$, $H \rhd o : dT$; (2) for every binding $P \mapsto dR \in d\Gamma$, $H \rhd dR$; and (3) if* `this` *$\mapsto o \in d\Gamma$, then $H \rhd o : C\texttt{<}dR\texttt{>}$, and param$(C) \mapsto dR \in d\Gamma$.*

Definition 2 (Valid heaps)**.** *A heap H is valid ($\rhd H$) if for each $o \in Dom(H)$, one of the following holds:*

1. *(a) $H \vdash o : C\texttt{<}dR\texttt{>}$ and (b) $H \rhd C\texttt{<}dR\texttt{>}$ and (c) for each field $T\ f$ **in** $R_f \in def(C)$, if $H(o)(f)$ is defined, then $H \rhd H(o)(f) : dT$ and $H \rhd dT$ and $H \rhd dT \leq T[o \leftarrow$ **this**$][dR \leftarrow param(C)]$; or*
2. *(a) $H \rhd o : dT[]\texttt{<}dR\texttt{>}\#i$ and (b) $H \rhd dT[]\texttt{<}dR\texttt{>}\#i$ and (c) if $H(o)(n)$ is defined, then $H \rhd H(o)(n) : dT'$ and $H \rhd dT$ and $H \rhd dT' \leq dT[i \leftarrow n]$.*

Definition 3 (Instantiation of static environments)**.** *A dynamic environment $d\Gamma$ instantiates a static environment Γ ($H \rhd d\Gamma \leq \Gamma$) if $\rhd\Gamma$, $\rhd H$, and $H \rhd d\Gamma$; the same variables appear in Dom(Γ) as in Dom$(d\Gamma)$; and for each pair $z \mapsto T \in \Gamma$ and $z \mapsto o \in d\Gamma$, $H \rhd v : dT$ and $H \rhd dT \leq d\Gamma(T)$.*

Theorem 1 (Preservation)**.** *For a well-typed program, if $\Gamma \rhd e : T, E$ and $H \rhd d\Gamma \leq \Gamma$ and $(e, d\Gamma, H) \to (o, H', dE)$, then (a) $\rhd H'$; and (b) $H' \rhd dT \leq d\Gamma(T)$, where $H' \rhd o : dT$; and (c) $H' \rhd dE$; and (d) $H' \rhd dE \subseteq d\Gamma(E)$.*

$$\text{(PROGRAM)}\ \frac{\triangleright \mathit{class}^* \quad \emptyset \triangleright e : T, E}{\triangleright \mathit{class}^*\ e} \qquad \text{(CLASS)}\ \frac{\{\texttt{this} \mapsto C\texttt{<}P\texttt{>}\} \triangleright \mathit{field}^*\ \mathit{method}^*\ \mathit{comm}^*}{\triangleright \texttt{class}\ C\texttt{<}P\texttt{>}\ \{\ \mathit{field}^*\ \mathit{method}^*\ \mathit{comm}^*\ \}} \qquad \text{(ENV)}\ \frac{\forall z \mapsto T \in \Gamma.\Gamma \triangleright T \quad \forall P \subseteq R \in \Gamma.\Gamma \triangleright R}{\triangleright \Gamma}$$

$$\text{(FIELD)}\ \frac{\Gamma \triangleright T \quad \Gamma \triangleright R}{\Gamma \triangleright T\ f\ \texttt{in}\ R} \qquad \text{(METHOD)}\ \frac{\Gamma \triangleright T_r, T_x, E \quad \Gamma' = \Gamma \cup \{x \mapsto T_x\} \quad \Gamma' \triangleright e : T', E' \quad \Gamma' \triangleright T' \leq T_r \quad \Gamma' \triangleright E' \subseteq E}{\Gamma \triangleright T_r\ m(T_x\ x)\ E\ \{\ e\ \}}$$

$$\text{(COMM)}\ \frac{\texttt{this} \mapsto C\texttt{<}P\texttt{>} \in \Gamma \quad \exists \mathrm{def}(C.m), \mathrm{def}(C.m')}{\Gamma \triangleright m\ \texttt{commuteswith}\ m'}$$

Figure 16. Rules for valid program elements. def$(C.m)$ means the definition of method m in class C.

$$\text{(RPL-ROOT)}\ \frac{}{\Gamma \triangleright \texttt{Root}} \qquad \text{(RPL-VAR)}\ \frac{z \mapsto C\texttt{<}R\texttt{>} \in \Gamma}{\Gamma \triangleright z} \qquad \text{(RPL-PARAM)}\ \frac{\texttt{this} \mapsto C\texttt{<}P\texttt{>} \in \Gamma \vee P \subseteq R \in \Gamma}{\Gamma \triangleright P} \qquad \text{(RPL-NAME)}\ \frac{\Gamma \triangleright R \quad \texttt{region}\ r \in \mathit{program}}{\Gamma \triangleright R : r}$$

$$\text{(RPL-INDEX)}\ \frac{\Gamma \triangleright R \quad i \in \Gamma}{\Gamma \triangleright R : [i]} \qquad \text{(RPL-STAR)}\ \frac{\Gamma \triangleright R}{\Gamma \triangleright R : *} \qquad \text{(UNDER-ROOT)}\ \frac{}{\Gamma \triangleright R \leq \texttt{Root}} \qquad \text{(UNDER-VAR)}\ \frac{z \mapsto C\texttt{<}R\texttt{>} \in \Gamma}{\Gamma \triangleright z \leq R}$$

$$\text{(UNDER-NAME)}\ \frac{\Gamma \triangleright R \leq R'}{\Gamma \triangleright R : r \leq R'} \qquad \text{(UNDER-INDEX)}\ \frac{\Gamma \triangleright R \leq R'}{\Gamma \triangleright R : [i] \leq R'} \qquad \text{(UNDER-STAR)}\ \frac{\Gamma \triangleright R \leq R'}{\Gamma \triangleright R : * \leq R'} \qquad \text{(UNDER-INCLUDE)}\ \frac{\Gamma \triangleright R \subseteq R'}{\Gamma \triangleright R \leq R'}$$

$$\text{(INCLUDE-NAME)}\ \frac{\Gamma \triangleright R \subseteq R'}{\Gamma \triangleright R : r \subseteq R' : r} \qquad \text{(INCLUDE-INDEX)}\ \frac{\Gamma \triangleright R \subseteq R'}{\Gamma \triangleright R : [i] \subseteq R' : [i]} \qquad \text{(INCLUDE-STAR)}\ \frac{\Gamma \triangleright R \leq R'}{\Gamma \triangleright R \subseteq R' : *}$$

$$\text{(INCLUDE-PARAM)}\ \frac{P \subseteq R \in \Gamma}{\Gamma \triangleright P \subseteq R} \qquad \text{(INCLUDE-FULL)}\ \frac{\Gamma \triangleright R \subseteq R_f}{\Gamma \triangleright R_f \subseteq R}$$

Figure 17. Rules for valid RPLs, nesting of RPLs, and inclusion of RPLs. The nesting and inclusion relations are reflexive and transitive (obvious rules omitted).

$$\text{(TYPE-CLASS)}\ \frac{\exists \mathrm{def}(C) \quad \Gamma \triangleright R}{\Gamma \triangleright C\texttt{<}R\texttt{>}} \qquad \text{(TYPE-ARRAY)}\ \frac{\Gamma \cup \{i\} \triangleright T, R}{\Gamma \triangleright T[]\texttt{<}R\texttt{>}\#i}$$

$$\text{(SUBTYPE-CLASS)}\ \frac{\Gamma \triangleright R \subseteq R'}{\Gamma \triangleright C\texttt{<}R\texttt{>} \leq C\texttt{<}R'\texttt{>}} \qquad \text{(SUBTYPE-ARRAY)}\ \frac{\Gamma \cup \{i\} \triangleright R \subseteq R'[i' \leftarrow i] \quad T \equiv T'}{\Gamma \triangleright T[]\texttt{<}R\texttt{>}\#i \leq T'[]\texttt{<}R'\texttt{>}\#i'}$$

Figure 18. Rules for valid types and subtypes. def(C) means the definition of class C. $T \equiv T'$ means that T and T' are identical up to the names of variables i.

$$\text{(EFFECT-EMPTY)}\ \frac{}{\Gamma \triangleright \emptyset} \qquad \text{(EFFECT-READS)}\ \frac{\Gamma \triangleright R}{\Gamma \triangleright \texttt{reads}\ R} \qquad \text{(EFFECT-WRITES)}\ \frac{\Gamma \triangleright R}{\Gamma \triangleright \texttt{writes}\ R} \qquad \text{(EFFECT-INVOKES)}\ \frac{\exists \mathrm{def}(C.m) \quad \Gamma \triangleright E}{\Gamma \triangleright \texttt{invokes}\ C.m\ \texttt{with}\ E}$$

$$\text{(EFFECT-UNION)}\ \frac{\Gamma \triangleright E \quad \Gamma \triangleright E'}{\Gamma \triangleright E \cup E'} \qquad \text{(SE-EMPTY)}\ \frac{}{\Gamma \triangleright \emptyset \subseteq E} \qquad \text{(SE-READS)}\ \frac{\Gamma \triangleright R \subseteq R'}{\Gamma \triangleright \texttt{reads}\ R \subseteq \texttt{reads}\ R'} \qquad \text{(SE-WRITES)}\ \frac{\Gamma \triangleright R \subseteq R'}{\Gamma \triangleright \texttt{writes}\ R \subseteq \texttt{writes}\ R'}$$

$$\text{(SE-READS-WRITES)}\ \frac{\Gamma \triangleright R \subseteq R'}{\Gamma \triangleright \texttt{reads}\ R \subseteq \texttt{writes}\ R'} \qquad \text{(SE-INVOKES-1)}\ \frac{\Gamma \triangleright E \subseteq E'}{\Gamma \triangleright \texttt{invokes}\ C.m\ \texttt{with}\ E \subseteq \texttt{invokes}\ C.m\ \texttt{with}\ E'}$$

$$\text{(SE-INVOKES-2)}\ \frac{}{\Gamma \triangleright \texttt{invokes}\ C.m\ \texttt{with}\ E \subseteq E} \qquad \text{(SE-UNION-1)}\ \frac{\Gamma \triangleright E \subseteq E' \vee \Gamma \triangleright E \subseteq E''}{\Gamma \triangleright E \subseteq E' \cup E''} \qquad \text{(SE-UNION-2)}\ \frac{\Gamma \triangleright E' \subseteq E \quad \Gamma \triangleright E'' \subseteq E}{\Gamma \triangleright E' \cup E'' \subseteq E}$$

Figure 19. Rules for valid effects and subeffects.

$$\text{(LET)}\ \frac{\Gamma \triangleright e : C\texttt{<}R\texttt{>}, E \quad \Gamma \cup \{x \mapsto C\texttt{<}R\texttt{>}\} \triangleright e' : T', E'}{\Gamma \triangleright \texttt{let}\ x = e\ \texttt{in}\ e' : T'[x \leftarrow R : *], E \cup E'[x \leftarrow R : *]} \qquad \text{(FIELD-ACCESS)}\ \frac{T\ f\ \texttt{in}\ R_f \in \mathrm{def}(C) \quad \texttt{this} \mapsto C\texttt{<}\mathrm{param}(C)\texttt{>} \in \Gamma}{\Gamma \triangleright \texttt{this}.f : T, \texttt{reads}\ R_f}$$

$$\text{(FIELD-ASSIGN)}\ \frac{\texttt{this} \mapsto C\texttt{<}\mathrm{param}(C)\texttt{>} \in \Gamma \quad z \mapsto T \in \Gamma \quad T'\ f\ \texttt{in}\ R_f \in \mathrm{def}(C) \quad \Gamma \triangleright T \leq T'}{\Gamma \triangleright \texttt{this}.f = z : T, \texttt{writes}\ R_f}$$

$$\text{(ARRAY-ACCESS)}\ \frac{z \mapsto T[]\texttt{<}R\texttt{>}\#i \in \Gamma}{\Gamma \triangleright z[n] : T[i \leftarrow n], \texttt{reads}\ R[i \leftarrow n]} \qquad \text{(ARRAY-ASSIGN)}\ \frac{\{z \mapsto T[]\texttt{<}R\texttt{>}\#i, z' \mapsto T'\} \subseteq \Gamma \quad \Gamma \triangleright T' \leq T[i \leftarrow n]}{\Gamma \triangleright z[n] = z' : T', \texttt{writes}\ R[i \leftarrow n]}$$

$$\text{(INVOKE)}\ \frac{\{z \mapsto C\texttt{<}R\texttt{>}, z' \mapsto T\} \subseteq \Gamma \quad T_r\ m(T_x\ x)\ E\ \{\ e\ \} \ \in \mathrm{def}(C) \quad \Gamma \cup \{P \subseteq R\} \triangleright T \leq T_x[\texttt{this} \leftarrow z][\mathrm{param}(C) \leftarrow P]}{\Gamma \triangleright z.m(z') : T_r[\texttt{this} \leftarrow z][\mathrm{param}(C) \leftarrow R], \texttt{invokes}\ C.m\ \texttt{with}\ E[\texttt{this} \leftarrow z][\mathrm{param}(C) \leftarrow R]}$$

$$\text{(VAR)}\ \frac{z \mapsto T \in \Gamma}{\Gamma \triangleright z : T, \emptyset} \qquad \text{(NEW-CLASS)}\ \frac{\Gamma \triangleright C\texttt{<}R\texttt{>}}{\Gamma \triangleright \texttt{new}\ C\texttt{<}R\texttt{>} : C\texttt{<}R\texttt{>}, \emptyset} \qquad \text{(NEW-ARRAY)}\ \frac{\Gamma \triangleright T[]\texttt{<}R\texttt{>}\#i}{\Gamma \triangleright \texttt{new}\ T[n]\texttt{<}R\texttt{>}\#i\ : T[]\texttt{<}R\texttt{>}\#i, \emptyset}$$

Figure 20. Rules for typing expressions. param(C) means the parameter of class C.

(DYN-LET) $$\frac{(e, d\Gamma, H) \to (o, H', dE) \quad (e', d\Gamma \cup \{x \mapsto o\}, H') \to (o', H'', dE')}{(\texttt{let}\ x = e\ \texttt{in}\ e', d\Gamma, H) \to (o', H'', dE \cup dE')}$$

(DYN-VAR) $$\frac{z \mapsto o \in d\Gamma}{(z, d\Gamma, H) \to (o, H, \emptyset)}$$

(DYN-FIELD-ACCESS) $$\frac{\texttt{this} \mapsto o \in d\Gamma \quad H \rhd o : C\texttt{<}dR\texttt{>} \quad T\ f\ \texttt{in}\ R_f \in \text{def}(C)}{(\texttt{this}.f, d\Gamma, H) \to (H(o)(f), H, \texttt{reads}\ d\Gamma(R_f))}$$

(DYN-FIELD-ASSIGN) $$\frac{\{\texttt{this} \mapsto o, z \mapsto o'\} \subseteq d\Gamma \quad H \rhd o : C\texttt{<}dR\texttt{>} \quad T\ f\ \texttt{in}\ R_f \in \text{def}(C)}{(\texttt{this}.f = z, d\Gamma, H) \to (o', H \cup \{o \mapsto (H(o) \cup \{f \mapsto o'\})\}, \texttt{writes}\ d\Gamma(R_f))}$$

(DYN-ARRAY-ACCESS) $$\frac{z \mapsto o \in d\Gamma \quad H \rhd o : dT[]\texttt{<}dR\texttt{>}\#i}{(z[n], d\Gamma, H) \to (H(o)(n), H, \texttt{reads}\ dR[i \leftarrow n])}$$

(DYN-ARRAY-ASSIGN) $$\frac{\{z \mapsto o, z' \mapsto o'\} \subseteq d\Gamma \quad H \rhd o : dT[]\texttt{<}dR\texttt{>}\#i}{(z[n] = z', d\Gamma, H) \to (o', H \cup \{o \mapsto (H(o) \cup \{n \mapsto o'\})\}, \texttt{writes}\ dR[i \leftarrow n])}$$

(DYN-INVOKE) $$\frac{H \vdash o : C\texttt{<}dR\texttt{>} \quad T_r\ m(T_x\ x)\ E\ \{\ e\ \} \in \text{def}(C) \quad (e, \{\texttt{this} \mapsto o, \text{param}(C) \mapsto dR, x \mapsto o'\}, H) \to (o'', H', dE)}{(z.m(z'), \{z \mapsto o, z' \mapsto o'\} \cup d\Gamma, H) \to (o'', H', \texttt{invokes}\ C.m\ \texttt{with}\ dE)}$$

(DYN-NEW-CLASS) $$\frac{o \notin \text{Dom}(H) \quad H' = H \cup \{o \mapsto \text{new}(C)\} \quad H' \rhd o : C\texttt{<}d\Gamma(R[:* \leftarrow \epsilon])\texttt{>}}{(\texttt{new}\ C\texttt{<}R\texttt{>}, d\Gamma, H) \to (o, H', \emptyset)}$$

(DYN-NEW-ARRAY) $$\frac{o \notin \text{Dom}(H) \quad H' = H \cup \{o \mapsto \texttt{new}(T[n])\} \quad H' \rhd o : d\Gamma(T)[]\texttt{<}d\Gamma(R[:* \leftarrow \epsilon])\texttt{>}}{(\texttt{new}\ T[n]\texttt{<}R\texttt{>}\#i, d\Gamma, H) \to (o, H', \emptyset)}$$

Figure 21. Rules for program evaluation.

(DISJOINT-LEFT-NAME) $$\frac{r \neq r' \quad \Gamma \rhd R \leq R_f : r \quad \Gamma \rhd R' \leq R_f : r'}{\Gamma \rhd R \# R'}$$

(DISJOINT-LEFT-INDEX) $$\frac{i \neq i' \quad \Gamma \rhd R \leq R_f : [i] \quad \Gamma \rhd R' \leq R_f : [i']}{\Gamma \rhd R \# R'}$$

(DISJOINT-LEFT-NAME-INDEX) $$\frac{\Gamma \rhd R \leq R_f : r \quad \Gamma \rhd R' \leq R_f : [i]}{\Gamma \rhd R \# R'}$$

(DISJOINT-RIGHT-NAME) $$\frac{r \neq r'}{\Gamma \rhd R : r \# R' : r'}$$

(DISJOINT-RIGHT-INDEX) $$\frac{i \neq i'}{\Gamma \rhd R : [i] \# R' : [i']}$$

(DISJOINT-RIGHT-NAME-INDEX) $$\frac{}{\Gamma \rhd R : r \# R' : [i]}$$

(DISJOINT-NAME) $$\frac{\Gamma \rhd R \# R'}{\Gamma \rhd R : r \# R' : r}$$

(DISJOINT-INDEX) $$\frac{\Gamma \rhd R \# R'}{\Gamma \rhd R : [i] \# R' : [i]}$$

Figure 22. Rules for disjointness of RPLs. The disjointness relation is symmetric (obvious rule omitted).

C.2 Disjointness

Figure 22 gives the rules for concluding that two static RPLs are disjoint; we extend them to dynamic RPLs as in Section B.

Definition 4 (Set interpretation of dynamic RPLs)**.** *Let* $\rhd H$ *and* $H \rhd dR$. *Then* $S(dR, H)$ *is defined as follows: (1)* $S(dR_f, H) = \{dR_f\}$*; (2)* $S(dR : r, H) = \{dR_f : r | dR_f \in S(dR, H)\}$*; (3)* $S(dR : [n], H) = \{dR_f : [n] | dR_f \in S(dR, H)\}$*; and (4)* $S(dR : *, H) = \{dR_f | H \rhd dR_f \leq dR\}$.

Definition 5 (Region of a field or array cell)**.** *If* $H \rhd o : C\texttt{<}dR\texttt{>}$ *and* $T\ f\ \textit{in}\ R_f \in \textit{def}(C)$*, then* $\textit{region}(o, f, H) = R_f[\textit{this} \leftarrow o][\textit{param}(C) \leftarrow dR]$. *If* $H \rhd o : dT[]\texttt{<}dR\texttt{>}\#i$*, then* $\textit{region}(o, n, H) = dR[i \leftarrow n]$.

(NI-READ) $$\frac{}{\Gamma \rhd \texttt{reads}\ R \# \texttt{reads}\ R'}$$

(NI-READ-WRITE) $$\frac{\Gamma \rhd R \# R'}{\Gamma \rhd \texttt{reads}\ R \# \texttt{writes}\ R'}$$

(NI-WRITE) $$\frac{\Gamma \rhd R \# R'}{\Gamma \rhd \texttt{writes}\ R \# \texttt{writes}\ R'}$$

(NI-INVOKES-1) $$\frac{\Gamma \rhd E \# E'}{\Gamma \rhd \texttt{invokes}\ C.m\ \texttt{with}\ E \# E'}$$

(NI-INVOKES-2) $$\frac{m\ \texttt{commuteswith}\ m' \in \text{def}(C)}{\Gamma \rhd \texttt{invokes}\ C.m\ \texttt{with}\ E \# \texttt{invokes}\ C.m'\ \texttt{with}\ E'}$$

(NI-EMPTY) $$\frac{}{\Gamma \rhd \emptyset \# E}$$

(NI-UNION) $$\frac{\Gamma \rhd E \# E'' \quad \Gamma \rhd E' \# E''}{\Gamma \rhd E \cup E' \# E''}$$

Figure 23. The noninterference relation on effects. Noninterference is symmetric (obvious rule omitted).

Proposition 1 (Disjointness of region sets)**.** *If* $H \rhd dR \# dR'$*, then* $S(dR, H) \cap S(dR', H) = \emptyset$.

Proposition 2 (Distinctness of disjoint regions)**.** *If* $H \rhd \textit{region}(o, f, H) \# \textit{region}(o', f', H)$*, then either* $o \neq o'$ *or* $f \neq f'$*; and if* $H \rhd \textit{region}(o, n, H) \# \textit{region}(o', n', H)$*, then either* $o \neq o'$ *or* $n \neq n'$.

C.3 Noninterference of Effect

Figure 23 gives the noninterference relation on static effects. We extend this relation to dynamic effects as in Section B.

Theorem 2 (Soundness of noninterference)**.** *If* $\Gamma \rhd e : T, E$ *and* $\Gamma \rhd e' : T', E'$ *and* $\Gamma \rhd E \# E'$ *and* $H \rhd d\Gamma \leq \Gamma$ *and* $(e, d\Gamma, H) \to (o, H', dE)$ *and* $(e', d\Gamma, H') \to (o', H'', dE')$*, then there exists* H''' *such that* $(e', d\Gamma, H) \to (o', H''', dE')$ *and* $(e, d\Gamma, H''') \to (o, H'', dE)$.

The Tasks with Effects Model for Safe Concurrency

Stephen T. Heumann Vikram S. Adve Shengjie Wang
University of Illinois at Urbana-Champaign
{heumann1,vadve,wang260}@illinois.edu

Abstract

Today's widely-used concurrent programming models either provide weak safety guarantees, making it easy to write code with subtle errors, or are limited in the class of programs that they can express. We propose a new concurrent programming model based on *tasks with effects* that offers strong safety guarantees while still providing the flexibility needed to support the many ways that concurrency is used in complex applications. The core unit of work in our model is a dynamically-created task. The model's key feature is that each task has programmer-specified *effects*, and a run-time scheduler is used to ensure that two tasks are run concurrently only if they have non-interfering effects. Through the combination of statically verifying the declared effects of tasks and using an effect-aware run-time scheduler, our model is able to guarantee strong safety properties, including data race freedom and atomicity. It is also possible to use our model to write programs and computations that can be statically proven to behave deterministically. We describe the tasks with effects programming model and provide a formal dynamic semantics for it. We also describe our implementation of this model in an extended version of Java and evaluate its use in several programs exhibiting various patterns of concurrency.

Categories and Subject Descriptors D.3.2 [*Software*]: Language Classifications—Concurrent, distributed, and parallel languages; D.3.3 [*Software*]: Language Constructs and Features—Concurrent Programming Structures; D.1.3 [*Software*]: Concurrent Programming

General Terms Languages, Verification, Design, Performance

Keywords Tasks, effects, task scheduling, concurrent and parallel programming, task isolation, data race freedom, atomicity, determinism

1. Introduction

Concurrency is used for many purposes in modern programs. To exploit the full capabilities of today's multicore processors, parallel algorithms must be used. But concurrency is also used for other purposes. This is perhaps particularly true of interactive programs, both on end-user devices and servers, where the behavior of the user or client is inherently concurrent with the program. In GUI programs, long-running operations should be run concurrently with user interface event processing in order to preserve responsiveness. It can also be convenient to express a full program as a set of of modules or actors [3] that can operate concurrently and communicate with each other. This can be a natural fit, for example, to the model-view-controller design of interactive programs.

Large programs often combine multiple types of concurrency. For example, an interactive application may separate long computations from the UI thread or use multiple concurrent modules, but also sometimes perform data-parallel computations. We believe a widely-applicable concurrent programming model should seek to support all of these forms of concurrency, since they are all widely used and are often combined within a single application.

Today, parallel and concurrent programs are commonly written using threads, with low-level mechanisms such as locks used for synchronization (or with carefully designed lock-free data structures). Such a programming model is flexible enough to express many forms of concurrency, but it does not guarantee any safety properties such as data race freedom, atomicity, deadlock-freedom, or determinism. It also provides little well-defined structure for the concurrent control flow and synchronization in programs, making it difficult to reason about them manually or automatically. In addition, complicated low-level details such as processor memory models [2] can affect the semantics of programs written in this style, further complicating the task of reasoning about them.

Many previous systems have attempted to address aspects of these problems. Some offer more structured parallel control and synchronization constructs, but sometimes with limitations that prevent them from expressing general, event-driven concurrency, and often without strong safety guarantees. Cilk [11] and Thread Building Blocks (TBB) [22], for example, provide structured parallelism constructs, but they do not offer checked guarantees of strong safety properties such as data race freedom.

Some other systems do seek to offer stronger guarantees. The Deterministic Parallel Java (DPJ) language [13, 14] offers a strong set of guarantees for programs that can be expressed in it. These include data race freedom, strong atomicity [1], deadlock freedom, and deterministic semantics with full sequential equivalence for parallel computations that do not explicitly use nondeterministic parallel constructs. These guarantees are very strong, but DPJ's parallelism model does not provide the flexibility that we seek. Most critically, DPJ is restricted to fork-join parallelism structures, which are not suitable for many concurrent programs.

In this paper, we propose a new model for concurrent programming, which gives strong safety guarantees while providing the flexibility needed to express a wide range of concurrent programs in it. We call our model *tasks with effects*.[1] It uses tasks that can execute concurrently as the fundamental units of work. Tasks are lighter-weight constructs than threads and support only limited operations for inter-task communication and synchronization. Concurrent work is launched by creating a new task, and it is possi-

[1] Two workshop papers gave preliminary descriptions of the tasks with effects model [20, 21].

PPoPP'13, February 23–27, 2013, Shenzhen, China.
Copyright © 2013 ACM 978-1-4503-1922/13/02. . . $10.00

ble for one task to await the completion of another. A scheduler is responsible for executing tasks in an efficient manner. Tasks provide a structured mechanism for concurrent control flow, while still preserving the flexibility to express a wide variety of concurrency patterns and parallel algorithms.

Several existing systems support task-based programming models, including Intel's TBB, Apple's Grand Central Dispatch and operation queues [6], Microsoft's Task Parallel Library in .NET [30], the `ForkJoinTask` framework in Java 7 [33], and the tasking operations in OpenMP 3.x [32]. However, they do not offer strong safety guarantees. It is possible for two concurrent tasks to perform conflicting accesses that give rise to data races or violations of intended atomicity properties, and it is generally the programmer's responsibility to manually reason that such accesses do not occur or are benign, or else to protect them using low-level synchronization mechanisms such as locks.

We propose instead to associate a checked effect specification with each task. The run-time system then schedules tasks so as to ensure that only tasks with non-interfering effects can run concurrently. Effect specifications can take many forms, but in this work we adopt the statically-checked effect system developed for Deterministic Parallel Java [13]. In this system, the compiler statically verifies that the memory accesses in each task or method are covered by its programmer-specified effects. By combining these static checks with our dynamic effect-based task scheduling system, we are able to guarantee the basic *task isolation* property that no two tasks with interfering effects may run concurrently with each other. This guarantee leads to a guarantee of data race freedom, and to a guarantee of atomicity for tasks or portions of tasks that do not create or wait for any other tasks.

We also define mechanisms based on *effect transfer* between tasks to further enhance the utility of our model. One mechanism is used to avoid a class of deadlocks, and also enables certain useful programming paradigms. Another form of effect transfer is used for nested parallel computations. It enables us to provide a compile-time guarantee of determinism for a class of deterministic programs and algorithms similar to those supported by DPJ. We are aware of no other programming model which provides equally strong safety guarantees while supporting the flexible control flow needed for general concurrent programs such as interactive applications and actor-like programs.

This paper makes the following contributions:

- We define the tasks with effects programming model, which supports flexible task-based concurrency while providing a strong set of safety guarantees.
- We describe the TWEJava language which implements this model, and describe our compiler and runtime system for it.
- We provide a formal dynamic semantics of tasks with effects, and describe how it guarantees task isolation, data race freedom, atomicity, and (for certain computations) determinism.
- We evaluate the expressiveness and performance of our language and implementation. We show that TWEJava can be used to write a variety of concurrent and parallel programs, including two interactive applications, and that we can achieve substantial parallel speedups.

The rest of this paper proceeds as follows. Section 2 presents the TWEJava language and describes the task-related operations used in it. Section 3 gives a dynamic semantics of tasks with effects. Section 4 describes the safety properties of our model. Section 5 discusses our implementation of TWEJava in a compiler and runtime system, and section 6 evaluates it on several benchmark programs. Finally, section 7 discusses related work and section 8 concludes.

```
public abstract class Task<type TRet, TArg, effect E> {
  // Code to be run when task is executed.
  public abstract TRet run(TArg arg) effect E;

  // Execute a task at some point in the future
  public final TaskFuture<TRet> executeLater(TArg arg);
  // Spawn a subtask of the current task, with effect transfer
  public final SpawnedTaskFuture<TRet, effect E> spawn(TArg arg);
}

public class TaskFuture<type TReturn> {
  // Await completion and get return value (no effect transfer)
  public TReturn getValue();
  // Check if task is done
  public boolean isDone();
}

public class SpawnedTaskFuture<type TReturn, effect E>
      extends TaskFuture<TReturn> {
  // Await completion and get return value, with effect transfer
  public TReturn join();
}
```

Figure 1. Operations supported by TWEJava. The abstract method `run` must be implemented in concrete subclasses of `Task`, giving the code to be run as a task. The other operations, although using the syntax of Java methods, are in fact new task-related language operations supported by our compiler and runtime system.

2. The TWEJava language

We implement the tasks with effects model for safe, flexible concurrency in an extended version of Java, which we call TWEJava. (TWEJava programs can use almost all Java language features, but they should not use Java's thread-based concurrency mechanisms or lock-based synchronization, which TWEJava is designed to replace.) Figure 1 shows the new operations supported by TWEJava.

Figure 2 shows how our task system can be used in an image editing program, which we will use as as running example. It illustrates a simplified version of a programming pattern used in the ImageEdit program that we have implemented in TWEJava (see section 6). The example code shows a class `Image` representing an image, with the pixel values held in two arrays, `topHalf` and `bottomHalf`. We would like to support operations in parallel on these two halves of the image. (We adopt this arrangement for simplicity. In the actual ImageEdit application, it is possible to use finer-grained parallelism.) We also want to support a variety of operations to read and manipulate the image, which may be invoked as asynchronous tasks. This is useful, for example, when the user directs the program to perform a lengthy operation that should not block the user interface while it runs.

We show the task `increaseContrast` (lines 6–16), which can be executed to increase the contrast of the image. It relies on the separate method `increasePixelContrast` (lines 18–26) to actually update the pixel values in each array. This enables the `increaseContrast` operation to work on the top and bottom halves of the image in parallel, by spawning a child task to work on the top half while the parent task works on the bottom half.

Figure 3 shows the tasks created in this computation. The GUI system executes the `increaseContrast` task in response to user input. That task in turn spawns a child task so that the two halves of the image can be processed in parallel, and then joins that child task after it completes. Meanwhile, the GUI system might execute additional tasks in response to further user input. (In this example, we show the GUI system as a task, responsible for processing low-level input data and launching tasks in response to UI events. This architecture would be possible, but for ease of implementation we have so far used Java's Swing GUI framework, with wrappers to launch tasks in response to Swing events.)

```
class Image {
  region Top, Bottom;
  final int[]<Top> topHalf;        // pixel values
  final int[]<Bottom> bottomHalf;
  ...
  public final Task<Void, Void, writes Top,Bottom>
  increaseContrast =
      new Task<>() {
        public Void run(Void _) {
          SpawnedTaskFuture<Void, writes Top> f =
              increasePixelContrast(topHalf).spawn(null);
          increasePixelContrast(bottomHalf).run(null);
          f.join();
          return null;
        }
      };

  private static <region runtime R> Task<Void, Void, writes R>
  increasePixelContrast(final int[]<R> pixels) pure {
    return new Task<>() {
      public Void run(Void _) {
        modify values in pixels array
        return null;
      }
    };
  }
}
```

Figure 2. Example computation.

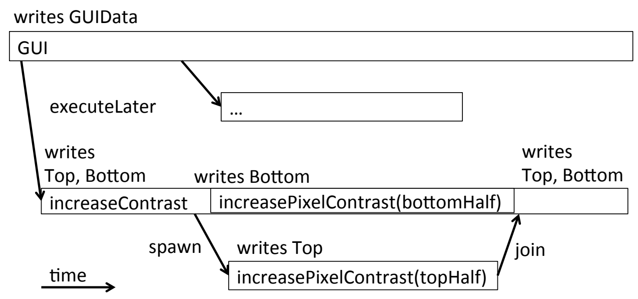


Figure 3. Tasks in example computation.

2.1 Tasks

In TWEJava, potentially concurrent work is made by creating a *task*, which will then be executed at some point when execution resources are available. It is possible to check if a task is completed or block awaiting its completion. A program written in TWEJava is started by invoking an initial task, and creating new tasks is the sole means of performing concurrent work. Tasks can also take parameters as input and return a result. (Wrapper classes support tasks with multiple parameters.) Three fundamental operations implement this basic tasking model: `executeLater` adds a task to the queue of tasks to be executed, `getValue` waits until a task is done and gives its return value, and `isDone` checks whether a task is done, without blocking.

Each type of task is specified by a subclass of the `Task` class, which takes type parameters giving its input and output types, and an effect parameter giving its effect (described below). An `executeLater` operation performed on a `Task` instance returns a `TaskFuture` object, which represents an actual execution of the task; `getValue` and `isDone` operations can be performed on this task future. In our example, the `increaseContrast` task is created by an `executeLater` operation in the GUI. (The `spawn` and `join` operations used within it will be described in section 2.5.)

The structure described above is similar to other existing task systems, but TWEJava has a key difference. In other systems, a task can generally be run at any time after it is queued for execution, without regard to what other tasks are running concurrently. Because of this, the programmer must take care to ensure that there are no data races between potentially concurrent tasks. This can be done by using synchronization mechanisms such as locks within tasks to guard access to shared data, or by carefully designing the pattern in which tasks are executed and joined in such a way that no two tasks accessing the same data might be executed concurrently. Using these mechanisms to guard against data races is often complex and error-prone, and traditional thread-based systems for concurrent programming generally do not provide a mechanism to automatically check that they have been used correctly.

Our system solves this problem by using *effects* to control the scheduling of tasks. Each task has an effect specification, which is checked at compile time to ensure that it accurately (conservatively) reflects the task's memory accesses. These effect specifications of tasks are in turn used at run time by the task scheduler, which will ensure task isolation—that is, that no two tasks with interfering effects can be running concurrently.

2.2 Effects and Regions

In order to perform effect-based scheduling of tasks, our system must know the effects of each task, and be able to check whether the effects of two different tasks interfere with each other. Intuitively, two tasks interfere if they could both access the same memory location and at least one of those accesses could be a write. Two tasks can only be run concurrently if their effects do not interfere, which is the core property enforced by the scheduler in our system.

In TWEJava, we use the effect system originally developed for the Deterministic Parallel Java (DPJ) language [13]. DPJ is an extended version of Java that uses type and effect annotations to enable the compiler to statically prove strong safety properties for programs written using its fork-join parallel constructs. In this work, however, we adopt its type and effect system for use in combination with our effect-based task scheduling system.

The DPJ type and effect system is based on a partitioning of memory into *regions*. The programmer can declare each object field and array cell to be in a specified region. Region-parameterized types and methods are also supported. This permits different instances of a class to have their fields in different regions by giving different region arguments when instantiating the class. In addition, nested hierarchies of regions are supported by using *region path lists* (*RPLs*), and *index-parameterized arrays* allow each element of an array to be placed in a distinct region. A wildcard `*` can be used in RPLs to specify effects covering a set of regions.

Using this partitioning of memory into regions, the effects of any operation in the program can be specified in terms of read and write effects on memory regions. The programmer declares the effects of each method as part of its method signature. The compiler can then statically verify that the declared effects of each method actually cover the effects of every operation in it. The DPJ type and effect system also defines formally under what circumstances two effects can be proven to be non-interfering. In DPJ, this information is used purely statically to verify that programs using simple fork-join parallelism constructs have no interference of effect between portions of code that can run concurrently.

TWEJava adopts DPJ's region-based type and effect system, but couples it with a runtime representation of effects that is used by a run-time scheduler to guarantee noninterference of effect between concurrent tasks. This allows it to support a much wider range of programs than DPJ can handle, including those that are inherently nondeterministic and do not use a fork-join style of concurrency.

We also use an extension of the basic DPJ type system to support effect parameters to types (in addition to region parameters), which was introduced in [12]. This allows us to use an effect parameter E in our definition of the abstract class Task, which will be extended by each actual task defined in the user's code. The definition of each actual type of task will instantiate this parameter with that task's effects, and the compiler will then be able to ensure statically that the effects of the supplied run method for that task are actually covered by the effect parameter E. Thus, our runtime system can safely use that effect parameter as a (possibly conservative) summary of the actual effects of the task.

In our example code, we declare two region names, Top and Bottom (line 2). We then declare the cells of the topHalf and bottomHalf arrays to be in those two regions, respectively. The increaseContrast task is declared with the effects writes Top, Bottom, meaning it can read and write the pixel values in both halves of the image. The increasePixelContrast method has a region parameter R corresponding to the region containing the cells of the array passed to it. Since the declared effect of the task it returns is writes R, increasePixelContrast(topHalf) will produce a task with the effect writes Top.

Like DPJ, we use purely static checks to ensure that each method and task complies with its effect declaration and that region- and effect-parameterized types are used soundly. TWEJava never requires runtime checks associated with individual memory accesses, which avoids a major source of overhead in some other systems such as STMs. However, our system does need to have information on the effects of tasks available at run time so that it can be used by the scheduler. We make this information available by introducing a set of internal runtime classes that represent dynamic regions and effects, and internally adding extra fields to classes which hold the runtime instantiation of their region and effect parameters, as well as extra arguments to constructors and methods corresponding to the region and effect parameters passed to them.

The scheduler only directly needs information about the effect parameters of task objects, but these may depend on other region and effect parameters in scope at the places where task classes are declared and instantiated, making it necessary to also track those additional parameters at run time. To minimize the overhead of this run-time tracking, we require programmers to annotate region parameters that need to be tracked at run time. This allows us to avoid generating run-time tracking code for the many region parameters that are used only in the compiler's static analysis. (Failing to provide such an annotation where needed will cause a compile-time error.)

2.3 Effect-Based Task Scheduling

The key property that our run-time task scheduler must enforce is that two tasks with interfering effects will not be run concurrently. To do this, the scheduler will have to delay the execution of tasks that are created while another task with interfering effects is already executing. It may also delay tasks for other reasons, e.g. waiting until execution resources are available.

In Figure 3, the increaseContrast task with effects writes Top, Bottom is run while the GUI task with effect writes GUIData continues to execute. To determine if the new task may be run concurrently with the already-executing task, the scheduler will check if these two sets of effects interfere with each other. In this case, the region GUIData is disjoint from Top and Bottom, so the two tasks have non-interfering effects and may be run concurrently.

If a third task is run with executeLater while these two tasks are executing, its effects will be checked against those of both existing tasks. Thus, another task trying to access the image data in the regions Top and Bottom would have to wait until the increaseContrast task is done, but a task accessing different regions might be able to run concurrently. (The increasePixelContrast(topHalf) task is run with the spawn operation, which uses effect transfer to avoid the need for these run-time checks; see section 2.5.)

Considerable variation is possible in the design of an effect-aware task scheduler. Our initial prototype implementation uses an approach based on a linear queue of tasks, which is described in section 5.2. For greater performance and scalability, the effect checking could be structured around regions, so that tasks accessing unrelated regions do not need to be explicitly checked against each other. A scheduler may also provide additional properties related to fairness or task ordering, in addition to the basic property of noninterference. For interactive programs, it is valuable to preserve responsiveness through fairness properties that avoid delaying the execution of one task excessively while other tasks execute ahead of it. But for efficiency in many parallel codes, it would be desirable to use algorithms similar to Cilk's work-stealing scheduler [11], which preferentially execute recently-created tasks on each processor. We believe that the design of high-performance effect-based task schedulers is a valuable area for future work.

2.4 Effect Transfer When Blocked

The model we have described so far envisions the effects of each task remaining unchanged while it runs, and says that two tasks with interfering effects may not run concurrently. This will lead to deadlock if one task blocks waiting for another task that has yet to run and which has effects that interfere with those of the first task. For example, if task A creates task B using executeLater, then blocks on B using getValue, and the effects of tasks A and B interfere, deadlock results because B cannot begin execution until A is complete.

We wish to prevent this form of deadlock and enable certain useful programming patterns involving this sort of blocking, so we introduce a mechanism for *effect transfer* from a blocked task to the task it is blocked on. The key idea is that a getValue or a join operation (described later) "transfers" enough effects from the blocking task to the target task to allow the target task to begin execution. For example, if a task A is blocked on another task B using getValue, we record this fact and ignore any effect conflict between A and B in deciding whether B can be executed. We also extend this to indirect blocking through chains of blocking operations. Note that a task that blocks will always remain blocked until all the tasks it directly or indirectly blocks on are done. Therefore, this mechanism does not enable two tasks with conflicting effects to be actively running at the same time.

This form of effect transfer prevents the type of deadlock described above, and it also allows some useful programming patterns. One of these is for one module of the program with effects on a certain region to launch and block on a task in another module, which may "call back" to the first module by launching and blocking on another task whose effects interfere with those of the first task. Another useful programming pattern enabled by this mechanism is similar to a locked or atomic block in other programming models. One task can launch a second task with a superset of its effects, and then use a getValue operation to wait for the second task. This transfers the first task's effects to the second task (allowing it to access the same regions as the first task), and leaves the second task to wait until it can acquire access to the regions covered by its other effects, which may correspond to a shared resource.

2.5 Effect Transfer for Nested Parallelism

Our system supports an additional form of effect transfer which is particularly suitable for nested parallelism, as used in fork-join style computations. It is a mechanism to transfer some of the effects of a parent task to a newly-created child task, and later transfer those effects back to the parent task when the child task completes. We call these operations spawn and join, respectively. A child task created with spawn may run immediately, since "ownership" of its effects is transferred directly from the parent to the child task, and thus no other tasks with conflicting effects may be running concurrently.

In Figure 2, these mechanisms are used to operate in parallel on the two halves of the image. We use the spawn operation to run the increasePixelContrast(topHalf) task (line 11). This transfers the effect writes Top directly from the parent increaseContrast task to the new child task, which means the new task can be enabled for execution immediately. The parent task also continues executing concurrently, with its remaining effect writes Bottom. The increasePixelContrast(bottomHalf) operation is run as a method within the parent task, which is possible since its remaining effect writes Bottom covers the effect of the method call. After that computation finishes, the parent task joins the spawned child task. This join operation also transfers the child task's effect writes Top back to the parent task. After this, both halves of the image will have been updated, so any other task that waits for the increaseContrast task to finish will know that the full operation is complete.

2.5.1 Spawning and joining child tasks

The spawn operation executes a new task, whose effects must be entirely covered by the effects of the parent task calling spawn. It immediately transfers those effects to the spawned task, which allows that task to be enabled for execution immediately, without going through the normal effect-based scheduling process required when using executeLater. Since the effects are transferred directly from the parent task to the child task, data in regions covered by those effects cannot be modified by any other task in the interim, so the child task reading that data is guaranteed to see the values last seen or written by the parent task.

The join operation permits effect transfer back to the parent task at the end of a child task. Apart from effect transfer, join behaves like getValue: it will await the completion of the joined task, and return the result value produced by it, if any. The difference is that join will transfer effects directly from a completed task to the task that joins it. This permits the task that called join to perform subsequent operations covered by the effects of the joined task. One application of this is that data written by the completed child task can be read by its parent task after the child task is done.

Only tasks executed with spawn are joinable, and this is reflected by the fact that spawn returns a SpawnedTaskFuture, which supports the join operation. Furthermore, only the parent task that spawns a task may join it, and a task may only be joined once (violating these rules causes an exception to be thrown). Also, the system implements an implicit join operation prior to returning from each method for all the tasks spawned by that method that have not already been explicitly joined. These measures ensure that all spawned tasks get joined, and that all the effects transferred from a method with spawn are returned to it through join operations before the method returns. This simplifies our static effect analysis, since a method never "gives up" effects from the perspective of its callers.

2.5.2 Covering Effect Analysis for Effect Transfer

Implementing effect transfer makes the static analysis of covering effects more complex, since a spawn or join operation subtracts or adds effects to the task in which it is executed, thereby changing the covering effects applicable to subsequent code in that task. This would be easy to address if we used dynamic checks to determine whether the effect of each memory operation is covered by the current effects of the task in which it appears, but we want to use a static analysis to determine this in order to minimize runtime overheads and detect as many errors as possible at compile time.

To do so, we added a dataflow analysis algorithm in the compiler to conservatively compute the *current covering effect* applicable to each expression in the program. The current covering effect at the beginning of a method is given by its declared method effect summary. When spawn operations are encountered, the statically declared effects of the spawned task are subtracted from the current covering effect, and a SpawnedTaskFuture parameterized by the effects of the spawned child task is returned. At join operations, the effects given by the static type of the joined SpawnedTaskFuture are added to the current covering effect. At control flow join points, a minimum of the current covering effects from the different control flow paths is used. Using an iterative dataflow analysis, we can thus conservatively compute the current covering effect applicable to each expression in the program. The effects of each expression can then be compared against its current covering effect to ensure the expression's effects will be covered.

In our example the covering effect of the increaseContrast task is initially writes Top, Bottom. When it spawns a child task (line 11), its covering effect then becomes writes Bottom, since the writes Top effect has been transferred to the spawned child task. When that task is joined (line 13), the covering effect of the parent task once again becomes writes Top, Bottom.

One detail that must be accounted for in this analysis is that effect-parameterized types in the static program code are in general only a conservative summary of the actual effects of tasks at run time, and they may contain wildcard elements in their region specifiers. The actual effects of the Task object used at run time may be smaller than the effects given in the static type, e.g. by omitting some of the effects that are included in the static type or replacing effects on RPLs containing wildcards (which can cover a set of regions) with effects on a fully-specified RPL designating a single region in that set. We generally use a conservative static analysis: spawns are treated as transferring away all the effects in the static type of the spawned task, including ones with wildcards. Subsequent operations in the parent task may not interfere with those transferred-away effects, which conservatively ensures that they cannot interfere with any of the actual effects of the child task at run time.

As an exception in this conservative analysis, however, we allow spawn operations even if we cannot be certain at compile time whether or not the effects of the spawned task will actually be covered at run time. In this case, we generate code to keep track of the run-time covering effects in the method containing the spawn operation (updated only when a spawn or join operation is performed). An exception will be thrown if the effects of the spawned task are not actually covered at run time. This limited dynamic checking is useful for cases where we do not have full information on the effects of spawned tasks at compile time. For example, a loop may spawn tasks to operate on different elements of an index-parameterized array, but our compiler cannot determine statically whether each of the elements is distinct, so this mechanism effectively enables the check to be performed dynamically instead.

For joins, we need to be sure that the actual run-time effects of the task being joined are not less than those specified in the static type. To do this, we statically treat joins as performing effect transfer only if the effect parameter of the joined task's static type is fully-specified (i.e. contains no wildcards). We also adopt the typing rule that an effect-parameterized type A is only treated as a subtype of another effect-parameterized type B if either the corresponding effect parameters are exactly equivalent, or the effect parameters in B are *not* fully-specified. This ensures that fully-specified effect parameters in the static types of SpawnedTaskFutures exactly match the actual parameters used when instantiating the task object at run time, so we may safely use those parameters in the static analysis of join operations.

CONFIGURATION:

$$\langle\langle\langle \$PGM \curvearrowright \texttt{execute}\rangle_{\mathsf{k}}\ \langle 0\rangle_{\mathsf{id}}\ \langle\cdot\rangle_{\mathsf{env}}\ \langle\cdot\rangle_{\mathsf{spawned}}\rangle_{\mathsf{task*}}\ \langle\cdot\rangle_{\mathsf{running}}\ \langle\cdot\rangle_{\mathsf{waiting}}\ \langle\cdot\rangle_{\mathsf{genv}}\ \langle\cdot\rangle_{\mathsf{store}}\ \langle 1\rangle_{\mathsf{nextLoc}}\rangle_{\mathsf{T}}$$

RULE EXECUTELATER

$$\left\langle \frac{(\lambda XTs\,.\,S) : (Tt\ \texttt{->}\ T)\mathit{Eff}.\texttt{executeLater}(Vs)}{\texttt{loc}(L)} \cdots\right\rangle_{\mathsf{k}} \left\langle \frac{L}{L +_{\mathit{Int}} 1}\right\rangle_{\mathsf{nextLoc}} \left\langle \cdots \frac{\cdot}{L \mapsto \mathit{TF}(\mathit{Eff}, \texttt{bindto}(XTs, Vs) \curvearrowright S, \bot_T)} \cdots\right\rangle_{\mathsf{store}} \left\langle \cdots \frac{\cdot}{L} \cdots\right\rangle_{\mathsf{waiting}}$$

RULE START-TASK

$$\left\langle \cdots \frac{L}{\cdot} \cdots\right\rangle_{\mathsf{waiting}} \langle \cdots L \mapsto \mathit{TF}(\mathit{Eff}, K, \_) \cdots\rangle_{\mathsf{store}} \left\langle R\ \frac{\cdot}{(L, \mathit{Eff}, \varnothing)}\right\rangle_{\mathsf{running}} \frac{\cdot}{\langle \cdots \langle K \curvearrowright \texttt{return nothing;}\rangle_{\mathsf{k}}\ \langle \mathit{GEnv}\rangle_{\mathsf{env}}\ \langle L\rangle_{\mathsf{id}} \cdots\rangle_{\mathsf{task}}} \langle \mathit{GEnv}\rangle_{\mathsf{genv}}$$

when $\forall (L_2, \mathit{Eff}_2, B) \in R : \mathit{Eff} \# \mathit{Eff}_2 \vee L \in B$

RULE SPAWN

$$\left\langle \frac{(\lambda XTs\,.\,S) : (Tt\ \texttt{->}\ T)\mathit{Eff}.\texttt{spawn}(Vs)}{\texttt{loc}(L)} \cdots\right\rangle_{\mathsf{k}} \left\langle \cdots \frac{\cdot}{L} \cdots\right\rangle_{\mathsf{spawned}} \left\langle \cdots \frac{\cdot}{L \mapsto \mathit{TF}(\mathit{Eff}, \cdot, \bot_T)} \cdots\right\rangle_{\mathsf{store}} \left\langle \frac{L}{L +_{\mathit{Int}} 1}\right\rangle_{\mathsf{nextLoc}}$$

$$\frac{\cdot}{\langle \cdots \langle \texttt{bindto}(XTs, Vs) \curvearrowright S \curvearrowright \texttt{return nothing;}\rangle_{\mathsf{k}}\ \langle \mathit{GEnv}\rangle_{\mathsf{env}}\ \langle L\rangle_{\mathsf{id}} \cdots\rangle_{\mathsf{task}}} \left\langle \cdots \frac{\cdot}{(L, \mathit{Eff}, \varnothing)} \cdots\right\rangle_{\mathsf{running}} \langle \mathit{GEnv}\rangle_{\mathsf{genv}}$$

RULE GETVALUE-SUCCEEDS

$$\left\langle \frac{\texttt{loc}(L).\texttt{getValue}()}{V} \cdots\right\rangle_{\mathsf{k}} \langle L_1\rangle_{\mathsf{id}} \langle \cdots L \mapsto \mathit{TF}(\_, \_, V) \cdots\rangle_{\mathsf{store}} \left\langle \cdots (L_1, \_, \frac{\_}{\varnothing}) \cdots\right\rangle_{\mathsf{running}}$$

RULE JOIN-SUCCEEDS

$$\left\langle \frac{\texttt{loc}(L).\texttt{join}()}{V} \cdots\right\rangle_{\mathsf{k}} \langle L_1\rangle_{\mathsf{id}} \left\langle \cdots \frac{L}{\cdot} \cdots\right\rangle_{\mathsf{spawned}} \langle \cdots L \mapsto \mathit{TF}(\_, \_, V) \cdots\rangle_{\mathsf{store}} \left\langle \cdots (L_1, \_, \frac{\_}{\varnothing}) \cdots\right\rangle_{\mathsf{running}}$$

RULE GETVALUE-BLOCKS

$$\langle \texttt{loc}(L).\texttt{getValue}() \cdots\rangle_{\mathsf{k}} \langle L_1\rangle_{\mathsf{id}} \left\langle \cdots (L_1, \_, \frac{\varnothing}{\{L\}}) \cdots\right\rangle_{\mathsf{running}}$$

RULE JOIN-BLOCKS

$$\langle \texttt{loc}(L).\texttt{join}() \cdots\rangle_{\mathsf{k}} \langle L_1\rangle_{\mathsf{id}} \left\langle \cdots (L_1, \_, \frac{\varnothing}{\{L\}}) \cdots\right\rangle_{\mathsf{running}}$$

RULE INDIRECT-BLOCKING

$$\left\langle \cdots (L, \_, ts_2)\ (\_, \_, \frac{ts_1}{ts_2 \cup ts_1}) \cdots\right\rangle_{\mathsf{running}}$$

when $(L \in ts_1) \wedge_{\mathit{Bool}} (ts_2 \not\subseteq ts_1)$

RULE RETURN

$$\left\langle \frac{\texttt{return}\ V\texttt{;} \curvearrowright \_}{\texttt{awaitSpawned} \curvearrowright (\texttt{setRetVal}\ V) \curvearrowright \texttt{done}}\right\rangle_{\mathsf{k}}$$

RULE SET-RETURN-VALUE

$$\left\langle \frac{\texttt{setRetVal}\ V}{\cdot} \cdots\right\rangle_{\mathsf{k}} \langle L\rangle_{\mathsf{id}} \left\langle \cdots L \mapsto \mathit{TF}(\_, \_, \frac{\_}{V}) \cdots\right\rangle_{\mathsf{store}}$$

RULE AWAIT-SPAWNED

$$\left\langle \frac{\texttt{awaitSpawned}}{((\texttt{loc}(L).\texttt{join}())\ \texttt{;}) \curvearrowright \texttt{awaitSpawned}} \cdots\right\rangle_{\mathsf{k}} \langle \cdots L \cdots\rangle_{\mathsf{spawned}}$$

RULE AWAIT-SPAWNED-DONE

$$\left\langle \frac{\texttt{awaitSpawned}}{\cdot} \cdots\right\rangle_{\mathsf{k}} \langle \cdot\rangle_{\mathsf{spawned}}$$

RULE DONE

$$\frac{\langle \cdots \langle \texttt{done}\rangle_{\mathsf{k}}\ \langle L\rangle_{\mathsf{id}} \cdots\rangle_{\mathsf{task}}}{\cdot} \left\langle \cdots \frac{(L, \_, \_)}{\cdot} \cdots\right\rangle_{\mathsf{running}}$$

RULE ISDONE-TRUE

$$\left\langle \frac{\texttt{loc}(L).\texttt{isDone}()}{\mathsf{true}} \cdots\right\rangle_{\mathsf{k}} \langle \cdots L \mapsto \mathit{TF}(\_, \_, V) \cdots\rangle_{\mathsf{store}}$$

RULE ISDONE-FALSE

$$\left\langle \frac{\texttt{loc}(L).\texttt{isDone}()}{\mathsf{false}} \cdots\right\rangle_{\mathsf{k}} \langle \cdots L \mapsto \mathit{TF}(\_, \_, \bot_T) \cdots\rangle_{\mathsf{store}}$$

Figure 4. Dynamic semantics of tasks with effects.

3. Dynamic Semantics of Tasks with Effects

We have formalized the dynamic semantics for the core operations of the tasks with effects model in the context of a basic imperative language. A program in this language consists of a set of global variable declarations and task declarations (which are similar to function declarations in a traditional language, but include an effect specification for each task). Here we present and describe only those semantic rules related to tasks, which are shown in Figure 4.

These rules are written using the K semantic framework [36], which is based on rewriting logic and operates on a configuration of nested cells which corresponds at any point to the current state of the execution. (Although the K framework is less common than the standard approach for operational semantics, it has significant advantages, especially in that it is more modular and flexible.) Each rule may apply when it can match the configuration elements on the top of it, and when it applies any elements with a horizontal line under them are replaced by what is below the line. K supports lists, sets, and maps, and a rule may match a single element from these structures, either anywhere in them or at the front of a list; in these cases, the remainder of the structure is denoted by ellipses. A dot represents the identity element of these structures, and an underline is a 'don't-care' element that can match anything. K rules also obey a locality principle, saying that a rule matching two subcells that appear within the same outer cell must match only two subcells within the same *instance* of that outer cell.

At the top of Figure 4, we show the initial configuration of the program. It consists of a *task* cell (of which there may later be more than one, indicated by the *); a *running* cell which will hold a set containing information on running tasks; a *waiting* cell which will contain a set of IDs of tasks waiting to execute; a *genv* cell holding the global environment (mapping identifiers to locations in the store); a *store* cell which will map locations (integers) to various objects; and a *nextLoc* cell giving the next available location in the store. Each *task* cell contains code to be executed in its *k* subcell; an ID in its *id* subcell (corresponding to a location in the store); the current environment in its *env* subcell; and a set of IDs of spawned child tasks in its *spawned* subcell. The initial configuration will pass the program code to a special operation `execute` (not shown) which initializes the store and global environment based on the declarations in the program and then runs the task named `main`.

Note that we present here only a dynamic semantics, which presupposes that the program has passed all static checks, including type checking and checking that the current covering effects at each point in each task correctly cover all the memory accesses it may perform. (Dynamic computations of current covering effects are not needed in this formalism, because the effects of each task are fully defined statically and there is no provision for dynamic instantiation of region or effect parameters.) These semantics are agnostic to the specific effect system used, but a formalism of the DPJ type and effect system used in TWEJava is presented in [13].

3.1 Starting Tasks

The first major class of rules in our semantics relates to starting tasks. The EXECUTELATER rule implements the `executeLater` operation. It will apply once the `executeLater` operation is the next piece of code to execute, after a task name in the code has been evaluated to a lambda expression (comparable to a `Task` object in TWEJava) and its arguments have been evaluated to values (simple rules not shown). The EXECUTELATER rule will allocate a new location L in the store, and store a *TF* tuple (corresponding to a `TaskFuture` in TWEJava). This tuple contains the effect of the task, the code to be executed when it is run, and the task's return value (initially $\bot_T$, indicating it has not yet been set). The rule adds the ID (location) of this task to the set of tasks waiting to run, and the result of the operation is a reference to that location.

The START-TASK rule is then responsible for actually starting one of the tasks in the *waiting* set. When it applies, it will create a new *task* cell in the configuration, containing the code of the new task to be run. (This cell may exist side-by-side with other *task* cells.) The rule also adds a tuple (L, *Eff*, $\varnothing$) to the *running* cell. This indicates that the task L is now running, and holds its effects and an initially-empty set of tasks that it is blocked on. Finally, the key element of this rule is the condition relating to the existing contents R of the *running* cell. This will contain information about all the other currently-running tasks, and we use it to ensure our model's basic property of task isolation. Specifically, the new task L cannot be started unless for every already-running task L_2, either the effects of L are non-interfering with those of L_2 (denoted by $\#$) or L is in the set of tasks on which L_2 is blocked. This latter case implements our mechanism for effect transfer when blocked.

The SPAWN rule is similar to a combination of the EXECUTE-LATER and START-TASK rules, since it allows a task to start immediately without the need for the effect checking in the START-TASK rule. One addition, however, is that the ID of the spawned task is added to the *spawned* set of its parent task, which keeps track of child tasks that have been spawned and not yet joined.

3.2 Awaiting Completed Tasks and Blocking

The next group of rules relates to the potentially blocking operations `getValue` and `join`. They both can be applied to a reference to a location containing a *TF* tuple. The GETVALUE-SUCCEEDS rule addresses the case where the task in question is complete, and as such has a return value V stored in its *TF* tuple. In this case, the result of the operation is that value. Since the task that executed the `getValue` operation (L_1) will no longer be blocked, we empty the blocked-on set in its *running* tuple. The JOIN-SUCCEEDS rule is similar, but also requires that the task being joined was in the current task's *spawned* set, and removes it. This reflects the fact that a task can only be joined once, and only by the task that spawned it. (If a `join` operation violates these rules, the task that executes it will hang in our formalism. In TWEJava, an exception is thrown.)

The next two rules, GETVALUE-BLOCKS and JOIN-BLOCKS, handle the case where the task L may not yet be done. These rules put L in the blocked-on set for the task L_1 that does a `getValue` or `join` operation on L. This potentially allows L to be started based on effect transfer, using the START-TASK rule. The INDIRECT-BLOCKING rule propagates entries in the blocked-on sets when there is a chain of blocked tasks, allowing effect transfer to be applied in the case of indirect blocking. (In the TWEJava implementation, this propagation is fully performed at the time a `getValue` or `join` operation is evaluated.)

3.3 Finishing Tasks and Checking If Tasks are Done

The next group of rules relates to finishing a task. The RETURN rule handles a `return` statement (which may be in the program's code, or the `return nothing;` that we insert at the end of each task when starting it, in case it does not explicitly return a value). The rule says to first await any spawned children of the current task that have not yet been joined, then set the task's return value in its *TF* tuple (which will signal that the task may be considered done), and finally erase its *task* cell and its entry in the *running* set. The next several rules implement these operations.

Finally, the last two rules implement the `isDone` operation. A task is considered done once its return value has been set to a value. If it is still undefined (indicated by $\bot_T$), then the task is not done.

4. Safety Properties

Our model guarantees strong safety properties, including our basic task isolation property, plus data race freedom and atomicity properties stemming from it. We also avoid a significant class of deadlocks and can prove that many computations are deterministic.

4.1 Task isolation

The task isolation property of our system is that no two tasks may be actively running concurrently with interfering covering effects. The basic check used to guarantee this is to record the effects of each running task in the *running* set, and compare the effects of new tasks against the effects of all existing tasks before allowing them to start in the START-TASK rule.

There are two cases where we can start tasks even though they might appear to have effects interfering with those of another running task. One is that a task A may be allowed to start while a task B with conflicting effects is in the *running* set if A is in the blocked-on set for B. In this case, our rules guarantee that B cannot resume execution until A has completed, so we allow A to run based on our first effect transfer mechanism.

The other case is the `spawn` operation. In this case, our covering effects analysis ensures that the spawned task's effects are subeffects of its parent task's effects (so they may not conflict with anything that the parent's effects do not) and that the parent task will not execute any operations that conflict with the effects of the spawned task between where it is spawned and where it is joined.

4.2 Data race freedom

Data race freedom follows from the combination of the task isolation property and the guarantee provided by our static checks that the specified effects of each task cover all its memory accesses.

The formalism in section 3 implicitly uses a sequentially-consistent memory model, but in fact the tasks with effects model requires memory updates to be visible only between operations ordered by a limited set of happens-before edges. Our model imposes some order on any two tasks with interfering effects. This gives rise to happens-before edges between the end of one task and the start of any subsequent task with interfering effects, analogous to those between a lock release and subsequent acquisition in other systems. A full happens-before relation for our model is given by the transitive closure over these edges as well as edges for task creation, waiting or checking for task completion, and the sequential program order within each task. Any two accesses to a memory location where at least one is a write will be ordered by this happens-before relation.

4.3 Atomicity

A task or portion of a task that does not create or wait for any other tasks behaves atomically. It has fixed effects that cover all the memory locations it can access, and the scheduler will ensure that no other tasks performing conflicting accesses run concurrently with it, which ensures it is atomic. This atomicity property also extends to portions of tasks that contain task creation operations, in the sense that the semantics are equivalent to those given by creating the new tasks only at the end of the parent task or just before the next `getValue` or `join` operation in it.

Atomicity does not always extend across `getValue` or `join` operations, as our mechanism for effect transfer when blocked may allow other tasks with conflicting effects to run before the blocking operation completes. However, this potential for non-atomicity is limited to running the task(s) that are directly or indirectly blocked on, and it does not occur in cases where those tasks have definitely finished prior to the `getValue` or `join` operation. Also, a deterministic computation (discussed below) effectively executes atomically, as it is semantically equivalent to a sequential execution with no task-related operations. As in languages with explicit atomic constructs, it remains the programmer's responsibility to identify sections of code that should behave atomically and write the code in a way that ensures they do so, e.g. by not using `getValue` or `join` operations within such sections.

4.4 Deadlock avoidance

Our model avoids deadlocks in the case that a task A directly or indirectly blocks on another task B whose effects conflict with A's effects, using the effect transfer mechanism discussed in section 2.4. While we do not prevent all deadlocks, we believe this class of deadlocks is significant, and we found our effect transfer mechanism to be useful in practice, particularly in the interactive FourWins program (see section 6).

4.5 Determinism

Many parallel algorithms are deterministic. That is, they always produce the same output given the same input state. Since this is an expected property of many algorithms, detecting violations of it is a useful way of finding bugs. Moreover, knowing that a program or an algorithm is deterministic makes it much easier to reason about: the user of the program or algorithm knows that it will always produce the same output given the same input, so they need not be concerned that different parallel interleavings of operations may produce different results. Determinism also makes a program or algorithm much simpler to debug, since one knows that the same result will be produced every time it is run with a given input.

DPJ [13] can provide a compile-time guarantee of determinism using the combination of its type and effect system and simple parallelism constructs supporting only fork-join patterns of parallelism. We provide a similar static guarantee of determinism for deterministic algorithms or programs written in TWEJava. All programming patterns for which DPJ can give a guarantee of determinism can also be expressed and proven deterministic using the tasks with effects model. Our model also allows us to give a static guarantee of determinism for certain computations in a program while still allowing the rest of the program to use the full flexibility of TWEJava (including non-fork-join concurrency structures), and guaranteeing our other safety properties for the whole program. Thus, our feature for guaranteed determinism can be used within programs that could not be expressed with DPJ.

To request that the compiler check and enforce the determinism of a certain task or method, the programmer can annotate it as `@Deterministic`. In code that has this annotation, the compiler will enforce that the only task-related operations used in the code are the `spawn` and `join` operations described in section 2.5. Also, code annotated as deterministic may only call other deterministic methods and spawn other deterministic tasks.

These restrictions ensure that the code invoked from a deterministic task or method (including through the creation of other tasks) accesses memory only as specified by its declared effects. Moreover, there is a defined order by which control of each region covered by those effects is transferred between tasks, as determined by `spawn` and `join` operations. (Note that the form of effect described in section 2.4 will never be needed for `join` operations within a deterministic computation, and thus will not occur.) Therefore, for a given input state of the memory in regions covered by the effects of the deterministic task or method, there is a deterministic output state that will not vary between executions of the code. This state is the same as the state produced if the code were executed sequentially with each task's code run at the point where the task is spawned. These deterministic computations are also deadlock-free.

5. Compiler and Runtime System

Our implementation of TWEJava consists of a compiler and a runtime system, which we briefly describe here.

5.1 Compiler

The compiler is based on the DPJ compiler, which checks that effect declarations are correct and that types are used correctly. Our extended version also supports the new features of TWEJava described in Section 2. These include generating code to record effect parameters and some region parameters for use at run time; performing a data flow analysis to determine the covering effects for each operation (accounting for operations that do effect transfer); and checking the use of the `@Deterministic` annotation.

To enable interoperation with existing Java code and libraries that do not have region and effect annotations (including the Java standard libraries), the compiler allows methods without effect annotations to be called within methods that have effect annotations. This produces a warning, but that warning can be suppressed for individual methods. Since we have not written an extensive standard library for TWEJava, we take advantage of this capability to use Java standard library features such as the Swing GUI system, I/O routines, and math functions. The compiler cannot give a full guarantee about the correctness of code making such calls, so the programmer has to manually reason about it, but that reasoning can be encapsulated by writing annotated wrapper methods that internally call unannotated library routines.

5.2 Runtime System

Code generated by our compiler can be run using our runtime system, which implements the various task-related operations in TWEJava. We use an effect-based scheduler to enforce our model's key property of task isolation. Our current prototype implementation uses a queue of tasks protected by a single lock to manage the effect-checking phase of task scheduling. The effect-based scheduler enables a task for execution only once it is safe to do so based on its effects. Once a task is enabled for execution by our scheduler, it is handed off to a version of the Java `ForkJoinPool` framework, which is responsible for actually executing tasks using a thread pool.

When attempting to execute a task, our implementation generally works by scanning from a task's position forward toward the head of the queue (which includes both running and waiting tasks), checking if the task's effects conflict with those of each task ahead of it. If a conflicting task is found when attempting to schedule a task, then the later task is marked as waiting for the earlier one to complete. This approach will generally run conflicting tasks in the order that they were enqueued, but there is also a mechanism for prioritizing tasks that a running task is blocked on.

We show below that with this relatively simple scheduling approach we can achieve substantial speedups on a range of benchmarks. However, the tasks with effects model could also be implemented with other more scalable scheduling approaches. In particular, if we associated information about enqueued tasks with regions, then tasks with effects on unrelated regions would not need to have their effects explicitly compared against each other, and we could also avoid the need for a single global task queue lock.

6. Evaluation

We have carried out an evaluation of the tasks with effects model and our TWEJava language by porting several concurrent programs to it and writing one new one from scratch. We are principally concerned with demonstrating that TWEJava and the tasks with effects model can express a variety of concurrent programming styles used in real-world applications, but we also show that substantial parallel speedups can be achieved with our current TWEJava implementation.

6.1 Expressiveness

We ported four existing concurrent programs to TWEJava and wrote one new application in it. The first ported program is an interactive Connect Four game implementation called FourWins, which was ported from an original code that used JCoBox [37], an actor-like concurrent programming system for Java. The FourWins code is structured in terms of modules that behave similarly to actors, including the game state, board state, game controller, GUI view, and human and computer players. These modules communicate by sending messages between each other, sometimes, but not always, blocking until the message is processed. Our general approach in most parts of this code was to introduce a region holding the data for each module, and to define a number of types of tasks corresponding to each message that may be sent to that module. Those tasks have either read or write effects on the module's region, as needed. This code also includes a parallel computation in the computer player's AI, to explore the tree of possible future moves. That recursive parallel computation consumes most of the execution time, and it is the portion for which we report performance results below. We note that the complex concurrency structure of this program, with code from multiple actors running concurrently and sending messages between each other, cannot be expressed in many more restrictive parallelism models that require structured parallelism (e.g. fork-join) or involve a single conceptual flow of control.

The other interactive GUI application we implemented is an image editing application called ImageEdit, which we wrote from scratch in TWEJava. It allows the user to open one or more images and apply various image editing filters to them. Each of the images is displayed in a separate window and updated as filters are applied to it. Each image has a region associated with it, and the actual pixel data for the image is broken up into a 2-D grid of blocks, with the data for each block placed in a separate region using index-parameterized arrays. (By default, and in our benchmarks, a block is simply a group of adjacent lines totaling about 100,000 pixels, but the user may specify other block dimensions.) Concurrency is possible both by doing concurrent operations on different images and by operating in parallel on one image at the level of blocks. ImageEdit currently includes filters for Gaussian blur, sharpening (unsharp mask), detecting edges in the image (based on the Canny edge detection algorithm [15]), darkening or brightening the image, and converting it to grayscale. All of the filters can use parallelism at the level of blocks, sometimes using several computation steps in sequence with parallelism in each step. The only non-parallel step in any of them is a short final step in the edge detection filter to identify edges in the input image that cross between two different blocks. Computation in this program is driven by user input events, so the program as a whole does not follow the fork-join computation model required by systems like DPJ. It could be written in other task-based concurrency models that do not use effects, but these would not provide the strong safety guarantees of TWEJava and would require the programmer to manually ensure that tasks performing conflicting memory operations cannot run concurrently.

The other three benchmarks were previously written in DPJ [13], and we ported our versions from the DPJ versions, following a similar pattern of regions and effects. These are the force computation from a Barnes-Hut n-body simulation; a k-means clustering algorithm (originally adapted from STAMP); and a Monte Carlo financial simulation, originally from the Java Grande parallel benchmarks. These three benchmarks allow us to evaluate the impact of the run-time scheduling overheads in our system by comparing against the original DPJ versions, which do not have any overheads related to effect-based scheduling at run time.

The Barnes-Hut force computation involves a parallel loop over a set of bodies, computing and adding up the forces on each body due to the other bodies. We create one task per thread using the `spawn` operation, each operating on a portion of the total set of bodies, which is divided using an index-parameterized array. The resulting computation is deterministic and has good parallelism.

The Monte Carlo simulation includes a deterministic parallel loop to compute an array of results, followed by a reduction step that updates globally shared data. In the DPJ version, this reduction step used DPJ's commutative annotation, which represents an unchecked assertion from the programmer that two invocations of a certain method are commutative and that it internally uses the necessary locking to correctly synchronize concurrent invocations. In the TWEJava version, this commutative method is replaced by a task, and our system automatically guarantees that this task behaves atomically. Thus, TWEJava offers a stronger safety guarantee than DPJ, since it does not require the programmer to correctly insert manual locking operations. As with Barnes-Hut, we create one task per thread in the parallel loop.

The k-means computation involves a parallel loop with a reduction step. In the original STAMP code, this reduction step is an atomic block, but in the DPJ version it is a commutative method with internal locking. In TWEJava, it is a task. As in Monte Carlo, the DPJ version relied on unchecked, manual locking, so TWEJava offers a stronger safety guarantee than DPJ. The structure of the reduction computation in k-means requires that we create many reduction tasks, independent of the number of threads.

We were able to express all the parallelism that was present in the original codes that we ported. Both the `executeLater/getValue` operations and structured parallelism with `spawn` are used in our benchmarks. The former are necessary for unstructured parallelism such as messaging between actors or modules, and for defining tasks that behave like atomic or synchronized blocks, while the latter can be used in parallel loops or recursive parallel computations.

6.2 Performance

We measured the performance of our benchmark codes on a machine with four Intel Xeon E7-4860 processors (40 total cores, 80 hardware threads using Hyper-Threading) and 128 GB of memory, running Scientific Linux 6.3 with kernel 2.6.32 and 64-bit Oracle JDK 7u9. Figures 5 and 6 report the speedups achieved in the parallel portion of each code. For ImageEdit, we report speedups for

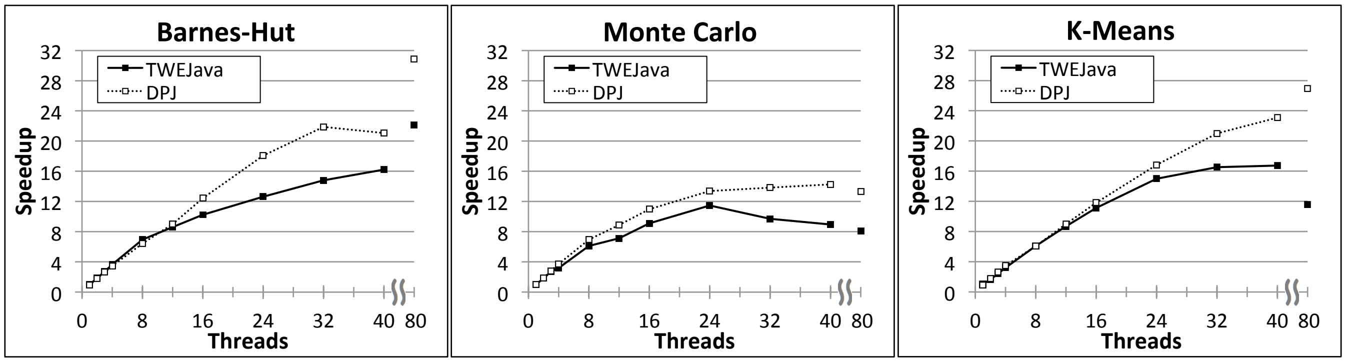


Figure 5. Parallel speedups of benchmarks ported from DPJ, showing performance of TWEJava and DPJ versions. These speedups are for the parallel portion of each code and are relative to the DPJ code compiled and run in sequential mode, in which the DPJ parallelism constructs are erased by the compiler, creating a sequential program with no run-time overheads related to parallelism constructs.

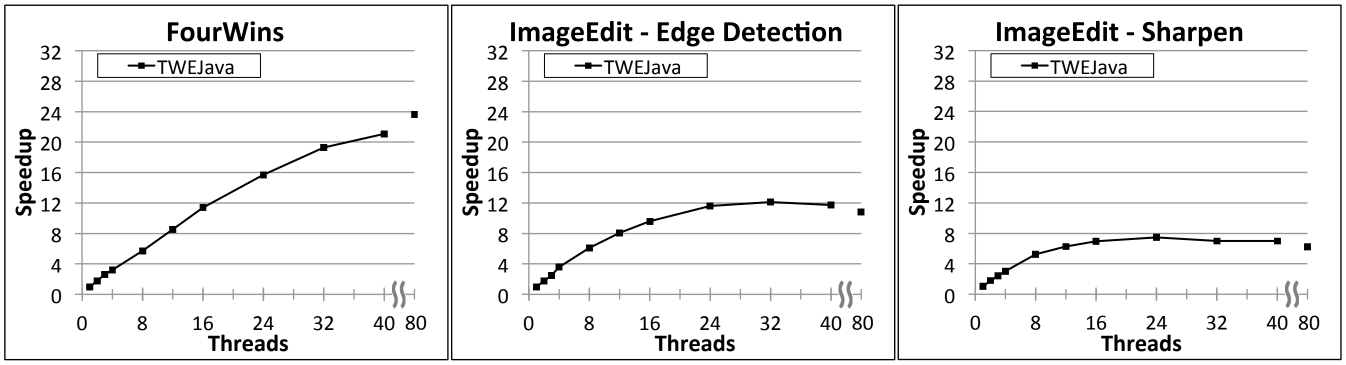


Figure 6. Speedups for the FourWins AI computation and two filters in the ImageEdit application. We did not have pure sequential versions of these programs available for comparison, so we give speedups relative to the TWEJava codes run using one worker thread and configured so that the major potentially-parallel computations in the codes each run as a single task, thereby minimizing task-related overheads.

both the edge detection filter and the sharpening filter. We also compared the parallel running times to DPJ for the codes where there is a DPJ version. The multi-threaded DPJ version internally executes tasks on a thread pool, but it does not have the overhead of run-time effect-based task scheduling, and previous work has shown it is generally quite efficient [13].

Each of our TWEJava benchmarks achieves significant speedups, with maximum speedups on the various benchmarks ranging from 7.5x to 23.6x. The Barnes-Hut and FourWins benchmarks continue scaling substantially up to 80 threads (with the gains going from 40 to 80 threads attributable to Hyper-Threading). The other benchmarks show good scaling at lower numbers of threads, but do not continue scaling above 24 to 32 threads.

The benchmarks for which we have DPJ versions perform very similarly to the DPJ versions up to at least eight threads, but show worse scaling at high numbers of threads. Several types of overhead in the TWEJava system may be responsible for these performance differences. The overheads of our effect-based run-time task scheduling system include the need to check the effects of tasks against each other to see whether they conflict, and the need to track some region parameters and all effect parameters at run time, rather than erasing them during compilation. These overheads can become larger with larger numbers of threads, because there will generally be more tasks active at once in such configurations. Another important factor limiting the scalability of our current implementation is that our effect-based scheduler uses a single queue protected by a single lock, so all the effect-based scheduling operations in the system are essentially serialized. Also, the DPJ runtime system can use recursive subdivision to split the iterations of parallel loops into tasks, while in TWEJava we converted these constructs to loops that sequentially spawn off child tasks. This may also contribute to the inferior scalability of the TWEJava codes. It would be possible to implement this sort of recursive subdivision in the tasks with effects model, but TWEJava currently does not have convenient language constructs for it.

We believe the overheads of effect-based task scheduling are particularly important factors in explaining the inferior scaling of our version of KMeans compared to the DPJ version on large numbers of threads, because the TWEJava version uses a task rather than a locked block for the reduction step. This is called a large number of times, regardless of the number of threads (550,000 times in our benchmark configuration). Since task scheduling is a heavier-weight procedure than simple locking around a short block, and particularly since (as noted above) the scheduling of each task is effectively sequentialized in our current implementation, this leads to poorer scalability for the TWEJava version of the code.

In the case on ImageEdit, one factor limiting the speedups achieved is that each time the image is updated, some sequential operations are necessary to actually change the image displayed in the GUI, which is implemented with Java's Swing framework and therefore needs to do GUI operations on a single thread, in accordance with Swing's architecture. This is a larger factor for the sharpening operation than for the edge detection operation, since the core parallel computation for sharpening is faster than for edge detection. We believe this at least partially accounts for the poorer scalability of sharpening compared to edge detection, as well as the overall scalability limits of the ImageEdit computations.

While our system has run-time overheads related to task scheduling and dynamic tracking of region and effect parameters, it still delivers significant parallel speedups, sometimes comparable to the DPJ versions of the codes (particularly on relatively low numbers of threads). We believe the scalability and performance of our system could be improved by implementing a scheduler that does not use a single lock and a compiler and scheduler that work together to minimize the number of dynamic effect comparisons (e.g. by avoiding the need for run-time checking of covering effects when child tasks are spawned in a loop). However, we think our current implementation without these optimizations still gives good enough performance to be used in many applications, particularly in desktop and mobile systems with relatively low numbers of cores.

7. Related Work

Traditional multithreaded systems such as Java or Posix threads are more flexible than TWEJava in the sense that they allow almost any desired concurrency and synchronization structure to be expressed, but they provide no guarantees about the absence of concurrency errors, and also have few or no facilities that simplify reasoning about such errors. Some systems, including OpenMP [32], Cilk [11], Threading Building Blocks (TBB) [22] (except for the "lower-level" task interfaces), and Java's `ForkJoinTask` [33] are more structured and easier to reason about than traditional threads. However, these systems still do not provide any correctness guarantees such as data race freedom or determinism. The programmer still has to reason manually to ensure that data sharing patterns are correct and synchronization is present when needed. These systems simplify such reasoning by limiting programs to use a particular parallelism structure (e.g. fork-join), but in doing so, can no longer express the forms of concurrency required by many programs such as interactive applications, servers, and actor-style programs. TWEJava is able to express all these kinds of programs and yet provides strong correctness guarantees.

RCCJava [18] can ensure data race freedom, but it does not provide structured concurrency constructs or guarantee other safety properties such as determinism. SharC [5] allows flexible concurrent control flow while providing a guarantee of data race freedom, but it also does not provide structured concurrency constructs and cannot guarantee stronger properties like determinism. CoreDet [8], Kendo [31], Grace [9], and DMP [16] allow multithreaded programs to be executed with a deterministic execution order that does not vary from run to run, but they do not provide structured parallelism constructs, and the deterministic execution order they provide is not related in an obvious way to the program code and may change if the code or input changes, which limits their utility as tools for reasoning about program behavior.

Many parallel and concurrent programming systems provide various correctness guarantees but have weaker expressive power than TWEJava. These include Jade [35], Prometheus [4], DPJ [13, 14], OoOJava [23], Dynamic Out-of-Order Java (DOJ) [17], Pāṇini [27], SvS [10], Legion [7], and Ke et al.'s system for parallelization with dependence hints [25]. Several of these systems, including Jade, Prometheus, OoOJava, DOJ, and Pāṇini, guarantee deterministic semantics (often with equivalence to a unique sequential program) but these systems are unable to express inherently nondeterministic algorithms, or programs where concurrency is due to external requests or user input and the input and its timing may affect the program's results. SMPSs [34] is also designed to provide sequential-equivalent semantics and uses a form of effect annotations for task scheduling, but these annotations are not verified, so the programmer is responsible for ensuring that the annotations are correct in order to ensure proper program behavior. Several systems, including (at least) Jade, SvS, Legion, DOJ, SMPSs, and Aida [28], have used effects in some form to guide run-time scheduling decisions, but TWEJava provides the ability to express programs not supported by any of these other languages and gives stronger safety guarantees than some of them.

DPJ, Legion and SvS can express nondeterministic programs, but not programs requiring flexible concurrency structures, identified above. DPJ supports programs with both deterministic and nondeterministic algorithms, and provides the strongest parallel correctness guarantees we know of, but because it is limited to fork-join parallel structures, it is not suitable for many concurrent programs. TWEJava supports a much broader class of programs than DPJ, and provides almost as strong correctness guarantees: its primary weakness compared to DPJ is that it only provides limited protection from deadlocks. Like DPJ, Legion cannot express programs with general concurrency and synchronization patterns because there are no mechanisms for explicit "join" synchronization between tasks (tasks block for other tasks only due to interfering effects, enforced by the scheduler) and the effects of a parent task must be a superset of the effects of its child tasks. Legion also provides significantly weaker correctness guarantees than DPJ or TWEJava, although it allows more dynamic assignment of data to regions, and explicit program management of locality via region maps. SvS executes tasks according to a statically-defined task graph, which limits the language to a narrower range of concurrent applications than TWEJava. SvS allows both deterministic and nondeterministic algorithms, and guarantees data race freedom to such programs. One key difference is that SvS *infers* potential conflicts due to implicit sharing of data between tasks, and uses an approximate run-time analysis of the memory possibly accessed by a task. While these features reduce the annotation burden on the programmer, they increase the likelihood of spurious dependences ("false positives") that prevent two tasks from executing in parallel. TWEJava does not suffer from such false positives when checking for effect interference between tasks.

Transactional memory systems [19] use speculative execution to enforce correctness guarantees such as atomicity. These systems guarantee that atomic blocks declared by the programmer execute in isolation from each other, performing rollback and retry if necessary. To date, implementations have often relied on software transactional memory (STM). STM systems generally have high overheads, stemming from the need to track memory accesses and check for conflicts, combined with wasted computation when rollbacks occur. In contrast, TWEJava only requires conflict checks (on task effect summaries) before a task begins execution and never rolls back partially-completed tasks. Also, to avoid exorbitantly high overheads, many STM systems only guarantee isolation between two atomic blocks (weak isolation). In these systems, statements outside atomic blocks may still race with other statements inside or outside atomic blocks, so there is not a full guarantee of data race freedom.

Several other systems also use optimistic parallelism. Galois [26] focuses on irregular algorithms and requires the programmer to specify which operations are semantically commutative and define inverse methods for use on rollback. Non-deterministic algorithms in DPJ also use atomic blocks implemented via an STM system, which has fairly poor absolute performance [14]. Aida [28] also focuses on irregular parallelism. It guarantees the absence of data races, deadlock and livelock, via a mechanism called "delegated isolation," where a task that conflicts with another concurrent task is rolled back and then "delegates" all its computation and data to the latter task. Galois, DPJ and Aida are all limited to highly structured, fork-join concurrency.

A more flexible style of concurrent programming is *actors* [3]. In the basic actor model, a concurrent system is composed of several actors, each potentially having local state, but no shared state between the actors. Actors communicate by sending messages to other actors, and computation is done at each actor in response to the messages received. Each actor processes only one message at a time, so all concurrency is due to the simultaneous execution of different actors. Actor-style programs are natural to express using our system: a region can be defined to correspond to each actor, and tasks with effects on that region can be thought of as equivalent to messages sent to and processed by that actor. Several actor-like programming models for shared memory systems [24, 29, 37] broaden the basic actor model to include some form of shared state between actors, but these systems are generally less flexible than our effect system, and in some cases do not guarantee data race freedom. Our system, when used to write actor-style programs, can express both shared state between actors and internal concurrency within actors, while guaranteeing data race freedom as well as, where desired, deterministic, sequential-equivalent semantics for parallel algorithms used within an actor.

8. Conclusion

We have described and defined the semantics of a new concurrent programming model based on tasks with effects, and presented a language called TWEJava that implements it. TWEJava can express a wide range of concurrent and parallel programs, while delivering very strong safety properties including task isolation, data race freedom, atomicity, and optionally determinism. We have implemented several concurrent programs in TWEJava and shown that our present implementation can give substantial parallel speedups.

Acknowledgments

This work was funded by the Illinois-Intel Parallelism Center at the University of Illinois at Urbana-Champaign. The Center is sponsored by the Intel Corporation.

References

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL*, 2008.

[2] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Comp., Special Issue on Shared-Mem. Multiproc.*, pages 66–76, December 1996.

[3] G. Agha. *Actors: A model of concurrent computation in distributed systems*. MIT Press, 1986.

[4] M. D. Allen, S. Sridharan, and G. S. Sohi. Serialization sets: A dynamic dependence-based parallel execution model. In *PPOPP*, 2009.

[5] Z. Anderson, D. Gay, R. Ennals, and E. Brewer. SharC: Checking data sharing strategies for multithreaded C. In *PLDI*, 2008.

[6] Apple. Concurrency Programming Guide. `http://developer.apple.com/library/mac/documentation/General/Conceptual/ConcurrencyProgrammingGuide/`, Dec. 2012.

[7] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *SC'12*, 2012.

[8] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *ASPLOS*, 2010.

[9] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe multithreaded programming for C/C++. In *OOPSLA*, 2009.

[10] M. J. Best, S. Mottishaw, C. Mustard, M. Roth, A. Fedorova, and A. Brownsword. Synchronization via scheduling: Techniques for efficiently managing shared state. In *PLDI*, 2011.

[11] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *PPOPP*, 1995.

[12] R. L. Bocchino and V. S. Adve. Types, regions, and effects for safe programming with object-oriented parallel frameworks. In *ECOOP*, 2011.

[13] R. L. Bocchino, V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for Deterministic Parallel Java. In *OOPSLA*, 2009.

[14] R. L. Bocchino, S. Heumann, N. Honarmand, S. V. Adve, V. S. Adve, A. Welc, and T. Shpeisman. Safe nondeterminism in a deterministic-by-default parallel language. In *POPL*, 2011.

[15] J. Canny. A computational approach to edge detection. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 8(6):679–698, June 1986.

[16] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic shared memory multiprocessing. In *ASPLOS*, 2009.

[17] Y. h. Eom, S. Yang, J. C. Jenista, and B. Demsky. DOJ: Dynamically parallelizing object-oriented programs. In *PPoPP*, 2012.

[18] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *PLDI*, 2000.

[19] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2nd Edition (Synthesis Lectures on Comp. Arch.)*. Morgan & Claypool, 2010.

[20] S. Heumann and V. Adve. Disciplined concurrent programming using tasks with effects. In *HotPar*, 2012.

[21] S. Heumann and V. Adve. Tasks with effects: A model for disciplined concurrent programming. In *WoDet*, 2012.

[22] Intel. Intel Thread Building Blocks Reference Manual. `http://software.intel.com/sites/products/documentation/hpc/tbb/referencev2.pdf`, Aug. 2011.

[23] J. C. Jenista, Y. h. Eom, and B. C. Demsky. OoOJava: software out-of-order execution. In *PPOPP*, 2011.

[24] R. K. Karmani, A. Shali, and G. Agha. Actor frameworks for the JVM platform: A comparative analysis. In *Principles and Practice of Programming in Java (PPPJ)*, 2009.

[25] C. Ke, L. Liu, C. Zhang, T. Bai, B. Jacobs, and C. Ding. Safe parallel programming using dynamic dependence hints. In *OOPSLA*, 2011.

[26] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI*, 2007.

[27] Y. Long, S. L. Mooney, T. Sondag, and H. Rajan. Implicit invocation meets safe, implicit concurrency. In *Generative Programming and Component Engineering (GPCE)*, 2010.

[28] R. Lublinerman, J. Zhao, Z. Budimlić, S. Chaudhuri, and V. Sarkar. Delegated isolation. In *OOPSLA*, 2011.

[29] Microsoft. Axum. `http://msdn.microsoft.com/en-us/devlabs/dd795202`.

[30] Microsoft. Task Parallel Library (TPL). `http://msdn.microsoft.com/en-us/library/dd460717.aspx`.

[31] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *ASPLOS*, 2009.

[32] OpenMP Architecture Review Board. OpenMP Application Program Interface, Version 3.1. `http://www.openmp.org/mp-documents/OpenMP3.1.pdf`, 2011.

[33] Oracle. Java Platform, Standard Edition 7 API specification. `http://download.oracle.com/javase/7/docs/api/`.

[34] J. M. Perez, R. M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *IEEE International Conference on Cluster Computing*, 2008.

[35] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of Jade. *TOPLAS*, 20(3):483–545, May 1998.

[36] G. Roşu and T. F. Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6), 2010.

[37] J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *ECOOP*, 2010.

Efficient Compilation of Fine-Grained SPMD-threaded Programs for Multicore CPUs

John A. Stratton[†*] Vinod Grover[†]
Jaydeep Marathe[†] Bastiaan Aarts[†]
Mike Murphy[†] Ziang Hu[†]
[†]NVIDIA Corporation
{vgrover, jmarathe, baarts, mmurphy, zhu}
@nvidia.com

Wen-mei W. Hwu[*]
[*] University of Illinois at Urbana-Champaign,
Center for Reliable and High-Performance
Computing
{stratton, hwu}@crhc.illinois.edu

Abstract

In this paper we describe techniques for compiling fine-grained SPMD-threaded programs, expressed in programming models such as OpenCL or CUDA, to multicore execution platforms. Programs developed for manycore processors typically express finer thread-level parallelism than is appropriate for multicore platforms. We describe options for implementing fine-grained threading in software, and find that reasonable restrictions on the synchronization model enable significant optimizations and performance improvements over a baseline approach. We evaluate these techniques in a production-level compiler and runtime for the CUDA programming model targeting modern CPUs. Applications tested with our tool often showed performance parity with the compiled C version of the application for single-thread performance. With modest coarse-grained multithreading typical of today's CPU architectures, an average of 3.4× speedup on 4 processors was observed across the test applications.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel Programming

General Terms Algorithms, Performance

Keywords CUDA, Multicore, CPU, SPMD

1. Introduction

In the coming years, commercial application developers will have a strong incentive to develop highly parallel software to take advantage of widespread parallel processors in the consumer market. However, it is unclear whether each potential user of an application will have a computing subtrate with a similar degree, granularity and style of parallelism. Even if an application is amenable to targeting a wide variety of parallel computational platforms, it is unclear whether a single expression of the application in any one programming model will be sufficient. The model must be powerful enough to effectively capture many applications, yet have enough constraints to enable a wide range of architectures to be effectively supported.

We present some initial findings of a case study testing one parallel programming model that industry is hoping will be such a portable model: fine-grained Single Program Multiple Data (SPMD) kernels, with limited thread cooperation, controlled by a centralized process. CUDA [16] and OpenCL [12], for example, are both built on an underlying programming model of fine-grained SPMD threads. For the experiments presented here, we will be working with the CUDA programming model, noting in advance that the same techniques would be applicable to OpenCL and other SPMD programming models as well.

The CUDA programming model is a hybrid of two parallel programming models initially tailored to GPU architectures. It supports bulk synchronous task parallelism [24], where each task is composed of fine-grained SPMD threads. Programmers have been using CUDA with significant success in many application fields, such as bioinformatics [19], molecular dynamics [21], machine learning [4], and medical imaging [22]. We view these successes as sufficient evidence that the fine-grained SPMD model is effective for programming a manycore architecture with explicit support for fine-grained threads. However, previously there has not been investigation of how such a model could effectively map to a more coarsely threaded architectures such as the current commodity multicore processors.

The contributions of this paper are:

- An implementation and comparison of two approaches to implementing a fine-grained SPMD programming model on a processor with coarse-grained thread-level parallelism.
- A description of programming model restrictions necessary to implement the intuitively more effective approach.
- Optimizations enabled by the serialization of a parallel model, primarily redundancy removal in both computation and data storage.
- Experimental evidence confirming the intuition, and comparing it with standard compiled C on current multicore CPUs.

The primary enabling factors for generating efficient C code from a fine-grained threading model are the restrictions on synchronization usage. These restrictions allow stronger reasoning in the compiler about execution semantics in the static code. The baseline microthreading approach to serializing an SPMD programming model is described in Section 4. The baseline approach represents what we believe to be the state of the art in implementing general finely-threaded programs on a system with significantly less thread-level parallelism. The second approach is summarized in our own previously published work [23] and that of Shirako et al. [20], describing a basic approach for generating structured code serializing fine-grained SPMD code. We have reimplemented and extended the functionality of these algorithms within a production-level compiler, and compile the full CUDA language without the limitations of the previous work. We show with experimental results that the structured approach enabled by restrictions on synchronization usage does indeed provide significant performance benefits over the more general baseline.

In the context of a serialized parallel model, several optimizations not available to the parallel form of the code are enabled. The optimizations detailed in Section 6 are notably analogous to existing redundancy removal optimizations in sequential programming models. However, we can leverage knowledge of explicit parallelism to reduce the burden of analysis or surpass the typical capabilities of commercial implementations.

We highlight some of the related work in cross-architecture parallel programming models in Section 2. A concise description of CUDA's execution and memory models relevant to this work is presented in Section 3. The general microthreading and structured microthreading techniques are discussed in Sections 4 and 5 respectively, followed by a description of enabled optimizations in Section 6. We describe the practical details of our compiler and runtime environment in Section 7 to provide a full context for our performance results presented in Section 8. We summarize the experiments and lessons learned in the concluding remarks of Section 9.

2. Related Work

The issue of mapping small-granularity parallel work units to CPU cores has been addressed in other programming models, such as parallel simulation frameworks [7] and dataflow or message-driven programming models [2, 3]. Such models typically implement a user-level microthreading technique similar to our baseline approach. Microthreading implementation is simplified when implemented within a single code object, as an SPMD programming model provides. OpenCL [12] is a programming model closely related to CUDA that claims such platform portability as we would like to explore. However, it has not matured to demonstrate such portability at this time. The methods and results presented here would be directly applicable to all finely-threaded SPMD programming models, including OpenCL.

Shirako et al. [20] applied many of the same transformation methodologies to serialize data-parallel loops containing barriers. We demonstrate how similar techniques can be utilized in an SPMD programming model, and demonstrate the further optimizations enabled by the application of these techniques.

Numerous other frameworks and programming models have been proposed for data-parallel applications for multiprocessor architectures. Some examples include OpenMP [17] and HPF [11]. Although widely used in a CPU symmetric multiprocessor environment, these models are yet to be proven for manycore chips. Lee et al. have described a system for compiling OpenMP programs to CUDA [13] which, if successful, could provide similar experimental benefit as extending CUDA to CPUs.

```
__global__ small_mm_list(float* A_list, float* B_list,
                         , const int size)
{
  float sum;
  int matrix_start, col, row, out_index, i;
  matrix_start = blockIdx.x * size * size;
  col = matrix_start + threadIdx.x;
  row = matrix_start + (threadIdx.y * size);

  sum = 0.0;

  for(i = 0; i < size; i++)
    sum += A_list[row + i] * B_list[col + (i*size)];

  //  Barrier before overwriting input data
  __syncthreads();

  out_index = matrix_start +
              (threadIdx.y * size) + threadIdx.x;
  A_list[out_index] = sum;
```

Figure 1: Multiplying many small matrices in CUDA.

Diamos has implemented a binary translation framework from GPU binaries to x86 [10]. While binary translators have advantages in knowing statically unavailable runtime parameters, compilers have more high-level program information available to them in the structured and symbolic source code. It is unclear which of the high-level transformations we propose would be possible without high-level compiler information available, if any.

Liao et al. designed a compiler for efficiently mapping the stream programming model to a multicore CPU architecture [14]. Their implementation attempted to build into the compiler capability for removing many of the restrictions of the stream programming model. In many ways, fine-grained SPMD-threaded models remove from the stream programming model those same limitations addressed by Liao et al.'s compiler. The programmer has control over tiling and kernel merging optimizations, the range of which is potentially broader than can be discovered and applied in an automated framework.

NVIDIA has released a toolset for CUDA program emulation on a CPU, designed for debugging. In the emulation framework, each fine-grained thread is executed by a separate runtime OS thread, incurring significant thread-scheduling overhead, and performing orders of magnitude more poorly than any of our approaches in informal experiments.

3. CUDA Programming Model

CUDA as a programming model has several interacting constructs for composing parallel programs on a shared-memory system [16]. The programming model allows sequential code in the standard C language with library APIs to control and manage *grids* of parallel execution specified by *kernel* functions. The *host* portion of the code is compiled using traditional methods and tools, while the kernel code introduces constructs for expressing SPMD parallelism. This work primarily focuses on the compilation and execution of the parallel kernel functions. We will be using the example kernel function of Figure 1 throughout this paper.

Within the SPMD kernel functions, *threads* are distinguished by an implicitly defined 3-tuple index uniquely iden-

```
tid = threadIdx.x;
while(i < end)
{
  x += input[i];
  if(i == end-1) {
    //segmented circular shift
    data[(tid + 1) % shift] = x;
    __syncthreads();
    output = data[tid];
    break;
  }
  else {
    i++;
  }
}
```

(a) Incorrect Usage

```
tid = threadIdx.x;
while(i < end)
{
  x += input[i];
  if(i == end-1) {
    break;
  }
  else {
    i++;
  }
}
//segmented circular shift
data[(tid + 1) % shift] = x;
__syncthreads();
output = data[tid];
```

(b) Correct Usage

Figure 2: Synchronization within control flow. (b) shows code semantically equivalent to that of (a), and obeys the synchronization usage constraints.

tifying threads within a thread *block*. Thread blocks themselves are distinguished by an implicitly defined 2-tuple variable. The ranges of these indexes are defined at runtime by the host code in special kernel invocation syntax. In the example of Figure 1, each thread block is computing one small matrix multiplication out of the list, while each thread is computing one element of the result matrix for its block.

CUDA guarantees that threads within a thread block will be live concurrently, and provides constructs for threads within a thread block to perform fast barrier synchronizations and local data sharing. Distinct thread blocks within a grid have no ordering imposed on their creation or execution. Atomic operations provide limited interblock communication.

CUDA uses textually-aligned static barrier semantics, such as those of the Titanium language [1]. For instance, it is illegal to invoke a barrier intrinsic in both paths of an if-else construct when CUDA threads may take different branches of the construct. Although all threads within a thread block will reach one of the intrinsics, they represent separate barriers, each requiring that either all or none of the threads reach it.

As a more general example, consider the constructed example of Figure 2. We assume that `end` is a function of the thread index, while the initial value of `i` is thread-invariant. Although each logical thread will hit the barrier exactly once, the code of Figure 2a will have unpredictable runtime behavior. Figure 2b shows how the code may be restructured to achieve the desired effect without violating this constraint.

CUDA is less restrictive than Titanium in that barriers can be dependent on statically thread-dependent expressions. It only requires that the dynamic evaluation of those expressions results in a uniform boolean value at runtime. For instance, if `end` and the initial value of `i` are functions of the thread index such that `(i - end)` is thread-invariant, the code of Figure 2a will function correctly, in constrast with the restrictions of Titanium that would prohibit this case as well.

The CUDA memory model, at the highest level, separates the host and device memory spaces, such that host code and kernel code can only access their respective memory spaces directly. The device memory spaces are the *global*, *constant*, *local*, *shared*, and *texture* memory spaces. A summary of the memory spaces is given in Table 1.

```
1   __global__ small_mm_list(float* A_list, float* B_list,
                             const int size)
    {
2     float sum[];
3     int matrix_start[], col[], row[], out_index[], i[];
      int current_restart, next_restart;
      next_restart = 0;
      // Loop over barrier synchronization intervals
      while (next_restart != -1) {
        current_restart = next_restart;
        //Loop over threads within an interval
        for(each tid) {
          switch (current_restart) {
            case 0:
              goto RESTART_POINT_0;
            case 1:
              goto RESTART_POINT_1;
          }

          // Original program beginning:
         RESTART_POINT_0:
4         matrix_start[tid] = blockIdx.x * size * size;
5         col[tid] = matrix_start[tid] + tid.x;
6         row[tid] = matrix_start[tid] + (tid.y * size);

7         sum[tid] = 0.0;

8         for(i[tid] = 0; i[tid] < size; i[tid]++)
9           sum[tid] += A_list[row[tid] + i[tid]] *
                        B_list[col[tid] + (i[tid]*size)];

          // restart point induced by syncthreads()
10        next_restart = 1;
          goto end_of_thread_loop;
         RESTART_POINT_1:
11        out_index[tid] = matrix_start[tid] +
                             (tid.y * size) + tid.x;
12        A_list[out_index[tid]] = sum[tid];
          next_restart = -1; // indicates "return"
         end_of_thread_loop:
        }
      } // while
    }
```

Figure 3: Microthreaded code for our example kernel

These memory spaces follow general microarchitecture principles. Large memory spaces are expected to have long latencies and limited random-access bandwidth, while small memory spaces can reliably satisfy low-latency accesses. Efficient CUDA programs make these cost trade-offs explicitly by using localized access patterns and limiting the active working set. However, if an application is written assuming significant hardware acceleration of texture processing operations, it could lead to design choices that perform poorly on processors implementing those features in software.

4. Baseline SPMD Microthreading

The term *microthreading* describes software techniques used in contexts where parallel work units are too small to efficiently schedule individually [2, 7]. The key concept is that software emulates the execution of multiple conceptually parallel threads or computation objects in a single, sequential program. The result of applying such a microthreading technique to the kernel of Figure 1 is shown in Figure 3. Note that the implicitly defined variable `threadIdx` has been shortened to `tid` for brevity. The compiler begins by labeling each barrier with a unique number, re-

Table 1: CUDA Device Memory Spaces in GPU Execution Context

Memory Space	Permissions	Scope of an Object	Capacity	Latency	Special Features
Global	Read/Write	All threads	DRAM capacity	High	Requires aligned, contiguous simultaneous accesses for best bandwidth.
Constant	Read-Only	All threads	64KB	Low (cached)	Single-banked cache with broadcast capability to multiple threads.
Local	Read/Write	Single thread	DRAM capacity	High	Most often promoted to private registers, which are shared between threads. Values not promoted to registers have long latency access.
Shared	Read/Write	Single thread block	16KB	Low	Scratchpad memory shared between thread blocks. More shared memory used per thread block means fewer thread blocks can be simultaneously active.
Texture	Read-Only	All threads	DRAM capacity, limits per object	High	Hardware interpolation, indexable by real-valued indexes, and other features for image processing.

serving the number zero for the implicit barrier at the beginning of the program. In our example, the single barrier gets labeled with the number 1. The original code for the program is modified, with each barrier replaced by a unique label, an assignment of the `next_restart` variable with the barrier's ID, and a jump to begin executing the next conceptual thread. All exit points from the function are replaced by statements assigning an exit flag (-1) to the `next_restart` variable. The compiler then generates the microthreading iteration structures. The master while-loop iterates over the number of times the threads will synchronize, each time updating the current restart point to the place the threads synchronized. A for-loop iterates over thread indexes, and uses a switch structure to begin each thread's execution at the current restart point. For each iteration of the conceptual thread for-loop, a single conceptual thread is advanced from its previous synchronization point to its next synchronization point. The master while-loop then iterates again to emulate all conceptual threads executing the original program from the barrier statement to the next point of syncronization, unless the original program end was reached by the conceptual threads being emulated.

In our example, the master while-loop control structure will begin executing the SPMD code of the original parallel program, marked by `RESTART_POINT_0`. The program executes the original, SPMD source code until it reaches statement 10, the original synchronization point. It then marks the synchronization point it reached, and program execution continues with the next conceptual thread at the original program beginning (statement 4). When all intances of conceptual threads have been iterated over (`each tid` is exhausted), the barrier is marked as the next restart point. This corresponds to the release of all conceptual threads from the barrier, so each microthread is executed again starting at the barrier release. Each conceptual thread then writes its output and reaches the original function's end. When all conceptual thread indexes have been processed again, the master while-loop detects that all conceptual threads have completed, and exits the function.

The memory model must also be adapted to fit a monolithic shared memory system. The globally visible memory regions already fit this model, and need not be changed. The features of the texture fetching functions must be implemented in a software library. The host and device memory spaces must generally be kept distinct, implying that API functions copying between host and device memory spaces should still operate as specified. Removing this overhead is a potential target for future work.

Local memory regions must be allocated per thread. The simplest method accomplishing this is to change each local memory object into an array of objects accessed by the CUDA thread index. The shared memory regions, private to a thread block, should be dynamically allocated for the thread blocks actively executing. For shared memory arrays of fixed size, this can be done using the program function stack. However, CUDA allows shared array of statically unspecified size, determined at kernel launch time. In C, this is most feasibly addressed by dynamically allocating a shared memory buffer of the appropriate size for each actively executing thread block. This is addressed in the runtime portion of the system.

The runtime environment is responsible for the execution of the programming model, given the adapted kernel functions generated by the compiler. Considering the thread blocks as work units, the runtime essentially implements a bulk-synchronous parallism model. It is responsible for the parallel processing of the work units within a grid, ensuring that different grids will be synchronized with each other and with the host.

5. Structured Microthreading

Consider a common case in which a kernel function has no synchronization. In this case, complex microthreading techniques are unnecessary, as the threads can be interleaved in any way we desire, including complete serialization. When barrier synchronization is present, complete serialization is not possible, but unstructured control flow caused by the added *goto* statements to and from the restart points of the previous approach is less easily analyzed by most compilers, especially for optimizations like automatic vectorization. The improved approach described in this section summarizes a variation of previous work [23] taking advantage of the synchronization restrictions to more efficiently implement microthreading.

Algorithm 1 partitions an SPMD program with textually-aligned static barriers and regular control flow into groups of statements not containing barrier synchronization. For each statement in sequence, we examine whether it is or contains a barrier statement. If not, it is included in the current partition. If it is a barrier statement, it defines a partition boundary, ending the current partition and beginning another. If it is a control-flow construct containing a barrier, then by the restrictions on the correct usage of barriers, all

```
Input: List of Statements $F$ in AST representation
Output: List $X$ of Code Partitions Free of Barriers
Begin new partition $P$;
while $F$ has next statement $S$ do
    switch type of statement $S$ do
        case barrier
            Add $P$ to $X$;
            $P$ = new partition;
        end
        case simple_statement
            Add $S$ to $P$;
        end
        case seq
            Prepend statements comprising $S$ to $F$;
        end
        otherwise
            if $S$ contains a barrier statement then
                Add $P$ to $X$;
                Invoke algorithm recursively on the body
                of $S$, producing a list $L$ of partitions
                within $S$; Append $L$ to $X$;
                $P$ = new partition;
            else
                Add $S$ to $P$;
            end
        end
    end
end
if $P$ not empty then
    Add $P$ to $X$;
end
```
Algorithm 1: Construction of code partitions free of barriers

```
1   __global__ small_mm_list(float* A_list, float* B_list,
                             , const int size)
    {
2     float sum[];
3     int matrix_start, col[], row[], out_index[], i[];
      for( each tid ) {

4       matrix_start = blockIdx.x * size * size;
5       col[tid] = matrix_start + tid.x;
6       row[tid] = matrix_start + (tid.y * size);

7       sum[tid] = 0.0;

8       for(i[tid] = 0; i[tid] < size; i[tid]++)
9         sum[tid] += A_list[row[tid] + i[tid]] *
                      B_list[col[tid] + (i[tid]*size)];
      }
10
      for( each tid ) {
11      out_index[tid] = matrix_start +
                         (tid.y * size) + tid.x;
12      A_list[out_index[tid]] = sum[tid];
      }
    }
```

Figure 4: Partitioned translation of our example kernel

threads must reach or not reach the construct, making it a valid partition boundary itself. The same algorithm is invoked recursively on the internal contents of the construct to partition the statements within.

These partitions define regions of code where the execution of different CUDA threads may be interleaved in any way, including complete serialization, as shown in Figure 4, where each partition is enclosed within a nested loop structure iterating through all thread indexes. Comparing Figure 4 to the previous Figure 3, we see that both perform the same sequential ordering of the original statements. However, Figure 4 does so with significantly less complex code in comparison, both inherently simpler and more easily analyzable for later optimization. For each statement of the program, the code generator also finds references to variables in the local memory space in that statement, and conservatively converts these into references to the replicated arrays.

6. Optimizations Enabled

Programmers writing parallel software make significant tradeoffs between the cost of redundant computation among parallel execution units and the cost of synchronization and communication. However, when these parallel applications are serialized to execute on a sequential processor, the cost of communication largely vanishes, and redundant computation often no longer makes sense. In sequential-program compilers, redundancy removal has been very successful, but somewhat limited by the conservative assumptions necessary to preserve sequential semantics when analysis falls short. However, when the sequential program is actually an explicitly parallel program serialized, the need for analysis is either greatly reduced or removed entirely, as interthread ordering semantics are much more loosely constrained than a typical sequential loop nest. While such optimizations should be possible within the baseline approach, it would not be possible to leverage the existing work on loop nest transformations in that context.

Variance Analysis Opportunities for redundancy removal are exposed by discovering what portions of the kernel code will produce the same value for all thread indexes. Computation that was previously performed redundantly by multiple CUDA threads now can be executed once in the single CPU thread. The core of variance analysis is the forward program slice of each element of the thread index tuple. We compute these program slices, annotating each statement with those program slices they comprise. We refer to these annotations as variance vectors. For instance, statement 9 of our example kernel has a variance vector of `(x,y)`, because it depends on the results of statements 5 and 6 that respectively read the `x` and `y` index components. Implicitly, atomic intrinsics are considered as a use of each element of the thread index, as their return value could vary for each CUDA thread.

When no statement in a partition contains a particular element in its variance vector, the partition does not need to be executed for each value in the index range of that element. Its results are independent of that element of the conceptual thread index. In the simplest case, and perhaps the most common, a programmer could intend to only use a subset of the elements of the thread index tuple to distinguish threads, implicitly assuming that all of the other elements will have a constant value of 1. In this case, the programmer writes a kernel never using some elements of the thread index tuple. The variance analysis will not annotate any statement with an unused component, directing the code generator to not create any loops over those elements of the thread index for any partition. This is the case for our example kernel, where the z index is unused.

Adaptive Loop Nesting Even when loops over certain elements are required for a partition, perhaps not all statements in a partition require execution for all thread indexes, analogous to loop invariant removal. However, we propose a technique called *adaptive loop nesting* that is more general in that it simultaneously evaluates transformations equivalent

```
1   __global__ mm_list(float* A_list, float* B_list,
                               , const int size)
    {
2     float sum[];
3     int matrix_start, col[], row[], out_index, i;

4     matrix_start = blockIdx.x * size * size;
      for(tid.x = 0; tid.x < blockDim.x; tid.x++) {
5       col[tid] = matrix_start + tid.x;

        for(tid.y = 0; tid.y < blockDim.y; tid.y++) {
6         row[tid] = matrix_start + (tid.y * size);
7         sum[tid] = 0.0;

8         for(i = 0; i < size; i++)
9           sum[tid] += A_list[row[tid] + i] *
                          B_list[col[tid] + (i*size)];
        }
      }
10
      for(tid.x = 0; tid.x < blockDim.x; tid.x++)
        for(tid.y = 0; tid.y < blockDim.y; tid.y++) {
11        out_index = matrix_start +
                               (tid.y * size) + tid.x;
12        A_list[out_index] = sum[tid];
      }
    }
```

Figure 5: Optimized translation of our example kernel

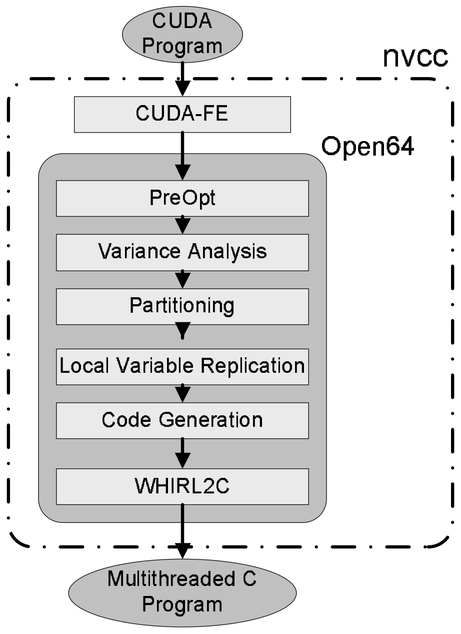


Figure 6: Compiler implementation diagram

to loop interchange, loop fission, and loop invariant removal to achieve the best redundancy removal, similar to polyhedral modeling of loop nests for sequential languages [8]. The significant distinction from typical loop-nest optimization is that all iterations can be assumed independent without analysis because of their origin from parallel threads.

The compiler may generate loops over thread index elements only around those statements that contain that element in their variance vector. To remove loop overhead, the compiler may fuse adjacent statement groups where one has a variance vector that is a subset of the other. All of the traditional cost analysis applied to loop fusion operations may apply here.

Typical cost analysis must be used to determine cases such as statements 5-9 of our example kernel. Statements 7-9 must be included in a loop nest over both x and y components of the conceptual thread index, as the computation is unique to each CUDA thread. As each of statements 5 and 6 is only dependent on one index element, either can be merged into a loop nest with statements 7-9, inside the outer loop over one component but before the inner loop of the other index. However, choosing either statement 5 or 6 to merge will lead to one of two choices for the other. We may choose to force the other into the innermost loop, causing unnecessary redundant execution, since it was independent of one of the loops now containing it. Otherwise, me must enclose it in an extra, separate loop nest for that statement alone, incurring extra control overhead. We chose a cost heuristic that in this case would determine that the extra control overhead is more costly, and would generate the control flow observed in Figure 5 that redundantly executes statement 6 for every x index.

Optimizing Local Variable Replication We note that because of the serialization of the computation in the fine-grained threads, not all data conceptually private to each thread must necessarily be instantiated as separate memory locations per thread. In particular, it is not necessary to create private memory locations for values that have a live range completely contained within a partition. In such cases, one memory location reused by all threads is sufficient. Another case is where, even though the variable is live through multiple partitions, its value is thread-invariant. This is the case when a variable definition has an empty variance vector.

Two cases arise in which variable replication must be applied to the output value of an assignment with a non-empty variance vector. The first is if a value defined by the assignment reaches a use in another partition. As stated previous, values with a live range completely contained within a partition will never need to be saved for the same conceptual thread to use in some later partition. The second is if, in the presence of the loop over thread indexes placed around the partition, the defined value would reach a use that it previously would not have.

Assignments with an empty variance vector technically never need to write to a replicated location, such as statement 4 of Figure 5. However, we decided that for any use reachable by at least one replicated definition, all its potential definitions must write to the replicated location for simplicity.

Minimal variable replication and adaptive loop nesting share an interesting interplay in that the maximal fusing of loops over indexes can introduce additional cases requiring replication. This has been well established in work on loop fusion. The final results of these optimization algorithms would result in a generated kernel code like that shown in Figure 5.

7. Implementation

The compiler is implemented within NVIDIA's production CUDA compilation toolchain. The toolchain provides a CUDA compiler driver, called nvcc. We added a new compiler flag enabling multicore compilation. The compiler structure is shown in Figure 6. At a high level the compiler consists of two main components: a frontend (CUDAFE) and the Open64 [9] high-level backend. CUDAFE is the standard CUDA production compiler front-end without modifications, just as it is used for GPU compilation.

In our implementation we generate HI-WHIRL intermediate representation (IR) for the Open64 backend infrastructure [9]. We implemented all the optimizing transformations at the HI-WHIRL level, chosen because almost all machine-independent analysis and optimization passes are available there [6]. The backend consists of five main components.

PreOpt- We use the standard Open64 optimizer to perform a few simple optimizations and, more importantly, to generate data flow information in the form of def-use chains.

Variance Analysis- The variance analysis we described earlier computes forward program slices on the thread index variables, annotating every statement with the components of the `threadIdx` variable on which that statement depends.

Partitioning- The partitioning algorithm described in Section 5 builds a list of partitions and, within each partition, collects a list of statements.

Local Variable Replication- Def-use chains restricted to the set of local variables of a function determine which variable references are read and written in multiple partitions. Each statement is annotated with the list of variable references within that statement needing to reference the expanded version of the variable.

Code Generation- This phase completes the generation of IR that is the complete, optimized transformation of the input into executable code. It traverses each partition, grouping adjacent statements if desirable given their varience vectors. It also transforms statements to use replicated versions of variables as necessary. Finally, it surrounds each grouped cluster of statements within a partition by the necessary thread loops, as required by the variance vectors of those statements.

WHIRL2C- We use the WHIRL2C [5] component from the Open64 distribution to generate C code from the transformed IR.

Thread blocks in the CUDA programming model represent independent tasks, each embodied by a sequential program following our compiler's translation. Many frameworks exist for distributing such parallel tasks to processors. Our implementation uses POSIX threads as an example. The runtime system creates several OS worker threads, the number of which can be controlled by an environment variable. At a kernel launch, the number of CUDA thread blocks in the grid to be launched is statically partitioned to the runtime threads. Each runtime thread executes its chunk sequentially and waits on a barrier. When all runtime threads reach the barrier, the grid has completed, and control is returned to the host thread.

8. Performance Evaluation

We present results on the eight CUDA benchmarks in Table 2 from application fields including fluid dynamics, astrophysics, and financial modeling. These applications were written specifically for a GPU target architecture, and have shown significant performance on that platform, some reported in previous work [18]. For benchmarking, we used an Intel Core2 Quad processor system running RedHat Enterprise Linux 4 (Update 7). We use gcc version 3.4.6 as the final C compiler, with -O3 optimization for all tests.

Table 3 shows that optimizations of the structured microthreading implementation dramatically reduced the number of replicated variables, with direct effect on reducing cache pressure. The number of references to replicated variables is also consequently reduced, intuitively leading gcc to promote a larger fraction of variable accesses to register accesses. The variance analysis correctly detected that, out of all of the benchmarks, only `tpacf` used two dimensions of the thread index, while all the other applications used only one.

Benchmark	App. domain	Kernel lines	Static barriers
petrinet	stochastic models	191	5
blinn	volume rendering	155	0
blackscholes	financial models	43	0
nbody	astrophysics sim.	180	3
lbm	fluid sim.	285	1
tpacf	astronomy data processing	98	4
binoption	financial models	121	5
FDTD	electromagnetic simulation	263	6

Table 2: Benchmark summary

Benchmark	Local objects	Static local object references	Replicated local objects	Static references to replicated objects
petrinet	72	623	0	0
blinn	93	343	0	0
blackscholes	35	133	0	0
nbody	82	498	18	141
lbm	110	1269	11	51
tpacf	36	196	6	25
binoption	51	215	6	6
FDTD	46	481	13	94

Table 3: Static Results of Optimizing Transformations

Figure 7 shows the benefits of our optimizations over a traditional microthreaded approach. Those applications with the least performance differences, `blinn` and `blackscholes`, do not use any synchronization within the CUDA kernel. In these cases, the performance benefits of the structured implementation are primarily due to the removal of the redundant local memory objects, as the control flow structure is practically the same between the two implementations. The rest of the applications do use synchronization, and gain significant performance benefits from the structured implementation, with an average of approximately 2$\times$ performance difference between the baseline and structured implementations of microthreading.

The most extreme cases of disparity between structured and unstructured microthreading were `BinOption` and `FDTD`. These were also the applications with the most synchronization, showing that the advantage of strutured microthreading and optimization generally increases with kernel program complexity.

Finally, we can see that the performance compared to a native C application varies widely. This is to be expected, as the implementation decisions were made in different programming models, although the task and general algorithm were fixed. `petrinet` and `FDTD` required the most parrallel algorithm implementation overhead, reflected in the comparison with sequential execution. Some applications even saw single thread performance gains over the existing C implementation. This indicates that the optimization effort spent

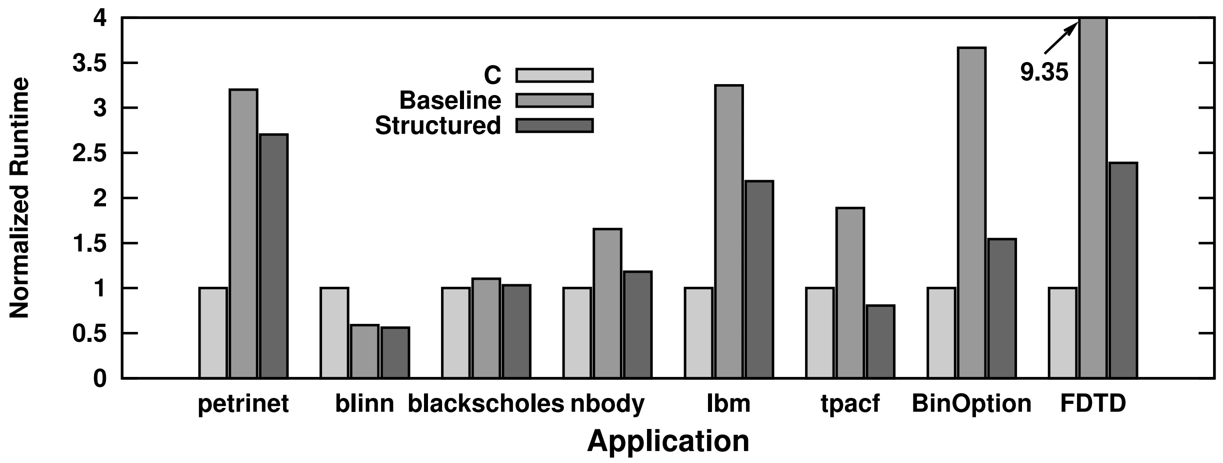


Figure 7: Translated CUDA application runtime relative to a native C implementation, each using one CPU execution thread. Only nominal programmer optimization effort was applied to either the C or CUDA versions of the code.

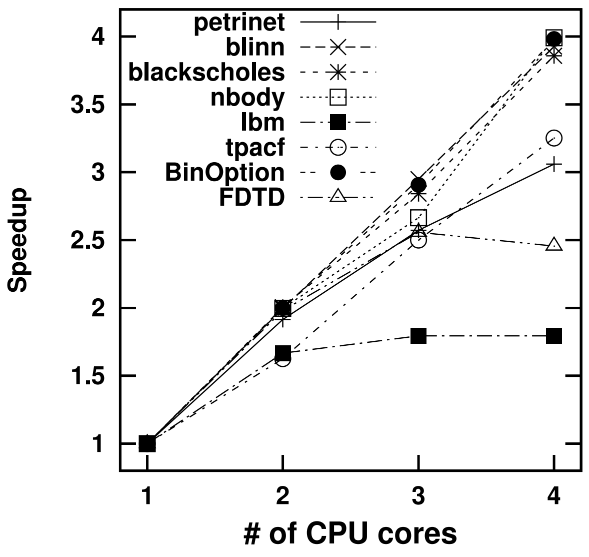


Figure 8: Application scaling from 1 to 4 threads.

on the CUDA implementation, for the GPU, was more effective for the CPU than the optimization effort spent on the C implementation.

All applications also saw significant performance gains from multithreading across the coarse-grained cores. We can see in Figure 8 that the performance scaling of the translated applications is very good, with close to ideal linear scaling for a small number of processor cores for most applications. The only application that reaches a scaling ceiling on our test system is `lbm`, as the application becomes bottlenecked by system memory bandwidth. Several other applications show somewhat less than ideal scaling, primarily due to load imbalance caused by our simplistic work partitioning implementation developed under the assumption of large numbers of equal-latency tasks. The two applications most affected by load imbalance are `tpacf` and `petrinet`, which have large variations in the runtimes of each block. A large existing body of work explores more effective dynamic work scheduling policies [15] applicable to our implementation would likely move some of the applications closer to the ideal scaling curve.

9. Conclusions

We have described techniques for efficiently implementing the CUDA programming model on a conventional multiprocessor CPU architecture. We have described a baseline microthreading approach, showing that a microthreading approach based on structured control flow has significant comparative performance advantages, in part due to additional optimizations that are enabled.

We observe that a fine-grained SPMD decomposition can be translated into more coarse-grained work units effectively, but only with reasonable restrictions on the synchronization model. Fine-grained threads that may interact arbitrarily must resort to some form of unstructured microthreading, which has shown to as much as double execution times compared to the structured approach, and in no case was it better. Our results also suggest that there is a class of parallel kernels where the finely-threaded version of the code shows parity with a native C implementation in single-thread performance.

Finally, our results have shown a particular software engineering advantage for current CUDA developers requiring some CPU fallback implementation when CUDA is not installed on a particular client's system. Using these techniques, such developers could translate their CUDA code directly into multithreaded C that is almost always better than a quickly written sequential program on a small multiprocessor typical in today's systems, while still keeping a single code base.

Acknowledgments

We would like the thank NVIDIA corporation for its support, and our formal and informal reviewers for their feedback.

References

[1] A. Aiken and D. Gay. Barrier inference. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 342–354, 1998.

[2] H. A. Andrade and S. Kovner. Software synthesis from dataflow models for embedded software design in the G programming language and the LabVIEW development environment. In *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, pages 1705–1709, 1998.

[3] G. Bikshandi, J. G. Castanos, S. B. Kodali, V. K. Nandivada, I. Peshansky, V. A. Saraswat, S. Sur, P. Varma, and T. Wen. Efficient, portable implementation of asynchronous multi-place programs. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 271–282, New York, NY, USA, 2009. ACM.

[4] B. Catanzaro, N. Sundaram, and K. Keutzer. Fast support vector machine training and classification on graphics processors. In *Proceedings of the 25th International Conference on Machine Learning*, pages 104–111, June 2008.

[5] W.-Y. Chen. Building a source-to-source UPC-to-C translator. Master's thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA, 2004.

[6] F. Chow, S. Chan, R. Kennedy, S. ming Liu, R. Lo, and P. Tu. A new algorithm for partial redundancy elimination based on ssa form. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 273–286, 1997.

[7] J. H. Cowie, D. M. Nicol, and A. T. Ogielski. Modeling the global internet. *Computing in Science and Engineering*, 1(1):42–50, 1999.

[8] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20:23–53, 1991.

[9] G. Gao, J. Amaral, J. Dehnert, and R. Towle. The SGI Pro64 compiler infrastructure. Tutorial. October 2000.

[10] Gregory Diamos. The design and implementation Ocelot's dynamic binary translator from PTX to multi-core x86. Technical Report GIT-CERCS-09-18, Georgia Institute of Technology, 2009.

[11] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Rice University, May 1993.

[12] Khronos OpenCL Working Group. The OpenCL Specification, May 2009.

[13] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *Proceedings of 14th ACM Symposium on Principles and Practice of Parallel Programming*, pages 101–110, 2008.

[14] S.-W. Liao, Z. Du, G. Wu, and G.-Y. Lueh. Data and computation transformations for Brook streaming applications on multiprocessors. In *Proceedings of the 4th International Symposium on Code Generation and Optimization*, pages 196–207, March 2006.

[15] E. P. Markatos and T. J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. In *Proceedings of the 1992 International Conference on Supercomputing*, pages 104–113, July 1992.

[16] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.

[17] OpenMP Architecture Review Board. OpenMP application program interface, May 2005.

[18] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. Kirk, and W. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, February 2008.

[19] M. Schatz, C. Trapnell, A. Delcher, and A. Varshney. High-throughput sequence alignment using graphics processing units. *BMC Bioinformatics*, 8(1):474, 2007.

[20] J. Shirako, J. M. Zhao, V. K. Nandivada, and V. N. Sarkar. Chunking parallel loops in the presence of synchronization. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 181–192, New York, NY, USA, 2009. ACM.

[21] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten. Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry*, September 2007.

[22] S. S. Stone, J. P. Haldar, S. C. Tsao, W.-M. W. Hwu, Z.-P. Liang, and B. P. Sutton. Accelerating advanced MRI reconstructions on GPUs. In *Proceedings of the ACM International Conference on Computing Frontiers*, pages 261–272, 2008.

[23] J. A. Stratton, S. S. Stone, and W. mei W. Hwu. MCUDA: An effective implementation of CUDA kernels for multi-core CPUs. In *Proceedings of the 21st International Workshop on Languages and Compilers for Parallel Computing*, pages 16–30, July 2008.

[24] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

Performance Portability in Accelerated Parallel Kernels

John A. Stratton, Hee-Seok Kim, Thoman B. Jablin, and Wen-Mei W. Hwu
{stratton, kim868, jablin, w-hwu}@illinois.edu

IMPACT Technical Report
IMPACT-13-01
University of Illinois at Urbana-Champaign
Center for Reliable and High-Performance Computing
May 18, 2013
Revision: May 22, 2013

Abstract

Heterogeneous architectures, by definition, include multiple processing components with very different microarchitectures and execution models. In particular, computing platforms from supercomputers to smartphones can now incorporate both CPU and GPU processors. Disparities between CPU and GPU processor architectures have naturally led to distinct programming models and development patterns for each component. Developers for a specific system decompose their application, assign different parts to different heterogeneous components, and express each part in its assigned component's native model. But without additional effort, that application will not be suitable for another architecture with a different heterogeneous component balance. Developers addressing a variety of platforms must either write multiple implementations for every potential heterogeneous component or fall back to a "safe" CPU implementation, incurring a high development cost or loss of system performance, respectively. The disadvantages of developing for heterogeneous systems are vastly reduced if one source code implementation can be mapped to either a CPU or GPU architecture with high performance.

A convention has emerged from the OpenCL community defining how to write kernels for performance portability among different GPU architectures. This paper demonstrates that OpenCL programs written according to this convention contain enough abstract performance information to enable effective translations to CPU architectures as well. The challenge is that an OpenCL implementation must focus on those programming conventions more than the most natural mapping of the language specification to the target architecture. In particular, prior work implementing OpenCL on CPU platforms neglects the OpenCL kernel's implicit expression of performance properties such as spatial or temporal locality. We outline some concrete transformations that can be applied to an OpenCL kernel to suitably map the abstract performance properties to CPU execution constructs. We show that such transformations result in marked performance improvements over existing CPU OpenCL implementations for GPU-portable OpenCL kernels. Ultimately, we show that the performance of GPU-portable OpenCL kernels, when using our methodology, is comparable to the performance of native multicore CPU programming models such as OpenMP.

I. Introduction

Heterogeneous computing systems are becoming very prevalent in many market segments. The number of accelerated supercomputers in the Top500 has been steadily increasing since 2009 [1]. The current #1 entry in the Top500 ranking of supercomputers uses an equal number of CPU and GPU chips [1]. Processors marketed towards consumer laptop and workstation systems often integrate heterogeneous CPU and GPU components on the same silicon die. Because GPUs and CPUs developed for somewhat distinct workloads historically, the programming models associated with each are mostly disjoint. Multicore CPU programmers may gravitate towards OpenMP or TBB, while GPU vendors have been encouraging the growth of CUDA [2], OpenCL [3], C++AMP [4], and OpenACC [5].

Divergent programming models and architectures are already causing significant costs in high-performance software development. Optimization is difficult for any architecture, and explicit heterogeneity increases that difficulty even for a single platform. An application developer must target program segments to the appropriate system component, balancing for the relative strengths and weaknesses of each, and then optimize each program segment for the targeted architecture. But software typically outlives systems, and an application with expected longevity has to be targeted towards a system unknown at development time.

When faced with the challenge of targeting an unknown but probably heterogeneous system, developers today typically make one of two choices. Some accept that each performance-sensitive program component must be capable of running on any CPU or GPU with high performance. Those developers bear the high cost of writing high-performance implementations for each, and guarantee performance portability through exhaustive specialization. Other developers lack the motivation or resources to pursue such a high-cost path, and choose instead to target some lowest common denominator, usually the CPU. These two choices drain development effort or leave significant performance opportunities on the table, respectively.

The root problem is a lack of *performance portability*, or the ability to write a single software implementation that can be targeted to either a CPU or GPU with high performance. For some languages, this is because the programming model itself is innately unsuitable for certain architectures. We are unaware of any serious work trying to implement POSIX threads on a GPU, for instance. Other programming models seem to hold much more promise. Stratton et al. presented an implementation of the CUDA language for CPU architectures within two years of the language's release [6]. OpenCL was specifically designed with portability between GPUs and CPUs in mind, with both AMD and Intel releasing x86 CPU implementations. Yet the current ecosystem fails to deliver satisfactory performance portability [7], [8]. This is primarily because a language specification, by itself, is insufficient for establishing performance portability.

Performance portability requires not only an agreed-on functional language specification, but a clear specification or convention for expressing a program's performance-related attributes. In practice, a vendor implementing a language may think first of how the language's functional components might most naturally map to the specific architecture. That implementation will then define some best practices for programmers targeting that architecture through that implementation. The problem is that when the best practices for different platforms diverge, the potential for performance portability is lost. To follow divergent best practices for different platforms, programmers have no choice but to write specialized implementations for each.

OpenCL is a language with both essential characteristics for performance portability in place. The parallelism constructs are abstract enough to transform into diverse lower-level implementations. Additionally, multiple GPU vendors have converged on similar conventions for how OpenCL kernel code should be written for good performance [9], [10], [11].

Our contributions over prior work include:

- Identifying the OpenCL community's emergent performance model.
- Formalizing the patterns necessary for performance portable OpenCL programming.
- Introducing two novel compiler transformations to efficiently map portable OpenCL codes to CPU architectures.
- Providing a best-of-breed OpenCL implementation for CPUs that exploits our performance and portability insights.

We review the OpenCL programming model and discuss the prevailing performance conventions in that model in Section II. While multiple other accelerator languages have similar parallelism models and performance conventions, OpenCL has the most examples of alternative CPU implementation methodologies. These alternative CPU implementations allow us to characterize the impact of failing to match the conventions for expressing performance in one or more significant ways. The characterization and qualitative analysis of current implementation methodologies is presented in Section III.

In Section IV, we describe one possible set of transformations that honors all of the major OpenCL performance conventions while mapping the programming model to a CPU architecture. Section V details the practical implementation of such transformations in an OpenCL compiler and runtime for experimental evaluation. We demonstrate that successfully exploiting the performance conventions achieves greater performance portability than previous work with two primary experiments. First, we show that OpenCL programs following the performance conventions perform as much as $10\times$ better than prior implementations that did not incorporate these performance insights (average $1.8\times$). Second, we demonstrate that with our implementation, the performance of OpenCL kernels on a

```
__kernel void MatMul(__global float *A,
                __global float *B, __global float *C) {
  float result;
  __global float *A_line = A + get_group_id(1)*A_WIDTH;

  result = 0.0f;
  for (int i = 0; i < A_WIDTH; i+= TILE_WIDTH) {

      for (int ii = 0; ii < TILE_WIDTH; ii++)
        result += A_line[i + ii] *
          B[(i+ii)*B_WIDTH + get_global_id(0)];

    barrier(CLK_LOCAL_MEM_FENCE);
  }
  C[C_WIDTH * get_group_id(1) + get_global_id(0)] =
      result;
}
```

Fig. 1. A simple, portable matrix multiplication kernel in OpenCL.

CPU is comparable to the performance of a native CPU implementation of the same algorithm in OpenMP. We summarize some additional related work and our conclusions in Sections VII and VIII, respectively.

II. A Proposed Consensus Performance Model for OpenCL

Clearly, if different vendors have seriously divergent expectations of how programmers should use the language, it is impossible for programmers to write portable code. In the early days of OpenCL, the NVIDIA OpenCL programming guide and AMD GPU programming guide offered conflicting guidelines, mainly regarding whether work-items should operate on scalar or vector data elements [12], [9]. For a variety of reasons, certainly including the pain developers felt from the lack of performance portability, later architectures and compilers from NVIDIA, AMD, and Intel converged on the general conventions described in this section.

The ultimate convention was attractive for several reasons. First, its focus on scalar work-item programs is simpler to use. Tools can aggregate scalar elements into a vector access if necessary, but cannot easily decompose a short vector access into multiple GPU SIMD-lanes [13]. Second, the convention has direct corallaries with dominant performance aspects of modern computer architectures: data-parallel (SIMD) execution, scalable task parallelismsuch, spatial locality, and temporal locality.

Figure 1 shows a listing of an OpenCL kernel for multiplying two matrices in single precision, which we will use as an ongoing example in this paper. If these practices could be performance-portable in theory, then achieving the goal of a portable language will require that both GPU and CPU implementations of OpenCL deliver high performance for codes written with these programming practices.

The kernel exhibits the major performance principles outlined in this section, but also demonstrates a limitation of those principles. The kernel program lacks more advanced algorithm-level data reuse and register tiling techniques commonly used to improve performance on real systems [14]. Nevertheless, the code is sufficient for demonstrating the performance guidelines. Note that these guidelines are primarily about programming for high hardware efficiency, i.e. reaching an achieved bandwidth and execution efficiency close to the architecture's peak. They will not advise whether or not particular algorithms would perform better, or whether bandwidth or execution throughput will be the ultimate limiting performance factor for a given architecture.

A. *Task and Data Parallelism*

The OpenCL programming model includes a two-level decomposition of work. Although the decomposition is just a two-level hierarchy of parallel tasks, the two levels have very distinct performance implications. All of the work-items in a work-group are guaranteed to be scheduled together, allowing them to coordinate more closely. Work-groups can not make any assumptions about scheduling or co-scheduling of other work groups, which means both that atomic operations must be used to guard critical sections, and in all other ways the groups are constrained to a bulk-synchronous programming model.

The groups of co-scheduled tasks are a clear source of thread-level parallelism, and are exploited in that way by nearly all implementations. Less obviously, perhaps, the work-items within a group are exploited as a source of vector-level parallelism on all GPU implementations known to the authors. The reasoning is that OpenCL's single-program multiple-data programming model will often naturally lead programmers to write groups of work-items with nearly-identical control flow in many situations. Even if it were not fully natural, the GPU implementations made this programming pattern fastest on their architectures from the beginning. Every known OpenCL GPU programming guide discourages "divergence", or writing programs such that different work-items within a work-group take different paths through the program, making SIMD less effective.

To be performance-portable, the amount of parallelism in and among workgroups needs to be flexible. Work-groups must be at least as wide as the native SIMD width of the machine, but never larger than the machine's capacity to schedule locally and simultaneously. The number of work-groups should be several times larger than the number of processors on the device to enable load-balancing. Given the variety of architectures, it is unclear whether these constraints can be met for all platforms with a fixed-size work-group. At minimum, performance-portable programs have to query the device parameters at runtime and choose group sizes appropriately, or allow the platform itself to choose a group size suitable for itself.

Our example kernel program computes a single output element with each work-item, so the number of work-items for a reasonably large matrix would be substantial. In line 15, `get_group_id(1)`, the second element of the group index tuple, is used as the output row index, indicating that each work-group should process some contiguous section of a particular output row. Therefore, every work-item in a work-group will need to access the same row of data from the A matrix. Also, there are no divergent branches in the kernel. Every condition is independent of the work-item index, so the entire path taken through the program execution is uniform across all work-items. Therefore, SIMD groups of work-items will be fully exercised, without the need to predicate any SIMD lanes at any time.

One interesting point about the OpenCL programming model is that by using groups of work-items as the basis for SIMD execution, the OpenCL implementations are providing the user a way to exploit SIMD without requiring them to program to a specific SIMD width. This is critical for portability, because different CPU and GPU architectures have widely varying SIMD widths. Work-groups that are significantly larger than an architecture's SIMD width enables additional thread- or instruction-level parallelism, as the group can be divided into multiple SIMD-width units. Subsequent proposed languages captured this insight particularly well also, such as the ISPC programming model that advocates SPMD programming as an easy and effective way of writing SIMD code for x86 CPUs [8].

B. *Spatial and temporal memory locality*

A large part of writing high-performance code is managing data locality well. In a recent survey article of seven broad GPU programming optimization techniques, only one was not directly related to memory locality management [15]. The OpenCL programming model makes explicit certain architectural realities that other languages try to keep abstracted away. Large, coherent memories are inherently more expensive to access than small, local data resources. As typical, we will divide our discussion of locality into two major classes: spatial locality and temporal locality.

Traditional mechanisms for capturing spatial locality were in response to the observation that in sequential programs, if a particular address was accessed, other addresses nearby were likely to be accessed soon in the future. Today, spatial locality is almost a performance requirement, because we build our entire memory systems out of large-line data transactions, such as cache lines and DRAM bursts. Furthermore, building hardware data structures with many ports for independent simultaneous access is very expensive. In particular, this means that even if a wide SIMD unit has gather and scatter capabilities, spatial locality among the addresses accessed will significantly reduce the number of unique memory lines touched by the access, which means a much reduced overall throughput demand on the memory system. In fact, OpenCL programmers are specifically encouraged to assign work-items to data elements such that memory accesses are "coalesced" [9], [10], [11]. Formally, an access is considered coalesced if it can be decomposed into the form:

```
uniform_base_address + (get_local_id(0) % SIMD_WIDTH).
```

A coalesced access causes all of the work-items in a particular SIMD execution bundle to access a set of contiguous elements in memory for the given instruction or expression. To be tolerant to varying SIMD widths

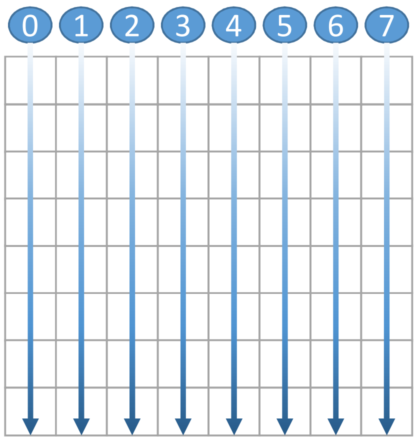


Fig. 2. Access pattern to a tile of matrix B in our example OpenCL program. Tasks within the work group perform wide, coalesced accesses to the memory system.

across architectures, many programs assign the entire work-group to a contiguous set of elements, such that any contiguous subdivision of the group will have a coalesced vector access.

Temporal locality is somewhat more complex to manage portably. Given that inter-group scheduling is out of the programmer's control, her focus is on temporal locality within each group. Futhermore, it is primarily temporal locality in accesses from multiple work-items that needs explicit management; task-private temporal locality is usually handled through simple register promotion. There are two possible approaches to achieving temporal locality in OpenCL: explicitly managed local memory buffers or assuming an implicitly managed cache. Older GPUs prevalent during OpenCL's drafting had very limited caching support, forcing programmers to manage temporal locality through explicitly-managed scratchpads. More recent GPUs from NVIDIA, AMD, and Intel all include memory caches all the way down to the L1 level, which simplifies but does not eliminate the need to specifically consider how temporal locality is managed. Even if a cache is present, it can be exploited one of two ways on a GPU: explicitly controlling the execution order with work-group barriers, or relying on implicit, round-robin scheduling patterns on GPUs to keep all work-items in a work-group roughly in phase with each other. Either of these mechanisms will ensure that memory locations accessed repeatedly by different work-items in the group will likely still be in cache.

In the given example, most accesses to global memory are perfectly coalesced across the entire work-group, because the index expressions on lines 11, and 15 are both of the form `uniform_base + get_local_id(0)`. In the example kernel, the priority of spatial locality is most clearly shown by the access to the input matrix B from line 11, which is shown graphically in Figure 2. The entire work group accesses wide lines of the matrix at a time before proceeding downward in the column direction. Therefore, because each access completely consumes an entire memory line, this kernel achieves a high percentage of peak global memory bandwidth consumption, even though more advanced tiling algorithms could reduce the total number of global memory accesses significantly.

The shown kernel also uses a functionally unnecessary barrier on line 13 to potentially improve temporal locality, particularly for accesses to the matrix A. Line 10 shows that the accesses to matrix A do not depend on the local work-item index. Accesses independent of work-item index are uniform across the work-items of the group, which is often an equally effective access pattern for GPUs with any kind of local caching mechanism. The presence of the barrier ensures that a particular tile of the A row remains in cache while all work-items in the group use it.

Note that controlling locality on GPUs always relies on being able to switch between actively executing work-items frequently and with low overhead. Round-robin instruction scheduling is another way of saying that the hardware makes frequent implicit moves between actively executing work-items to balance their progress. Frequent barriers are effectively programmer commands to suspend currently executing work-items at the barrier so that other work-items can catch up. Therefore, we can say that a fundamental requirement of an OpenCL implementation that supports performance portability for current developer practices is a low-overhead mechanism for switching execution between work-items.

```
 1 struct wi_state {
 2   int local_id[3], group_id[3], global_id[3];
 3   float result;
 4   float *A_line;
 5   int i;
 6   int ii;
 7   void *restart_point;
 8 }
 9
10 struct wi_state group_state[WORKGROUP_SIZE];
11 struct wi_state *awi;  //Active work-item
12
13 void barrier(int fence, void* restart) {
14   (awi++)->restart_point = restart;
15   if (awi = group_state + WORKGROUP_SIZE)
16     awi = group_state;
17   goto awi->restart_point;
18 }
19
20 void MatMul(float *A, float *B, float *C,
21             int g_id[3], g_size[3]) {
22   setup_wi_contexts(group_state);
23   awi = group_state;
24 kernel_start:
25   awi->result = 0.0f;
26   awi->A_line = A + awi->group_id[1]*A_WIDTH;
27
28   for (awi->i = 0; awi->i < A_WIDTH;
29        awi->i+= TILE_WIDTH) {
30
31       for (awi->ii = 0; awi->ii < TILE_WIDTH; awi->ii++)
32         awi->result += awi->A_line[awi->i + awi->ii] *
33             B[(awi->i+awi->ii)*B_WIDTH +
34                                   awi->global_id[0]];
35     barrier(CLK_LOCAL_MEM_FENCE, &&restart_0);
36     restart_0:
37   }
38   C[C_WIDTH*awi->group_id[1] + awi->global_id[0]] =
39       awi->result;
40   barrier(0, &&kernel_finish);
41 }
```

Fig. 3. C-like pseudocode representing AMD's OpenCL implementation.

III. Prior Implementations of OpenCL on CPUs

Much previous work has addressed the challenge of implementing OpenCL on x86 processors, both published academically and implemented industrially. Here, we cover those related works most directly related to our methodology and those that are most popularly used today. We will study each implementation as it relates to the running example in Figure 1.

The AMD CPU OpenCL language implementation is based on the Twin Peaks technology [16]. The primary insight of the implementation is that modern, multicore, superscalar, x86 CPUs support a relatively low level of thread-level parallelism, but a very high degree of instruction-level parallelism. Therefore, it makes most sense to combine all of the work-items in a group into a single CPU thread. The AMD CPU stack accomplishes this with user-level threading techniques, using irregular control flow to "simulate" multiple parallel work-items with a single user thread.

Figure 3 shows a pseudocode example of how this user-level threading is accomplished. First, the implementation declares a data structure suitable for holding all the data private to a single work-item, and then initializes a collection of such data structures to hold the state of all work-items in the group (details not shown.) The CPU thread calls the `MatMul` function with a particular work-group index, which the compiler has modified such that it will complete the execution of all work-items in the specified work-group. It initializes the local state of the work-group on line 22, and selects the work-item with index 0 to be the first *active* work-item. At any given time, the active work-item is the one being advanced through the program. To support multiple work-items with the same kernel code, a level of indirection is added, with `awi` pointing to the private data of the active work-item.

The program execution follows the original OpenCL kernel's operations, referring to the active work-item's private storage through `awi` for references to private variables. Execution of the first work-item continues until the

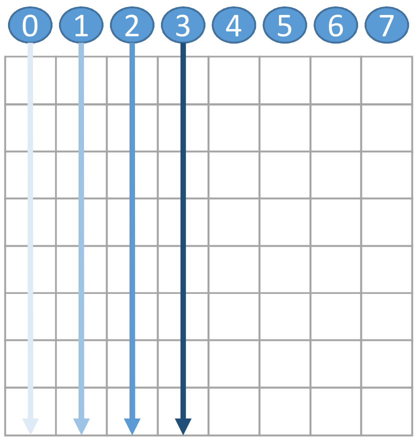


Fig. 4. Access pattern to a tile of matrix B in our example OpenCL program using the Twin Peaks methodology. Only accesses from the first few threads of the work-group are shown.

barrier statement on line 35. At the barrier, the framework saves the program location where the current work-item should be restarted, in this case the label `restart_0` on line 36. It then updates the `awi` pointer to the next work-item's private state, and performs an indirect jump to the location at which the newly activated work-item should be restarted. At initialization, all work-items have their restart points set to the `kernel_start` label, so in this case, the indirect jump takes takes the second work-item to the beginning of the program, as it should. The second work-item will then follow the same program path to the same barrier, at which point the framework will make the third work-item the active work-item, and so on, until all work-items in the group have reached the first barrier.

When the last work-item executes the first barrier, the condition on line 15 will evaluate to true for the first time, and restart the first work-item at `restart_0`, allowing it to continue execution where it left off. It will do so until it reaches the barrier again, where it will again save its current restart point and switch to the second work-item. The constant switching of active work-items continues until work-items begin to reach the kernel's end. At that point, the work-items save their restart point as some sentinel value that will lead the framework to the cleanup code to finish the current work-group, and prepare to execute another work-group if available.

The Twin Peaks methodology has several aspects that make it ill-suited to support the programming practices outlined in Section II. First, the overhead of changing the active work-item is significant. The example shown uses illegal label-passing to illustrate the concepts, but the real implementation is based on `setjmp` and `longjmp`. Even after significant optimization of those low-level routines for this context, the Twin Peaks authors claim an overhead of 10ns or thirty clock cycles per work-item change. Additionally, the micro-threading approach makes no effort to capture vector-level parallelism across work-items. Each work-item is executed in isolation, and any vectorization is limited to opportunities within the code of a single work-item.

Finally, the Twin Peaks implementation does not capture spatial locality as expected by the developer. Figure 4 shows a graphical representation of a single work-group's accesses to the input matrix B over the course of one tile. In a GPU implementation, with wide SIMD vectors and round-robin scheduling, large collections of contiguous addresses are accessed and consumed together. However, a serialization of work-items with the Twin Peaks methodology effectively executes all of a single work-item's accesses first, before the accesses of any other work-items. The kernel follows the guidelines to support SIMD across work-items, leading to interleaved accesses among work-items but strided accesses in the access stream of a single work-item. Figure 4 shows how the serialized implementation accesses a wide range of addresses in a short amount of time for the first work-item, followed by another set of strided accesses from the second work-item, and so on. If the tile size or the memory footprint of the work-group's total state gets large enough, this kind of access pattern will cause significant cache thrashing, and result in very poor spatial locality usage.

Figure 5 shows pseudocode for the result of prior region-based serialization work [17]. Some private variables such as `result` are expanded into an array of values, with one element for each work-item. However, analysis can often detect cases where private variables always store values uniform across the entire work-group, such as

```
 1 void MatMul( float *A, float *B, float *C,
 2     int g_id[3], // work-group ID
 3     int g_size[3]){ // work-group size
 4   float result[WORKGROUP_SIZE];
 5   float *A_line = A + g_id[1]*A_WIDTH;
 6   for (__x__ = 0; __x__ < g_size[0]; ++__x__) {
 7     result[__x__] = 0.0f;
 8   }
 9   for (int i = 0; i < A_WIDTH; i+= TILE_WIDTH) {
10
11     for (__x__ = 0; __x__ < g_size[0]; ++__x__) {
12       for (int ii = 0; ii < TILE_WIDTH; ii++)
13         result[__x__] += A_line[i + ii] *
14           B[(i+ii)*B_WIDTH + g_size[0]*g_id[0]+__x__];
15     }
16     //barrier(CLK_LOCAL_MEM_FENCE);
17   }
18   for (__x__ = 0; __x__ < g_size[0]; ++__x__) {
19     C[C_WIDTH*g_id[1] + g_size[0]*g_id[0]+__x__] =
20       result[__x__];
21   }
22 }
```

Fig. 5. C-like pseudocode representing region-based loop serialization.

the variable A_line, and avoid creating separate memory locations to store redundant information.

Instead of adding functionality to the barrier function, the compiler uses the very presense of the barrier function to inform analysis of the kernel code. The prior work describes the transformations more rigorously, but effectively, the kernel code is split up into contiguous regions that contain no barriers. Each region is then serialized with an inserted counted loop over the work-item indexes.

In Figure 5, one region occupies lines 6-8, initializing the private variable result for all work-items. A second region on lines 11-15 performs the primary computation, accumulating inner products for each column of B. The final region on lines 18-21 copies the final results to the correct region of the output space. The code regions themselves constitute nodes in a dynamic control flow graph independent of work-item index, with each dynamic region executed for all work-items. In the example kernel, there is a loop over the second regions, executing it for each tile of the input data, while the first and last regions are executed only once each.

The inserted serialization loops themselves then maintain the semantics of the original barrier, not letting any operations following the barrier in the dynamic execution completing before any operation before that barrier. Therefore, the barrier itself can be removed from the final code, as it adds no information or constraint not already represented by the serialized code.

From a portability standpoint, the region-based serialization methodology has several advantages over the Twin Peaks technique. First, the overhead of executing a barrier is significantly reduced. In this methodology, a barrier only adds a cost of a loop branch and loop counter increment in the worst case. In practice, the overhead is even smaller, because optimizing compilers apply optimizing transformations such as loop unrolling to the serialization loops. Such optimizing loop transformations are practically prohibited by the indirect jumps of the Twin Peaks methodology. Second, this implementation could indirectly result in SIMD vectorization across work-items, if the inserted serialization loops happen to be innermost loops, and a vectorizing compiler is able to conservatively prove the vectorizability of those loops. And finally, the implementation does not fundamentally solve the spatial locality expectation mismatch, as the access patterns remain largely unchanged. The CPU Scalar Access Pattern in Figure 4 still accurately describes the serialized access pattern of the main computation region: strided accesses along a column of the B matrix, followed by more strided accesses along subsequent columns.

Intel's implementation of OpenCL for x86 is both the most recent and the least explicitly disclosed or studied. Our best understanding is that the Intel implementation would behave somewhat like the pseudocode in Figure 6. The figure assumes that the implementation uses region-based serialization for simplicity, but this is not necessarily clear. What is more clear, and noteworthy, is the implementation's focus on explicitly combining multiple work-items into vectorized execution bundles. Instead of creating private, scalar data elements for every work-item, it will create vector data elements for each SIMD bundle, as the declaration of the variable result on line 4 shows. All serialization loops are effectively unrolled by a factor of SIMD_W, the width of the SIMD units, with each iteration performing operations on vector values.

```
 1 void MatMul( float *A, float *B, float *C,
 2     // work-group ID and size
 3     int g_id[3], int g_size[3]) {
 4   simd_float result[WORKGROUP_SIZE/SIMD_W];
 5   float *A_line = A + g_id[1]*A_WIDTH;
 6   for (__x__ = 0; __x__ < g_size[0]; __x__+=SIMD_W) {
 7     simd_store(result[__x__], simd_expand(0.0f));
 8   }
 9   for (int i = 0; i < A_WIDTH;
10        i+= TILE_WIDTH) {
11
12     for (__x__ = 0; __x__ < g_size[0]; __x__+=SIMD_W) {
13       for (int ii = 0; ii < TILE_WIDTH; ii++)
14         simd_accumulate(&result[__x__] , A_line[i + ii] *
15             simd_load(&B[(i+ii)*B_WIDTH +
16                          g_size[0]*g_id[0]+__x__]));
17     }
18     //barrier(CLK_LOCAL_MEM_FENCE);
19   }
20   for (__x__ = 0; __x__ < g_size[0]; __x__+=SIMD_W) {
21     simd_store(&C[C_WIDTH*g_id[1] +
22         g_size[0]*g_id[0]+__x__], result[__x__]);
23   }
24 }
```

Fig. 6. C-like pseudocode representing Intel's vectorizing OpenCL implementation.

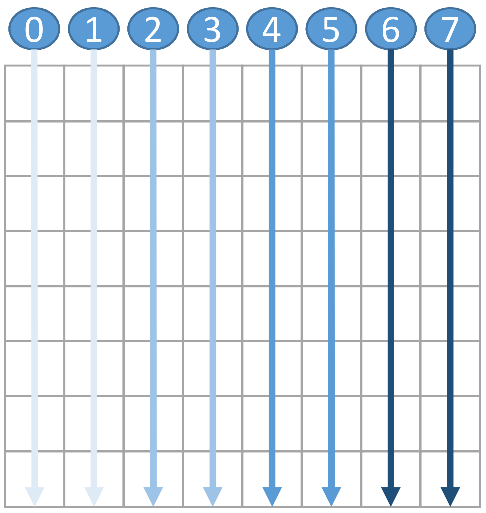


Fig. 7. Access pattern to a tile of matrix B in our example OpenCL program using the work-item SIMD bundling. For illustration, we assume a SIMD width of 2.

In practice, for recent CPUs, Intel's methodology works very well compared to the other techniques already described. It does map multiple work-items to the SIMD units of the architecture, mirroring the expected behavior as described in Section II. The barrier overhead of the implementation is not clear from the disclosed materials, but experimentally seems to be somewhere between the region-based methods and the Twin Peaks method. The explicit combining of work-items into SIMD units does assist in the capturing of spatial locality, but still does not use the caches as effectively as they could. The CPU 2-wide SIMD access pattern in Figure 7 shows why. For GPUs, the effective SIMD width of the processor is very wide, and the cache line size is closely matched to the SIMD data vector width for 32-bit words. In CPUs, while the SIMD widths have increased recently, the cache lines are still significantly larger than the SIMD data vector width. Therefore, a single SIMD access will utilize a smaller portion of the cache line by itself. In a kernel written according to the OpenCL programming guidelines, other work-items in adjacent SIMD bundles would be consuming the rest of that data. However, the overall control flow of the compute region on lines 15-20 of Figure 6 still executes all of the accesses for one SIMD group before any

Array Declaration	`int array[ARRAY_SIZE]`
Full array slice	`array[:]`
Bounded array slice	`array[100:100]`
Indirect gather or scatter	`array[indexes[:]]`

Fig. 8. CEAN array slice notation examples.

accesses from the next SIMD group. The final result is an access pattern that looks like the Figure 7, somewhere between the completely serialized and completely vectorized access patterns.

In summary, in the previous implementation methodologies we show some common performance insights and common portability oversights. It is clear that the work-items in a single work-group should be combined into a single, sequential CPU thread. Work-items within a group are primarily a source of vector- and instruction-level parallelism, both of which CPU architectures exploit from within a single CPU thread. The CPU implementations vary widely in their approach to serializing work-items and capturing SIMD parallelism from the work-items, with the Twin Peaks method vectorizing only explicit vector operations within a work-item, region serialization relying on autovectorization technology, and Intel's methodology directly targeting SIMD instructions. And finally, no current CPU implementation does an excellent job of handling spatial locality given the most common OpenCL programming practices. Instead, they each result int some kind of strided access pattern by executing one or more work-items as long as possible instead of interleaving the accesses of the work-items that would consume the elements of a particular cache line.

IV. Mapping OpenCL Performance Conventions to CPUs

The previous two sections summarized the implicit performance assumptions held by most OpenCL developers, and how prior work implementing OpenCL on CPUs fails to match some of those assumptions. In this section, we present one potential approach to mapping the performance conventions of OpenCL programs to CPU architectures. The goal of this implementation is to evaluate the practical impact to be had by honoring those performance conventions.

We propose a vector-based serialization of each work-group. Even though the physical SIMD width of a machine is of a fixed and limited value, the programming model's usages would benefit from executing work-groups in a way that emulates a work-group-wide vector machine. If instead of advancing only a small number of work items until they are forced to yield, an implementation could execute each dynamic statement for all work-items in the group before moving on to the next statement. The C Extensions for Array Notation (CEAN) programming model provides an excellent mechanism for describing just such execution semantics.

A. C Extensions for Array Notation

Intel introduced CEAN as part of their production compiler in 2010. It has also been implemented in gcc, although not integrated into the trunk, and proposed to the C++ standards committee as an industry-standard extension of C and C++ It is very similar to, and likely inspired by, FORTRAN-style array operations. The basic syntax is shown in Figure 8. An *array slice expression* is an array subscript expression (C99 6.5.2.1) that uses an *array slice operator*. The two most relevant array slice operator types are the *full slice* and *bounded slice* operators. A full slice operator, syntactically expressed with a single semicolon as the subscript expression, can only be used on arrays with a known size, and evaluates to the entire contents of the array. A bounded slice operator can be used on any array or pointer, and is a subscript expression of the form:

array_ptr [*base_index* `semicolon` *extent*].

The base index determines the offset of the first element of the slice, and the extent value determines the number of contiguous elements that should be extracted in the slice. The example bounded array slice in Figure 8 accesses a 100-element slice from the array, beginning with index 100 and ranging to index 199. A bounded slice will always result in an array value with a number of elements equal to the extent. Multi-dimensional slices are permitted, but we will restrict ourselves to single-dimensional array operations for this paper. Array expressions can also be used

```
 1 void MatMul( float *A, float *B, float *C,
 2     // work-group ID and size
 3     int g_id[3], int g_size[3]) {
 4   float result[WORKGROUP_SIZE];
 5   float *A_line = A + g_id[1]*A_WIDTH;
 6
 7   result[:] = 0.0f;
 8   for (int i = 0; i < A_WIDTH; i+= TILE_WIDTH) {
 9
10       for (int ii = 0; ii < TILE_WIDTH; ii++)
11         result[:] += A_line[i + ii] *
12           B[(i+ii)*B_WIDTH + g_id[0]*g_size[0]:g_size[0]];
13     //barrier(CLK_LOCAL_MEM_FENCE);
14   }
15   C[C_WIDTH*g_id[1] + g_id[0]*g_size[0]:g_size[0]] =
16     result[:];
17 }
```

Fig. 9. CEAN-based result of our proposed OpenCL implementation.

as array subscript expressions into other arrays, which is useful for defining indirect gather and scatter accesses. Indirect accesses tend to be significantly slower than full or bounded accesses in practice.

Operations on multiple array expressions must operate per-element across the extent of all involved array expressions. For instance, adding a scalar to an array expression will result in a new array expression with the scalar addition applied to every element of the array. Adding a pair of array expressions means an element-wise addition, and requires that the two array expressions have the same number of elements.

B. *Implementing OpenCL with CEAN*

Figure IV-A shows how we can apply CEAN-style transformations similar to the way previous work applies loop-based serialization. As with previous work, we expand the `result` private variable into an array, because its value depends on work-item index. However, instead of introducing loops over the kernel code, we simply replace the scalar expressions in the code with array slices where appropriate. Accesses to the `result` local variable on lines 7, 11, and 16 use a full slice expression over the array. Accesses to the global memory could use the indirect array access expression syntax in the general case. However, in the example code, all global memory accesses are provably coalesced across the entire work-group. They can therefore be converted into the faster bounded slice operations by decomposing the index operation into the form *base_index + get_local_id(0)*. Once the base index expression has been identified, the compiler can generate a bounded array slice beginning at that base index and with an extent equal to the work-group size. For the accesses to `B` and `C`, the compiler must first apply the following equivalence:

```
get_global_id(0) ==
get_group_id(0)*get_group_size[0] + get_local_id(0)
```

The group index and size are then incorporated to the base index expression for the array slice expressions, as seen on lines 12, and 15. The result of all these transformations is a program that expresses the execution of the work-group as a sequence of vector operations over local variables.

CEAN has two important properties that make it very well suited to describing OpenCL work-group execution. First, array expression operations were specifically introduced to support SIMD execution on CPUs. Operations over array expressions are explicitly independent across all elements, and therefore directly targeted as vectorization opportunities. Second, the execution semantics are such that each statement using array slice expressions is evaluated in its entirety before the next statement executes, just as it would if all operations were only scalar. This creates the kind of access pattern that actually achieves the spatial locality the developer intended. And like in the prior region-based serialization approach, barriers are rendered irrelevant in the final code. The array slice ordering constraints essentially provide the same ordering as if there were a barrier between every pair of statements.

Note that this choice of scheduling has strong implications for the local layout of data within the work-group. The Twin Peaks authors specifically defends their choice of storing all the private data for a single work-item contiguously in memory in a data structure. Their claim is that such a layout will get the best spatial locality [16] (although admittedly stating that more research was needed on the topic.) This makes sense given their execute-until-yield serialization model. If one work-item is going to be executed for a long time, it makes most sense that all its private data would be close together, and not interleaved with the data from other work-items. However, it makes vectorization across work-items inefficient. In order to efficiently combine multiple work-items into a SIMD bundle, all instances of the local variables for work items in that bundle should be contiguously stored.

V. Practical Implementation and Methodology

To the best of our knowledge, prior implementations of region-based serialization were only available for the CUDA language. For comparisons with a uniform codebase, we implemented both region-based serialization and vector serialization in our compiler, so that each could be compared with each other and with industry implementations.

Our compiler is implemented as a two-stage framework, the first of which performs source-to-source translation from OpenCL to C, performing the high-level transformations in the process. It will transform the kernel code itself into a new function representing the computation of a single work-group. It will also generate a wrapper function to marshall arguments and execute each work-group as an iteration of an OpenMP parallel-for loop. The second phase of the compiler is an off-the-shelf C compiler with OpenMP support, in our case using version 13.0 of Intel's C compiler with the optimization flags `-O3 -xHOST` enabled. The resulting object file is linked with a library implementing all of the OpenCL built-in kernel functions, resulting in a shared library that can be dynamically loaded. The wrapper function for a kernel is invoked by the OpenCL runtime at kernel launch to perform the kernel computation.

In the case of region-based serialization, the high-level transformations performed by the translator include region formation alanysis, as described in previous work [17]. Region formation is the identification of textual regions of input code that can be safely iterated sequentially for all work-items in a work-group. The primary constraints are that regions must be properly nested with the existing control flow constructs within the program itself, and must not contain any barrier operations. Once the regions have been identified, each declared private variable is analyzed for control- or data-dependence on the local work-item index. Those variables that do vary across work-items are expanded into an array of values for each work-item.

When the translator performs vector-based serialization of work-items, it performs the same region formation and work-item-dependence analyses as described above. It then checks the contents of each region to determine whether the region can be serialized with CEAN notation. For instance, loops with work-item-dependent trip counts cannot be expressed in CEAN notation. When such cases are detected, the compiler tries to divide the region into smaller subregions, so that as many statements as possible are within regions that meet the requirements for vector serialization.

Our translator is implemented as an automatic source-to-source transform in the Clang frontend [18] of the LLVM compiler infrastructure [19]. We evaluate our proposed OpenCL implementation on an Intel Core$^{\text{TM}}$i7-3770 CPU, with 256-bit vector units supporting the AVX instruction set, using version 13.0 of Intel's C compiler with the optimization flags `-O3 -xHOST` enabled, and running Ubuntu GNU-Linux (Linux kernel version 3.2.0-32).

VI. Experimental Results

We demonstrate the performance of the proposed approach by running OpenCL and OpenMP benchmarks. We have experimented with two benchmark suites; Parboil [20] and Rodinia [21]. They both come with benchmarks that contains functionally equivalent OpenCL and OpenMP implementations in most cases.

We run the OpenCL benchmarks over four different OpenCL implementations analyzed in this paper. The Intel and AMD implementations are the publicly available versions. The prior region-based serialization proposal and the new CEAN-based vector serialization proposal are both enabled in our own implementation. Figure 10 shows the speedup of the OpenCL implementations relative to the slowest implementation for each benchmark.

The benchmarks that both follow and rely on implicit performance conventions heavily show a dramatic performance increase with our vector-based serialization, including `kmeans`, `sgemm`, `lud`, `cfd`, `backprop`, and `fft`. This is not to say that these benchmarks are simple; the `fft` butterly access pattern is far from trivial to form into coalesced operations, for instance.

Although the benchmarks were written specifically for a GPU architecture, not all of them follow the performance guidelines we describe in this paper. In particular, the `bfs` benchmark is fundamentally a task-parallel algorithm, and does not embody good data-parallelism or memory locality. Other benchmarks may follow the performance

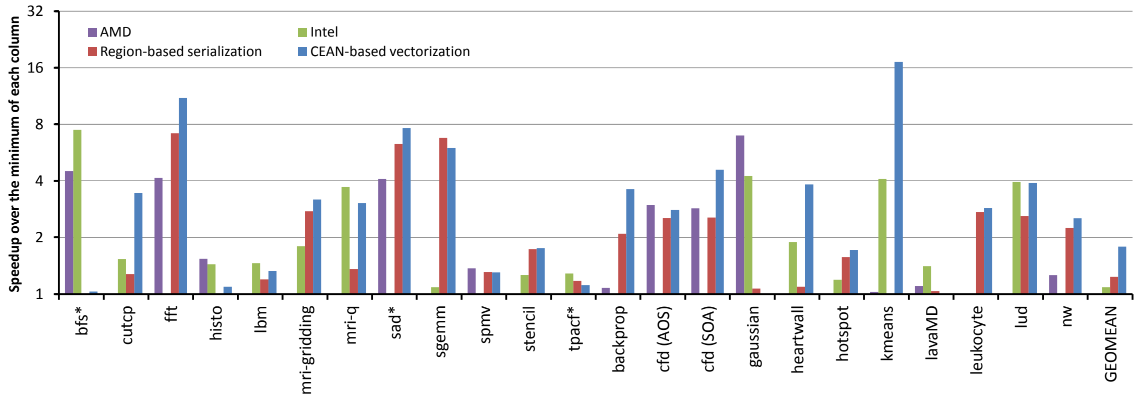


Fig. 10. Comparison of performance of OpenCL implementations. The baseline for each benchmark is an implementation takes the longest execution time.

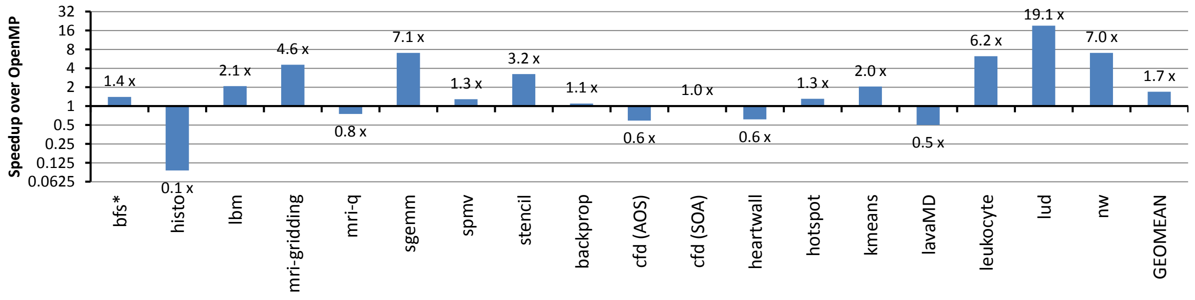


Fig. 11. Comparison of the MxPA OpenCL implementation with OpenMP.

conventions as much as possible, but are naturally less sensitive to implementations that diverge from those performance considerations, such as `lavaMD` or `tpacf`. Such cases usually indicate that the benchmarks use short vector types, enabling the AMD implementation to successfully vectorize operations, and are heavily compute-bound, making them less sensitive to differences in how locality is exploited in the target platform. Still other benchmark have very small runtimes, and we have experimentally determined that the overhead for the OpenMP work distribution used in the region- and vector-based serialization implementations significantly impacts the runtime for the `gaussian`, `bfs`, and `lud` benchmarks. Even when these cases are taken into account, the CEAN vector-based serialization is a full 1.7× faster than AMD's implementation on average.

We also compare the performance of OpenCL kernels on the CPU with OpenMP implementations. Such comparisons are difficult, because the source code implementations can vary significantly. For instance, the default implementation of histo for OpenMP uses a very small degree of parallelism, with significanly less overhead compared to the highly scalable OpenCL implementation. Other cases fall the other way, where the SIMD and locality expressions in the OpenCL kernels were superior to those of the OpenMP implementation, as we can see in lud and sgemm in particular. But with a large number of benchmarks, we can see that, on average, the OpenCL implementations are, if anything, higher-performing on the CPU architecture than the OpenMP implementations, even though the OpenCL implementations were written with a GPU platform specifially in mind. This result boasts the success of performance portability in practice, even if, in theory, the OpenMP implementations could be improved. The OpenCL implementations could also be improved to more closely conform to the OpenCL performance conventions we describe, increasing their performance on the CPU archtecture as well.

VII. Related Work

Performance portability has been a concern as long developers have had the desire to target multiple architectures. For instance, Jiang et al. were concerned about the performance portability between Multi-socket CPU systems with and without hardware cache coherence as far back as 1997 [22]. Performance portability between different GPU devices was noted a concern by early adopters [23], [24], but that since been tempered by the adoption of the portability guidelines described here, supported by newer tools [13].

Aside from the prior OpenCL and CUDA implementations we discuss at length, other have built systems for executing GPU-style kernels on CPU platforms. Kerr et al. implemented the CUDA programming model on CPUs with the GPUOcelot project [25]. However, performance seemed to be a secondary concern, as the tools is primarily developed to enable better debugging, profiling, and tracing of CUDA applications. It does take a more vector-execution approach, similar to the one we propose, but sacrifices performance for flexibility and the other high-level features mentioned. The Portable Compute Language project [26] aspires to be an open-source, high-performance implementation of OpenCL on CPUs. It has two methods of code generation, roughly corresponding to the region-based and vector-execution methodologies described in this paper. However, the project is somewhat immature, and performance results comparing itself with industry implementations have not been published to the best of our knowledge.

VIII. Conclusions

Performance portability requires a programming language and well-defined performance convention within that language. The natural pull for a language implementor is to map language constructs to architecture constructs in a straightforward way, minimizing the cost of the implementation. However, implementors for different architectures will be pulled in different directions if this methodology is followed. A collection of implementations guided by architecture idiosyncrasies eliminates performance portability, and creates a huge programming burden for application developers.

However, the community is already moving towards an abstract performance convention for OpenCL for portability between multiple GPU devices. The performance convention embodies preferred methods for expressing data-parallelism, task parallelism, spatial locality, and temporal locality. To achieve performance portability, vendor implementations of the language must determine how to adopt those abstract performance expressions to their architecture, in addition to obeying the functional specification of the language. We have demonstrated that a CPU implementation of OpenCL following those guidelines outperforms the currently available implementations for OpenCL workloads following these conventions.

Programming conventions have limitations. There is some measurable cost paid to conform not only to a language specification but to a particular performance model. The OpenCL benchmarks we studied even included several examples of algorithms that simply have no tenable expression that conforms to the portable performance guidelines. Future work should focus on what the boundaries of the current performance conventions, and add features to the language that broaden the scope of algorithms that can be portably expressed.

Our proposed methodology has been rigorously tested and used for several industry OpenCL application projects. The lead developer of one of those projects, a video processing library, once sent an email with these words. "I just happened to re-run the multi-core performance tests for the VPL today and discovered that the changes we have done recently to improve AMD (GPU) performance have also improved our (CEAN-based) performance." This is what developers want, the ability to optimize a piece of software once, and have those optimization efforts be favorably reflected on many architectures. We have shown in this paper that performance portability is feasible, when implementations of a language focus on a common performance convention.

Acknowledgements

This research was conducted with the partial support of the following agencies: the Gigascale Research Center of the Semiconductor Research Corporation, the Microsoft/Intel Universal Parallel Computing Research Center, NVIDIA Corporation, MulticoreWare Inc., and the Department of Energy under the Vancouver project (grant #DE-SC0005515.) The opinions expressed in this paper are solely those of the authors, and do not necessarily reflect those of any organization here mentioned.

References

[1] Top500.org, “Top500 list,” Sept. 2012.

[2] J. Nickolls, I. Buck, M. Garland, and K. Skadron, “Scalable parallel programming with CUDA,” *ACM Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[3] Khronos OpenCL Working Group, “The OpenCL specification, version 1.2,” nov 2011.

[4] Microsoft Corporation, “C++ AMP: Language and programming manual, version 1.0,” aug 2012.

[5] “The OpenACC application programming interface, version 1.0,” nov 2011.

[6] J. A. Stratton, S. S. Stone, and W.-m. W. Hwu, “MCUDA: An effective implementation of CUDA kernels for multi-core CPUs,” in *Proceedings of the 21st International Workshop on Languages and Compilers for Parallel Computing*, pp. 16–30, July 2008.

[7] S. Seo, G. Jo, and J. Lee, “Performance characterization of the NAS parallel benchmarks in OpenCL,” in *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pp. 137–148, nov 2011.

[8] M. Pharr and W. Mark, “ispc: A SPMD compiler for high-performance CPU programming,” in *Proceedings of the IEEE conference on Innovative and Parallel Computing*, 2012.

[9] NVIDIA Corporation, “Nvidia cuda programming guide 5.0,” 2012.

[10] Advanced Micro Devices, “”amd” accelerated parallel processing ”opencl” programming guide,” may 2012.

[11] Intel Corporation, “Intel OpenCL optimization guide,” Apr. 2012.

[12] AMD, “ATI Stream SDK openCL Programming Guide,” Mar. 2010.

[13] M. Delahaye, “Quickly optimize OpenCL applications with SlotMaximizer,” 2002.

[14] V. Volkov and J. W. Demmel, “Benchmarking GPUs to tune dense linear algebra,” in *Proceedings of the ACM/IEEE conference on Supercomputing*, (Piscataway, NJ, USA), pp. 31:1–31:11, IEEE Press, 2008.

[15] J. A. Stratton, C. Rodrigues, I.-J. Sung, L.-W. Chang, N. Anssari, G. Liu, W.-m. W. Hwu, and N. Obeid, “Algorithm and data optimization techniques for scaling to massively threaded systems,” *Computer*, vol. 45, no. 8, pp. 26–32, 2012.

[16] J. Gummaraju, L. Morichetti, M. Houston, B. Sander, B. R. Gaster, and B. Zheng, “Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors,” in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT ’10, (New York, NY, USA), pp. 205–216, 2010.

[17] J. A. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu, and W.-m. W. Hwu, “Efficient compilation of fine-grained spmd-threaded programs for multicore CPUs,” in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pp. 111–119, ACM, 2010.

[18] “clang: a C language family frontend for llvm.”

[19] C. Lattner and V. Adve, “”llvm”: A compilation framework for lifelong program analysis & transformation,” in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, (Washington, DC, USA), p. 75, ”IEEE” Computer Society, 2004.

[20] J. A. Stratton, C. Rodrigrues, I.-J. Sung, N. Obeid, L. Chang, G. Liu, and W.-M. W. Hwu, “Parboil: A revised benchmark suite for scientific and commercial throughput computing,” Tech. Rep. IMPACT-12-01, University of Illinois at Urbana-Champaign, Urbana, Mar. 2012.

[21] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” *IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[22] D. Jiang, H. Shan, and J. P. Singh, “Application restructuring and performance portability on shared virtual memory and hardware-coherent multiprocessors,” in *Proceedings of the sixth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP ’97, (New York, NY, USA), pp. 217–229, ACM, 1997.

[23] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, “From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming,” *Parallel Comput.*, vol. 38, pp. 391–407, Aug. 2012.

[24] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi, “Evaluating performance and portability of opencl programs,” in *The Fifth International Workshop on Automatic Performance Tuning*, June 2010.

[25] A. Kerr, G. Diamos, and S. Yalamanchili, *GPU Computing GEMS Jade Edition, 1st Edition*, ch. 30. Morgan Kaufmann, 2011.

[26] P. Jääskeläinen, C. de La Lama, P. Huerta, and J. Takala, “Opencl-based design methodology for application-specific processors,” in *Embedded Computer Systems (SAMOS), 2010 International Conference on*, pp. 223–230, July.

Scheduling of Stream-Based Real-Time Applications for Heterogeneous Systems

Bruno Virlet, Xing Zhou, Jean-Pierre Giacalone[†], Bob Kuhn[†],
María Jesús Garzarán, and David Padua

University of Illinois at Urbana-Champaign
{virlet1,zhou53,garzaran,padua}@illinois.edu

[†]Intel Corporation
{jean-pierre.giacalone,bob.kuhn}@intel.com

Abstract

Designers of mobile devices face the challenge of providing the user with more processing power while increasing battery life. Heterogeneous systems offer some opportunities to solve this challenge. In an heterogeneous system, multiple classes of processors with dynamic voltage and frequency scaling functionality are embedded in the mobile device. With such a system it is possible to maximize performance while minimizing power consumption if tasks are mapped to the class of processors where they execute the most efficiently.

In this paper, we study the scheduling of tasks in a real-time context on a heterogeneous system-on-chip that has dynamic voltage and frequency scaling functionality. We develop a heuristic scheduling algorithm which minimizes the energy while still meeting the deadline. We introduce the concept of cross-platform task heterogeneity and model sets of tasks to conduct extensive experiments. The experimental results show that our heuristic has a much higher success rate than existing state of the art heuristics and derives a solution whose energy requirements are close to those of the optimal solution.

Categories and Subject Descriptors I.2.8 [*Problem Solving, Control Methods, and Search*]: Scheduling; C.1.3 [*Other Architecture Styles*]: Heterogeneous (hybrid) systems

General Terms Algorithms, Performance

Keywords Dynamic voltage and frequency scaling, heterogeneous system, scheduling

1. Introduction

Advances in portable devices demand higher speed of computation as well as longer operational autonomy. To address the challenge of these opposite goals we study techniques that balance program execution speed and energy consumption in the context of typical portable device applications. We focus on streaming computations, which are networks of tasks that operate on a data stream that flows into the computation at a fixed rate [10, 20]. Upon completion, each task passes its output to its successor(s) in the network. Streaming computations are often used to implement video post-processing applications, which are among the most important for the future of mobile devices. In video post-processing computations, tasks implement filters and the data stream is a sequence of video frames. The computation is typically subjected to a real-time constraint which is to display between 15 and 30 frames per second for many of today's video post-processing applications.

The objective of the techniques discussed in this paper is to minimize the energy consumed by streaming computations under the constraint of a minimum output rate. We assume that there could be multiple classes of processors embedded in the mobile device and that they have dynamic voltage and frequency scaling (DVFS) functionality. Since the maximum possible frequency is not typically required to achieve the desired output rate, energy consumption can often be minimized by lowering the voltage and hence the frequency as much as the real time constraint allows. This minimization process is complicated by three situations. First, only a discrete number of frequencies are possible in today's machines. Second, changing the voltage/frequency consumes time, which means that such changes must be applied judiciously. Third, energy efficiency can be controlled not only by DVFS but also by choosing the class of processor on which to map each task since different processors are efficient for different types of tasks. A GPU will excel in vector operations but will be inferior to a conventional CPU for applications rich in control flow. An example of Systems-on-a-Chip (SOC) that integrate a multiprocessor core with graphics cores is the AMD Fusion Zacate E350/E240 [1]. Another example is the PowerVR SGX 5XT from Imagination [2] for smartphones and other mobile devices, which can be integrated with a multiprocessor core to obtain a heterogeneous SOC. Although these systems are not yet available for mobile devices, we expect them to reach the market soon.

The technique that we propose in this paper takes the form of a two step heuristic that first chooses on what class of processors to map each task and then uses a homogeneous scheduling algorithm to apply the voltage and frequency scaling within each homogeneous subsystem. We allow frequency scaling at the granularity of the tasks. This enables us to place the code for frequency scaling at natural locations. Our heuristic outperforms other heuristics such as Greedy and LR, both presented in [22]. It has a high success rate at finding a feasible schedule and, for most of the cases we studied, the resulting schedule is within 5% of the one of the optimal schedule. It also offers very stable results when the architecture heterogeneity and task uniformity vary. Additionally, the memory usage is linear in the size of the input. Finally, we demonstrate the importance of the concept of the cross-platform heterogeneity of a

LCTES'11, April 11–14, 2011, Chicago, Illinois, USA.
Copyright © 2011 ACM 978-1-4503-0555-6/11/04...$10.00

task in the choice of the processors and show how combining two heuristics enables even better results.

This paper is organized as follows. Section 2 discusses related work. Section 3 describes the problem we are trying to solve. Section 4 discusses our scheduling algorithm. Section 5 gives some details about other techniques that we use to compare against our proposed scheduling algorithm. Section 6 describes the environmental setup that we use to run the experiments in Section 7. Finally, Section 8 discuss future work and our final conclusions.

2. Related Work

Task scheduling for power efficiency is a fairly recent problem. There have been studies on scheduling for one processor with DVS capabilities [19, 23, 12, 14] and for homogeneous multiprocessors also with DVFS capability [16, 17, 5]. In Section 4.1.3, we discuss one of the heuristics for homogeneous multiprocessors, the SpringS algorithm [17]. We use this algorithm as part of our scheduling strategy for heterogeneous systems.

Mixed software-hardware strategies in which the application is partitioned between hardware and software components [11] have also been studied but they require some of the scheduling to happen at hardware design time. Mixed strategies may offer higher performance because they make use of specialized hardware.

The problem of scheduling on heterogeneous processors is discussed only in a few papers. The case of heterogeneous single-voltage setup system has been studied in [9, 13]. The single-voltage setup problem consists in choosing a fixed frequency for each processor. This technique helps system designers choose the most efficient operating point for the products. However, such systems clearly lack flexibility when optimizing different applications.

Luo and Jha addressed the heterogeneous scheduling problem with continuous voltage scaling [18]. They assume that the frequency can be scaled continuously between a minimum and a maximum value. Yang, Chen, Kuo and Thiele [21] study the heterogeneous multi-level voltage scheduling problem and propose an algorithm which accepts a factor $\rho > 1$ and generates schedules whose energy consumption are less than ρ times the optimal possible energy required.

To the best of our knowledge the paper by Yu and Prasanna [22] is the only one so far to propose a reasonably efficient algorithm for the problem of scheduling in a heterogeneous system with multi-level discrete frequencies. They propose the LR heuristic that we discuss in Section 5.3 and evaluate in our experimental results in Section 7. The LR heuristic is a fast heuristic algorithm based on the linear relaxation of the linear-programming problem. The heuristic we introduce in this paper outperforms the LR heuristic by succeeding in multiple cases where LR fails, as shown in the experimental results in Section 7.

3. Problem formulation

3.1 Hardware and Power Model

We assume that the target system contains m processing elements (PE) P_j for $j = 1 \dots m$. We also assume a finite number of possible frequencies. The system considered is heterogeneous, which means that it contains different classes of processors (or processor species): we assume that there are q different processor types $S_1 \dots S_q$ in the system. We say that $P_j \in S_k$ if P_j is of type S_k.

For each processor type S_k there is a set F_k of allowed frequencies. If the processor type does not support DVS, the set F_k is reduced to the singleton containing the only frequency offered by processors of type S_k.

The power consumed by the system will fluctuate over time depending on which processor units are in use. The power function indicating the average energy consumption rate of the processor as a function of the frequency can be obtained from specifications or can be measured. For each frequency $f \in F_k$, let $p_k(f)$ be the associated average dynamic power of a processor of type S_k when running at frequency f. We will not consider the static power because we assume that it is constant. We assume that $\forall k \in [1, q]$, $f \longmapsto p_k(f)$ is an increasing function of the frequency. Generally, $p_k(f)$ is also convex such that a slight increase in the frequency at low frequency will not have much impact on power whereas the same increase at high frequency produces a much higher power increase. In fact, for CMOS DVS processors, the dynamic power function $p_k(f)$ can be approximated by $p_k(f) = Cf^3/\kappa^2$ where C is the switch capacitance and κ is a design specific constant [8].

3.2 Application Model

The scheduling problem of dependent tasks without dependence cycles can be reduced to the scheduling of a kernel of independent tasks as it has been shown by Liu et al. [17]. We will discuss this point more in depth in section 4.1.1.

Therefore, we will consider a set of n independent tasks T_i ($i = 1, 2, \dots, n$) and a deadline d defined as the maximum time allotted to process one dataset (in the case of video post-processing, this would be the time required to generate one video frame). We define C^S_{ik} as the number of cycles required to execute task T_i on a processor of type S_k. C_{ij} is also defined as the number of cycles to execute T_i on processor P_j. We assume that the number of cycles is not data dependent. This is the case for the video post-processing filters we studied. If it was data dependent, C^S_{ik} could be defined as the worst-case number of cycles so that the algorithm can guarantee the feasibility of the found schedule.

Let $\hat{C}^S_{ik} = \frac{1}{q}\sum_{k=1}^{q} C^S_{ik}$ be the average cycles of a task on the different types of processing units. We define the cross-platform *heterogeneity* of a task by:

$$H_i = \frac{\sum_{k=1}^{q}(C^S_{ik} - \hat{C}^S_{ik})^2}{\hat{C}^S_{ik}}$$

A task T_{i_1} is *more heterogeneous* than another task T_{i_2} if $H_{i_1} > H_{i_2}$. This means that choosing the right processor for T_{i_1} has a more significant impact on its execution time than choosing the right processor for T_{i_2} would have on T_{i_2}'s execution time.

Notice that the execution time of task T_i on processor P_j and at frequency f is $\frac{C_{ij}}{f}$.

3.3 Scheduling Problem

Let $V_z = (P_{j_z}, f_z)$ be a processor-frequency pair. Our goal is to find a mapping of each task T_i onto a pair V_z that minimizes energy consumption and produces results at the rate of $1/d$ results per second, where d is determined by the real-time constraint of the application. For the algorithm presented in this paper we ignore the communication costs. Thus, our model does not account for the time consumed to bring the data to the processor the first time that they are accessed or the time required to move data from the cache or memory in a processing element to the cache or memory of another processing element, independently of whether these two processing elements are of the same or of different type. To measure to what extent this simplification could affect our results we ran some experiments with four filters of a video post-processing application from MJPEGTools [3] and we observed that the processor was very effective at hiding the latency the first time the frame was brought to a core, most likely because the hardware prefetcher managed to hide the latency of the memory accesses due to the regular access pattern of the filters in this application. We do not have data for the transfer costs of the data between processing units of different types, but we expect that in a SOC the communication cost between processing elements of different types will be significantly smaller than today's cost between the processor cores and the off-chip GPU. In addition, prefetching (hardware or software) should be able to help at decreasing communication costs. Next, we describe more formally the scheduling problem that the algorithm presented in this paper is trying to solve.

Given S_k, the set of processing elements of type k and F_k, the set of frequencies allowed in each processor type, there are $v = \sum_{k=1}^{q} |S_k| \times |F_k|$ pairs. Let x_{iz} be 1 if task T_i is mapped to the pair V_z (which means that the task T_i will run on processor P_{j_z} at frequency f_z) and 0 otherwise. The value $e_{iz} = p_k(f_z)\frac{C_{ij_z}}{f_z}$ is the energy consumed by task T_i running at frequency f_z on processor P_{j_z} of type S_k. Given $V_z = (P_{j_z}, f_z)$, the fraction of processor P_{j_z} utilized by the task T_i when mapped to a pair V_z is $u_{iz} = \frac{C_{ij_z}}{d \times f_z}$ where d is the deadline. If this fraction is smaller than 1, there is still room in the processor to accommodate other tasks and still meet the deadline. The utilization U_j of processor P_j is the sum of the utilizations for all tasks: $U_j = \sum_{i=1}^{n} \sum_{V_z} u_{iz} x_{iz}$, where $V_z \in \{(P_{j_z}, f_z) | j_z = j\}$. For each processor P_j, the real-time constraint requires that $U_j \leq 1$.

The scheduling problem can be formulated as an integer linear programming problem ILP0 which consists in the minimization of:

$$\sum_{i=1}^{n} \sum_{z=1}^{v} e_{iz} x_{iz} \tag{1}$$

such that:

$$U_j \leq 1 \qquad 1 \leq j \leq m \tag{2}$$

$$\sum_{z=1}^{v} x_{iz} = 1 \qquad 1 \leq i \leq n \tag{3}$$

$$x_{iz} \in \{0, 1\} \qquad 1 \leq i \leq n, 1 \leq z \leq v \tag{4}$$

While Equation 2 states that the deadline must be respected, Equations 3 and 4 specify that each task has to be mapped entirely to one processor. We call *optimal solution* any solution to this problem, if it exists. We call *feasible schedule* any solution to this problem without the minimization constraint (Equation 1). Finally, a mapping of all the tasks which does not respect the time constraint (Equation 2) but satisfies Equations 3 and 4 is called an *unfeasible schedule*.

For discrete frequencies, finding the optimal schedule that minimizes the energy consumption while meeting the time constraint is clearly a NP-hard problem since the scheduling problem is NP-hard even ignoring the energy issues. We therefore must solve the problem with a heuristic to avoid the exponential complexity. The algorithm we use to address this problem is presented in the next section.

4. Scheduling Algorithm

4.1 Algorithm

Before scheduling an application, it is necessary to identify which tasks to schedule. In a first section we study how to build a kernel of independent tasks. Our heuristic has then two phases which apply to this kernel. The first, *mapping*, chooses the processor type for each task. The power function is used in that phase to guide the choice. The second phase, *frequency choice*, chooses one of the processors within the type selected in the first phase and the frequency for each task.

4.1.1 Task Set Identification

A stream-based application consists of a set of filters where each filter is applied in sequence to each input item. Thus, the stream application contains a total of n tasks, where $n = number\_of\_filters \times$

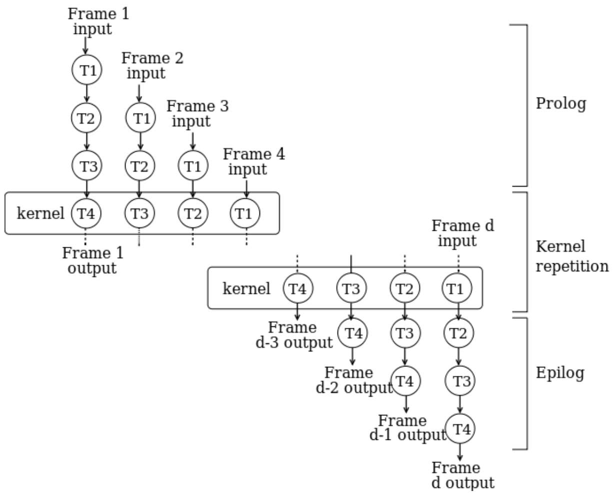


Figure 1: Kernel identification for four video filter tasks and d frames.

$number\_of\_input\_items$. In order to take advantage of the several processing units, we need to partition the total number of tasks into sets of independents tasks or kernels that can execute in parallel. To extract these kernels we follow an approach similar to software-pipelining [17]. This strategy can be easily applied when there are not data dependence cycles between the tasks, i.e., the task graph can be represented as a directed acyclic graph or DAG.

If we call $T_i[k]$ to filter T_i when applied to item k, the normal chain of dependencies would be $T_1[k] \rightarrow T_2[k], \cdots \rightarrow T_n[k]$. To obtain independent tasks one can select filters that apply to different items. This way $T_1[k], T_2[k+1], T_3[k+2], \ldots T_{m-1}[k+m-2], T_m[k+m-1]$ form a kernel whose tasks are independent on each other, as they operate on different items. Figure 1 illustrates an example when $m = 4$, where the tasks in the kernel repeat several times, and a prolog step at the beginning and an epilog at the end fill and drain the pipeline, respectively.

Notice that in the case of the video post-processing application we considered four different filters from the MJPEGTools [3]: denoise, sharpen, increase frame rate, and up-scale, where the input items to these filters were the frames of the video being processed.

Once a kernel of independent tasks has been built, it is usually possible to create sub-tasks. For instance, in the case of video post-processing applications, the filters can be easily tiled. Then, instead of considering task T_i working on frame k, we can consider two subtasks, $T_{i,1}$ and $T_{i,2}$ working respectively on two subsets of frame k. The result is a more flexible scheduling, as the increase of the number of tasks increases the number of tasks for the scheduler to choose from. In fact, our experimental results in Section 7 show that increasing the number of tasks has an impact on the optimality of the schedule.

Finally, consider the case where the dependence graph of the tasks is not a DAG, that is, there are back arcs due to dependences across data stream entities. In this case, applying software pipelining may produce a kernel with multiple steps instead of the single step that arises when the dependence graph is a DAG. The methodology described below can be applied to each one of these steps.

4.1.2 Mapping Phase

This Section describes the mapping of each task in the kernel to a processor type. The mapping algorithm is shown in Algorithm 1. The power function is used in this phase to guide the search. The power function is an increasing function, that is, the faster a processor computes, the more energy it consumes. Thus, to minimize energy consumption this algorithm uses a heuristic that maps tasks to processor types at the lowest possible frequency that still meets the deadline.

Algorithm 1 Heuristic algorithm

```
1: Input: kernel of independent tasks $T_i$, set of processors $P_j$.
2: Output: variables $x_{iz}$ set to 1 if task $i$ is mapped to the pair
   processor/frequency $V_z$.
3: for all i, j do
4:     $x_{ij} = 0$
5: end for
6: for each task $T_i$ in decreasing $H_i$ order do
7:     for each processor $P_j$ of type $S_k$ do
8:         $f_{opt,j} = \frac{\sum_{i'} x_{i'j}C_{i'j}+C_{ij}}{d}$.
9:         $f_{new,j} = f_{opt,j}$ round to the next discrete frequency
           available on $S_k$ or $max(F_k)$ if $f_{opt,j} \geq max(F_k)$.
10:        $\delta e_j = p_k(f_{new,j})[(\sum_{i'} \frac{x_{i'j}C_{i'j}}{f_{new,j}}) + \frac{C_{ij}}{f_{new,j}}] -$
           $p_k(f_j)\sum_{i'} \frac{x_{i'j}C_{i'j}}{f_j}$
11:    end for
12:    Choose the pair $V_z$ such that $\delta e_z$ is minimal and task $T_i$ fits
       on $P_{j_z}$ if $T_i$ and all the tasks already assigned to $P_{j_z}$ were to
       run at maximum frequency. Fail if no such processor exists.
13:    $x_{iz} = 1$.
14: end for
```

The algorithm assigns processors to tasks following a decreasing cross-platform heterogeneity order (line 6), where the cross-platform heterogeneity of a task is computed as shown in Section 3.2. By following this order, we are giving more choices to those tasks whose energy consumption is more affected by the type of processor where they run. The end goal is to minimize the overall energy consumption. Our experimental results in Section 7.2 will show the effectiveness of this heuristic.

Then, the algorithm computes the “insertion frequency” of this task on each processing element and the corresponding energy increase if the task were to be mapped on this processing element (lines 7 to 11).

The “insertion frequency” $f_{opt,j}$ is the ideal frequency [15, 6] at which the processor P_j should run so that all the tasks already assigned to P_j finish within the time constraint while minimizing the energy consumed. This frequency converges to an approximation of the frequency at which all the tasks on the processor should run in an ideal situation. In general this frequency is not available and the tasks would have to run at the smallest higher discrete frequency available $f_{new,j}$. If the insertion frequency is higher than the highest available frequency, this processor will not be able to execute this task and all the previously assigned tasks within the deadline d. In this case, if space is not available on another processor, the scheduling algorithm fails.

At line 12 we choose the processor which results in the minimum energy increase when the task is assigned to that processor at the insertion frequency. We also make sure that the deadline is always met when running at the maximum frequency. By checking this, the algorithm makes sure that it is not generating an infeasible schedule. It might be possible that, at this step, no processor can run an additional task within the defined deadline d. If this is the case, the heuristic fails. This, however, does not mean that no feasible schedule exists. Failure is inherent to the heuristic approach. In section 4.2, we discuss ways to reduce the failure rate in finding a feasible solution.

Finally, we record the pair processor/frequency V_z where the task under consideration will run (line 13) and proceed to the next task.

4.1.3 Frequency Choice Phase

Once this first phase finishes, each task is associated with a given processor type S_k. In the process, we actually assigned each task to a specific processor. In this last phase, we rearrange the tasks mapped to processors of the same type and assign them a final processor and frequency. We consider each group of processors of the same type in Algorithm 2 and apply to them an homogeneous scheduling technique as shown in Algorithm 3. We chose to use the SpringS algorithm by Liu et al. [17]. SpringS reorganizes the tasks mapped to the processors of the same type and search for the appropriate frequency. Applying the SpringS algorithm is only possible because the processors of the same type have the same characteristics (frequency and cycles for each task). This algorithm starts with an existing schedule. In our application, we start with the schedule found by the mapping phase and we reset all the frequencies to the minimum frequency (Algorithm 3, line 3). Then the SpringS algorithm, as its name indicates, behaves like a Spring. If a processor does not meet the deadline, that is, its utilization is larger than 1, it will find the best task for which to increase the frequency (the one that results in the smallest energy increase) and try to reschedule a subset of the tasks (lines 7 to 13). On the other hand, if the schedule has some slack to meet the deadline, it tries to slow down a task to save some energy (lines 15 to 17). After this phase, the variables x_{iz} are final and define a feasible schedule for ILP0.

Notice that in the case of homogeneous scheduling, the mapping phase can be skipped and our heuristic is reduced to the SpringS heuristic.

Algorithm 2 Frequency Choice Phase

```
for each set $\mathcal{S}_k$ do
    Apply the SpringS algorithm to $\mathcal{S}_k$ with the schedule found
    by the mapping phase.
end for
```

4.2 Improving the Heuristic

Heuristics can fail to find a feasible solution, as shown by our experimental results in Section 7. Thus, we have search different solutions to improve the success rate of the heuristic. Heuristics run fast. Hence, the first solution is to run a different heuristic in combination with our heuristic. We call this combination the *hybrid heuristic*. This heuristic selects the best results between LR-heuristic [22], which we will present later, and our heuristic. As we will see in the next section, not only does the hybrid heuristic help improve the optimality but it also reduces the failure rate of the scheduler. In addition, since both heuristic run fast, this can also be used as a strategy to improve a given schedule.

Additionally, it is clear that solving the scheduling problem for a deadline tighter than the required constraint also satisfies the original problem. For instance, if the original problem requires a constraint d, any solution to the same problem with a new constraint βd with $0 \leq \beta < 1$ is also a solution to the original problem. Although the failure rate is higher for tighter constraints, the heuristic algorithms are sensitive to small changes in the constraint since the deadline is involved in the computation of the optimal insertion frequency (line 8 of Algorithm 1). Therefore, if our heuristic fails for the initial constraint, we can retry with a slightly tighter one and may find a valid schedule.

5. Other Approaches

We compare our results with a greedy heuristic and a heuristic based on the linear relaxation of the integer linear programming

Algorithm 3 SpringS Algorithm. The operators argmin and argmax refer respectively to the index of the minimum and of the maximum in an ordered set.

```
1: Input: initial schedule of tasks in $\mathcal{T}$ onto the processing elements in $S_k$.
2: Output: optimized schedule of tasks in $\mathcal{T}$ onto the processing elements in $\mathcal{S}_k$.
3: $\forall T_i \in \mathcal{T}$ such that $x_{iz} = 1$: $x_{iz} = 0$ and $x_{im} = 1$, where $\{V_m = (P_{j_m}, f_m) | (P_{j_m} = P_{j_z})$ and $(f_m = min(F_k))\}$
4: while true do
5:   $j_0 = \text{argmax}(\{U_j | P_j \in \mathcal{S}_k\})$
6:   if $U_{j_0} > 1$ then
7:     Find the task $T_{ref}$ on $P_{j_0}$ with the minimum energy increase when increasing its assigned frequency to the next step $f_{ref+1}$ if its frequency is not already the highest.
8:     Let $\mathcal{R}$ be the set of tasks whose current execution time is smaller than the execution time of $T_{ref}$ running at $f_{ref+1}$.
9:     for each task $T_i$ in $\mathcal{R}$ in decreasing number of cycles order do
10:      $j_1 = \text{argmin}(\{U_j | P_j \in \mathcal{S}_k\})$
11:      Assign $T_i$ to $P_{j_1}$ without changing its frequency if all the tasks assigned to $P_{j_1}$ and not in $\mathcal{R}$ fit at maximum frequency. If not, return the schedule previously found.
12:      Remove $T_i$ from $\mathcal{R}$.
13:    end for
14:  else
15:    $j_1 = \text{argmin}(\{U_j | P_j \in \mathcal{S}_k\})$
16:    On $P_{j_1}$, find the task $T$ with the minimum execution time increase when decreasing its frequency to the lower step.
17:    Decrease $T$'s frequency if possible. If not, return the current schedule.
18:  end if
19: end while
```

problem (LR-heuristic) both presented in [22]. To be able to compare the algorithms in an absolute fashion, we also search for the optimal solution. We present these different approaches in the next subsections.

5.1 Solving Integer Linear Programming

The problem ILP0 is an NP-hard problem. For problems of small size, it can be solved by exhaustive search. We search for a solution of this previously defined integer linear programming problem by using *lp_solve*, an open-source linear solver [7]. By using a branch-and-bound approach, *lp_solve* gives us the optimal solution if it exists. For problems small enough, *lp_solve* will find the solution in a reasonable amount of time, which allows us to compare the optimality of the algorithms. Searching for the optimal solution to ILP0 may be slower than expected for small problem sizes depending on the characteristics of the input data.

5.2 The Greedy Heuristic

The Greedy heuristic will pick a task after another – order doesn't matter – and for each task consider all the possible processors and frequencies, and choose the task mapping that minimizes the energy consumption among all the possible combinations that are within the deadline. The Greedy heuristic is presented in Algorithm 4.

Clearly a problem with the Greedy algorithm is that it tends to allocate the first tasks considered at a low frequency and then the remaining tasks might not fit anymore. Therefore, the success rate of Greedy is expected to be low.

Algorithm 4 The Greedy heuristic

```
T = set of all tasks
while T is not empty do
  For each task $T_i$, find the pair $V_z$ such that $e_{iz}$ is minimum and that $U_{j_z} \leq 1$. Save the value as $(T_i, V_z, e_{iz})$.
  Among all the triplets, choose the one with the minimum energy $e_{iz}$ and map the corresponding $T_i$ it to the processor $P_z$ and frequency $f_z$: $x_{iz} = 1$.
  Remove this task $T_i$ from T.
end while
```

5.3 The LR-heuristic

The LR-heuristic [22] uses properties of the linear relaxation of the scheduling problem to iteratively map tasks to processors while reducing the size of the problem at each step by removing the tasks that are already mapped. The LR-heuristic considers the integer linear programming problem ILP0 defined previously. However, in that problem, the variables x_{iz} are constrained to be binary. The LR-heuristic solves the more general relaxed problem LP0 in which Equation 4 is changed into

$$x_{iz} \in [0, 1] \qquad 1 \leq i \leq n, 1 \leq z \leq v \tag{5}$$

The only difference with ILP0 is that the variables x_{iz} are allowed to take any value between 0 and 1. The LR-heuristic is then as described in Algorithm 5.

Algorithm 5 The LR-heuristic (LinRel)

```
repeat
  Remove all the useless $x_{iz}$ variables which set to 1 would make $U_{j_z} > 1$.
  Solve the linear relaxation problem LP0. As proved in [22], at least one variable $x_{iz}$ will be equal to 1, in spite of being able to take any value between 0 and 1.
  All the variables $x_{iz} = 1$ are fixed and removed from the problem.
until all tasks are mapped or no feasible schedule is found
```

6. Environmental Setup

In this Section we describe the environmental setup that we use to run our experiments.

6.1 Task Set Generation

To demonstrate the quality of our heuristic compared to the previously described heuristics, we ran a large number of experiments using synthetic task sets. In order to generate a synthetic task set, we define two parameters, the single-platform task uniformity τ [1] and the architecture heterogeneity η as presented in [4]. τ represents how different the cycles number of the tasks will be on the same platform. η allows to tweak how different this number will be between the various platforms. For a given task i, we draw τ_i from a uniform distribution $U(1, \tau)$ and for each processor type we draw $\eta_{i,k}$ from a uniform distribution $U(1, \eta)$ and we set C_{ik}^S to $\tau_i \eta_{i,k}$ cycles.

In order to have realistic parameters for these task sets, we use numbers from experiments on Intel(r) Atom(tm). We ran the five filters denoise, sharpen, color correction, increase frame rate and up-scale from MJPEGTools, and measured an average cycle

[1] Single-platform task uniformity is called task heterogeneity in [4]. We call it here single-platform task uniformity to avoid confusion with the cross-platform heterogeneity defined in Section 3.2

number of 10^{10}. Therefore, we chose $\tau = 10^5$ and $\eta = 10^5$ for our experiments to achieve an average cycle number of 10^{10}. In Section 7.3, we will consider different values of τ and η.

For the experiments we generate task sets of different sizes. The limit to the task set size is set by the execution time of the linear solver. We run experiments with up-to 40 tasks.

6.2 Time Constraint

Once we have a synthetic task set, we want to generate a reasonable deadline. The minimum execution time of a task i (t_i^{min}) is the execution time of this task on the processor best suited for this task at the maximum frequency available on this processor:

$$t_i^{min} = min(\{\frac{C_{ik}^S}{max(F_k)}|k \in [1..q]\})$$

We define the tight time constraint as the sum of the minimum execution time of the n tasks distributed among the m available processors:

$$d_{tight} = \frac{\sum_{i=1}^{n} t_i^{min}}{m}$$

Unless the tasks were perfectly balanced, the tight time constraint would be impossible to meet. This is why we define the relaxed time constraint:

$$d = \alpha \cdot d_{tight}$$

where α is an input parameter of the experiment. By varying α we obtain a constraint more or less tight. In our experiments with MJPEGTools, a deadline of 30 frames per second translated to a value $\alpha = 1.5$. In section 7, we present results for different values of α: 1.1, 1.5 and 2.0.

6.3 Hardware Configurations

We consider two different hardware configurations, configuration 1 and 2. Configuration 1 is composed of three processing units. Two of the processing units follow the power function and the available frequencies of an Atom CPU as described in Table 1a. The other processing unit has the power characteristics of a GPU with only one power state of 344 mW at 800 MHz as shown in Table 1b. Configuration 2 is another platform composed of four Atom-like and two GPU-like processing units with the power characteristics presented in Tables 1a and 1b, respectively.

6.4 Algorithms evaluated

We run experiments with five different algorithms. Greedy is the Greedy algorithm described in Section 5.2. LinRel is the LR heuristic described in Section 5.3. Heuristic is the algorithm proposed in the paper and described in Section 4.1. Heuristic with retry and Hybrid are the algorithms described in Section 4.2. The Heuristic with retry tightens the constraint βd up to 20% of the original constraint ($\beta = 0.80$) with a new try every 1%. This of course requires 20 runs of the heuristics and, consequently, the Heuristic with retry can run up to 20 times slower. We based the heuristic with retry on our heuristic. It would be possible to base it on the LR-heuristic which might improve it, or even on the hybrid heuristic which would of course lead to even better results.

7. Experiment Results

In this section, we present our experimental results. Section 7.1 evaluates the optimality of the different algorithms for different numbers of tasks and different constraints. Section 7.2 discusses different ways of sorting the tasks and evaluates the impact on the optimality of the schedule. Section 7.3 analyzes the sensitivity of the different heuristics to different values of the task uniformity and architecture heterogeneity τ and η. Section 7.4 compares the execution time of the different algorithms. Finally, Section 7.5 summarizes our results.

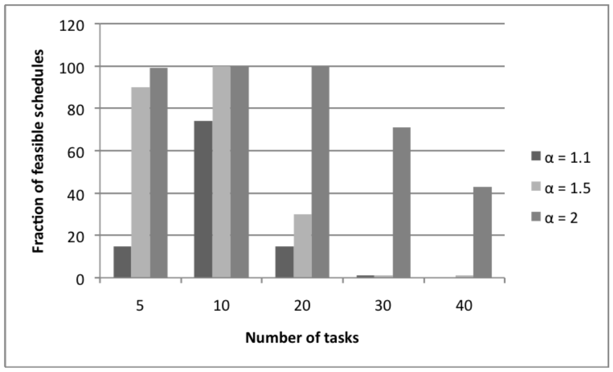


Figure 2: Fraction of feasible schedules for the linear solver with the hardware configuration 1. This shows what percentage of the generated task sets leads to feasible schedules. If the linear solver fails, it means that there is no feasible schedule for this task set.

7.1 Energy Savings and Success Rate

For each experiment, we generate one thousand synthetic task sets. Not every task set allows a feasible schedule but the linear solver (Section 5.1) will always find the optimal schedule if there is one. Running the linear solver first tells us if there is a feasible schedule and, if there is one, gives us the optimal energy. If there is no feasible schedule, we discard the task set. Figure 2 reports the percentage of feasible schedules over the one thousand task sets generated for different values of α and different number of tasks.

For each heuristic we measure its *success rate* as the ratio of number of schedules found to the number of feasible schedules. In addition, if the heuristic succeeds, it returns a schedule and its associated energy. We measure the optimality of the different heuristics as the ratio of the energy found to the optimal energy. We call it *error to optimal*.

Configuration 1: Two CPUs and one GPU The first set of experiments considers $\alpha = 1.1$ with configuration 1, two CPUs and one GPU. Such a value for α leads to a very tight constraint. The number of feasible schedules is very low and is close to 0% for more than 20 tasks, as seen on Figure 2. Therefore, we only present results up to 20 tasks for these experiments. The success rate of all the heuristics is shown in Figure 4a. The ratio of extra energy required when the heuristic does not find the optimal schedule is shown as the error to optimal plot in Figure 3a. As Figure 4a shows Heuristic outperforms the Greedy and the LR-heuristic, especially for 10 and 20 tasks. The Greedy heuristic finds only a few schedules. Although the success rates are low, the heuristics perform well when they find a schedule, with an average error of less than 5%, as shown in Figure 3a.

For $\alpha = 1.5$, success rate and error to optimal are shown in Figures 4b and 3b, respectively. As it can be seen the success rate of the heuristics improves significantly. We only show results for 20 tasks because the number of feasible schedules is still close to 0% for more than 20 tasks (See Figure 2). The Greedy heuristic is still lagging behind with less than 30% success rate for 5 tasks and almost 0% for 10 or more tasks. The LR-heuristic has a decent success rate that is above 70%, but all our heuristics outperforms it with, with close to 100% success rate.

Figure 3c presents the results for $\alpha = 2$. We can see that the success rate of Heuristic is close to the optimal (in average less than 3%) and outperforms the LinRel and the Greedy heuristic. For

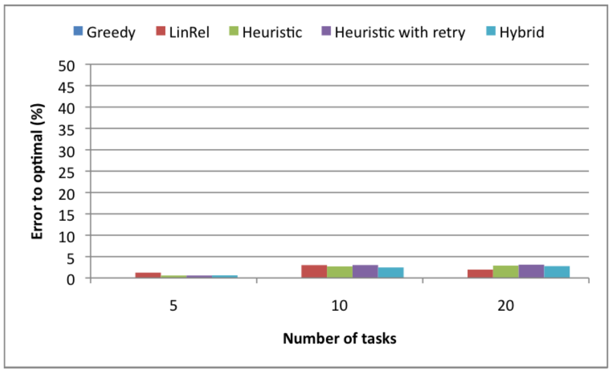
Frequencies (GHz)	0.8	1.0	1.2	1.4	1.6	1.8	2.0	2.4
Power (mW)	240	300	360	750	1100	1620	2160	3240

(a) ATOM CPU power function

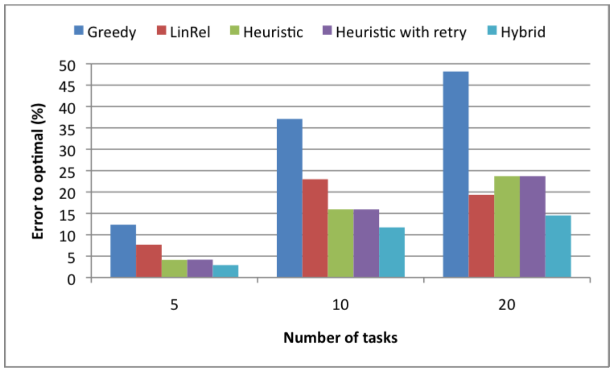
Frequencies (GHz)	0.8
Power (mW)	344

(b) GPU power function

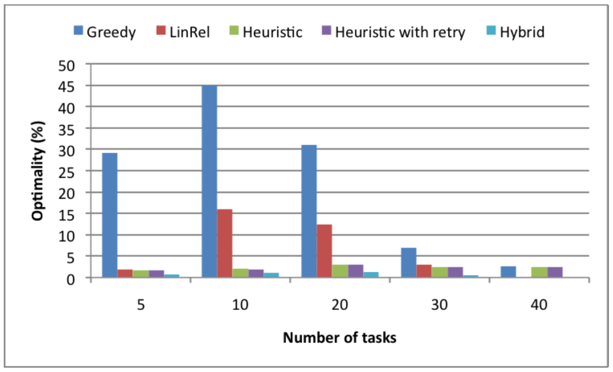
Table 1: Power functions of the processing units



(a) $\alpha = 1.1$

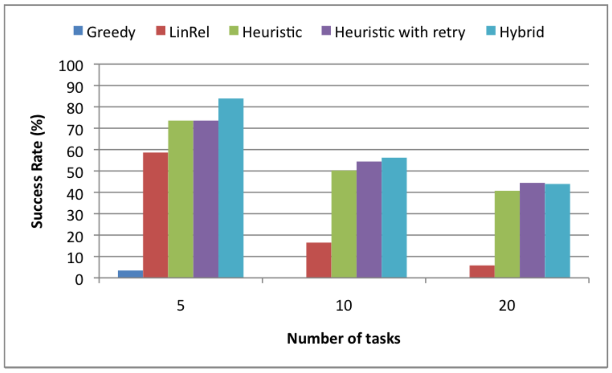


(b) $\alpha = 1.5$

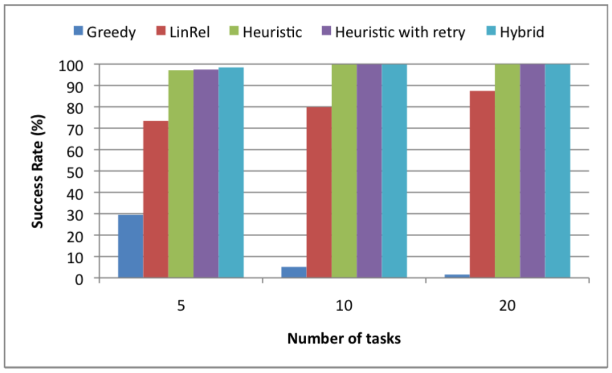


(c) $\alpha = 2$

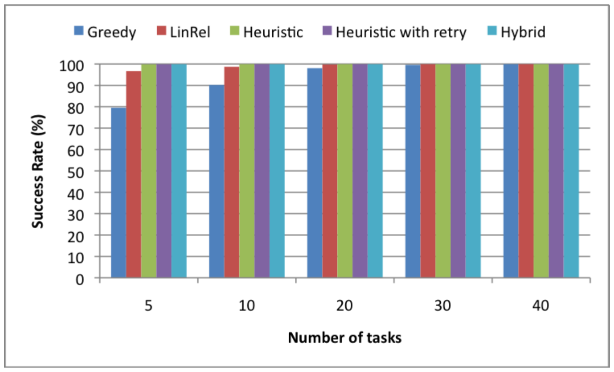
Figure 3: Error of the heuristics for different values of α. Two CPUs and one GPU. The horizontal axis is the number of tasks. The vertical axis represents how far from the optimal the heuristics are. The optimal is computed with the linear solver.



(a) $\alpha = 1.1$



(b) $\alpha = 1.5$



(c) $\alpha = 2$

Figure 4: Success rate of the heuristics. Two CPUs and one GPU. A success rate of 100% means that the heuristic found a schedule each time the linear solver found one.

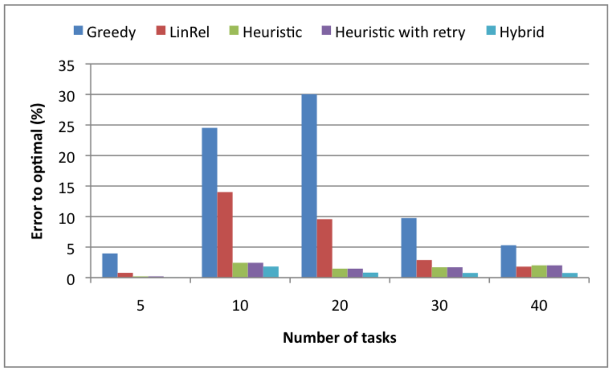


Figure 5: Error of the heuristics. Four CPUs and two GPUs with $\alpha = 2$. The horizontal axis is the number of tasks. The vertical axis represents how far from the optimal the heuristics are. The optimal is computed with the linear solver.

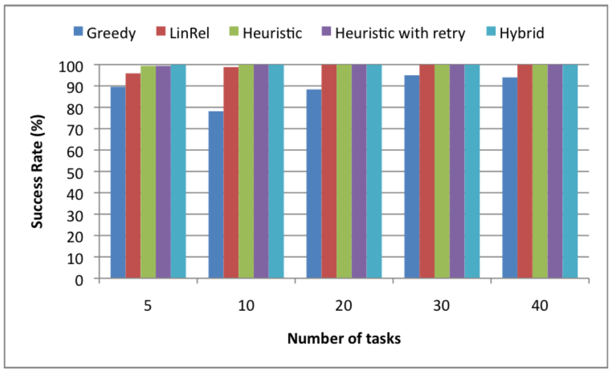


Figure 6: Success rate of the heuristics. Four CPUs and two GPUs with $\alpha = 2$. A success rate of 100% means that the heuristic found a schedule each time the linear solver found one.

a small number of tasks, Greedy and LinRel do not perform very well. As shown in Figure 4c, the success rate of our heuristic is 100%, whereas the Greedy heuristic does not succeed in finding a good schedule 20% of the time for 5 tasks, and 10% of the time for 10 tasks. Similarly, LR-heuristic also fails to find a feasible schedule for some small task sets.

Notice that the error is small (¡ 5%) for all our heuristics when α is 1.1 or 2.0. When α is 1.5, all the heuristics have a higher error. We believe the reason for this is that for values for α of 1.1 the constraint is very tight. So if the algorithm cannot find the correct schedule, it fails. Most likely there are very few feasible schedules. In that situation, we believe all the feasible schedules are likely to be similar. For values for α of 1.5 there are more feasible schedules, but the constraint is still tight and there is not much freedom for the algorithm to rearrange tasks or map a task to the best processor. In this situation, the heuristic can find a solution very different from the optimal, but still feasible. For values of α of 2.0, the constraint is looser, so it is much easier for all the algorithms to find a good schedule.

Finally, notice that the Hybrid heuristic is useful to reduce the error when Heuristic fails. Hybrid also helps improving the success rate. This is particularly useful for tight constraints as in Figure 4a when heuristics are more likely to fail. Finally, the heuristic with retry helps to slightly improve the success rate for the very tight schedules generated with $\alpha = 1.1$.

Configuration 2: Four CPUs and two GPUs Figure 5 and Figure 6 present the results for Configuration 2, that uses four CPUs and two GPUs, when $\alpha = 2$. Our heuristic stays under 2.5% of error for all the task counts considered. The Greedy and LR-heuristic improve a lot with an increasing number of tasks thanks to the greater freedom in scheduling allowed by more granularity. For $\alpha = 1.1$ or $\alpha = 1.5$ and for 1000 experiments run, only a small fraction gave a feasible schedule. Comparing the algorithms for such a small number of schedules would not be significant enough.

7.2 Sorting by Cross-Platform Task Heterogeneity vs. Sorting by Task Size

As discussed in Section 4.1, a key point in our heuristic is sorting tasks by decreasing cross-platform heterogeneity at the beginning of the mapping phase. One alternative logic could consider sorting the tasks by decreasing size to first map the largest tasks which could be the most difficult to fit in the schedule; this logic is still better than picking tasks in a random order or sorting them

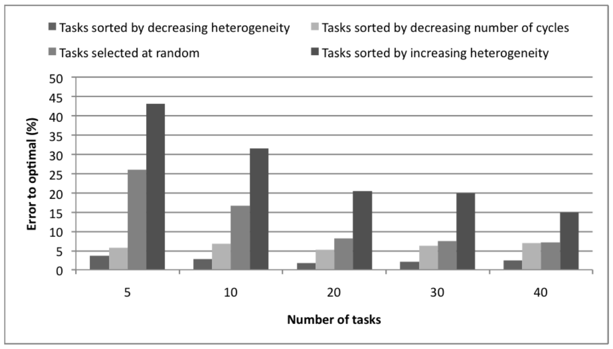


Figure 7: Error to optimal for the first hardware configuration when sorting tasks by decreasing cross-platform heterogeneity, by decreasing cycles, not sorting at all or sorting by increasing cross-platform heterogeneity, respectively. The y-axis shows the average error to the optimal for 1000 experiments with $\alpha = 2$

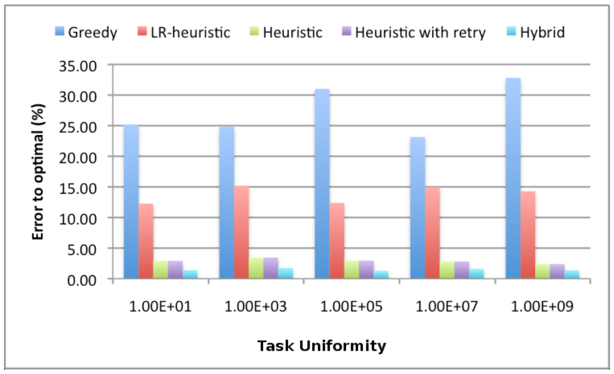
by increasing cross-platform heterogeneity, as shown in Figure 7. Notice our heuristic of sorting tasks by decreasing cross-platform task heterogeneity divides the error by more than 2 in most cases. Success rate for all experiments presented in this figure is greater than 99%.

7.3 Sensitivity Analysis

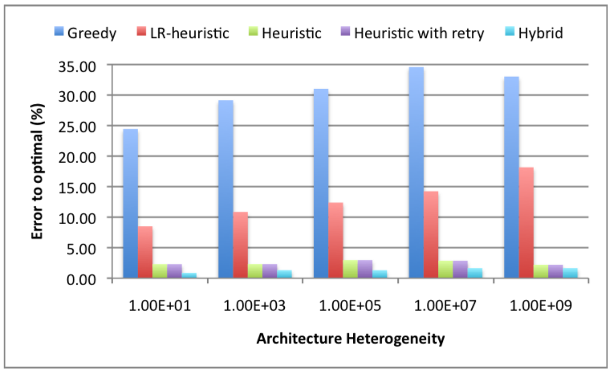
In this subsection, we analyze the sensitivity of the heuristics to the single-platform task uniformity and architecture heterogeneities. We chose the hardware configuration 1 and let the single-platform task uniformity τ and the architecture heterogeneity η vary respectively between 10 and 10^9. Figures 8a and 8b present the results. Our different heuristics are not sensitive to variations on the whole spectrum of heterogeneity considered. On the other hand, both the LinRel and Greedy are sensible to changes in η and τ; on Figure 8b, we see that Greedy and the LinRel are negatively impacted by an increase in architecture heterogeneity.

7.4 Execution Time of the Heuristic

A fast scheduler allows to test a lot of different configurations in a short time. It is also better for online scheduling, especially on real-time systems since the scheduling will consume time and power. In addition, if the scheduler is embedded in a compiler, it will allow shorter compilation times.



(a) Sensitivity to single-platform task uniformity τ: error to optimal as a function of τ



(b) Sensitivity to architecture heterogeneity η: error to optimal as a function of η

Figure 8: Error to optimal of the different heuristics as a function of the task uniformity and architecture heterogeneity.

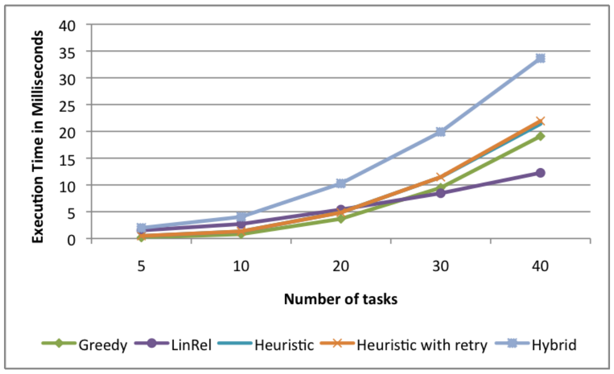


Figure 9: Comparison of the execution time in milliseconds as a function of the number of tasks for scheduling on the first hardware configuration with $\alpha = 2$. The heuristic and the heuristic with retry curve are almost the same because the retry only happens when no schedule is found.

#tasks	5	10	20	30	40
Exhaustive	8.05	88.27	13,000	172,470	420,090

Table 2: Execution time in milliseconds of the linear solver as a function of the number of tasks.

Our heuristic was implemented in C++ without specific performance optimizations. We also reimplemented in C++ the LR-heuristic and the Greedy heuristic presented in [22]. We used the *lp_solve* library [7] to solve the linear programming problems, both for the optimal case and for the LR-heuristic. For the optimal case, we simply make a call to the library, whereas for the LR-heuristic we wrap the call to *lp_solve* in some code controlling the different iterations of Algorithm 5. We ran 100 serial experiments on an Intel(r) Core(tm) i7 machine and timed each experiment with `gettimeofday()`. The results of these experiments are shown in Figure 9, where the average running time of each algorithm is plotted. We kept the exhaustive search numbers apart in Table 2 due to the huge difference in execution time.

Results on Figure 9 show that all the heuristics always perform very fast compared to the linear solver. The linear solver average execution time increases considerably as soon as the number of tasks increases. However, we noticed that in a large number of instances of the problem, even if the number of tasks is high, the execution time of the exhaustive search can be short depending on whether the branch-and-bound approach of the linear solver can eliminate more or less branches depending on the constraints. The average is very high because some instances of the problem take an extremely long time to explore. In the experiment presented in Figure 9, the heuristic with retry performs exactly the same as the heuristic because $\alpha = 2$ is a loose enough constraint that there is almost no need for retry. The hybrid heuristic execution time is higher and is exactly the sum of our heuristic and the LR-heuristic execution times.

7.5 Summary

Our experimental results in the previous Sections show that our Heuristic performs significantly better than Greedy and LR-Heuristic for those cases where there is little flexibility in the scheduling. In our experiments those cases appear when the value of α is low (α = 1.1) or when the number of tasks is small (20 or less). Our experiments on the Configuration 1 also show that when the number of tasks is small (20 or less) our Heuristic is faster than LR-Heuristic, and only slightly slower than Greedy, which performs very poorly. When the scheduling is easier due to extra freedom to map the tasks, all the algorithms obtain better results in terms of success rate and error rate. In particular our experiments for Configuration 1 show that for 40 tasks and α=2 LR-Heuristic is a good heuristic, as it runs faster than our Heuristic and have similar success and error rate. Notice that the are only two cases where the error rate of the LR-Heuristic is smaller than that of our Heuristic. Both cases occur in Configuration 1: the first one occurs with α = 1.5 and 20 tasks and the second one when α=2 and 40 tasks. In the first case our Heuristic has a higher success rate; in the second case (where all the heuristics have a 100% success rate), the error of our Heuristic is less than 3%. In addition, the Hybrid heuristic can take advantage of those cases where the LR-Heuristic performs better, although at the expense of some extra execution time.

Our results also show that the Greedy algorithm performs poorly; in fact, Greedy only performs well with large values of α (α=2) and large number of tasks.

Finally, our Heuristic seems very insensitive to architecture or task heterogeneity, because the heuristic that we use to do the mapping takes care of these variations.

8. Conclusions and future work

In this paper, we explored the scheduling of real-time tasks on a heterogeneous platform with energy minimization as a goal. Our heuristic offers a high success rate and significantly improves the state of the art heuristics, especially for small task sets. Our algorithm is stable in its results when exploring different task sets and platforms of different heterogeneities. Our experiments also evaluate the importance of the order in which the tasks are selected for scheduling. Furthermore, we have shown how to improve the results by combining two heuristics and how to improve the success rate by using the sensitivity of the scheduler to the tightness of the constraint.

Our scheduling strategy relies on a kernel composed of independent tasks and use software-pipelining to build this kernel of independent tasks. However, this can create a problem because it increases the pressure on cache and memory traffic, because now several items are in-flight at the same time. In the particular case of video postprocessing that we were studying, the number of frames in-flight will increase. Thus, we plan to study different approaches to refactor the task set for scheduling while improving the locality and reducing the data traffic.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Awards CCF 0702260, CNS 0509432, and by the Universal Parallel Computing Research Center at the University of Illinois at Urbana-Champaign, sponsored by Intel Corporation and Microsoft Corporation.

References

[1] Amd fusion.http://fusion.amd.com.

[2] http://www.imgtec.com/news/release/index.asp?newsid=557.

[3] Mjpeg tools. http://mjpeg.sourceforge.net/.

[4] Shoukat Ali, Howard Jay Siegel, Muthucumaru Maheswaran, Sahra Ali, and Debra Hensgen. Task execution time modeling for heterogeneous computing systems. In *HCW '00: Proceedings of the 9th Heterogeneous Computing Workshop*, page 185. IEEE Computer Society, 2000.

[5] James H. Anderson and Sanjoy K. Baruah. Energy-efficient synthesis of periodic task systems upon identical multiprocessor platforms. In *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 428–435, Washington, DC, USA, 2004. IEEE Computer Society.

[6] Hakan Aydi, Pedro Mejía-Alvarez, Daniel Mossé, and Rami Melhem. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *RTSS '01: Proceedings of the 22nd IEEE Real-Time Systems Symposium*, page 95. IEEE Computer Society, 2001.

[7] Michel Berkelaar, Kjell Eikland, and Peter Notebaert. Lpsolve. http://lpsolve.sourceforge.net/5.5/.

[8] Jian-Jia Chen, Chuan-Yue Yang, Tei-Wei Kuo, and Chi-Sheng Shih. Energy-efficient real-time task scheduling in multiprocessor dvs systems. *Asia and South Pacific Design Automation Conference*, 0:342–349, 2007.

[9] Edward T.-H. Chu, Tai-Yi Huang, and Yu-Che Tsai. An optimal solution for the heterogeneous multiprocessor single-level voltage-setup problem. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(11):1705–1718, 2009.

[10] Michael Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[11] Jörg Henkel and Yanbing Li. Energy-conscious hw/sw-partitioning of embedded systems: a case study on an mpeg-2 encoder. In *CODES/CASHE '98: Proceedings of the 6th international workshop on Hardware/software codesign*, pages 23–27, Washington, DC, USA, 1998. IEEE Computer Society.

[12] Chung-Hsing Hsu and Ulrich Kremer. The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 38–48. ACM, 2003.

[13] Tai-Yi Huang, Yu-Che Tsai, and Edward T.-H. Chu. A near-optimal solution for the heterogeneous multi-processor single-level voltage setup problem. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–10, 2007.

[14] Christopher J. Hughes, Jayanth Srinivasan, and Sarita V. Adve. Saving energy with architectural and frequency adaptations for multimedia applications. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 250–261. IEEE Computer Society, 2001.

[15] Tohru Ishihara and Hiroto Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *ISLPED '98: Proceedings of the 1998 international symposium on Low power electronics and design*, pages 197–202. ACM, 1998.

[16] Jian Li and José F. Martínez. Power-performance considerations of parallel computing on chip multiprocessors. *ACM Trans. Archit. Code Optim.*, 2(4):397–422, 2005.

[17] Hui Liu, Zili Shao, Meng Wang, and Ping Chen. Overhead-aware system-level joint energy and performance optimization for streaming applications on multiprocessor systems-on-chip. In *ECRTS '08: Proceedings of the 2008 Euromicro Conference on Real-Time Systems*, pages 92–101, Washington, DC, USA, 2008. IEEE Computer Society.

[18] Jiong Luo and Niraj K. Jha. Power-efficient scheduling for heterogeneous distributed real-time embedded systems. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 26(6):1161–1170, 2007.

[19] Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 89–102. ACM, 2001.

[20] William Thies and Saman Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Sep 2010.

[21] Chuan-Yue Yang, Jian-Jia Chen, Tei-Wei Kuo, and Lothar Thiele. An approximation scheme for energy-efficient scheduling of real-time tasks in heterogeneous multiprocessor systems. In *ACM/IEEE Conference of Design, Automation, and Test in Europe (DATE)*, pages 694–699, 2009.

[22] Yang Yu and Viktor K. Prasanna. Power-aware resource allocation for independent tasks in heterogeneous real-time systems. In *ICPADS '02: Proceedings of the 9th International Conference on Parallel and Distributed Systems*, page 341. IEEE Computer Society, 2002.

[23] Wanghong Yuan and Klara Nahrstedt. Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 149–163. ACM, 2003.

Improved Multithreaded Unit Testing

Vilas Jagannath, Milos Gligoric, Dongyun Jin,
Qingzhou Luo, Grigore Roşu, Darko Marinov
Department of Computer Science, University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{vbangal2, gliga, djin3, qluo2, grosu, marinov}@illinois.edu

ABSTRACT

Multithreaded code is notoriously hard to develop and test. A multithreaded test exercises the code under test with two or more threads. Each test execution follows some schedule/interleaving of the multiple threads, and different schedules can give different results. Developers often want to enforce a particular schedule for test execution, and to do so, they use time delays (`Thread.sleep` in Java). Unfortunately, this approach can produce false positives or negatives, and can result in unnecessarily long testing time.

This paper presents IMUnit, a novel approach to specifying and executing schedules for multithreaded tests. We introduce a new language that allows explicit specification of schedules as *orderings on events* encountered during test execution. We present a tool that automatically instruments the code to control test execution to follow the specified schedule, and a tool that helps developers migrate their legacy, sleep-based tests into event-based tests in IMUnit. The migration tool uses novel techniques for inferring events and schedules from the executions of sleep-based tests. We describe our experience in migrating over 200 tests. The inference techniques have high precision and recall of over 75%, and IMUnit reduces testing time compared to sleep-based tests on average 3.39x.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Verification, Reliability

Keywords

IMUnit, Unit Testing, Multithreaded Code

1. INTRODUCTION

Multicore processors are here to stay. To extract greater performance from multicore processors, developers need to write parallel code. The predominant paradigm for parallel code is that of shared memory where multiple threads of control communicate by reading and writing shared data objects. Shared-memory multithreaded code is often afflicted by concurrency bugs, which are hard to detect because multithreaded code can demonstrate different behavior based on the scheduling of threads, and the bugs may only be triggered by a small specific set of schedules.

To validate their multithreaded code, developers write multithreaded unit tests. A multithreaded test creates and executes two or more threads (and/or invokes code under test that itself creates and executes two or more threads). Each test execution follows some schedule/interleaving of the multiple threads, and different schedules can give different results. Developers often want to enforce a particular schedule for a test. For example, consider two threads, one executing a method m and the other executing a method m'. Developers may want to ensure in one test that m finishes before m' starts and in another test that m' finishes before m starts (and in more tests that m and m' interleave in certain ways). Without controlling the schedule, it is impossible to write precise assertions about the execution because the results can differ in the two scenarios, and it is impossible to guarantee which scenarios were covered during testing, even if multiple runs are performed.

To control the schedule of multithreaded tests, developers mostly use a combination of timed delays in the various test threads. In Java, the delay is performed with the `Thread.sleep` method, so we call this approach *sleep-based*. A sleep pauses a thread while other threads continue execution. Using a combination of sleeps, developers attempt to enforce the desired schedule during the execution of a multithreaded test, and then assert the intended result for the desired schedule. A sleep-based test can fail when an undesired schedule gets executed even if the code under test has no bug (false positive). Dually, a sleep-based test can pass when an unintended schedule gets executed even if the code under test has a bug (false negative). Indeed, sleeps are an unreliable and inefficient mechanism for enforcing schedules. To use sleeps, one has to estimate the real-time duration for which to delay a thread while the other threads perform their work. This is usually estimated by trial and error, starting from a small duration and increasing it until the test passes consistently on the developer's machine. The estimated duration depends on the execution environment (hardware/software configuration and the load on the machine). Therefore, when the same test is executed in a different environment, the intended schedule may not be enforced, leading to false positives/negatives. Moreover, sleep can be very inaccurate even on a single machine [20]. In an attempt to mitigate the unreliability of sleep, developers often end up over-estimating the duration, which in turn leads to slow running multithreaded tests.

Researchers have previously noted the numerous problems with using sleeps to specify schedules in multithreaded tests and have developed frameworks such as ConAn [22, 23], ConcJUnit [27], MultithreadedTC [26], and ThreadControl [13] to tackle some problems in specifying and enforcing schedules in multithreaded unit tests. However, despite these frameworks, multithreaded unit testing still has many issues that could be categorized as follows:

Readability: Most current frameworks force developers to reason about the execution of threads relative to a global clock. This is unintuitive since developers usually reason about the execution of their multithreaded code in terms of event relationships (such as m finishing before m' starts). Some frameworks require users to write schedules in external scripts, which makes it even more difficult to reason about schedules. In other frameworks the schedule is implicit, as a part of the unit test code, and hence it is difficult to focus on the schedule and reason about it separately at a higher level.

Modularity: In some current frameworks, the intended schedule is intermixed with the test code and effectively hard-coded into a multithreaded unit test. This makes it difficult to specify multiple schedules for a particular unit test and/or to reuse test code among different tests.

Reliability: Some current frameworks, as well as the legacy sleep-based tests, rely on real time. As explained, this makes them very fragile, leading to false positives/negatives and/or slow testing time.

Migration Costs: Most current frameworks are very different from the traditional sleep-based tests. This makes it costly to migrate the existing sleep-based tests.

We present a new framework, called *IMUnit* (*pronounced "immunity"*), which aims to address these issues with multithreaded unit testing. Specifically, we make the following contributions:

Schedule Language: IMUnit introduces a novel language that enables natural and explicit specification of schedules for multithreaded unit tests. Semantically, the basic entity in an IMUnit schedule is an *event* that an execution can produce at various points (e.g., a thread starting/finishing the execution of a method, or a thread getting blocked). We call the IMUnit approach *event-based*. An IMUnit schedule itself is a (monitorable) property [10,24] on the sequence of events. More precisely, each schedule is expressed as a set of desirable *event orderings*, where each event ordering specifies the order between a pair of events (note that an IMUnit schedule need not specify a total order between all events but only the necessary partial order).

While the ideas of IMUnit can be embodied in any language, we have developed an implementation for Java. Syntactically, the IMUnit constructs are represented using Java annotations. A developer can use `@Event` and `@Schedule` annotations to describe the events and intended schedules, respectively, for a multithreaded unit test.

Automated Migration: We have developed two inference techniques and a tool to ease migration of legacy, sleep-based tests to IMUnit, event-based tests. Our inference techniques can automatically infer likely relevant events and schedules from the execution traces of existing sleep-based tests. We implemented our migration tool as an Eclipse plugin which uses the results of inference to automatically refactor a given multithreaded test into an IMUnit test.

Execution and Checking: We have implemented a tool for execution of IMUnit multithreaded unit tests. The tool can work in two modes. In the active mode, it controls the thread scheduler to enforce a given IMUnit schedule during test execution. In the passive mode, it checks whether an arbitrary test execution, controlled by the regular JVM thread scheduler, follows a given IMUnit schedule. To enforce/check the schedules, our tool uses the JavaMOP monitoring framework [10,24]. We also include a new runner for the standard JUnit testing framework to enable execution of IMUnit tests with our enforcement/checking tool.

Evaluation: To guide and refine our design of the IMUnit language, we have been inspecting over 200 sleep-based tests from several open-source projects. We manually translated 198 of those tests into IMUnit, adding events and schedules, and removing sleeps. As a result, the current version of IMUnit is highly expressive, and we were able to remove all sleeps from all but 4 tests.

We evaluated our inference techniques by automatically inferring events/schedules for the original tests that we manually translated (the subprojects on manual translation and automatic inference were performed by different authors to reduce the direct bias of manual translation on automatic inference). Computing the precision and recall of the automatically inferred events/schedules with respect to the manually translated events/schedules, we find our techniques to be highly effective, with over 75% precision and recall.

We also compared the execution time of the original tests and our translated tests. Because the main goal of IMUnit is to make tests more readable, modular, and reliable, we did not expect IMUnit to run faster. However, IMUnit did reduce the testing time, on average 3.39x, compared to the sleep-based tests, with the sleep duration that the original tests had in the code. As mentioned earlier, these duration values are often over-estimated, especially in older tests that were written for slower machines. In summary, IMUnit not only makes multithreaded unit tests more readable, modular, and reliable than the traditional sleep-based approach, but IMUnit can also make test execution faster.

This paper makes progress on our vision for improving multithreaded unit testing; our position paper [15] proposed the idea of event-based specification of schedules, but the IMUnit language and algorithms/tools for inference and execution are completely new.

2. EXAMPLE

We now illustrate IMUnit with the help of an example multithreaded unit test for the ArrayBlockingQueue class in `java.util.concurrent` (JSR-166) [17]. ArrayBlockingQueue is an array-backed implementation of a bounded blocking queue. One operation provided by ArrayBlockingQueue is `add`, which performs a non-blocking insertion of the given element at the tail of the queue. If `add` is performed on a full queue, it throws an exception. Another operation provided by ArrayBlockingQueue is `take`, which removes and returns the object at the head of the queue. If `take` is performed on an empty queue, it blocks until an element is inserted into the queue. These operations could have bugs that get triggered when the `add` and `take` operations execute on different threads. Consider testing some scenarios for these opera-

```
1  @Test
2  public void testTakeWithAdd() {
3    ArrayBlockingQueue<Integer> q;
4    q = new ArrayBlockingQueue<Integer>(1);
5    new Thread(
6      new CheckedRunnable() {
7        public void realRun() {
8          q.add(1);
9          Thread.sleep(100);
10         q.add(2);
11       }
12     }, "addThread").start();
13   Thread.sleep(50);
14   Integer taken = q.take();
15   assertTrue(taken == 1 && q.isEmpty());
16   taken = q.take();
17   assertTrue(taken == 2 && q.isEmpty());
18   addThread.join();
19 }
```

(a) JUnit

```
public class TestTakeWithAdd
    extends MultithreadedTest {
  ArrayBlockingQueue<Integer> q;
  @Override
  public void initialize() {
    q = new ArrayBlockingQueue<Integer>(1);
  }
  public void addThread() {
    q.add(1);
    waitForTick(2);
    q.add(2);
  }
  public void takeThread() {
    waitForTick(1);
    Integer taken = q.take();
    assertTrue(taken == 1 && q.isEmpty());
    taken = q.take();
    assertTick(2);
    assertTrue(taken == 2 && q.isEmpty());
  }
}
```

(b) MultithreadedTC

```
@Test
@Schedule("finishedAdd1->startingTake1,
          [startingTake2]->startingAdd2")
public void testTakeWithAdd() {
  ArrayBlockingQueue<Integer> q;
  q = new ArrayBlockingQueue<Integer>(1);
  new Thread(
    new CheckedRunnable() {
      public void realRun() {
        q.add(1);
        @Event("finishedAdd1")
        @Event("startingAdd2")
        q.add(2);
      }
    }, "addThread").start();
  @Event("startingTake1")
  Integer taken = q.take();
  assertTrue(taken == 1 && q.isEmpty());
  @Event("startingTake2")
  taken = q.take();
  assertTrue(taken == 2 && q.isEmpty());
  addThread.join();
}
```

(c) IMUnit

Figure 1: Example multithreaded unit test for ArrayBlockingQueue

tions (in fact, the JSR-166 TCK provides over 100 tests for various scenarios for similar classes).

Figure 1 shows a multithreaded unit test that ArrayBlockingQueue exercises `add` and `take` in two scenarios. In particular, Figure 1(a) shows the test written as a regular JUnit test method, with sleeps used to specify the required schedule. We invite the reader to consider what scenarios are specified with that test (without looking at the other figures). It is likely to be difficult to understand which schedule is being exercised by reading the code of this unit test. While the sleeps provide hints as to which thread is waiting for another thread to perform operations, it is unclear which operations are intended to be performed by the other thread before the sleep finishes.

The test actually checks that `take` performs correctly both with and without blocking, when used with `add` from another thread. To check both scenarios, the test exercises a schedule where the first `add` finishes before the first `take` starts, and the second `take` blocks before the second `add` starts. Line 13 shows the first sleep that is intended to pause the `main` thread[1] while the `addThread` finishes the first `add`. Line 9 shows the second sleep which is intended to pause the `addThread` while the `main` thread finishes the first `take` and then proceeds to block while performing the second `take`. If the specified schedule is not enforced during the execution, there may be a false positive/negative. For example, if both `add` operations execute before a `take` is performed, the test will throw an exception and fail even if the code has no bug, and if both `take` operations finish without blocking, the test will not fail, even if the blocking `take` code had a bug.

Figure 1(b) shows the same test written using MultithreadedTC [26]. Note that it departs greatly from traditional JUnit where each test is a method. In MultithreadedTC, each test has to be written as a class, and each method in the test class contains the code executed by a thread in the test. The intended schedule is specified with respect to a global, logical clock. Since this clock measures time in *ticks*, we call the approach tick-based. When a thread executes a `waitForTick`, it is blocked until the global clock reaches the required tick. The clock advances implicitly when all threads are blocked (and at least one thread is blocked in a `waitForTick`). While a MultithreadedTC test does not rely on real time, and is thus more reliable than a sleep-based test, the intended schedule is still not immediately clear upon reading the test code. It is especially not clear when `waitForTick` operations are blocked/unblocked, because ticks are advanced implicitly when all the threads are blocked.

Figure 1(c) shows the same test written using IMUnit. The interesting events encountered during test execution are marked with `@Event` annotations[2], and the intended schedule is specified with a `@Schedule` annotation that contains a comma-separated set of *orderings* among events. An ordering is specified using the operator `->`, where the left event is intended to execute before the right event. An event specified within square brackets denotes that the thread executing that event is intended to block after that event. It should be clear from reading the schedule that the `addThread` should finish the first `add` before the `main` thread starts the first `take`, and that the `main` thread should block while performing the second `take` before the `addThread` starts the second `add`.

We now revisit, in the context of this example, the issues with multithreaded tests listed in the introduction. In terms of *readability*, we believe that making the schedules explicit, as in IMUnit, allows easier understanding and maintenance of schedules and code for both testing and debugging. In terms of *modularity*, IMUnit allows extracting the `addThread` as a helper thread (with its events) that can be reused in

[1]JVM names the thread that starts the execution `main` by default, although the name can be changed later.

[2]Note that `@Event` annotations appear on statements. The current version of Java (ver. 6) does not support annotations on statements, but the upcoming version of Java (ver. 7) will add such support. For now, `@Event` annotations can be written as comments, e.g., `/* @Event("finishedAdd1") */`, which IMUnit translates into code for test execution.

```
<Schedule>      ::= { <Ordering> [","] } <Ordering>
<Ordering>      ::= <Condition> "->" <Basic Event>
<Condition>     ::= <Basic Event> | <Block Event>
                  | <Condition> "||" <Condition>
                  | <Condition> "&&" <Condition>
                  | "(" <Condition> ")"
<Basic Event> ::= <Event Name> ["@" <Thread Name>]
                  | "start" "@" <Thread Name>
                  | "end" "@" <Thread Name>
<Block Event> ::= "[" <Basic Event> "]"
<Event Name>  ::= { <Id> "." } <Id>
<Thread Name> ::= <Id>
```

Figure 2: Syntax of the IMUnit schedule language

other tests (in fact, many tests in the JSR-166 TCK [17] use such helper threads). In contrast, reusing thread methods from the MultithreadedTC test class is more involved, requiring subclassing, parametrizing tick values, and providing appropriate parameter values. Also, IMUnit allows specifying multiple schedules for the same test code (Section 4.3). In terms of *reliability*, IMUnit does not rely on real time and hence has no false positives/negatives due to unintended schedules. In terms of *migration costs*, IMUnit tests resemble legacy JUnit tests more than MultithreadedTC tests. This similarity eases the transition of legacy tests into IMUnit: in brief, add `@Event` annotations, add `@Schedule` annotation, and remove `sleep` calls. Section 4 presents our techniques and tool that automate this transition.

3. SCHEDULE LANGUAGE

We now describe the syntax and semantics of the language used in IMUnit's schedules.

3.1 Concrete Syntax

Figure 2 shows the concrete syntax of the implemented IMUnit schedule language. An IMUnit schedule is a comma-separated set of *orderings*. Each ordering defines a condition that must hold before a basic event can take place. A *basic event* is an event name possibly tagged with its issuing thread name when that is not understood from the context. An *event name* is any identifier, possibly prefixed with a qualified class name. There are two implicit event names for each thread, `start` and `end`, indicating when the thread starts and terminates. Any other event must be explicitly introduced by the user with the `@Event` annotation (see Figure 1(c)). A *condition* is a conjunctive/disjunctive combination of basic and block events, where block events are written as basic events in square brackets. A *block event* $[e']$ in the condition c of an ordering $c \to e$ states that e' must precede e and, additionally, the thread of e' is blocked when e takes place.

3.2 Schedule Logic

It is more convenient to define a richer logic than what is currently supported by our IMUnit implementation; the additional features are natural and thus may also be implemented in the future. The semantics of our logic is given in Section 3.3; here is its syntax:

$$\begin{array}{lcl}
a & ::= & \mathit{start} \mid \mathit{end} \mid \mathit{block} \mid \mathit{unblock} \mid \text{event names} \\
t & ::= & \text{thread names} \\
e & ::= & a@t \\
\varphi & ::= & [t] \quad \mid \quad \varphi \to \varphi \quad \mid \quad \text{usual propositional connectives}
\end{array}$$

The intuition for $[t]$ is "thread t is blocked" and for $\varphi \to \psi$ "if ψ held in the past, then φ must have held at some moment before ψ". We call these two temporal operators the *block* and the *ordering* operators, respectively. For uniformity, all events are tagged with their thread. There are four implicit events: *start*$@t$ and *end*$@t$ were discussed above, and *block*$@t$ and *unblock*$@t$ correspond to when t gets blocked and unblocked[3].

For example, the following formula in our logic

$$\begin{array}{ll}
 & (a_1@t_1 \wedge ([t_2] \vee (\neg(\mathit{start}(t_2) \to a_1@t_1)))) \to a_2@t_2 \\
\wedge & (a_2@t_2 \wedge ([t_1] \vee (\mathit{end}(t_1) \to a_2@t_2))) \to a_2@t_2
\end{array}$$

says that if event a_2 is generated by thread t_2 then: (1) event a_1 must have been generated before that and, when a_1 was generated, t_2 was either blocked or not started yet; and (2) when a_2 is generated by t_2, t_1 is either blocked or terminated. As explained shortly, every event except for *block* and *unblock* is restricted to appear at most once in any execution trace. Above we assumed that $a_1, a_2 \notin \{\mathit{block}, \mathit{unblock}\}$.

Before we present the precise semantics, we explain how our current IMUnit language shown in Figure 2 (whose design was driven exclusively by practical needs) is a smaller fragment of the richer logic. An IMUnit schedule is a conjunction (we use comma instead of $\wedge$) of orderings, and schedules cannot be nested. Since generating *block* and *unblock* events is expensive, IMUnit currently disallows their explicit use in schedules. Moreover, to reduce their implicit use to a fast check of whether a thread is blocked or not, IMUnit also disallows the explicit use of $[t]$ formulas. Instead, it allows *block events* of the form $[a@t]$ (note the square brackets) in conditions. Since negations are not allowed in IMUnit, and since we can show (after we discuss the semantics) that $(\varphi_1 \vee \varphi_2) \to \psi$ equals $(\varphi_1 \to \psi) \vee (\varphi_2 \to \psi)$, we can reduce any IMUnit schedule to a Boolean combination of orderings $\varphi \to e$, where φ is a conjunction of basic events or block events. All that is left to show is how block events are desugared. Consider an IMUnit schedule $(\varphi \wedge [a_1@t_1]) \to a_2@t_2$, saying that $a_1@t_1$ and φ must precede $a_2@t_2$ *and* t_1 is blocked when $a_2@t_2$ occurs. This can be expressed as $((\varphi \wedge a_1@t_1) \to a_2@t_2) \wedge ((a_2@t_2 \wedge [t_1]) \to a_2@t_2)$, relying on $a_2@t_2$ happening at most once.

3.3 Semantics

Our schedule logic is a carefully chosen fragment of *past-time linear temporal logic (PTLTL)* over special well-formed multithreaded system execution traces.

Program executions are abstracted as finite traces of events $\tau = e_1 e_2 \ldots e_n$. Unlike in conventional LTL, our traces are finite because unit tests always terminate. Traces must satisfy the obvious condition that events corresponding to thread t can only appear while the thread is alive, that is, between *start*$@t$ and *end*$@t$. Using PTLTL, this requirement states that for any trace τ and any event $a@t$ with $a \notin \{\mathit{start}, \mathit{end}\}$, the following holds:

$$\tau \models \neg \Diamond (a@t \wedge (\Diamond \mathit{end}@t \vee \neg \Diamond \mathit{start}@t))$$

where $\Diamond$ stands for "eventually in the past". Moreover, except for *block*$@t$ and *unblock*$@t$ events, we assume that each

[3]It is expensive to explicitly generate *block*/*unblock* events in Java precisely when they occur, because it requires polling the status of each thread; our currently implemented fragment only needs, through its restricted syntax, to check if a given thread is currently blocked or not, which is fast.

event appears at most once in a trace. With PTLTL, this says that the following must hold ($\odot$ is "previously"):

$$\tau \models \neg \diamondsuit (a@t \wedge \odot \diamondsuit a@t)$$

for any trace τ and any $a@t$ with $a \notin \{block, unblock\}$.

The semantics of our logic is defined as follows:

$$
\begin{array}{lll}
e_1e_2 \dots e_n \models e & \text{iff} & e = e_n \\
\tau \models \varphi \ \wedge/\vee \ \psi & \text{iff} & \tau \models \varphi \text{ and/or } \tau \models \psi \\
e_1e_2 \dots e_n \models [t] & \text{iff} & (\exists 1 \le i \le n)\ (e_i = block@t \text{ and} \\
 & & \quad (\forall i < j \le n)\ e_j \ne unblock@t) \\
e_1e_2 \dots e_n \models \varphi \to \psi & \text{iff} & (\forall 1 \le i \le n)\ e_1e_2 \dots e_i \not\models \psi \text{ or} \\
 & & (\exists 1 \le i \le n)\ (e_1e_2 \dots e_i \models \psi \text{ and} \\
 & & \quad (\exists 1 \le j \le i)\ e_1e_2 ... e_j \models \varphi)
\end{array}
$$

It is not hard to see that the two new operators $[t]$ and $\varphi \to \psi$ can be expressed in terms of PTLTL as

$$
\begin{array}{rcl}
[t] & \equiv & \neg unblock@t\ \mathcal{S}\ block@t \\
\varphi \to \psi & \equiv & \boxdot \neg\psi \ \vee \ \diamondsuit(\psi \wedge \diamondsuit\varphi)
\end{array}
$$

where $\mathcal{S}$ stands for "since" and $\boxdot$ for "always in the past".

4. MIGRATION

We now describe the process of migrating legacy, sleep-based tests to IMUnit, event-based tests. First we present the steps that are typically performed during manual migration and then we describe the automated support that we have developed for key steps of the migration.

4.1 Manual Migration

Based on our experience of manually migrating over 200 tests, the migration process typically follows these steps:

Step 1: Optionally add explicit names for threads in the test code (by using a thread constructor with a name or by adding a call to `setName`). This step is required if events are tagged with their thread name (e.g. `finishedAdd1@addThread`) in the schedule, because by default the JVM automatically assigns a name (e.g. `Thread-5`) for each thread created without an explicit name, and the automatic name may differ between JVMs or between different runs on the same JVM.

Step 2: Introduce `@Event` annotations for the events relevant for the intended schedule. Some of these annotations will be used for block events and some for basic events.

Step 3: Introduce a `@Schedule` annotation for the intended schedule. Steps 2 and 3 are the hardest to perform as they require understanding of the intended behavior of the sleep-based test. Note that a schedule with too few orderings can lead to failing tests that are false positives. On the other hand, a schedule with too many orderings may lead to false negatives whereby a bug is missed because the schedule is over-constraining the test execution.

Step 4: Check that the orderings in the introduced schedule are actually satisfied when running the test with sleeps (Section 5 describes the passive, checking mode).

Step 5: Remove sleeps.

Step 6: Optionally merge multiple tests with different schedules (but similar test code) into one test with multiple schedules, potentially adding schedule-specific code (Section 4.3).

4.2 Automated Migration

We have developed automated tool support to enable easier migration of sleep-based tests to IMUnit. In particular, we have developed inference techniques that can compute

```
enum EntryType { SLEEP_CALL, SLEEP_RETURN, BLOCK_CALL,
       BLOCK_RETURN, OTHER_CALL, OTHER_RETURN, TH_START,
       TH_END, EVENT }
class LogEntry { EntryType type; ThreadID tid; String info; StmtID sid; }
```

Figure 3: Log Entries

likely relevant events and schedules for sleep-based tests by inspecting the execution logs obtained from test runs. We next describe the common infrastructure for logging the test runs. We then present the techniques for inferring events and schedules.

4.2.1 Lightweight Logging

Our inference of events and schedules from sleep-based tests is dynamic: it first instruments the test code (using AspectJ [19]) to emit entries potentially relevant for inference, then runs the instrumented code (several times, as explained below) to collect logs of entries from the test executions, and finally analyzes the logs to perform the inference.

Figure 3 shows the generic representation for log entries, although event and schedule inference require slightly different representations. Each log entry has a type, name/ID of the thread that emits the entry, potential info/parameters for the entry, and the ID of the statement that creates the entry (which is used only for event inference). The types of log entries and their corresponding `info` are as follows:

`SLEEP_CALL`: Invocation of `Thread.sleep` method. (Only used for inferring events.)

`SLEEP_RETURN`: Return from `Thread.sleep` method.

`BLOCK_CALL`: Invocation of a thread blocking method (`LockSupport.park` or `Object.wait`).

`BLOCK_RETURN`: Return from a thread blocking method.

`OTHER_CALL`: Invocation of a method (other than those listed above) in the test class. The `info` is the method name. (Only used for inferring events .)

`OTHER_RETURN`: Return from a method executed from the test class.

`TH_START`: Invocation of `Thread.start`. The `info` is the ID of the started thread. (Only used for inferring schedules.)

`TH_END`: End of thread execution.

`EVENT`: Execution of an IMUnit event. The `info` is the name of the event. (Only available while inferring schedules.)

Note that any logging can affect timing of test execution. Because sleep-based tests are especially sensitive to timing, care must be taken to avoid false positives. We address this in three ways. First, our logging is lightweight. The instrumented code only collects log entries (and their parameters) relevant to the inference. For example, `OTHER_CALL` is not collected for schedule inference. Also, the entries are buffered in memory during test execution, and they are converted to strings and logged to file only at the end of test execution. While keeping entries in memory would not work well for very long logs, it works quite well for the relatively short logs produced by test executions. Second, our instrumentation automatically scales the duration of sleeps by a given constant N to compensate for the logging overhead. For example, for $N = 3$ it increases all sleep times 3x. Increasing all the durations almost never makes a passing test fail, but it does make the test run slower. Third, we perform multiple runs of each test and only collect logs for passing runs. This increases the confidence that the logs indeed correspond to the intended schedules specified with sleeps.

4.2.2 Inferring Events

Figure 4 presents the algorithm for inferring IMUnit events from a sleep-based test. The input to the algorithm consists of a set of logs (as described in Section 4.2.1) and a `confidenceThreshold`. The output is a set of inferred events. Each event includes the code location where `@Event` annotation should be added and the name of the event. The intuition behind the algorithm is that `SLEEP_CALL` log entries are indicative of code locations for events. More precisely, a thread t calls `sleep` to wait for one or more events to happen on other threads (those will be "finished" events) before an event happens on t (that will be a "starting" event). Recall our example from Section 2. When the `main` thread calls `sleep`, it waits for `add` to finish before `take` starts, and thus `finishedAdd1` executes before `startingTake1`.

For each log, the algorithm first computes a set of *regions*, each of which is a sequence of log entries between `SLEEP_CALL` and the matching `SLEEP_RETURN` executed by the same thread. The log entries executed by other threads within a region are potential points for the "finished" events. Regions from different threads can be partially or completely overlapping, but regions from the same thread are disjoint (i.e., each `SLEEP_CALL` is followed directly by `SLEEP_RETURN` before any other statement is executed by the thread). Figure 5 shows two regions for a simplified log produced by our running example. In pseudo-code, each region is represented as a pair of `int`s that point to the beginning and end of the region in the list of log entries. For each region, the algorithm first calls `addFinishedEvents` to potentially add some "finished" events for threads other than the region's thread. If an event is added, the algorithm calls `addStartingEvent` to add the matching "starting" event.

The procedure `addFinishedEvents` potentially adds an inferred event for each thread that executes at least one statement in the region. For each such thread, the procedure first discovers a *relevant* statement, which is one of `SLEEP_CALL`, `BLOCK_CALL`, and `TH_END`. Only threads that have exactly one relevant statement in the region are considered. The intuition is that sleeps usually wait for exactly one event in each other thread. If a thread executes none or multiple relevant statements, it is most likely independent of the thread that started the region and therefore can be ignored. Figure 5 shows the relevant statements for each region. The procedure then finds the `OTHER_RETURN` statement immediately before the relevant statement for each thread. This statement determines the name for the new "finished" `StaticEvent`, whereas the relevant statement determines the location. Note that logging only method calls would not be enough to properly determine the previous statement since the call can come from a helper method in the test class. For our example, these before log entries are `OTHER_RETURN(add), addThread, 326` and `OTHER_RETURN(take), main, 336` (Fig. 5).

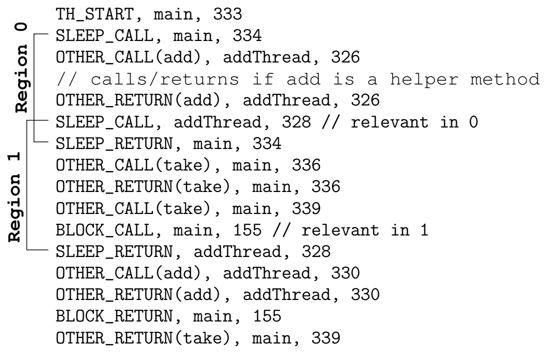
The procedure `addStartingEvent` adds an event for the thread that starts the region. The event is placed just before the first statement that follows the end of the region. The type of the statement can be any, including `OTHER_CALL`. The same statement is used for naming the event. In Figure 5, `OTHER_CALL(take), main, 336` and `OTHER_CALL(add), addThread, 330` are found following the algorithm.

```
// Input
Set⟨List⟨LogEntry⟩⟩ logs;
float confidenceThreshold;
// Output
class StaticEvent { StmtID sid; String name; }
Set⟨StaticEvent⟩ events;
// State
Bag⟨StaticEvent⟩ inferred := ∅;

class Region { int start; int end; }

void inferEvents() {
  foreach (List⟨LogEntry⟩ log in logs) {
    foreach (Region r in computeRegions(log)) {
      boolean addedFinished := addFinishedEvents(r, log);
      if (addedFinished) { addStartingEvent(r, log); }
    }
  }
  filterOutLowConfidence(confidenceThreshold);
  events := inferred.toSet();
}
Set⟨Region⟩ computeRegions(List⟨LogEntry⟩ log) {
  return { new Region(i, j) | log(i).type = SLEEP_CALL ∧
      j := min{ k | log(i).tid = log(k).tid ∧
      log(k).type = SLEEP_RETURN } }
}
boolean addFinishedEvents(Region r, List⟨LogEntry⟩ log) {
  boolean result := false;
  foreach (ThreadID t in { log(i).tid | i ∈ r } − { log(r.start).tid }) {
    Set⟨int⟩ relevant := { i ∈ r | log(i).tid = t ∧
        log(i).type ∈ { SLEEP_CALL, BLOCK_CALL, TH_END } ∧
         ¬(∃ j ∈ r | log(j).tid = t ∧
            log(j).type ∈ { SLEEP_RETURN, BLOCK_RETURN }) }
    if (relevant.size() ≠ 1) continue;
    int starting := max{ j < relevant | log(j).tid = t ∧
          log(j).type = OTHER_RETURN }
    addEvent(relevant, "finished", starting);
    result := true;
  }
  return result;
}
void addStartingEvent(Region r, List⟨LogEntry⟩ log) {
  int finished := min{ j > r.start | log(j).tid = log(r.start).tid ∧
        log(j).type ∈ { OTHER_CALL, TH_END } }
  addEvent(finished, "starting", finished);
}
void addEvent(int location, String namePrefix, int suffixIdx) {
    StmtID sid = log(location).sid;
    events ∪= new StaticEvent(sid, namePrefix +
        log(suffixIdx).info + sid);
}
```

Figure 4: Events-Inference Algorithm


Region 0
Region 1

```
TH_START, main, 333
SLEEP_CALL, main, 334
OTHER_CALL(add), addThread, 326
// calls/returns if add is a helper method
OTHER_RETURN(add), addThread, 326
SLEEP_CALL, addThread, 328 // relevant in 0
SLEEP_RETURN, main, 334
OTHER_CALL(take), main, 336
OTHER_RETURN(take), main, 336
OTHER_CALL(take), main, 339
BLOCK_CALL, main, 155 // relevant in 1
SLEEP_RETURN, addThread, 328
OTHER_CALL(add), addThread, 330
OTHER_RETURN(add), addThread, 330
BLOCK_RETURN, main, 155
OTHER_RETURN(take), main, 339
```

Figure 5: Snippet from a Log for Inferring Events

4.2.3 *Inferring Schedules*

Figure 6 presents the algorithm to infer an IMUnit schedule for a sleep-based multithreaded unit test that already contains IMUnit event annotations. These annotations can be automatically produced by our event inference or manually provided by the user. The input to the algorithm is a set of logs obtained from the passing executions of the sleep-based test. Figure 7 shows a snippet from one such log for our running example sleep-based test shown in Figure 1(a). The input also contains a `confidenceThreshold` which will be described later. The output is an inferred schedule, i.e., a set of orderings that encodes the intended schedule for the test. The main part of the algorithm is the `addSleepInducedOrderings` procedure. It captures the intuition that a thread normally executes a sleep to wait for the other active threads to perform events. Recall line 13 from our example in Figure 1(a) where the `main` thread sleeps to wait for the thread `addThread` to perform an `add` operation, and line 9 where `addThread` sleeps to wait for the `main` thread to first perform one `take` operation and then block while performing the second `take` operation.

For each log, the procedure scans for `SLEEP_RETURN` entries (line 31). As shown in Figure 7, the log for our example contains two `SLEEP_RETURN` entries, one each in the `main` thread and `addThread`. For each `SLEEP_RETURN` that is found, the procedure does the following:

1) Retrieves the next `EVENT` entry for the same thread (line 33). This event will be used as the `after` event in `Orderings` induced by the `SLEEP_RETURN`. In the example log, the two `after` events are `startingTake1` for the first `SLEEP_RETURN` and `startingAdd2` for the second `SLEEP_RETURN`.

2) Computes the other threads that were *active* between the `SLEEP_RETURN` and the `after` event (line 34). In the example, for the first `SLEEP_RETURN`, the only other active thread is `addThread` and for the second `SLEEP_RETURN`, the only other active thread is `main`.

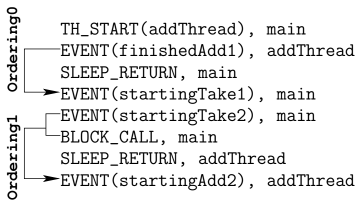
3) Finds for each active thread the last `EVENT` entry that is before the `after` event. This event will be the `before` event in the `Ordering` induced by the `SLEEP_RETURN` with the corresponding active thread (line 38). Note that this `before` event on another thread can be even *before* the `SLEEP_RETURN`. Effectively, this event is the *current* last entry and not the last entry at the time of the sleep. In the example, the two `before` events are `finishedAdd1` and `startingTake2` for the first and second `SLEEP_RETURN`s, respectively.

4) Creates an `Ordering` for each `before` and `after` event pair and inserts it into the `inferred` bag. If a `before` event is followed immediately by a `BLOCK_CALL` (within entries for the same thread), a `BlockingOrdering` is created; otherwise, a `NonBlockingOrdering` is created (line 41). In the example, since `startingTake2` is followed by a `BLOCK_CALL`, the ordering between `startingTake2` and `startingAdd2` will be a `BlockingOrdering`, while the other ordering between `finishedAdd1` and `startingTake1` will be a `NonBlockingOrdering`.

Before the `addSleepInducedOrderings` procedure is invoked, each `log` is modified by the `preprocessLogs` procedure. This procedure looks for `SLEEP_RETURN` entries followed immediately by `TH_START` entries for the same thread. For every such instance, it swaps the `SLEEP_RETURN` and `TH_START` entries and sets the `tid` of the `SLEEP_RETURN` entry to be the ID of the thread that is *started* by the `TH_START` event. The intuition is that a `SLEEP_RETURN` followed by a `TH_START` signifies that the *started* thread, rather than the starting thread perform-

```
 1 class Event { String eventName; ThreadID tid; }
 2 abstract class Ordering { Event before; Event after; }
 3 class NonBlockingOrdering extends Ordering {};
 4 class BlockingOrdering extends Ordering {};
 5 // Input
 6 Set⟨List⟨LogEntry⟩⟩ logs;
 7 float confidenceThreshold;
 8 // Output
 9 Set⟨Ordering⟩ orderings;
10 // State
11 Bag⟨Ordering⟩ inferred := ∅;
12
13 void inferSchedules() {
14   foreach (List⟨LogEntry⟩ log in logs) {
15     List⟨LogEntry⟩ preprocessed := preprocessLog(log);
16     addSleepInducedOrderings(preprocessed);
17   }
18   minimize();
19 }
20 List⟨LogEntry⟩ preprocessLog(List⟨LogEntry⟩ log) {
21   List⟨LogEntry⟩ result := log.clone();
22   foreach ({ i | log(i).type = SLEEP_RETURN }) {
23     int j := min{j > i | log(j).tid = log(i).tid };
24     if (log(j).type = TH_START) {
25       result(j) := new LogEntry(SLEEP_RETURN, _, log(j).info);
26       result(i) := log(j);
27     } }
28   return result;
29 }
30 void addSleepInducedOrderings(List⟨LogEntry⟩ log) {
31   foreach ({ i ∊ log.indexes() | log(i).type = SLEEP_RETURN }) {
32     ThreadID t := log(i).tid;
33     int j := min{ n > i | log(n).tid = t ∧ log(n).type = EVENT };
34     Set⟨ThreadID⟩ active := { t' | ( ∃ n < j |
35                      log(n).tid = t' ∧ log(n).type = EVENT ) ∧
36                      ( ∃ n > i | log(n).tid = t' ∧ log(n).type = TH_END
                               ) };
37     foreach (ThreadID t' in active − { t }) {
38       int j' := max{ n < j | log(n).tid = t' ∧ log(n).type = EVENT };
39       Event before := new Event(log(j').info, t');
40       Event after := new Event(log(j).info, t);
41       if (log(min{ n > j' | log(n).tid = t' }).type ≠ BLOCK_CALL) {
42         inferred ∪= new NonblockingOrdering(before, after);
43       } else { // before.type = BLOCK_CALL
44         inferred ∪= new BlockingOrdering(before, after);
45       } } } }
46 void minimize(List⟨LogEntry⟩ log) {
47   Set⟨Ordering⟩ graph := inferred.toSet() ∪ computeSeqOrderings(log);
48   removeCyclicOrderings(graph);
49   performTransitiveReduction(graph);
50   inferred.onlyRetainOrderingsIn(graph);
51   filterOutLowConfidence(confidenceThreshold);
52   orderings := inferred.toSet();
53 }
54 void Set⟨Ordering⟩ computeSeqOrderings(List⟨LogEntry⟩ log) {
55   return { new NonblockingOrdering(log(i), log(j)) |
56            i < j ∧ log(i).tid = log(j).tid ∧
57            log(i).type = log(j).type = EVENT ∧
58            ¬(∃ k | i < k < j ∧ log(j).tid = log(k).tid
59            ∧ log(k).type = EVENT) };
60 }
```

Figure 6: Schedule-Inference Algorithm



```
TH_START(addThread), main
EVENT(finishedAdd1), addThread
SLEEP_RETURN, main
EVENT(startingTake1), main
EVENT(startingTake2), main
BLOCK_CALL, main
SLEEP_RETURN, addThread
EVENT(startingAdd2), addThread
```

Figure 7: Snippet from a Log for Inferring Schedules

ing the TH_START, should wait for the other active threads to perform events. Many of the sleep-based tests that we migrated included instances of this pattern. Effectively, this swap makes it appear as if the sleep was at the beginning of the run method for the started thread, although the sleep was actually before the start method.

After each log is processed by the preprocessLogs and addSleepInducedOrderings procedures, the inferred bag is populated with all the inferred orderings. However, the inferred orderings may contain cycles (e.g., a->b and b->a) and transitively redundant orderings (e.g., a->b, b->c, and a->c, where the last ordering is redundant). The minimize procedure removes such orderings. It first creates an ordering graph by combining the edges from the inferred orderings with the edges implied by the sequential orderings of events within each thread (the latter edges being computed by the computeSeqOrderings procedure). It then removes all the edges of the graph that participate in cycles. It finally performs a transitive reduction on the acyclic graph and updates the inferred bag by removing all orderings not included in the reduced graph. We use an open-source implementation [12] of the transitive reduction algorithm introduced by Aho et al. [1]. Since the transitive reduction is performed on an acyclic graph, we can use a simpler case of the general algorithm.

The last step of the minimize procedure is to remove the orderings that were inferred with low confidence. Recall that the input to our inference is a set of logs from several (passing) runs of the test being migrated. The confidence of an inferred ordering is the ratio of the count of that ordering in the inferred bag and the number of logs/runs. For example, an ordering may be inferred in only 60% of runs, say 3 out of 5. The confidenceThreshold defines the lowest acceptable confidence. All inferred orderings with confidence lower than the specified threshold are discarded.

4.2.4 *Eclipse Plugin*

We have developed a refactoring plugin for Eclipse to enable automated migration of existing sleep-based unit tests into event-based IMUnit tests. The plugin is implemented using the generic refactoring API provided by Eclipse. The refactoring automates the most important steps required to migrate a sleep-based test into an IMUnit test: introduction of events and schedule (using inference techniques) and checking of the introduced schedule. The refactoring can also help the user name the threads in the test.

4.3 Multiple Schedules

As mentioned in Step 6 of Section 4.1, after converting sleep-based tests to event-based IMUnit tests, developers can merge several similar tests with different schedules into one test with multiple IMUnit schedules. Recall our example sleep-based test from Figure 1(a). Its intended schedule is an add followed by a non-blocking take and a blocking take followed by another add. Suppose that the same test class contained another sleep-based test whose indented schedule is an add followed by a non-blocking take and another add followed by another non-blocking take. Although these two sleep-based tests would be almost identical (with the sleep at line 9 moved to before line 16), they cannot share the common code without using additional conditional statements to enable the appropriate sleeps during execution. In contrast, after both tests are migrated to IMUnit tests, they can be easily replaced by just one new test. This new test would have the same code as in Figure 1(a), with two added annotations: (1) @Event("finishedAdd2") added after the add(2) call, and (2) @Schedule("finishedAdd1->startingTake1, finishedAdd2->startingTake2") added before the test method.

5. ENFORCING & CHECKING

We now describe the IMUnit Runner, our tool for enforcing/checking schedules for IMUnit tests. It is implemented as a custom test runner for the JUnit testing framework. It executes each test for each IMUnit schedule and has two operation modes. In the *active mode*, it controls the thread scheduler to enforce an execution of the test to satisfy the given schedule. Note that this mode avoids the main problem of sleep-based tests, that of false positives and negatives due to the execution of unintended schedules. In the *passive mode*, our tool observes and checks the execution provided by the JVM against the given schedule.

Our runner is implemented using JavaMOP [10, 24], a high-performance runtime monitoring framework for Java. JavaMOP is generic in the property specification formalism and provides several such formalisms as *logic plugins*, including past-time linear temporal logic (PTLTL). Although our schedule language is a semantic fragment of PTLTL (Section 3), enforcing PTLTL specifications in their full generality on multithreaded programs is rather expensive.

Instead, we have developed a custom JavaMOP logic plugin for our current IMUnit schedule language from Figure 2. Since JavaMOP takes care of all the low-level instrumentation and monitor integration details (after a straightforward mapping of IMUnit events into JavaMOP events), we here only briefly discuss our new JavaMOP logic plugin. It takes as input an IMUnit schedule and generates as output a monitor written in pseudo-code; a *Java shell* for this language then turns the monitor into AspectJ code [19], which is further woven into the test program. In the active mode, the resulting monitor enforces the schedule by blocking the violating thread until all the conditions from the schedule are satisfied. In the passive mode, it simply prints an error when its corresponding schedule is violated.

A generated monitor for an IMUnit schedule observes the defined events. When an event e occurs, the monitor checks all the conditions that the event should satisfy according to the schedule, i.e., a Boolean combination of basic and block events (Figure 2). The status of each basic event is maintained by a Boolean variable which is true iff the event occurred in the past. The status of a block event is checked as a conjunction of this variable and its thread's blocked state. In the active mode, the thread of e will be blocked until this Boolean expression becomes true. If the condition contains any block event, periodic polling is used for checking thread states. Thus, IMUnit pauses threads only if their events are getting out of order for the schedule. Note that the user may have specified an infeasible schedule, which can cause a deadlock where all threads are paused. Our runner includes a low-overhead runtime deadlock detection that detects and reports deadlocks.

As an example, Figure 8 shows the active-mode monitor generated for the schedule in Figure 1(c). When events finishedAdd1 and startingTake2 occur, the monitor just sets the corresponding Boolean variables, as there is no condition for those events. For event startingTake1, it checks if there was an event finishedAdd1 in the past by checking the vari-

```
switch (event) {
case finishedAdd1:
  occurred_finishedAdd1 = true; notifyAll();
case startingTake2:
  thread_startingTake2 = currentThread();
  occurred_startingTake2 = true; notifyAll();
case startingTake1:
  while (!occurred_finishedAdd1)
    wait();
  occurred_startingTake1 = true; notifyAll();
case startingAdd2:
  while (!(occurred_startingTake2 &&
           isBlocked(thread_startingTake2)))
    wait();
  occurred_startingAdd2 = true; notifyAll(); }
```

Figure 8: Monitor for the schedule in Figure 1(c)

able `occurred_finishedAdd1`; if not, the thread will be blocked until `finishedAdd1` occurs. For event `startingAdd2`, in addition to checking the Boolean variable for `startingTake2`, it also checks whether the thread of the event `startingTake2` is blocked; if not, the thread of the event `startingAdd2` will be blocked until both conditions are satisfied.

6. EVALUATION

To evaluate the IMUnit contributions—schedule language, automated migration, and schedule execution—we analyzed over 200 sleep-based tests from several open-source projects. Table 1 lists the projects and the number of sleep-based tests that we manually migrated to IMUnit. We first describe our experience with the IMUnit language. We then present results of our inference techniques for migration. We finally discuss the test running time.

6.1 Schedule Language

It is hard to quantitatively evaluate and compare languages, be it implementation or specification languages, including languages for specifying schedules. One metric we use is how *expressive* the language is, i.e., how many sleep-based tests can be expressed in IMUnit such that *sleeps can be removed altogether.* Note that IMUnit conceptually subsumes sleeps: sleeps and IMUnit events/schedules can co-exist in the same test, and developers just need to make sleeps long enough to account for the IMUnit schedule enforcement. While every sleep-based test is trivially an IMUnit test, we are interested only in those tests where IMUnit allows removing sleeps altogether.

We were able to remove sleeps from 198 tests, in fact all sleeps from all but 4 tests. While the current version of IMUnit is highly expressive, we have to point out that we refined the IMUnit language based on the experience with migrating the sleep-based tests. When we encountered a case that could not be expressed in IMUnit, we considered how frequent the case is, and how much IMUnit would need to change to support it. For example, blocking events are very frequent, and supporting them required a minimal syntactic extension (adding events with square brackets) to the initial version of our language. However, some cases would require bigger changes but are not frequent enough to justify them. The primary example is events in a loop. IMUnit currently does not support the occurrence of an event more than once in a trace. We did find 4 tests that would require multiple event occurrences, but changing the language to support them (e.g., adding event counters or loop indices to events) would add a layer of complexity that is not justified by the small number of cases. However, as we apply IMUnit to more projects, and gain more experience, we expect that the language could grow in the future.

Subject	Tests	Events	Orderings
Collections [4]	18	51	32
JBoss-Cache [18]	27	105	47
Lucene [6]	2	3	4
Mina [7]	1	2	1
Pool [5]	2	8	3
Sysunit [11]	9	33	34
JSR-166 TCK [17]	139	577	277
$\sum$	198	779	398

Table 1: Subject Programs Statistics

6.2 Inference of Events and Schedules

To measure the effectiveness of our migration tool in inferring events/schedules, we calculated precision and recall of automatically inferred events/schedules with respect to the manually written events/schedules (i.e., the manual translations from sleep-based schedules). Calculating precision and recall requires comparing the automatically inferred and manually written events/schedules. For event inference, the input is a sleep-based test, and the output is a set of events. Our current comparison uses only the *source-code location* (line number) of the static events and not their name. For schedule inference, the input is a sleep-based test *with manually written (not automatically inferred) events*, and the output is a schedule. Our comparison considers all orderings from the automatically inferred and manually written schedules; two orderings match only if they have exactly the same *both before and after events* (including their name and type that can be basic or block). We performed the comparisons for all but 14 (discussed below) of our 198 tests. Table 2 shows for each project precision and recall values, averaged over the tests from that project.

Columns two and three show the results for event inference. In most cases, precision and recall are fairly high. We inspected the cases with lower precision and identified two causes for it. The first cause is due to our evaluation setup and not the algorithm itself. Namely, our current comparison requires the exact match of source-code locations. If the locations differ, the inferred event counts as a false negative, even if it was only a few lines from the manually written event, and even if those locations are equivalent with respect to the code. In the future, we plan to improve the setup by analyzing the code around the automatically inferred and manually written events to determine if their locations are equivalent. The second reason is that some tests use sleeps that are not relevant for the thread schedule (e.g., JBoss-Cache has such sleeps in the helper threads shared among tests, and Lucene has similar sleeps while interacting with the I/O library). These extra sleeps mislead our inference, which assumes that every sleep is relevant for the schedule and infers events for every sleep.

Columns four and five show the results for schedule inference. The results are even more impressive than for event inference, with precision and recall of over 75% in all cases. We identified two causes for misses. The first cause is that some threads can be independent. The algorithm always forms edges from all threads to the thread that invokes sleep method, but this should not be done for independent threads. In the future, we plan to consider an abstraction similar to regions (Figure 4) as a mechanism to detect inde-

Subject	Inferring Events		Inferring Schedules	
	Precision	Recall	Precision	Recall
Collections	0.75	0.82	0.96	0.97
JBoss-Cache	0.83	0.86	0.87	0.96
Lucene	0.75	1.00	1.00	0.75
Mina	0.22	1.00	1.00	1.00
Pool	0.90	1.00	1.00	1.00
Sysunit	0.76	0.87	0.89	0.89
JSR-166 TCK	0.67	0.74	0.98	0.98
Overall	0.75	0.79	0.96	0.94

Table 2: Precision and Recall for Inference

Subject	Original	CR	TR	LC
Collections	33	0	0	0
JBoss-Cache	39	2	3	0
Lucene	5	0	1	1
Mina	1	0	0	0
Pool	3	0	0	0
Sysunit	39	0	5	0
JSR-166 TCK	306	0	30	1

Table 3: Numbers of Removed Orderings

pendent threads. The second cause is the same as for event inference, namely unnecessary sleeps.

A known issue in information retrieval is that some result sets may be empty, which corresponds to infinite precision and zero recall. For 14 of 198 tests, our inference techniques returned empty sets of events/schedules because these tests do not use sleeps to control schedules. Instead, these tests use `while (condition) { Thread.sleep/yield }` or `wait/notify` or `CountDownLatch` and other concurrent constructs to control schedules. We excluded these 14 tests from the evaluation of our inference techniques.

Our inference algorithms use `confidenceThreshold` to select some of the events/schedules, with the default value of 0.5 (for Table 2). We performed a set of experiments to evaluate how sensitive our inference is to the value of `confidenceThreshold`. We found that the results are quite stable. For example, for schedule inference, when changing the value from 0.5 to 0.1, only for Lucene the precision drops from 1 to 0.75. When changing the value from 0.5 to 0.9, only for JBoss-Cache the precision and recall drop from 0.87 and 0.96 to 0.86 and 0.93, respectively. For all other cases, everything else is inferred exactly the same for the values 0.1 and 0.9 as for the default value 0.5.

The other input to our inference algorithms is the set of logs obtained from passing runs of the legacy tests. By default, we collect 5 passing logs for each test (for Table 2). Different runs of the legacy test can produce different logs that can in turn result in different sets of events/schedules being inferred. Therefore, depending on the number of logs, inferred events/schedules could differ. So we evaluated how sensitive our inference is to the number of logs. We found that the logs are quite stable, and almost identical results were obtained for 1, 5, and 10 logs. For instance, going from 5 to 10 logs only the recall for JBoss-Cache drops from 0.96 to 0.94, and everything else remains the same.

Lastly, our schedule-inference algorithm runs a minimization phase after processing all the logs. Table 3 summarizes the results of this phase. It tabulates, for each project, the number of schedule orderings originally inferred before minimization (Original) and the numbers of orderings removed by cycles removal (CR), by transitive reduction (TR), and due to low confidence (LC). As it can be seen, the minimization phase does not remove many orderings. However, it is important to remove the orderings it does remove. For example, without removing the cycle for JBoss-Cache, not only would inference have a lower precision but it would also produce a schedule that is unrealizable.

Subject	Original	IMUnit [s]		Speedup	
	[s]	DDD	DDE	DDD	DDE
Collections	4.96	1.06	1.67	4.68	2.97
JBoss-Cache	65.58	31.25	31.76	2.10	2.06
Lucene	11.02	3.57	6.12	3.09	1.80
Mina	0.26	0.17	0.20	1.53	1.30
Pool	1.43	1.04	1.04	1.38	1.38
Sysunit	17.67	0.35	0.45	50.49	39.27
JSR-166 TCK	15.20	9.56	9.56	1.59	1.59
GeometricMean				3.39	2.76

Table 4: Test execution time. DDD - deadlock detection disabled; DDE - deadlock detection enabled

6.3 Performance

Table 4 shows the execution times of the 198 original, sleep-based tests and the corresponding IMUnit tests (for IMUnit, with deadlock detection both disabled and enabled). We ran the experiments on an Intel i7 2.67GHz laptop with 4GB memory, using Sun JVM 1.6.0_06. Our goal for IMUnit is to improve readability, modularity, and reliability of multithreaded unit tests, and we did not expect IMUnit execution to be faster than sleep-based execution. In fact, one could even expect IMUnit to be slower because of the additional code introduced by the instrumentation and the cost of controlling schedules. It came as a surprise that IMUnit is faster than sleep-based tests, on average 3.39x. Even with deadlock detection enabled, IMUnit was on average 2.76x faster. This result is with the sleep durations that the original tests had in the code.

We also compared the running time of IMUnit with MultithreadedTC on a common subset of JSR-166 TCK tests that the MultithreadedTC authors translated from sleep-based to tick-based [25]. For these 129 tests, MultithreadedTC was 1.36x faster than IMUnit. Although MultithreadedTC is somewhat faster, it has a much higher migration cost, and in our view, produces test code that is harder to understand and modify than the IMUnit test code. Moreover, we were surprised to notice that running MultithreadedTC on these tests, translated by the MultithreadedTC authors, can result in some failures (albeit with a low probability), which means that these MultithreadedTC tests can be unreliable and lead to false positives in test runs.

7. RELATED WORK

Three areas of work are related to IMUnit: (1) unit testing of multithreaded code, (2) enforcement of schedules, and (3) automated inference of specifications. We briefly discuss each of them. (1) ConAn [22, 23] and MultithreadedTC [26] introduce unit testing frameworks that allow developers to specify schedules to be used during the execution of multithreaded unit tests. However, the schedules in both frameworks are specified relative to a global clock (real time for ConAn and logic time for MultithreadedTC), which makes it difficult to reason about the schedules. Also, neither framework supports automated migration of sleep-based tests. ConcJUnit [27] extends JUnit to propagate exceptions raised by child threads up to the main thread and also checks whether all child threads have finished at the end of a test method. ThreadControl [13] proposes a tool to ensure that assertions are performed without interference from other threads. (2) There has been some previous work on using formally specified sequencing constraints to verify multithreaded programs [28]. The specifications are over sync events with LTL-like constraints, and the verification ensures that the implementation is faithful to the specification. In contrast, IMUnit schedule specifications are used to enforce ordering between user-specified events while the system is tested. Carver and Tai [9] use deterministic replay for concurrent programs. LEAP [14] is a more recent system using a similar record-and-replay approach to reproduce bugs. In comparison, our enforcement and checking mechanism targets ensuring the user-specified schedule rather than replaying a previously observed execution. (3) Work on automated mining of specifications for programs [2, 3, 8, 21] is related to our automated inference of events and schedules. However, most existing work focuses on mining API usage patterns/rules in a single threaded scenario, while our techniques mine the intention of sleep-based tests i.e. interesting events and event orderings across multiple threads.

8. CONCLUSIONS

Current approaches for unit testing of multithreaded code have issues with readability, modularity, reliability, and/or migration cost. We presented IMUnit, a novel approach that addresses these issues. IMUnit includes a new language that makes tests more readable and modular as it allows explicitly specifying schedules on the events during test execution. We described inference techniques and a tool that can help in migrating sleep-based tests to IMUnit. We also described a tool that can reliably execute the specified schedule to avoid false positives/negatives. The promising results with IMUnit encourage us to further explore this approach, e.g., for automatic generation of multithreaded tests (both test code and schedules) only from the code under test, or for regression testing of code with IMUnit schedules [16].

Acknowledgements

We would like to thank Feng Chen, Steven Lauterburg and Traian Şerbănuţă for initial discussion on this work. Also, we would like to thank the participants of the IWMSE 2010 workshop for useful feedback. This work is partially supported by the National Science Foundation under Grant Nos. CCF-1012759, CNS-0958199, CCF-0916893, CCF-0746856, and CNS-0720512, by Intel and Microsoft via the Universal Parallel Computing Research Center (UPCRC), by NASA contract NNL08AA23C, by an NSA grant, by a UIUC Campus Research Board Award, and by a Samsung SAIT grant.

9. REFERENCES

[1] A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1972.

[2] R. Alur, P. Cerný, M. Parthasarathy, and W. Nam. Synthesis of interface specifications for Java classes. In *POPL*, 2005.

[3] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL*, 2002.

[4] Apache Software Foundation. Apache Commons Collections. `http://commons.apache.org/collections/`.

[5] Apache Software Foundation. Apache Commons Pool. `http://commons.apache.org/pool/`.

[6] Apache Software Foundation. Apache Lucene. `http://lucene.apache.org/`.

[7] Apache Software Foundation. Apache MINA. `http://mina.apache.org/`.

[8] J. Burnim and K. Sen. DETERMIN: Inferring likely deterministic specifications of multithreaded programs. In *ICSE*, 2010.

[9] R. H. Carver and K. Tai. Replay and testing for concurrent programs. *IEEE Software*, 1991.

[10] F. Chen and G. Roşu. Mop: An efficient and generic runtime verification framework. In *OOPSLA*, 2007.

[11] Codehaus. Sysunit. `http://docs.codehaus.org/display/SYSUNIT/Home`.

[12] S. Cotton. graphlib. `http://www-verimag.imag.fr/~cotton/`.

[13] A. Dantas, F. V. Brasileiro, and W. Cirne. Improving automated testing of multi-threaded software. In *ICST*, 2008.

[14] J. Huang, P. Liu, and C. Zhang. LEAP: Lightweight deterministic multi-processor replay of concurrent Java programs. In *FSE*, 2010.

[15] V. Jagannath, M. Gligoric, D. Jin, G. Rosu, and D. Marinov. IMUnit: Improved multithreaded unit testing (position statement). In *IWMSE*, 2010.

[16] V. Jagannath, Q. Luo, and D. Marinov. Change-aware preemption prioritization. In *ISSTA*, 2011.

[17] Java Community Process. JSR 166: Concurrency utilities. `http://g.oswego.edu/dl/concurrency-interest/`.

[18] JBoss Community. JBoss Cache. `http://www.jboss.org/jbosscache`.

[19] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP*, 2001.

[20] Lassi Project. Sleep testcase. `http://tinyurl.com/4hk9zdr`.

[21] C. Lee, F. Chen, and G. Roşu. Mining parametric specifications. In *ICSE*, 2011.

[22] B. Long, D. Hoffman, and P. A. Strooper. A concurrency test tool for Java monitors. In *ASE*, 2001.

[23] B. Long, D. Hoffman, and P. A. Strooper. Tool support for testing concurrent Java components. *IEEE TSE*, 2003.

[24] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the MOP runtime verification framework. *Springer STTT*, 2011.

[25] W. Pugh and N. Ayewah. MultithreadedTC - A framework for testing concurrent Java applications. `http://code.google.com/p/multithreadedtc/`.

[26] W. Pugh and N. Ayewah. Unit testing concurrent software. In *ASE*, 2007.

[27] M. Ricken and R. Cartwright. ConcJUnit: Unit testing for concurrent programs. In *PPPJ*, 2009.

[28] K. Tai and R. H. Carver. Use of sequencing constraints for specifying, testing, and debugging concurrent programs. In *ICPADS*, 1994.

Predicting Null-Pointer Dereferences in Concurrent Programs *

Azadeh Farzan † P. Madhusudan ‡ Niloofar Razavi † Francesco Sorrentino ‡

Abstract

We propose null-pointer dereferences as a target for finding bugs in concurrent programs using testing. A null-pointer dereference prediction engine observes an execution of a concurrent program under test and predicts alternate interleavings that are likely to cause null-pointer dereferences. Though accurate scalable prediction is intractable, we provide a carefully chosen novel set of techniques to achieve reasonably accurate and scalable prediction. We use an abstraction to the shared-communication level, take advantage of a static lock-set based pruning, and finally, employ precise and relaxed constraint solving techniques that use an SMT solver to predict schedules. We realize our techniques in a tool, ExceptioNULL, and evaluate it over 13 benchmark programs and find scores of null-pointer dereferences by using only a single test run as the prediction seed for each benchmark.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging

Keywords Testing, Concurrency, SMT, Null-pointers, Data-Races

1. Introduction

Errors in concurrent programs often occur under subtle interleaving patterns that the programmer had not foreseen. There are too many interleavings to explore, even on a single test input for a concurrent program, making concurrency testing a hard problem. With the rise of multicore hardware platforms, finding solutions to this problem is very important as testing is still the most effective way of finding bugs today. Current testing technologies such as stress testing have proved largely inadequate in exposing such subtle interleavings.

Prediction-based testing has emerged as a promising approach to testing concurrent programs. It involves taking one arbitrary concurrent execution of the program under test, and from that predict alternate interleavings that are more likely to contain bugs (interleavings that lead to data-races, interleavings that violate atomicity, etc.). Prediction replaces systematic search with a search for interleavings that are *close* to observed executions only, and hence is more tractable, and at the same time explores interesting interleavings that are likely to lead to errors [9, 10, 26, 27, 31].

In this paper, we explore a new target for predictive testing of concurrent programs that is fundamentally very different from data-races or atomicity errors: we propose to target executions that lead to *null-pointer dereferences*. Given an arbitrary execution of a concurrent program under test, we investigate fundamental techniques to accurately and scalably predict executions that are likely to lead to null-pointer dereferences.

Null-pointer dereferences can occur in a thread when dereferencing local variables. Consequently, an *accurate* prediction of null-pointer dereferences requires, by definition, handling local variables and the computation of threads. This is in sharp contrast to errors like data-races and atomicity violations, which depend only on *accesses to shared variables*.

The prediction algorithm aims to find some interleaving of the events in the observed run that will result in a null-pointer dereference. The naive approach to this problem is to reduce it to a constraint satisfaction problem; the set of constraints capture the semantics of local computations as well as the interaction of threads using reads and writes, concurrency control like locks, etc. A constraint solver could then solve these constraints and hence synthesize an interleaving that causes a null-pointer dereference to occur (this is similar to logic-based bounded model-checking for concurrent programs [24, 25]). However, this simply does not scale in realistic dynamic testing setting where the number of events that model reads/writes and computation can span millions of events. Consequently, accurate null-pointer dereference prediction seems intractable.

The main goal of this paper is to achieve useful and scalable prediction for finding runs that cause null-pointer dereferences. We propose a combination of four carefully chosen techniques to achieve this:

1. **Approximation:** An approximation of the prediction problem that ignores local computation entirely, and recasts the problem involving only the events that observe shared reads, writes, and concurrency control events (which we call the shared communication level) to achieve scalability. The prediction at the shared communication level can be more efficiently solved, using a combination of a lock-set based static analysis that identifies null read-write pairs and a constraint satisfaction problem to predict executions that force these null reads. Predicted runs in this model will be feasible but may not actually cause a null-pointer dereference, though they are likely to do so.
2. **Static Pruning:** An aggressive pruning of executions using static analysis based on vector-clocks that identifies a small segment of the observed run on which the the prediction effort can be focused. This greatly improves the scalability of using sophisticated logic solvers. Pruning of executions does not affect feasibility of the runs, but may reduce the number of runs predicted. However, we show that in practice no additional errors were found without pruning.
3. **Relaxed prediction:** A formulation of the prediction at the shared communication level that allows some leeway so that the prediction algorithm can predict runs with mild *deviations* from the observed run which could be interesting runs; this makes the class of predicted runs larger at the expense of possibly making them *infeasible*, though in practice we found the majority of the predicted runs to be feasible.

* This work was funded partly by the NSERC Discovery Grant at University of Toronto and by the Illinois-Intel Parallelism Center at the University of Illinois at Urbana-Champaign and by NSF Career Award #0747041.

† University of Toronto.

‡ University of Illinois.

SIGSOFT'12/FSE-20, November 11–16, 2012, Cary, North Carolina, USA.
Copyright © 2012 ACM 978-1-4503-1614-9/12/11...$10.00

4. Re-execution: The runs predicted using the techniques may be infeasible, or may be feasible and yet not cause any null-pointer dereference. We mitigate this by a re-execution engine that executes predicted schedules accurately to check if a null-pointer dereference actually occurs. Errors reported hence are always real (i.e., they cause an uncaught exception or result in failing the test harness), and hence we incur no false positives.

We explain these techniques below, and motivate their choice and their impact.

Approximation to the shared communication level: Though null-pointer dereferences could occur on local variables and shared variables (unlike data-races and atomicity, which are defined only at the level of shared variables), a prediction that takes into account local variables and local computation simply does not scale (we have tried many experiments on such a model that validate this claim). We propose an approximation to null-pointer dereference prediction that works at the shared communication level. Consider a thread T that in some interleaving reads a shared variable x and subsequently does some computation locally using its value, and consider the task of predicting whether this could result in a null-pointer dereference. Our approximation of the prediction problem at the shared communication level asks for a run that forces the thread T to read a value of *null*. Note that this approximation is neither sound nor complete— thread T may read *null* for x but may not dereference the pointer (e.g., it could check if x is null), and there may be runs where the value read is not *null* and yet the local computation causes a null pointer dereference. However, such an approximation is absolutely necessary to scale to large runs, as it is imperative that local computation is not modeled. As our experiments demonstrate, this approximation does not inhibit us from finding interesting executions with null-pointer dereferences.

The above approximation poses the problem as a prediction problem at the shared communication level, which is a well-studied problem. In particular, there is a known *maximal causal model* that allows prediction at this level [23] and is the most precise prediction one can achieve at the shared communication level. Moreover, this prediction can be achieved using automatic *constraint solvers* that solve a constraint that demands a sequentially-consistent interleaving that respects the semantics of read/write of shared variables and the concurrency control mechanisms (locks, barriers, threads-creation, etc.). Maximal causality prediction using constraint solvers has been done in the past for data-races, atomicity, etc. (as they are properties already defined at the shared communication level) [22] and we utilize a similar technique with additional optimizations to approximately predict null-pointer dereferences.

Static pruning of executions: Prediction at the shared communication level, though more scalable than when modeling local computation, still has scalability issues in practice, as there can be in the order of hundreds of thousands of events involving shared variables. We propose pruning the execution using a simple scalable lock-set and vector-clock based analysis that determines a large prefix of the observed run that can be cut out *soundly*. The prediction algorithm hence is only applied to the remnant segment, which is smaller by an order of magnitude. If the prediction of the smaller segment succeeds, we are guaranteed that there is a way to stitch back the removed prefix to predict a full execution. Furthermore, the relaxed prediction technique (discussed below) being only applied to the smaller segment, introduces inaccuracies solely within that segment.

Relaxed prediction: The above two techniques of approximate prediction at the shared communication level and static pruning address scalability issues, and ensure that predicted runs are always feasible. However, they do not always work well in practice because sometimes *no solution* to the constraints exist. Demanding that the predicted run be absolutely guaranteed to be feasible (using the maximal causal model) seems too stringent a requirement, as it rules out runs that even mildly deviate from the requirements. For instance, if there is a read of y with value 23 in the original run, the predicted run is required to make the same read-event read the value 23, while in reality reading a different value, say 27, may not cause the program to completely deviate from the current path.

We propose a novel relaxed prediction algorithm that models the constraints to allow *bounded wiggle-room*. We propose to explore, for an increasing threshold k, whether there is a predictable run that violates at most k constraints that specify the values of shared reads. According to our experiments, even setting k to a small number (e.g., 5) is often sufficient to predict runs causing null-pointer dereferences. These predicted runs are not guaranteed to be feasible, but we show empirically that most of them actually are.

This technique, as well as the approximation to the shared communication level, causes unsoundness (predicted runs may be feasible or not cause a null-pointer dereference), which is handled by using an accurate re-execution tool that checks whether the predicted runs are indeed feasible and cause a null-pointer dereference.

The main technical contributions of this paper is the mechanism of targeting null-pointer dereference prediction using the approximation to the shared-communication level, the static pruning for scalability, and the prediction by relaxing constraints so as to make the prediction useful.

Evaluation: The final contribution of this paper is a full-fledged implementation realizing the new prediction-based testing tool that targets null-pointer dereferences, called EXCEPTIONULL . EXCEPTIONULL monitors and reschedules interleavings for Java programs using Java bytecode rewriting (building over the PENELOPE [27] infrastructure), and implements the identification of *null-WR* pairs, the pruning, and the logic-based procedures for both precise and relaxed prediction, using the SMT solver Z3 from Microsoft Research [11]. We show that EXCEPTIONULL is effective in predicting a large number of feasible runs in a suite of concurrent benchmarks. Evaluated over a suite of 13 benchmarks, we show the discovery of a slew of about 40 null-pointer dereference errors just be predicting from single test executions, and with no false positives. We know of no current technique that can find anywhere close to these many null-pointer dereferences on these benchmarks. We believe that the techniques proposed here hence make a significant leap forward in testing concurrent programs.

We also study some of the effects of techniques we have introduced, and estimate the inaccuracies caused and the scalability gained. We show that the pruning technique provides significant scalability benefits, while at the same time does not prohibit the prediction from finding any of the errors. We also show experimentally that the relaxation technique allows us to predict many more runs than the maximal causal model that result in errors (14 of the 41 errors were found due to relaxation).

We also show the efficacy of the relaxation technique by adapting our relaxed prediction to find *data-races*. Note that data-races are already defined at the shared communication level, and hence the approximation technique is not relevant. However, even in this setting, the static pruning allows us to scale more and the relaxation technique allows us to predict a lot more runs that have data-races than the strict prediction on the maximal causal model can (the latter is the current state-of-the-art in predicting data-races). Our tool discovers 60 data-races over our benchmarks of which 17 are found using relaxed prediction, showing that relaxed prediction is a very effective for other types of errors as well.

Related Work: The closest work related to ours in the realm of logic-based methods are those that stem from *bounded model-*

checking for finding executions of *bounded length* in concurrent programs that have bugs. Typically, a program's loops are unrolled a few times to get a bounded program, and using a logical encoding of the runs in this bounded program, a constraint solver is used to check if there is an error. We refer the reader to the papers from the NEC Labs group [24, 25] ([24] gives a clean encoding) as well as work from Microsoft Research [2, 12, 16, 21], where the programs are converted first to a sequential program from which bounded run constraints are generated. The crucial difference in our work is that we use logic in the *testing* setting to predict alternate interleavings. Another closely related work is CONMEM [35] (see also [34]), where the authors target a variety of memory errors in testing concurrent programs, including null-pointer dereferences, but the prediction algorithms are much weaker and quite inaccurate compared to our robust prediction techniques. Furthermore, there is no accurate rescheduling engine which leads the tool to have many false positives.

There are two promising approaches that have emerged in testing concurrent programs: *selective interleavings* and *prediction-based testing* (and combinations of these). The selective interleaving approach is to focus in testing a *small* but carefully chosen subset of interleavings. There are several tools and techniques that follow this philosophy: for instance, the CHESS tool from Microsoft [18] tests all interleavings that use a bounded number preemptions (unforced context-switches), pursuing the belief that most errors can be made to manifest this way. Several tools concentrate on testing *atomicity violating patterns* (for varying notions of what atomicity means), with the philosophy that they are much more likely to contain bugs [17, 19, 20, 33]. However, systematically testing even smaller classes of interleavings is often impossible in practice, as there are often too many of them.

There are several work on prediction-based testing that do not use logical methods. These algorithms may focus on predicting runs violating *atomicity* or containing *data-races*: those by Sorrentino et al. [13, 27], those by Wang and Stoller [31, 32], and [14] by Huang and Zhang. A more liberal notion of generalized dynamic analysis of a single run has also been studied in a series of papers by Chen et al. [9, 10]. JPREDICTOR [10] offers a predictive runtime analysis that uses *sliced causality* [9] to exclude the irrelevant causal dependencies from an observed run and then exhaustively investigates all of the interleavings consistent with the sliced causality to detect potential errors. The main drawback of non-logical prediction approaches is that the predicted runs may not be feasible. In fact, they ignore data which makes them less effective in finding bugs that are data-dependent such as null-pointer dereferences.

Logic-based prediction approaches, target precise prediction. Given an execution of the program, several work [29, 30] model the whole computation (local as well as global) logically to guarantee feasibility. The research presented in the above related work has too big an overhead to scale to large executions. Maximal Causality Model (MCM) [23], on the other hand, allows prediction at the level of shared communication and is the most precise prediction one can achieve at this level. MCM has been used by Said et al. [22] for finding data-race witnesses. We also use this model to predict runs leading to null-pointer dereferences.

2. Motivating Example

Consider a code extract from the `Pool 1.2` library [5] in the `Apache Commons` collection, presented in Figure 1. The object `pool`'s state, *open* or *closed*, is tested outside the synchronized block in method `returnObject`, by checking whether the flag variable `isClosed` is true. If so, then some local computation occurs, followed by a synchronized block that dereferences the shared object `pool`. A second method `close` closes the pool and sets `isClosed` to true to signal that the pool has been closed.

An error in this code (and such errors are very typical) stems from the fact that the check of `isClosed` in the method `return-Object` is not within the synchronized block; hence, if a thread executes the check at line ℓ, and then a concurrent thread executes the method `close()` before the synchronized block begins, then the access to the pool object at line ℓ' will raise an uncaught *null-pointer dereference exception*.

In a dynamic testing setting, consider the scenario where we observe an execution σ with two threads, where T executes the method `returnObject` first, and then, T' executes the method `close` after T finishes executing `returnObject`. There is no null-pointer dereference in σ. Our goal is to predict an alternate scheduling of events of σ that causes a null-pointer dereference.

Our prediction for null-pointer dereferences works as follows. In the run σ, a read of the shared variable `pool` at ℓ' occurs in T and the read value is not null. Also, a write to `pool` occurs in T' at ℓ'' which writes the value `null`. We ask whether there exists an alternative run σ' in which, the read at ℓ' (in T) can read the value `null` written by the write at location ℓ'' (in T') (as illustrated by the arrow in Figure 1).

Our prediction algorithm observes the shared events (such as shared reads/writes) but suppresses the semantics of local computations entirely and does not even observe them; they have been replaced by "..." in the figure as they play no role in our analysis.

Prediction of runs that force the read at ℓ' to read the null value written at ℓ'' must meet several requirements. Even if the predicted run respects the synchronization semantics for locks, thread creation, etc., the run may diverge from the observed run due to reading a different set of values for shared variables which will result in a different local computation path (e.g. the condition check at ℓ will stop the computation of the function right away if the value of `isClosed` is true). Therefore, we also demand that all other shared variable reads read the same value as they did in the original observed run, in order to guarantee that unobserved local computations will unfold in the same way as they did in the original run. This ensures the feasibility of the predicted runs.

Relaxed prediction: The requirement for all shared variable reads to read the same values, however, can be too strict in some cases. For instance, in our example, the variable `modCount` is a global counter keeping track of the number of modifications made to the `pool` data structure, and does not play any role in the local control flow reaching the point of null-pointer dereference at ℓ''. In the real execution leading to this null-pointer dereference, which is the one where block b_1 (from T) is executed first, followed by b_3 (from T') and then b_2 (from T), the read of `modCount` will read a different value than the corresponding value read in σ. However, this does not affect the feasibility of the run (in contrast to the value read for `isClosed`, which plays an important role in reaching the null-pointer dereference).

Our relaxed prediction model gives a slack threshold k, allowing predicted runs to have at most k reads that do not have to read the same values as in σ. By increasing the threshold k iteratively, our technique will find an execution that violates the read condition on `modCount`, but yet finds a feasible run that causes the null-pointer dereference in this example.

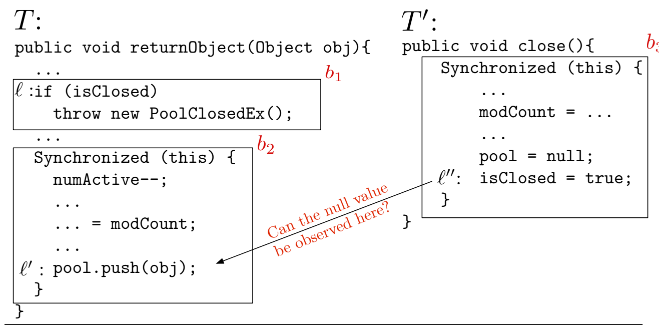
3. Preliminaries

Here, we present an overall overview of our prediction-based approach and set up a formal notation to describe the predicted runs.

3.1 Overview of proposed approach

Given a concurrent program P and an input I, we perform the following steps:

- **Monitoring:** We execute P on the input I and observe an arbitrarily interleaved run σ.



```
public void returnObject(Object obj){
  ...
ℓ :if (isClosed)
    throw new PoolClosedEx();
  ...
  Synchronized (this) {
    numActive--;
    ...
    ... = modCount;
    ...
ℓ' : pool.push(obj);
  }
}
```

```
public void close(){
  Synchronized (this) {
    ...
    modCount = ...
    ...
    pool = null;
ℓ'': isClosed = true;
  }
}
```

Figure 1. Code snippet of the buggy implementation of Pool.

- **Run Prediction:** We analyze the run σ to find a set of pair of events $\alpha = (e, f)$ in σ such that: (i) e is a write to shared variable x that writes a *null* value, (ii) f is a read from the same shared variable x that reads a non-null value in another thread, and (iii) static analysis on the run determines that there is a run $\widehat{\sigma}$ of P, obtained from reshuffling of events in σ, that respects locks and in which f reads the null value written by e. We call such pair of events $\alpha = (e, f)$ a *null-WR* pair. For each *null-WR* pair, we logically encode the set of runs and use SMT solvers to predict concrete runs $\widehat{\sigma}$ that force null-reads.
- **Rescheduling:** For each $\widehat{\sigma}$ generated by the *run prediction* phase, we re-execute the program, on the same input I, forcing it to follow $\widehat{\sigma}$. If it succeeds, then the null value read at f *may* later result in an error, such as a *null-pointer dereference exception*; we report all such confirmed errors.

We now set up the formal notation to describe the run prediction phase. In particular, the prediction algorithm will *ignore* computation of threads, and interleave at the level of blocks of local computations that happen between two reads/writes to global variables.

3.2 Modeling program runs, suppressing local computation

We model the runs of a concurrent program as a *word* where each letter describes the action done by a thread in the system. The word will capture the essentials of the run— shared variable accesses, synchronizations, thread-creating events, etc. However, we will *suppress* the local computation of each thread, i.e. actions a thread does by manipulating local variables, etc. that are not (yet) visible to other threads, and model the local computation as a *single* event *lc*. (In the formal treatment, we will ignore other concurrency constructs such as barriers, etc.; these can be accommodated easily into our framework.)

We fix an infinite countable set of thread identifiers $T = \{T_1, T_2, ..\}$ and define an infinite countable set of shared variable names SV that the threads manipulate. Without loss of generality, we assume that each thread T_i has a *single* local variable lv_i that reflects its entire local state. Let $V = SV \bigcup_i \{lv_i\}$ represent the set of all variables. Let $\mathit{Val}(x)$ represent the set of possible values that variable $x \in SV$ can get, and define $\mathit{Init}(x)$ as the initial value of x. We also fix a countable infinite set of locks L.

The actions that a thread T_i can perform on a set of shared variables SV and global locks L is defined as:

$$\begin{aligned}\Sigma_{T_i} = &\{T_i{:}\mathit{read}_{x,val},\ T_i{:}\mathit{write}_{x,val} \mid x \in SV, val \in \mathit{Val}(x)\}\\ &\cup \{\ T_i{:}\mathit{lc}\} \cup \{T_i{:}\mathit{acquire}(l),\ T_i{:}\mathit{release}(l) \mid l \in L\}\\ &\cup \{T_i{:}\ tc\ T_j \mid T_j \in T\}\end{aligned}$$

Actions $T_i{:}\mathit{read}_{x,val}$ and $T_i{:}\mathit{write}_{x,val}$ correspond to the thread T_i reading the value val from and writing the value val to the shared variable x, respectively. Action $T_i{:}\mathit{lc}$ corresponds to a local computation of thread T_i that accesses and changes the local state lv_i. Action $T_i{:}\ acquire(l)$ represents acquiring the lock l and the action $T_i{:}\ release(l)$ represents releasing of the lock l, by thread T_i. Finally, the action $T_i{:}\ tc\ T_j$ denotes the thread T_i creating the thread T_j.

We define $\Sigma = \bigcup_{T_i \in T} \Sigma_{T_i}$ as the set of actions of all threads. A word w in Σ^*, in order to represent a run, must satisfy several obvious syntactic restrictions, which are defined below.

Lock-validity, Data-validity, and Creation-validity: There are certain semantic restrictions that a run must follow. In particular, it should respect the semantics of locks and semantics of reads, i.e. whenever a read of a value from a variable occurs, the last write to the same variable must have written the same value, and the semantics of thread creation. These are captured by the following definitions ($\sigma|_A$ denotes the word σ projected to the letters in A).

DEFINITION 3.1 (Lock-validity). *A run $\sigma \in \Sigma^*$ is* lock-valid *if it respects the semantics of the locking mechanism. Formally, let $\Sigma_l = \{T_i{:}\ acquire(l), T_i{:}\ release(l) \mid T_i \in T\}$ denote the set of locking actions on lock l. Then σ is lock-valid if for every $l \in L$, $\sigma|_{\Sigma_l}$ is a prefix of*

$$\left[\bigcup_{T_i \in T} (T_i{:}acquire(l)\ T_i{:}release(l))\right]^* \qquad \blacksquare$$

DEFINITION 3.2 (Data-validity). *A run $\sigma \in \Sigma^*$ over a set of threads T, shared variables SV, and locks L, is* data-valid *if it respects the read-write constraints. Formally, for each n such that $\sigma[n] = T_i{:}\mathit{read}_{x,val}$, one of the following holds:*
(i) The last write action to x writes the value val. I.e. there is a $m < n$ such that $\sigma[m] = T_j{:}\mathit{write}_{x,val}$ and there is no $m < k < n$ such that $\sigma[k] = T_q{:}\mathit{write}_{x,val'}$ for any val' and any thread T_q, or (ii) there is no write action to variable x before the read, and val is the initial value of x. I.e. there is no $m < n$ such that $\sigma[m] = T_j : \mathit{write}_{x,val'}$ (for any val' and any thread T_j), and $val = \mathit{Init}(x)$. $\blacksquare$

DEFINITION 3.3 (Creation-validity). *A run $\sigma \in \Sigma^*$ over a set of threads T is* creation-valid *if every thread is created at most once and its events happen after this creation, i.e., for every $T_i \in T$, there is at most one occurrence of the form $T_j{:}\ tc\ T_i$ in w, and, if there is such an occurrence, then all occurrences of letters of Σ_{T_i} happen after this occurrence.* $\blacksquare$

Program Order: Let $\sigma = a_1...a_n$ be a run of a program P. The *occurrence* of actions in runs are referred to as *events* in this paper. Formally, the set of events of the run is $E = \{e_1, \ldots, e_n\}$, and there is a labeling function λ that maps every event to an action, given by $\lambda(e_u) = a_u$.

While the run σ defines a total order on the set of events in it $(E, \leq)$, there is an induced total order between the events of each thread. We formally define this as $\sqsubseteq_i$ for each thread T_i, as follows: for any $e_s, e_t \in E$, if a_s and a_t belong to thread T_i and $s \leq t$ then $e_s \sqsubseteq_i e_t$. The partial order that is the union of all the program orders is $\sqsubseteq = \cup_{T_i \in T} \sqsubseteq_i$.

The Maximal Causal Model for prediction: Given a run σ corresponding to an actual execution of a program P, we would like our prediction algorithms to synthesize new runs that interleave the events of σ to cause reading of null values. However, we want to predict *accurately*; in other words we want the predicted runs to be feasible in the actual program.

We now give a sufficient condition for a partial run predicted from an observed run to be *always* feasible. This model of prediction was defined by Şerbănuţă et al., and is called the *maximal causal model* [23]; it is in fact the most liberal prediction model that ensures that the predicted runs are always feasible in the program that work purely dynamically (i.e. no other information about the program is known other than the fact that it executed this set of observable events, which in turn do not observe computation).

We generalize the model slightly by taking into account thread creation.

DEFINITION 3.4 (Maximal causal model of prediction [23]). *Let σ be a run over a set of threads T, shared variables SV, and locks L. A run σ' is* precisely predictable *from σ if (i) for each $T_i \in T$, $\sigma'|_{T_i}$ is a prefix of $\sigma|_{T_i}$, (ii) σ' is lock-valid, (iii) data-valid, and (iv) creation-valid. Let $PrPred(\sigma)$ denote the set of all runs that are precisely predictable from the run σ.* ∎

The first condition above ensures that the events of T_i executed in σ' is a prefix of the events of T_i executed in σ. This property is crucial as it ensures that the local state of T_i can evolve correctly. Note that we are forcing the thread T_i to read the same values of global variables as it did in the original run. Along with data-validity, this ensures that the thread T_i reads precisely the same global variable values and updates the local state in the same way as in the original run. Lock-validity and creation-validity are, of course, required for feasibility. We will refer to runs predicted according to the maximal causal model (i.e. runs in $PrPred(\sigma)$) as the precisely predicted runs from σ.

The following soundness of the prediction that assures all predicted runs are feasible, follows:

THEOREM 3.5 ([23]). *Let P be a program and σ be a run corresponding to an execution of P. Then every precisely predictable run $\sigma' \in PrPred(\sigma)$ is feasible in P.* ∎

The above theorem is independent from the class of programs. We will assume however that the program is locally deterministic (non-determinism caused by threads interleaving is, of course, allowed). The above theorem, in fact, even holds when local computations of P are *non-deterministic*; i.e. the predicted runs will still be feasible in the program P. However, in order to be able to execute the predicted runs, we need to assume determinism of local actions. In this case, we can schedule the run σ' precisely and examine the outcomes of the tests on these runs.

3.3 The prediction problem for null-reads

We are now ready to formally define the precise prediction problem for forcing null-reads.

DEFINITION 3.6 (Precisely predictable null-reads). *Let σ be a run of a program P. We say that σ' is a* precisely predictable run that forces null-reads *if there is a thread T_i and a variable x such that the following are satisfied: (i) $\sigma' = \sigma''.f$ where f is of the form T_i: read$_{x,null}$, (ii) σ'' is a precisely predictable run from σ using the maximal causal model, and (iii) there is some $val \neq null$ such that $(\sigma''|_{\Sigma_i})$. T_i: read$_{x,val}$ is a prefix of $\sigma|_{\Sigma_i}$.* ∎

Intuitively, the above says that the run σ' must be a precisely predictable run from σ followed by a read of null by a thread T_i on variable x, and further, in the observed run σ, thread T_i must be executing a non-null read of variable x after executing its events in σ''. The above captures the fact that we want a precisely predictable run followed by a single null-read that corresponded to a non-null read in the original observed run. Note that σ' itself is not in $PrPred(\sigma)$, but is always feasible in the program P, and results in a null-read by thread T_i on variable x that had not happened in the original run.

The precisely predictable runs that force null-reads are hence excellent candidates to re-execute and test; if the local computation after the read does not check the null-ness of x before dereferencing a field of x, then this will result in an exception or error.

4. Identifying *null-WR* pairs using lock-sets

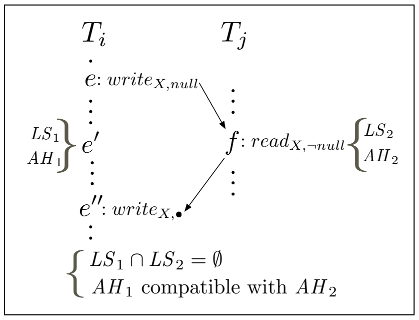
The first phase of our prediction is to identify *null-WR* pairs $\alpha = (e, f)$ where e is a write of null to a variable and f is a read of the same variable, but where the read in the original run reads a non-null value. Moreover, we would like to identify pairs that are feasible at least according to the hard constraints of thread-creation and locking in the program. For instance, if a thread writes to a shared variable x and reads from it in the same lock-protected region of code, then clearly the read cannot match a write protected by the same lock in another thread. Similarly, if a thread initializes a variable x to a non-null and then creates another thread that reads x, clearly the read cannot read an uninitialized x. We use a lock-set based static analysis of the run (without using a constraint solver) to filter out such impossible read-write pairs. The ones that remain are then subject to a more intensive analysis using a constraint solver.

Using a static analysis on the observed run σ, we first collect all *null-WR* pairs $\alpha = (e, f)$. Then, we prune away *null-WR* pairs for which there is no lock-valid run in which f is reading the null value written by e. Then, for each *null-WR* pair $\alpha = (e, f)$ left, we use our precise logical prediction algorithm to obtain a lock-valid, data-valid and creation-valid run in which f is reading the null value written by e. However, instead of using run σ for the purposes of the prediction, we slice a *relevant* segment of it, and use the segment instead. The reason for this is twofold: (1) these run segments are often orders of magnitude smaller than the complete run, and this increases the scalability of our technique and (2) when a precisely predictable run does not exist, we use a more relaxed version of the constraints to generate a new run, limiting the improvisation to a smaller part of run increases our chances of obtaining a feasible execution. We formally define this *relevant run segment* and how it is computed in Section 7.

In this static analysis, the idea is to check if the *null-WR* pair $\alpha = (e, f)$ can be realized in a run that respects lock-validity and creation-validity constraints only (and not data-validity). Creation validity is captured by computing a vector clock associated with each event, where the vector clock captures only the hard causality constraints of thread creation. If f occurs before e according to this relation, then clearly it cannot occur after e and the pair is infeasible. Lock-validity is captured by reducing the problem of realizing the pair (e, f) to *pairwise reachability* under nested locking [15], which is then solved by computing lock-sets and acquisition histories for each event. We describe only the lock-validity checking below. Similar techniques have been exploited for finding *atomicity* violations in the tool PENELOPE [27].

Checking lock-valid reachability Consider a *null-WR* pair $\alpha = (e, f)$ and a run σ in which f (a read in thread T_j) occurs first, and later the write event e is performed by T_i.

Let us assume that e'' is the next write event (to the same variable accessed in e and f) in T_i after e. If there exists a lock-valid run σ' (obtained by permuting the events in σ) in which f reads the null value provided by e, then in σ', f should be scheduled after e, but before e''; if f is scheduled also after e'', then the write in e'' overwrites the null value written by e before it reaches f. This means that there should exist an event e' of thread T_i, occurring between events e and e'', that is executed right before (or after f) in σ'; in other words, e' and f are co-reachable.


T_i
T_j
e: $write_{X,null}$
LS_1
AH_1
e'
f: $read_{X,\neg null}$
LS_2
AH_2
e'': $write_{X,\bullet}$
$LS_1 \cap LS_2 = \emptyset$
AH_1 compatible with AH_2

We use a simple technique [15] to check if there is an event e' in T_i between e and e'' such that e' and f are co-reachable in a lock-valid run. The co-reachability check is done by examining the lock-sets and ac-

$$\psi \equiv PO \wedge CV \wedge DV \wedge LV$$

$$PO = (\textstyle\bigwedge_{i=1}^{n} PO_i) \wedge C_{init}$$

$$C_{init} = \textstyle\bigwedge_{i=1}^{n} (ts_{e_{init}} < ts_{e_{i,1}})$$

$$PO_i = \textstyle\bigwedge_{j=1}^{m_i - 1} (ts_{e_{i,j}} < ts_{e_{i,(j+1)}})$$

$$DV = \textstyle\bigwedge_{x} \quad \bigwedge_{val \in Val(x)} \quad \bigwedge_{r \in R_{x,val}} \Big(\bigvee_{w' \in W_{x,val}} Coupled_{r,w'} \Big)$$

$$Coupled_{r,w} = (ts_w < ts_r) \wedge \bigwedge_{e'' \in W_x - \{w\}} ((ts_{e''} < ts_w) \vee (ts_r < ts_{e''}))$$

$$LV = LV_1 \wedge LV_2$$

$$LV_1 = \bigwedge_{i \neq j \in \{1,..,n\}} \bigwedge_{lock\ l} \bigwedge_{\substack{[e_{ac}, e_{rel}] \in L_{i,l} \\ [e'_{ac}, e'_{rel}] \in L_{j,l}}} \Big(ts_{e_{rel}} < ts_{e'_{ac}} \vee \ ts_{e'_{rel}} < ts_{e_{ac}} \Big)$$

$$LV_2 = \bigwedge_{i \neq j \in \{1,..,n\}} \bigwedge_{lock\ l} \bigwedge_{\substack{e_{ac} \in NoRel_{i,l} \\ [e'_{ac}, e'_{rel}] \in L_{j,l}}} \Big(ts_{e'_{rel}} < ts_{e_{ac}} \Big)$$

Figure 2. Constraints capturing the maximal causal model.

quisition histories at e' and f: the lock-sets at e' and f must be disjoint and the acquisition histories at e' and f must be compatible.

Note that the above condition is *necessary* for the existence of the lock-valid run σ', but not *sufficient*; hence filtering out pairs that do not meet this condition is sound.

5. Precise prediction by logical constraint solving

We now describe how we solve the problem of precisely predicting a run that realizes a *null-WR* pair $\alpha = (e, f)$. This problem is to predict whether there is an alternate schedule in the maximal causal model that forces the read at f to read the null value written by e. We solve this using a logic constraint solver (an SMT solver); the logic of the constraints is in a fragment that is efficiently decidable.

Prediction according to the maximal causal model is basically an encoding of the creation-validity, data-validity, and lock-validity constraints using logic, where quantification is removed by expanding over the finite set of events under consideration. Modeling this using constraint solvers has been done before ([22]) in the context of finding data races. We reformulate this encoding briefly here for several reasons. First, this makes the exposition self-contained, and there are a few adaptations to the null-read problem that need explanation. Second, we perform a wide set of carefully chosen optimizations on this encoding, whose description needs this exposition. And finally, the relaxation technique, which is one of the main contributions of this paper, is best explained by referring directly to the constraints.

Capturing the maximal causal model using logic

Given a run σ, we first encode the constraints on all runs predicted from it using the maximal causal model, independent of the specification that we want runs that match a given *null-WR* pair . A predicted run can be seen as a total ordering of the set of events E of the run σ. We use an integer variable ts_e to encode the *timestamp* of event $e \in E$ when e occurs in the predicted run. Using these timestamps, we logically model the constraints required for precisely predictable runs (see Definition 3.4), namely that the run respect the program order of σ, that it be lock-valid, data-valid, and creation-valid.

Figure 2 illustrates the various constraints. The constraints are a conjunction of program order constraints (PO), creation-validity constraints (CV), data-validity constraints (DV), and lock-validity constraints (LV).

The program order constraint (PO) captures the condition that the predicted run respect the program order of the original observed run. Suppose that the given run σ_α consists of n threads, and let $\sigma_\alpha|_{T_i} = e_{i,1}, e_{i,2}, ..., e_{i,m_i}$ be the sequence of events in σ_α that relates to thread T_i. Then the constraint PO_i demands that the time-stamps of the predicted run obey the order of events in thread T_i, and PO demands that all threads meet their program order. We also consider an initial event e_{init} which corresponds to the initialization of variables. This event should happen before any thread starts the execution in any feasible permutation, and is encoded as the constraint C_{init}.

Turning to creation-validity, suppose that $e_{tc(i)}$ is the event that creates thread T_i. Then the constraint CV demands that the first event of T_i can only happen after $e_{tc(i)}$. Combined with program order constraint, this means that all events before the creation of T_i in the thread that created T_i must also occur before the first event of T_i.

The data-validity constraints DV (see Definition 3.3) capture the fact that reads must be coupled with appropriate writes; more precisely, that every read of a value from a variable must have a write before it writing that value to that variable, and moreover, there is no other intermediate write to that variable. Let $R_{x,val}$ represent the set of all read events that read value val from variable x in σ_α, W_x represent the set of all write events to variable x, and $W_{x,val}$ represent the set of all write events that specifically write value val to variable x. For each read event $r = read_{x,val}$ and write event $w \in W_{x,val}$, the formula $Coupled_{r,w}$ represents the requirement that w is the most recent write to variable x before r and hence r is coupled with w. The constraint DV demands that all reads be coupled with writes that write the same value as the read reads.

Lock-validity is captured by the formula LV. We assume that each lock acquire event ac of lock l in the run is matched by precisely one lock release event rel of lock l in the same thread, unless the lock is not released by the thread in the run. We call the set of events in thread T_i between ac and rel a lock block corresponding to lock l represented by $[ac, rel]$. Let $L_{i,l}$ be the set of lock blocks in thread T_i regarding lock l. Then LV_1 asserts that no two threads can be simultaneously inside a pair of lock blocks $[e_{ac}, e_{rel}]$ and $[e'_{ac}, e'_{rel}]$ corresponding to the same lock l. Turning to locks that never get released, the constraint LV_2 handles asserts that the acquire of lock l by a thread that never releases it must always occur after the releases of lock l in every other thread. In this formula, $NoRel_{i,l}$ stands for lock acquire events in T_i with no corresponding later lock release event.

Optimizations

The constraints, when written out as above, can be large. We do several optimization to control the formula bloat (while preserving the same logical constraint).

The data-validity constraint above is expensive to express, as it is, in the worst case, cubic in the maximum number of accesses to any variable. There are several optimizations that reduce the number of constraints in the encoding. Suppose that $r = read_{x,val}$ is performed by thread T_i.

- Each write event w' to x that occurs after r in T_i, i.e. $r \sqsubseteq_i w'$, can be excluded in the constraints related to coupling r with a write in constraint DV above.
- Suppose that w is the most recent write to x before r in T_i. Then, each write event w' before w in T_i, (i.e. $w' \sqsubseteq_i w$), can be excluded in the constraints related to coupling r with a write in constraint DV above.
- When r is being coupled with $w \in W_{x,val}$ in thread T_j, each write event w' before w in T_j, i.e. $w' \sqsubseteq_j w$, can be excluded as candidates for e'' in the formula $Coupled_{r,w}$.

- Suppose that r is being coupled with $w \in W_{x,val}$ in thread T_j and w' is the next write event to x after w in thread T_j. Then each write event w'' after w' in T_j, i.e. $w' \sqsubseteq_j w''$, can be excluded as candidates for e'' in the formula $Coupled_{r,w}$.
- Event r can be coupled with e_{init} only when there is no other write event to x before r in T_i, i.e. $\nexists w.\ (w \sqsubseteq_i r \wedge w \in W_x)$. Furthermore, it is enough to check that the first write event to x in each thread (if it exists) is performed after r.

The lock-validity formula above, which is quadratic in the number of lock blocks, is quite expensive in practice. We can optimize the constraints. If a read event r in thread T_i can be coupled with only *one* write event w which is in thread T_j then in all precisely predictable runs, w should happen before r. Therefore, the lock blocks according to lock l that are in T_j before w and the lock blocks according to lock l that are in T_i after r are already ordered. Hence, there is no need to consider constraints preventing T_i and T_j to be simultaneously in such lock blocks. In practice, this greatly reduces the number of constraints. Furthermore, when considering lock acquire events with no corresponding release events in LV_2 above, it is sufficient to only consider the *last* corresponding lock blocks in each thread and exclude the earlier ones from the constraint.

Predicting runs for a *null-WR* pair

We adapt the above constraints for predicting in the maximal causal model to predict whether a *null-WR* pair $\alpha = (e, f)$ is realizable. Suppose that σ and $\alpha = (e, f)$ are a run and a *null-WR* pair passed to the prediction phase, respectively. Notice that in the original run f reads a non-null value while we will force it to read null in the predicted run by coupling it with write event e. Indeed, this is the whole point of predicting runs— we would like to diverge from the original run at f by forcing f to read a null value. Note that once f reads a different value, we no longer have any predictive power on what the program will do (as we do not examine the code of the program but only its runs). Consequently, we cannot predict any events causally later than f.

The prediction problem is hence formulated as follows:

Given a run σ, and a null-WR pair $\alpha = (e, f)$ in σ, algorithmically find a precisely predictable run from σ that forces null-reads according to α; i.e. f is the last event and reads the null value written by e.

The prediction problem is to find precisely predicted runs that execute e followed by f, while avoiding any other write to the corresponding variable between e and f. The constraints that force the read f be coupled with the write e is $NC = Coupled_{f,e}$.

Furthermore, recall that the feasibility of the run that we are predicting needs to be ensured only *up to* the read f. Consequently, we drop from the data-validity formula that the value read at f (in the original run) match the last write (it should instead match e as above).

A further complication is scheduling events that happen after e in the same thread. Note that some of these events may need to occur in order to satisfy the requirements of events before f (for instance a read before f may require a write after e to occur). However, we may not want to predict some events after e, as we are really only concerned with f occurring after e. Our strategy here is to let the solver figure out the precise set of events to schedule after e (and before the next write to the same variable as e is writing to) in the same thread.

For events after e in T_i, we enforce lock-validity and data-validity constraints only if they are scheduled *before* f. More precisely, we replace $\vee_{w'} Coupled_{r_i,w'}$ in the formula DV to $(ts_r < ts_f \Rightarrow \vee_{w'} Coupled_{r_i,w'})$. Similarly, we drop the lock constraints on events occurring after f (this relaxation is more involved but straightforward).

In summary, we have reduced the problem of predicting a run according to the maximal causal model that causes the null write-read pair to be realizable to a satisfiability of a formula ψ in logic. The constraints generated fall within the class of quantifier-free difference logic constraints which SMT solvers efficiently solve in practice.

6. Relaxed prediction

The encoding proposed in the previous section is sound, in the sense that it guarantees feasibility of the predicted runs. However, as demonstrated by the example in Section 2, sound prediction under the maximal causal model can be too restrictive and result in predicting no runs. Slightly diverging from the original can sometimes lead to prediction of runs that are feasible in the original program.

We hence have a tension between two choices— we would like to maintain the same values read for as many shared variable reads as possible to increase the probability of getting a feasible run, but at the same time allow a few reads to read different values to make it possible to predict some runs. Our proposal, which is one of the main contributions of this paper, is an iterative algorithm for finding the *minimum* number of reads that can be exempt from data-validity constraints that will allow the prediction algorithm to find at least one run. We define a suitable *relaxed* logical constraint system to predict such a run. Our experiments show that exempting a few reads from data-validity constraints greatly improves the flexibility of the constraints and increases the possibility of predicting a run, and at the same time, the predicted runs are often feasible.

The iterative algorithm works as follows. Let's assume there are n shared variable reads that are required to be coupled with specific write by the full set of data-validity constraints. The data-validity constraints are expressed so that we specifically ask for n shared reads to be coupled correctly. If we fail to find a solution satisfying constraints for all n reads, then we repeatedly decrement n, and attempt to find a solution that couples $n-1$ reads in the next round, and so on. The procedure stops whenever a run (solution) is found. The change required in the encoding to make this possible is described below.

For every read event $r_i \in R$, we introduce a new Boolean variable, b_i, that is true if the data-validity constraint for r_i is satisfied, and false otherwise. In addition, we consider an integer variable $bInt_i$ which is initially 0, and set to 1 only when b_i is true. This is done through a set of constraints, one for each $r_i \in R$: $[(b_i \rightarrow bInt_i = 1) \wedge (\neg b_i \rightarrow bInt_i = 0)]$. Also, for each $r_i \in R$, we change the sub-term $\vee_{w'} Coupled_{r_i,w'}$ to $(ts_r < ts_f) \Rightarrow (b_i \Rightarrow \vee_{w'} Coupled_{r_i,w'})$ in DV, forcing the data-validity constraint for read r_i to hold when b_i is true. Note that with these changes, we require a different theory, that is *Linear Arithmetic* in the SMT solver to solve the constraints, compared to the *Difference Logic* which was used for our original set of constraints.

Initially, we set a threshold η to be $|R|$, the number of all read events. In each iteration, we assert the constraint $\sum_{1 \le i \le |R|} bInt_i = \eta$, which specifies the number (η) of data-validity constraints that should hold in that iteration. If no run can be predicted with the current threshold η (i.e. the constraint solver reports unsatisfiability), then η is decremented in each iteration, until the formula is satisfiable. This way, when a satisfying assignment is found, it is guaranteed to have the maximum number of reads that respect data-validity possible for predictable run.

Note that once $\eta < |R|$, the predicted run is not theoretically guaranteed to be a feasible run. However, in practice, when η is close to $|R|$ and a run is predicted, this run is usually feasible in the program.

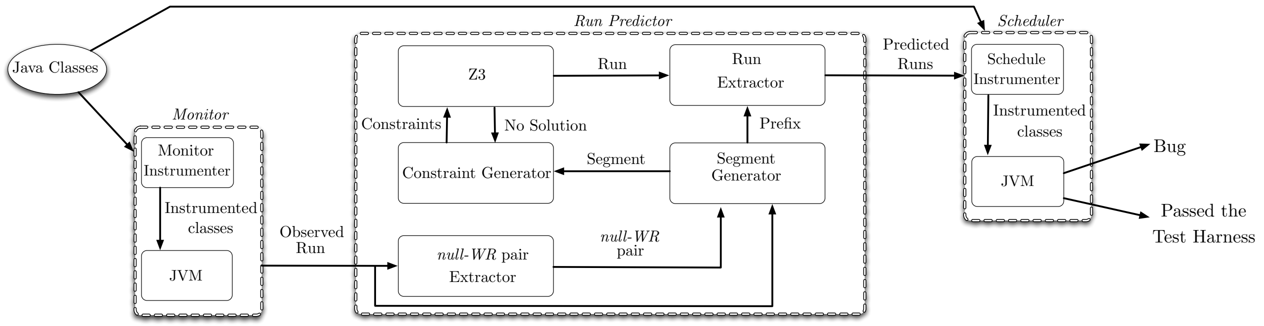


Figure 3. EXCEPTIONULL.

7. Pruning executions for scalability

Identifying *null-WR* pairs using the lock-set based analysis and then subjecting them to constraint checking is a precise method to force null reads. However, in our experiments, we discovered that the constraint solving approach does not scale well when runs get larger. In this section, we propose a pruning technique for the runs that removes a large prefix of them while maintaining the property that any run predicted from the suffix will still be feasible. While this limits the number of predictable runs in theory, we show that in practice, it does not prevent us from finding errors (in particular, *no error* was missed due to pruning in our experiments). Furthermore, we show that in practice pruning improves the scalability of our technique, in some cases by an order of magnitude.

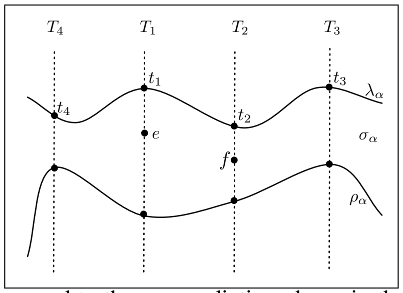
Consider an execution σ and a *null-WR* pair $\alpha = (e, f)$. The idea behind pruning is to first construct the causal partial order of events of σ, and then remove two sets of events from it. The first set consists of events that are *causally after* e and f (except for some events, as described in detail below). The second set is a causally prefix-closed set of events (a configuration) that are *causally before* e and f, and where all the locks are free at the end of execution of this configuration. The intuition behind this is that such a configuration can be replayed in the newly predicted execution precisely in the same way as it occurred in the original run, and then stitched to a run predicted from the suffix, since the suffix will start executing in a state where no locks are held.

In order to precisely define this run segment, we define a notion of partial order on the set of events E that captures the causal order. Let D denote the dependency relation between actions that relates two actions of the same thread, reads and writes on the same variable by different threads, and lock acquisition and release actions of the same lock in different threads. We define the partial order $\preceq \subseteq E \times E$ on the set of program events as the *least* partial order relation that satisfies the condition that $(e_i, e_j) \in \preceq$ whenever $a_i = \sigma[i]$, $a_j = \sigma[j]$, $i \le j$, and $(a, a') \in D$ where a_i and a_j are actions performed by events e_i and e_j, respectively.

Let us define ρ_α as the *smallest* subset of events of σ that satisfies the following properties: (1) ρ_α contains events e and f, (2) for any event e' in ρ_α, all events $e'' \preceq e'$ are in ρ_α, and (3) for every event corresponding to a lock *acquire* in ρ_α, its corresponding *release* event is also in ρ_α.

The intuition is that events that are not in ρ_α are not relevant for the scheduling of the *null-WR* pair ; they are either far enough in the future, or are not dependent on any of the events in ρ_α. The figure below presents a run of a program with 4 threads that is projected into individual threads. Here, e belongs to thread T_1 and f belongs to thread T_2. The cut labeled ρ_α marks the boundary after which all events are not *causally before* e and f, and hence, need not be considered for the generation of the new run.

Next, we identify a causally prefix-closed set of events before e and f to remove. For the *null-WR* pair α, define λ_α as the *largest* subset of events of ρ_α that has the following properties: (1) it does not contain e or f, (2) for any event e' in λ_α, all events $e'' \preceq e'$ are in λ_α, and (3) for any event e' in T_i such that e' is the last event of T_i in λ_α (with respect to $\sqsubseteq_i$), the lockset associated to e' in T_i is empty. In the above figure, the curve labeled λ_α marks the boundary of λ_α, and events $T_1, \ldots, T_4$ have empty lock-sets.



The run segment relevant to a *null-WR* pair α is then defined as the set of events in $\sigma_\alpha = \rho_\alpha \setminus \lambda_\alpha$ scheduled according to the total order in σ ($\le$). One can use a simple worklist algorithm to compute both ρ_α and λ_α, and consequently σ_α. This run segment is passed to the run prediction phase, in the place of the whole run σ.

8. Implementation

We have implemented our approach in a tool named EXCEPTIONULL . Figure 3 demonstrates the architecture of EXCEPTIONULL . It consists of three main components: a monitor, a run predictor, and a scheduler. The monitor and scheduler are built on top of the PENELOPE tool framework, with considerable enhancements and optimizations, including the extension of the monitoring to observe values of shared variables at reads and writes. In the following, we will explain each of these components in more details.

Monitor: The monitor component has an instrumenter which uses the Bytecode Engineering Library (BCEL) [4] to (automatically) instrument every class file in bytecode so that a *call* to an event recorder is made after each *relevant* action is performed. These relevant actions include field and array accesses, acquisition and releases of locks, thread creations and thread joins, etc., but exclude accesses to local variables. The instrumented classes are then used in the Java Virtual Machine (JVM) to execute the program and get an observed run. For the purpose of generating the data-validity constraints, the values read/written by shared variable accesses are also recorded. For variables with primitive types (e.g. Boolean, integer, double, etc), we just use the values read/written. Objects and arrays are treated differently; the object *hash code* (by `System.identityHashCode()`) is used as the value every time an object or an array is accessed.

Run Predictor: The run predictor consists of several components: *null-WR* pair extractor, segment generator, constraint generator, Z3 SMT solver, and run extractor. The *null-WR* pair extractor generates a set of *null-WR* pairs from the observed run by the static lock analysis described in Section 4. The segment generator component, for each *null-WR* pair $\alpha = (e, f)$, isolates a part of

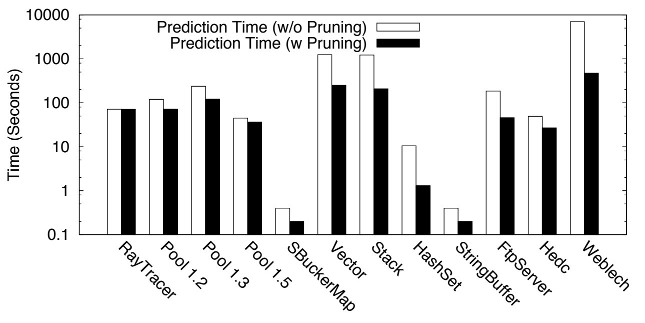


Figure 4. Prediction times with/without pruning in log scale.

run ρ that is *relevant* to α as described in Section 7 and passes it to the constraint generator. Given a *null-WR* pair and the relevant segment, the constraint generator produces a set of constraints, based on the algorithm presented in Section 5, and passes it to the Z3. Any model found by Z3 corresponds to a concurrent schedule. The run extractor component generates a run based on the model returned by Z3. When Z3 cannot find a solution, the constraint generator iteratively weakens the constraints (see Section 6) and calls Z3 until a solution is found.

Scheduler: The scheduler is implemented using BCEL [4] as well; we instrument the scheduling algorithm into the Java classes using bytecode transformations, so that the program interacts with the scheduler when it is executing the same set of events that were monitored. The scheduler, at each point, looks at the predicted run, and directs the appropriate thread to perform a sequence of n steps. The threads wait for a signal from the scheduler to proceed, and only then do they execute the number of (observable) events they are instructed to execute. Afterwards, the threads communicate back to the scheduler, relinquishing the processor, and await further instructions. The communication between the scheduler and the threads is implemented using wait-notify synchronization which allows us to have a finely orchestrated scheduling process.

Data-Race Prediction: Our proposed monitoring, logic-based precise and relaxed prediction on statically pruned runs, and the rescheduling is a general framework and can be adapted to errors other than null-pointer dereferences as well. In order to study the effects of relaxed prediction and static pruning, we implemented a data-race prediction unit as well to our tool, as data-races are a more well studied class of errors for which precise prediction has been studied. Due to lack of space, we do not discuss the details of the data race detection unit here.

9. Evaluation

We subjected EXCEPTIONULL to a benchmark suite of 13 concurrent programs, against several test cases and input parameters. Experiments were performed on an Apple MacBook with 2 Ghz Intel Core 2 Duo processors and 2GB of memory, running OS X 10.4.11 and Sun's Java HotSpot 32-bit Client VM 1.5.0.

Benchmarks. The benchmarks are all concurrent Java programs that use `synchronized` blocks and methods as means of synchronization. They include `RayTracer` from the Java Grande multi-threaded benchmarks [3], `elevator` from ETH [28], `Vector`, `Stack`, `HashSet` and `StringBuffer` from Java libraries, `Pool` (three different releases) and `StaticBucketMap` from the Apache Commons Project [5], `Apache FtpServer` from [7], `Hedc` from [6], and `Weblech` from [8]. `elevator` simulates multiple lifts in a building; `RayTracer` renders a frame of an arrangement of spheres from a given view point; `Pool` is an object pooling API in the Apache Commons suite; `StaticBucketMap` is a thread-safe implementation of the Java Map Interface; `Apache FtpServer` is a FTP server by Apache; and `Vector`, `Stack`, `HashSet` and `StringBuffer` are Java libraries that respectively implement the concurrent vector, the concurrent stack, the HashSet and the StringBuffer data structures. `Hedc` is a Web crawler application and `Weblech` is a websites download tool.

Experimental Results. Table 1 illustrates the experimental results for *null-pointer dereference prediction*; information is provided about all the three phases of monitoring, run prediction, and scheduling.

In the monitoring phase, the number of threads, shared variables, locks, the number of potential interleaving points (i.e. number of global events), and the time taken for monitoring are reported. For the prediction phase, we report the number of *null-WR* pairs in the observed run, the number of precisely predicted runs, and the additional number of predicted runs after relaxing the data-validity constraints (when there is no precisely predicted run for a null read-write pair). In the scheduling phase, we report the total number of schedulable predictions among the predicted ones. Finally, we report the average time for prediction and rescheduling of each run, the total time taken to complete the tests (for on all phases on all predicted executions), and also the number of errors found using the precise and relaxed predicted runs.

Errors Found. In almost all the cases, the errors manifested in the form of raised exceptions during the execution. In `Weblech`, in addition to a null-pointer dereference, an unwanted behavior occurred (the user is asked to push a stop button even after the website is downloaded completely, resulting in non-termination!). `RayTracer` has a built-in validation test which was failed in some of the predicted runs. For some of the test cases of `Vector` and `Stack` the output produced was not the one expected. We report the errors found in two categories; those that were found through the precise prediction algorithm, and those that were found after weakening data-validity constraints (relaxation).

The effect of pruning: Figure 4 illustrates the substantial impact of our pruning algorithm in reducing prediction time. We present prediction times with and without using the pruning algorithm. Note that the histogram is on a logarithmic scale. For example, in the case of `Weblech`, the prediction algorithm is about 16 times faster with pruning. Furthermore, all errors found without the pruning were found on the pruned runs, showing that the pruning did not affect the quality of error-finding on our benchmarks.

Data Race Detection. Table 2 presents the results of data-race prediction on our benchmarks using the same observed runs as in the null-reads prediction. For each benchmark we report the total number of data-races found; these are all distinct races identified by the code location of the racy access. We also report the number of distinct variables involved in data-races. For brevity, information about different test cases is aggregated for each benchmark (see [1] for more details).

Observations: EXCEPTIONULL performs remarkably well, predicting a large number of feasible program runs on which there are null-pointer dereferences and data-races. In total, it finds about 40 executions with null-pointer dereferences and 60 races, which to our knowledge, is the most successful attempt at finding errors on

Application	Elevator	RayTracer	Pool 1.2	Pool 1.3	Pool 1.5	SBucketMap	Vector	Stack	HashSet	StringBuffer	FtpServer	Hedc	Weblech	**Total**
Num. of Data-Races (by precise prediction)	-	3	5	-	8	1	1	1	6	-	8	5	5	**43**
Additional Data-Races (by relaxation)	-	3	0	-	2	0	1	1	3	-	3	0	4	**17**
Number of Distinct Involved Variables	-	1	3	-	4	1	2	2	3	-	7	4	3	**31**

Table 2. Experimental Results for data race prediction.

			Monitoring					Prediction			Scheduling				
Application (LOC)	Input	Base	Num. of Threads	Num. of Shared Variables	Num. of Locks	Num. of Potential Interleaving Points	Time to Monitor	Num. of *null-WR* Pairs	Num. of Precisely Predicted Runs	Additional Predicted Runs by Relaxation	Num. of Schedulable Predictions	Average Time per Predicted Run	Total Time	Null Pointer Deref. by Precise Prediction	Additional Null-Pointer Deref. by Relaxation
`Elevator` (566)	Data	7.3s	3	116	8	14K	7.4s	0	-	-	-	-	7.9s	**0**	**0**
	Data2	7.3s	5	168	8	30K	7.4s	0	-	-	-	-	8.9s	**0**	**0**
	Data3	19.2s	5	723	50	150K	19.0s	0	-	-	-	-	58.5s	**0**	**0**
`RayTracer` (1.5K)	A-10	5.0s	10	106	10	648	5.0s	9	9	-	9	5.6s	50.5s	**1***	**0**
	A-20	3.6s	20	196	20	1.7K	4.4s	19	19	-	19	6.7s	2m15s	**1***	**0**
	B-10	42.4s	10	106	10	648	42.5s	9	9	-	9	42.7s	6m24s	**1***	**0**
`Pool 1.2` (5.8K)	PT1	<1s	4	28	1	98	<1s	3	2	1	3	<1s	1.6s	**2**	**0**
	PT2	<1s	4	29	1	267	<1s	3	0	0	-	-	8.8s	**0**	**0**
	PT3	<1s	4	20	3	180	<1s	26	0	23	16	1.2s	27.0s	**0**	**3**
	PT4	<1s	4	24	3	360	<1s	32	2	21	15	2.5s	57.8s	**0**	**1**
`Pool 1.3` (7K)	PT1	<1s	4	30	1	100	<1s	3	0	3	3	<1s	2.6s	**0**	**0**
	PT2	<1s	4	31	1	271	<1s	3	0	0	-	-	9.8s	**0**	**0**
	PT3	<1s	4	20	3	204	<1s	35	0	30	19	1.4s	42.9s	**0**	**0**
	PT4	<1s	4	23	3	422	<1s	62	1	48	29	2.2s	1m49s	**0**	**1**
`Pool 1.5` (7.2K)	PT1	<1s	4	33	2	124	<1s	2	0	1	1	1.5s	1.5s	**0**	**0**
	PT2	<1s	4	34	2	306	<1s	5	0	1	0	10.5s	10.5s	**0**	**0**
	PT3	<1s	4	15	2	108	<1s	3	0	0	-	-	4.1s	**0**	**0**
	PT4	<1s	4	18	2	242	<1s	18	1	7	8	3.4s	27.4s	**0**	**1**
`SBucketMap` (750)	SMT	<1s	4	123	19	892	<1s	2	2	-	2	<1s	1.3s	**1**	**0**
`Vector` (1.3K)	VT1	<1s	4	44	2	370	<1s	21	11	10	21	<1s	14.3s	**2**	**0**
	VT2	<1s	4	34	2	536	<1s	31	21	10	31	1.1s	33.0s	**1**	**0**
	VT3	<1s	4	34	2	443	<1s	32	22	10	32	<1s	22.1s	**1**	**0**
	VT4	<1s	4	29	2	517	<1s	30	0	30	30	2s	59.4s	**0**	**1***
	VT5	<1s	4	29	2	505	<1s	85	1	84	82	2s	2m57s	**0**	**1***
`Stack` (1.4K)	ST1	<1s	4	29	2	205	<1s	11	6	5	11	<1s	5.5s	**2**	**0**
	ST2	<1s	4	24	2	251	<1s	16	11	5	15	<1s	10.9s	**1**	**0**
	ST3	<1s	4	24	2	248	<1s	17	12	5	17	<1s	10.3s	**1**	**0**
	ST4	<1s	4	29	2	515	<1s	30	0	30	30	1.8s	53.2s	**0**	**1***
	ST5	<1s	4	29	2	509	<1s	85	1	84	83	2.0s	2m51s	**0**	**1***
`HashSet` (1.3K)	HT1	<1s	4	76	1	432	<1s	7	7	-	7	<1s	3.2s	**1**	**0**
	HT2	<1s	4	54	1	295	<1s	0	-	-	-	-	<1s	**0**	**0**
`StringBuffer` (1.4K)	SBT	<1s	3	16	3	80	<1s	2	2	-	2	<1s	1.3s	**1**$^+$	**0**
`Apache FtpServer` (22K)	LGN	1m2s	4	112	4	582	60s	116	78	32	65	1m13s	2h14m46s	**9**	**3**
`Hedc` (30K)	Std	1.7s	7	110	6	602	1.74s	18	9	1	10	11.7s	1m57s	**1**	**0**
`Weblech v.0.0.3` (35K)	Std	4.9s	3	153	3	1.6K	4.92s	55	10	29	30	16.26s	10m34s	**1**	**1**$^@$
											Total Number of Errors			**27**	**14**

Table 1. Experimental results for predicting null-reads. Errors tagged with * represent test harness failures. Errors tagged with $^+$ represent array-out-of-bound exceptions. Errors tagged with $^@$ represent unexpected behaviors. All other errors are null-pointer dereference exceptions.

these benchmarks in the literature. Furthermore, all the errors are completely reproducible deterministically using the scheduler.

We count exceptions raised in different parts of the code as separate errors. More precisely, each error reported in Table 1 consists of a unique read-write pair in the code that were forced to perform a null-read and that resulted in an error. For example, the 12 exceptions in `FtpServer` are raised in 7 different functions and at different locations inside the functions, and involve null-pointer dereferences on 5 different variables.

The prediction algorithm works extremely well— while there were several runs that were predicted in the precise model, the relaxed prediction gives a lot more predictions, and a large fraction of these were schedulable. The time taken for prediction and scheduling are very reasonable for the kind of targeted analysis that we perform, despite the use of fairly sophisticated static analysis and logic-solvers.

The number of data-races found using relaxed prediction further shows the efficacy of relaxed prediction. A further 17 data-races were found using relaxed prediction, showing that predicting beyond the maximal causal model can be effective even in finding errors other than null-pointer dereferences.

References

[1] http://www.cs.uiuc.edu/∼sorrent1/penelope/exceptionull.

[2] http://research.microsoft.com/en-us/projects/poirot.

[3] http://www.javagrande.org/.

[4] http://jakarta.apache.org/bcel.

[5] http://commons.apache.org.

[6] http://www.hedc.ethz.ch.

[7] http://mina.apache.org/ftpserver.

[8] http://weblech.sourceforge.net.

[9] F. Chen and G. Roşu. Parametric and sliced causality. In *CAV*, pages 240–253, 2007.

[10] F. Chen, T.F. Serbanuta, and G. Roşu. JPredictor: a predictive runtime analysis tool for java. In *ICSE*, pages 221–230, 2008.

[11] L. de Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.

[12] M. Emmi, S. Qadeer, and Z. Rakamaric. Delay-bounded scheduling. In *POPL*, pages 411–422, 2011.

[13] A. Farzan, P. Madhusudan, and F. Sorrentino. Meta-analysis for atomicity violations under nested locking. In *CAV*, pages 248–262, 2009.

[14] J. Huang and C. Zhang. Persuasive prediction of concurrency access anomalies. In *ISSTA*, pages 144–154, 2011.

[15] V. Kahlon, F. Ivancic, and A. Gupta. Reasoning about threads communicating via locks. In *CAV*, pages 505–518, 2005.

[16] S.K. Lahiri, S. Qadeer, and Z. Rakamarić. Static and precise detection of concurrency errors in systems code using smt solvers. In *CAV*, pages 509–524, 2009.

[17] Zhifeng Lai, S.C. Cheung, and W.K. Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *ICSE*, pages 235–244, 2010.

[18] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, pages 446–455, 2007.

[19] C-S Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *FSE*, pages 135–145, 2008.

[20] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *ASPLOS*, pages 25–36, 2009.

[21] Z. Rakamarić. STORM: static unit checking of concurrent programs. In *ICSE*, pages 519–520, 2010.

[22] M. Said, C. Wang, Z. Yang, and K. Sakallah. Generating data race witnesses by an smt-based analysis. In *Proceedings of the Third international conference on NASA Formal methods*, NFM'11, pages 313–327, Berlin, Heidelberg, 2011. Springer-Verlag.

[23] T.F. Şerbănuţă, F. Chen, and G. Roşu. Maximal causal models for sequentially consistent systems. Technical report, University of Illinois at Urbana-Champaign, October 2011.

[24] N. Sinha and C. Wang. Staged concurrent program analysis. In *FSE*, pages 47–56, 2010.

[25] N. Sinha and C. Wang. On interference abstractions. In *POPL*, pages 423–434, 2011.

[26] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan. Sound predictive race detection in polynomial time. In *POPL*, pages 387–400, 2012.

[27] F. Sorrentino, A. Farzan, and P. Madhusudan. PENELOPE: weaving threads to expose atomicity violations. In *FSE*, pages 37–46, 2010.

[28] C. von Praun and T. R. Gross. Object race detection. *SIGPLAN Not.*, 36(11):70–82, 2001.

[29] C. Wang, S. Kundu, M. Ganai, and A. Gupta. Symbolic predictive analysis for concurrent programs. In *Proceedings of the 2nd World Congress on Formal Methods*, FM '09, pages 256–272, Berlin, Heidelberg, 2009. Springer-Verlag.

[30] C. Wang, R. Limaye, M. Ganai, and A. Gupta. Trace-based symbolic analysis for atomicity violations. In *TACAS*, pages 328–342, 2010.

[31] L. Wang and S. D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *PPoPP*, pages 137–146, 2006.

[32] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Transactions on Software Engineering*, 32:93–110, 2006.

[33] J. Yi, C. Sadowski, and C. Flanagan. SideTrack: generalizing dynamic atomicity analysis. In *PADTAD*, pages 1–10, 2009.

[34] W. Zhang, J.Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. W. Reps. Conseq: detecting concurrency bugs through sequential errors. In *ASPLOS*, pages 251–264, 2011.

[35] W. Zhang, C. Sun, and S. Lu. Conmem: detecting severe concurrency bugs through an effect-oriented approach. In *ASPLOS*, pages 179–192, 2010.

QuickRec: Prototyping an Intel Architecture Extension for Record and Replay of Multithreaded Programs*

Gilles Pokam, Klaus Danne,
Cristiano Pereira, Rolf Kassa, Tim Kranich,
Shiliang Hu, Justin Gottschlich
Intel Corporation
{gilles.a.pokam, klaus.danne,
cristiano.l.pereira, rolf.kassa, tim.kranich,
shiliang.hu, justin.e.gottschlich}
@intel.com

Nima Honarmand, Nathan Dautenhahn,
Samuel T. King, Josep Torrellas
University of Illinois at Urbana-Champaign
{honarma1, dautenh1, kingst, torrella}
@illinois.edu

ABSTRACT

There has been significant interest in hardware-assisted deterministic Record and Replay (RnR) systems for multithreaded programs on multiprocessors. However, no proposal has implemented this technique in a hardware prototype with full operating system support. Such an implementation is needed to assess RnR practicality.

This paper presents *QuickRec*, the first multicore Intel Architecture (IA) prototype of RnR for multithreaded programs. QuickRec is based on *QuickIA*, an Intel emulation platform for rapid prototyping of new IA extensions. QuickRec is composed of a Xeon server platform with FPGA-emulated second-generation Pentium cores, and *Capo3*, a full software stack for managing the recording hardware from within a modified Linux kernel.

This paper's focus is understanding and evaluating the implementation issues of RnR on a real platform. Our effort leads to some lessons learned, as well as to some pointers for future research. We demonstrate that RnR can be implemented efficiently on a real multicore IA system. In particular, we show that the rate of memory log generation is insignificant, and that the recording hardware has negligible performance overhead. However, the software stack incurs an average recording overhead of nearly 13%, which must be reduced to enable *always-on* use of RnR.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors) - MIMD Processors; C.4 [**Performance of Systems**]: Design Studies; C.0 [**General**]: Hardware/software interfaces

Keywords

Deterministic Record and Replay, Shared Memory Multiprocessors, Hardware-Software Interface, FPGA Prototype.

*This work is supported in part by the Illinois-Intel Parallelism Center (I2PC).

1. INTRODUCTION

Deterministic Record and Replay (RnR) of multithreaded programs is an appealing mechanism for computer systems builders. RnR can recreate past states and events, by recording key information while a program runs, restoring to a previous checkpoint, and replaying the recorded log to force the software down the same execution path. With this mechanism, system designers can debug applications [1, 4, 6, 17, 32, 34, 41], withstand machine failures [5], and improve the security of their systems [15, 16].

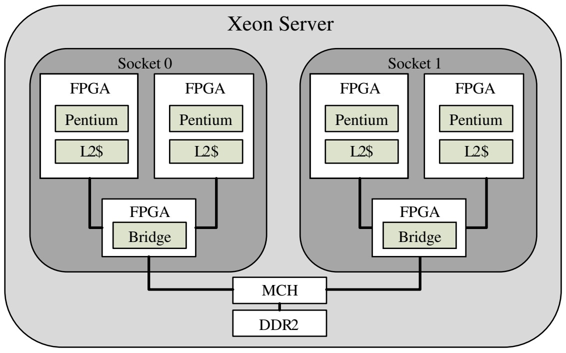
To replay a program, an RnR system must capture all sources of non-determinism. For multithreaded programs running on multicores, there are two key sources of non-determinism. The first is the inputs to the execution, such as effects and return values of system calls or occurrence of signals. The second is the order of the inter-thread communications, which manifests as the interleaving of the inter-thread data dependences through the memory system. While the first source of non-determinism can be captured in software with relatively low overhead, doing the same to record the second source typically imposes significant slowdowns.

To record memory access interleaving with low overhead, researchers have proposed several hardware assisted RnR designs (e.g., [3, 7, 12, 13, 23, 24, 25, 26, 30, 31, 36, 39, 40]). These proposals have outlined RnR systems that have negligible overhead during execution recording and can operate with very small log sizes. To evaluate these systems, the authors typically implement their techniques in software-based simulators. In addition, they typically run their simulations without an operating system that manages and virtualizes their special hardware. The exceptions are LReplay [7], which extends and simulates the RTL (Register Transfer Level) model of a chip multiprocessor and does not discuss system software issues, and Capo [24] and Cyrus [12], which use an RnR-aware operating system on top of simulated hardware.

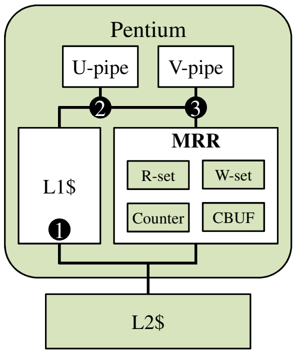
Although this evaluation approach helps assess the efficacy of the proposed algorithms, it ignores practical aspects of the design, such as its integration with realistic cache coherence hardware, coping with relaxed memory models, and virtualizing the recording hardware. In addition, promoting RnR solutions into mainstream processors requires a co-design with the system software that controls the hardware, and omitting software effects from the evaluation presents only part of the overall performance picture.

To evaluate the practical implementability of hardware-assisted RnR, we have built *QuickRec*, the first multicore IA-based prototype of RnR for multithreaded programs. QuickRec is based on *QuickIA* [37], an Intel emulation platform for rapid prototyping of

(a)



(b)



(c)

Figure 1: Photograph of the QuickRec prototype with FPGAs in CPU sockets (a); architecture of the QuickIA processor-emulation platform (b); and architecture overview of the extended Pentium core in QuickRec, where circled numbers identify the main CPU *touch points* required to enable recording (c).

new IA extensions. QuickRec is composed of a Xeon server platform with FPGA-emulated second-generation Pentium cores, and *Capo3*, a full software stack for managing the recording hardware from within a modified Linux kernel.

This paper focuses on identifying and characterizing RnR-related implementation issues. Specifically, we describe how QuickRec records the memory access interleaving of threads, and how to integrate this support into a commodity IA multicore. We discuss subtle issues related to capturing the ordering of instructions with multiple memory accesses, and the interaction with the memory consistency model. We also discuss how Capo3 records the inputs to processes, manages the replay logs, and virtualizes the hardware components. We provide data characterizing QuickRec's recording performance and log parameters. Overall, our evaluation demonstrates that RnR can be practical for real IA multicore systems.

This effort has led to some lessons learned, as well as to some pointers for future research directions. In particular, we find that the main challenge of RnR systems is to take into account the idiosyncrasies of the specific architecture used, such as single instructions producing multiple memory transactions. Further, we find that the software stack has a dominant role in the overall system performance, as it manages the logs. Based on these experiences, we suggest focusing future research on recording input events efficiently, and on replay techniques that are tolerant of the micro-architectural details of the system.

The main contributions of this work are the following:

• The implementation of the first IA multicore prototype of RnR for multithreaded programs. The prototype includes an FPGA design of a Pentium multicore and a Linux-based full software stack.

• A description of several key implementation aspects. Specifically, we show how to efficiently handle x86 instructions that produce multiple memory transactions, and describe the elaborate hardware-software interface required for a working system.

• An evaluation of the system. We show that the rate of memory log generation is insignificant, given today's bus and memory bandwidths. In addition, the recording hardware has negligible performance overhead. However, the software stack incurs an average recording overhead of nearly 13%, which must be reduced to enable always-on use of RnR.

This paper is organized as follows: Section 2 introduces the QuickRec recording hardware; Section 3 describes the Capo3 system software; Section 4 characterizes our prototype; Section 5 discusses using replay for validation; Section 6 outlines related work; Section 7 describes lessons learned; and Section 8 concludes.

2. QuickRec RECORDING SYSTEM

The QuickRec recording system prototyped in this work is built on a FPGA processor-emulation platform called QuickIA. This section introduces QuickIA and then describes the changes we added to support RnR. Figure 1a shows a picture of the QuickRec recording system testbed.

2.1 QuickIA Processor Emulation Platform

The QuickIA processor emulation platform [37] is a dual-socket Xeon server board in which Xeon CPUs are substituted with FPGA modules from XtreamData [38]. Each such FPGA module is composed of two Compute FPGAs and one Bridge FPGA, as shown in Figure 1b. Each Compute FPGA implements a second-generation Pentium core with private L1 and L2 caches. The Bridge FPGA implements the interconnect between the two Compute FPGAs and the Intel Front Side Bus (FSB), which connects the two CPU sockets to the Memory Controller Hub (MCH) on the platform. This allows both CPU sockets to be fully cache coherent, with full access to memory and I/O. The QuickIA system implements a MESI coherence protocol with L2 as the point of coherence.

The Pentium cores used in the QuickIA emulation platform are fully synthesizable. Each core features a dual-pipeline in-order CPU with floating-point support. In addition, each core is extended with a set of additional features to reflect the state of the art of modern processors. These changes include L1 cache line size increase to 64 bytes, Memory Type Range Registers, physical address extension, and FSB xAPICs.

The four emulated Pentium cores run at 60MHz. While this clock frequency is low, the memory bandwidth is also low (24MB/s), which means that the ratio between CPU speed and memory bandwidth is similar to that of today's systems. The QuickIA system includes 8GB of DDR2 memory and basic peripherals (network, graphics card and HDD), and can boot a vanilla SUSE Linux distribution. The basic platform parameters are shown in Table 1.

2.2 Recording Interleaving Non-Determinism

To record the non-determinism of memory access interleaving, the RTL of the synthesizable Pentium core is augmented to capture the order of memory accesses. This support includes mechanisms to break down a thread's execution into *chunks* (i.e., groups of consecutive dynamic instructions), and then order the chunks across cores. A significant effort was invested in integrating this support into the Pentium core without adding unnecessary complexity. Some of the main challenges we faced include dealing with the IA memory model, and coping with x86 instructions with multi-

Cores	4 Pentium cores
Clock	60MHz
L1 data cache	32KB, private, WB, 8-way assoc, 64B line size, 1-cycle latency
L2 cache	512KB, private, WB, 16-way assoc, 64B line size, 4-cycle latency
Coherence	MESI
Memory	8GB DDR2, 24MB/s bandwidth (measured by STREAM [22]), 90-cycle round-trip latency

Table 1: QuickIA platform parameters.

ple memory accesses. The extended Pentium core is then synthesized and downloaded into FPGAs to boot up the QuickRec emulation platform. A high-level overview of the extended Pentium core is shown in Figure 1c. In the figure, the *Memory Race Recorder (MRR)* box implements the functionality for recording memory access interleaving, while the circled numbers indicate the CPU *touch points* required to enable it.

2.2.1 Capturing and Ordering Chunks

The QuickRec recording system implements a mechanism similar to the Intel MRR [30] to divide a thread's execution into chunks. It adds Bloom filters next to the L1 cache to capture the read and write sets of the memory accesses in a chunk (*R-set* and *W-set* in Figure 1c). The line addresses of the locations accessed by loads and stores are inserted into their respective set at retirement and at global observation time, respectively. A thread's chunk is terminated when the hardware observes a memory conflict (i.e., a data dependence) with a remote thread. Conflicts are detected by checking the addresses of incoming snoops against addresses in the read and write sets. When a conflict is detected, a counter (*Counter* in Figure 1c) with the current chunk size is logged into an internal chunk buffer (*CBUF* in Figure 1c), along with a timestamp that provides a total order of chunks across cores. The chunk-size counter counts the number of retired instructions in the chunk. After a chunk is terminated, the read and write sets are cleared, and the chunk-size counter is reset.

In addition to terminating a chunk on a memory conflict, QuickRec can be configured to terminate a chunk when certain system events occur as well, such as an exception or a TLB invalidation. A chunk also terminates when the 20-bit chunk-size counter overflows. Additionally, the addresses of lines evicted from L2 are looked up in the read and write sets and, in case of a hit, the chunk also ends. This is done because the read and write sets would not observe future coherence activity on these evicted lines. Further information on chunk termination is provided in Section 2.3.

Figure 1c shows the main CPU touch points required to enable the chunking mechanism described above. The first CPU touch point is hooked-up to the external L1 snoop port to allow snoops to be forwarded to the MRR for address lookups. The second and third CPU touch points are hooked-up to the U and V integer execution pipelines of the Pentium core. They provide diverse functionalities, such as forwarding load and store line addresses to the MRR for insertion into the read and write sets, and forwarding the instruction retirement signal to the MRR to advance the chunk-size counter.

One of the complexities we encountered when integrating the chunking mechanism into the Pentium core was keeping updates to the read and write sets within one cycle, so that they can be performed in parallel with a cache access. The problem is that only the lower bits of the addresses are available at the beginning of a cache cycle, as the upper bits (tag bits) are provided usually late in the cycle, after a DTLB access. To preserve a single cycle for the read and write set update, addresses (tag plus set bits) are buffered into a latch stage before they are fed to the Bloom filter logic. To compensate for the delayed update of the read and write sets, these buffers are also looked-up on external snoops, at the cost of additional comparators for each address buffer.

2.2.2 Integration into the IA Memory Model

The IA memory model allows a load to retire before a prior store to a different address has committed, hence effectively ordering the load before the prior store in memory. This memory model is called Total Store Order (TSO). In this situation, using the retired instruction count is not sufficient to guarantee that loads and stores are ordered correctly during replay. This is because, during replay, instructions are executed in program order. Hence, regardless of when the store committed to memory during the recorded execution, the store is evaluated before the load during replay. To address this problem, QuickRec implements a solution similar to the one proposed in CoreRacer [31] to handle TSO. The idea is to track the number of pending stores in the store buffer awaiting commit and, at chunk termination, append the current number to the logged entry. This number is called the Reordered Store Window (RSW) count. The MRR is hooked-up to the memory execution unit to enable this functionality.

2.2.3 Instruction Atomicity Violation

In the x86 ISA, an instruction may perform multiple memory accesses before completing execution. For instance, a split cache line access, which is an access that crosses a cache line boundary, requires more than one load or store operation to complete. In addition, some complex instructions require several memory operations. For example, the increment instruction (INC) performs a load and a store operation. At the micro-architecture level, these instructions are usually broken down into multiple micro-operations or µops. An Instruction Atomicity Violation (IAV) occurs if an event causes the QuickRec recording system to log a chunk in CBUF in the middle of such an instruction execution. An example of such an event is a memory conflict. Because software is usually oblivious of split cache line accesses and µop execution, IAVs make it difficult for software to deterministically reproduce a program execution.

Figure 2 shows an example. Thread T0 executes instruction INC A, which increments the value in memory location A. The instruction breaks down into the three µops shown in the figure: a read from A into user-invisible register r_{tmp}, the increment of r_{tmp}, and the store of r_{tmp} into A. At the same time, thread T1 writes A. Suppose that the operations interleave as shown in the time line.

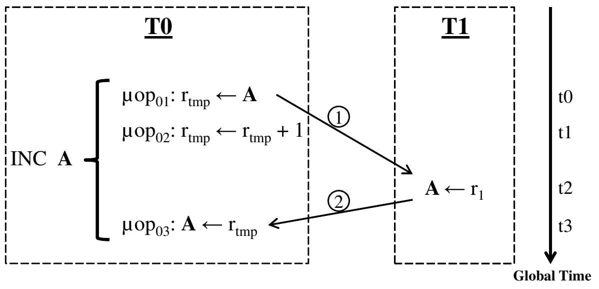


Figure 2: Instruction atomicity violation (IAV) example.

When the store in T1 executes at time t2, a conflict with T0 is detected, since μop_{01} has read from the same address at t0. Therefore, QuickRec terminates the chunk in T0 and logs an entry in T0's CBUF. This chunk is ordered before the store in T1. However, since the INC instruction has not yet retired, INC is not counted as belonging to the logged chunk. Then, when the INC instruction executes μop_{03} and retires at t3, a conflict with T1 is detected. This causes QuickRec to terminate the chunk in T1 and log an entry in T1's CBUF that contains the store. The logged chunk is ordered before the currently-executing chunk in T0, which is assumed to include the INC instruction. Consequently, in this naive design, the replay would be incorrect. Indeed, while during recording, μop_{01} occurred before the store in T1, which in turn occurred before μop_{03}, during replay, the store in T1 will be executed before the whole INC instruction.

This problem occurs because the INC instruction suffers an IAV. Although the instruction has performed some memory transactions during the earlier chunk in T0, since the instruction has not retired when the chunk in T0 is logged, the instruction is counted as belonging to the later chunk in T0.

The QuickRec recording system solves this problem by monitoring the retirement of the multiple memory accesses during the execution of the instruction. Specifically, it uses a dedicated IAV counter to count the number of retired memory transactions for a multi-line or multi-operation instruction (Figure 3). The IAV counter is incremented at every retired memory transaction, and is reset when the instruction retires. At chunk termination, if the IAV counter is not zero, the current instruction has not retired, and an IAV has been detected. In this case, QuickRec saves the value of the IAV counter in the log entry of the terminated chunk. Since, during replay, we know exactly the number (and sequence order) of the memory transactions that need to occur in a given instruction, by reading the IAV counter and examining the RSW count (Section 2.2.2), we know how many memory operations of the subsequent instruction need to be performed before completing the current chunk. In our actual implementation, the IAV counter is incremented by 1 for each access in a split cache line reference, and by 2 for any other access. With this design, an odd counter value indicates that the chunk terminated between the accesses of a split cache line reference.

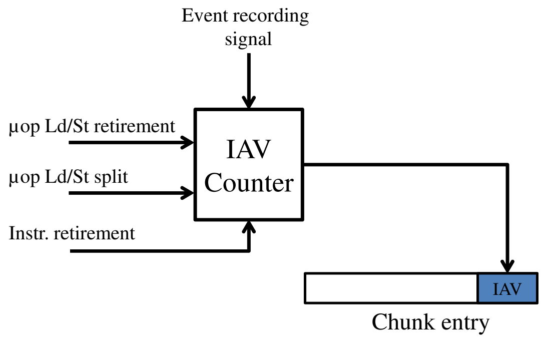


Figure 3: IAV counter mechanism.

Consider again the example of Figure 2. When T1 executes the store at time t2 and a conflict is detected in T0, the INC instruction has not yet retired. The IAV counter in T0 is 2, since the only retired access is that of μop_{01}. Therefore, an IAV is detected. The QuickRec recording system terminates the chunk in T0 and, as it logs the chunk, appends to it the value of the IAV counter. This log entry conveys to the replayer the information that an IAV has occurred in the chunk and that only the first memory μop had retired at the time of chunk termination.

Instruction atomicity violation was first introduced in [29] and then described in [31]. The main difference with [31] is that we log the number of retired memory transactions instead of the number of transferred bytes. The advantage of logging memory transactions over transferred bytes is the reduction in the log size.

2.2.4 *Log Management*

CBUF is organized into four entries, where each is as large as a cache line. When a chunk terminates, a 128-bit chunk packet is stored in CBUF. When a CBUF entry is full, it is flushed by hardware to a dedicated memory region called CMEM. To minimize the performance impact, this is done lazily, during idle cycles, by bypassing the caches and writing directly to memory. Occasionally, however, the chunking mechanism must stall the execution pipeline to allow CBUF to drain to CMEM to avoid overflow.

There are two main packet types inserted into CBUF, namely the timestamp packet (TSA) and the chunk packet. Both are very conservatively sized as 128-bit long. Once a TSA is logged for a thread, subsequent chunk packets for that thread only need to log the timestamp difference (TSD) with respect to the last TSA. The TSA is then logged again when the value in TSD overflows. Note that this also causes a chunk termination. Figure 4 shows the format of these two packets. The chunk packet contains the TSD, chunk size (CS), and RSW and IAV counts. It also contains a *Reason* field, which indicates why the chunk was terminated — e.g., due to a RAW, WAR or WAW conflict, an exception, or a chunk-size overflow. Table 2 lists the main reasons for terminating chunks.

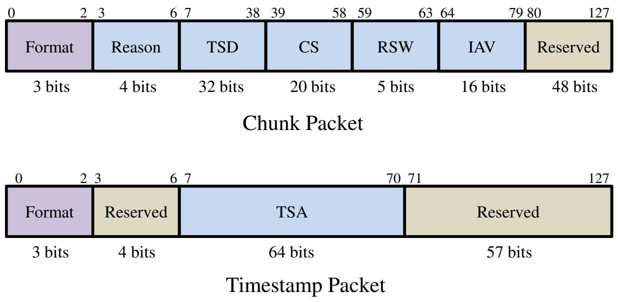


Figure 4: Packet formats in QuickRec.

Type	Reason
RAW	RAW conflict between chunks
WAR	WAR conflict between chunks
WAW	WAW conflict between chunks
WAB	Both WAR and WAW conflicts between chunks
EXCEPT	Exception, interrupt, far call, or far return
EVICT	Line eviction from L2 that hits the R-set or W-set
CS_OVERFLOW	Chunk size overflow
TLBINV	TLB invalidation
XTC	Explicit chunk termination instruction

Table 2: Main reasons for terminating chunks. WAB (Write-After-Both) is when a write in one chunk hits in both the read and the write set of another chunk.

2.3 Programming Interface

The QuickRec recording system contains a set of registers to configure and program the hardware. For instance, using these registers, the hardware can be programmed to record memory non-determinism for user-level code only, or for both user- and system-level code. It can also be programmed to terminate a chunk under certain conditions only, such as a specific type of conflict or exception. Privileged software can also specify where in memory the logs are written for each recorded thread. The QuickRec recording system also has a status register that is updated at chunk termination time to capture the state of the machine at that point. Among other information, it captures the reason for the chunk termination. Some of its information is copied to the Reason field of the logged chunk packet. A more detailed discussion of the programming interface, and how the system software uses it to manage the QuickRec hardware is provided in Section 3.3.

QuickRec extends the ISA with two new instructions: one that terminates the current chunk (*XTC*), and one that terminates the current chunk and flushes CBUF to memory (*XFC*). The use of these two instructions is restricted to privileged software. Examples of their use are discussed in Sections 3.4 and 3.6.

2.4 Other Issues

Because the main purpose of this work is to demonstrate the feasibility of hardware-assisted RnR, this prototype only addresses the issues that are critical to support RnR for the majority of programs. For instance, the prototype only supports Write-Back (WB) memory [14], which constitutes the majority of memory accesses in current programs. Memory accesses to Uncacheable (UC) or Write-Combining (WC) memory are not tracked, and cause the system to terminate a chunk. Chunking is resumed when the next access to WB memory occurs.

In some cases, the IA memory model allows accesses to WB memory to have different ordering semantics than TSO. For instance, in fast string operations, a store to WB memory can be reordered with respect to a prior store. To ensure that QuickRec's RSW and IAV support work properly, we disable this feature, so that all loads and stores obey TSO semantics.

Although we do not discuss how to extend our mechanisms to support Hyperthreading, the changes required to do so are minimal. In modern IA cores, there already exist mechanisms for detecting conflicts between the different hardware thread contexts sharing the same cache. Therefore, in order to enable RnR on a Hyperthreaded core, one would only need to replicate certain resources for each hardware thread context (e.g., the read and write sets).

3. Capo3 SYSTEM SOFTWARE

To manage the QuickRec hardware, we built a software system called *Capo3*. Capo3 draws inspiration and borrows many of the concepts and principles from Capo [24], a system designed for hardware-assisted RnR. However, Capo3 must run on real hardware, and as such, we encounter several issues that were abstracted away in Capo due to using simulated hardware. In this section, we compare Capo3 with Capo, describe its architecture, and focus on several of its key aspects.

3.1 Comparing Capo3 with Capo

Capo3 uses some of the basic ideas introduced by Capo, including the *Replay Sphere* and the *Replay Sphere Manager* (RSM). The Replay Sphere abstraction is the single application (or a group of applications) that should be recorded/replayed in isolation from the rest of the system. The Replay Sphere Manager is a software component that is responsible for correctly capturing non-deterministic input and memory access interleaving.

Capo3 also uses the same basic techniques as Capo to record program inputs, including interactions between the operating system and processes (e.g., system calls and signals), and non-deterministic instructions (i.e., *rdtsc* and *cpuid*). Recording these input events guarantees that, during replay, the same data can be injected into the user-mode address space. However, some system calls also affect the kernel-mode data structures of the program. Hence, to ensure that their effects are deterministically recreated during replay, we re-execute these system calls during replay.

To correctly capture kernel state, like in Capo, the RSM enforces a *total order* of input events during recording. The same total order is enforced during replay. This total order has major performance and correctness implications, as shown in Sections 3.6 and 4.

Capo3 uses a different software architecture than Capo. Specifically, it places the bulk of the RnR logic in the kernel — whereas Capo used *ptrace* to capture key events with user-mode logic. Moreover, since Capo3 must virtualize real hardware, its design must support a hardware/software interface to enable context switches, record memory access interleaving when the kernel is running with interrupts enabled, and manage subtle interactions between QuickRec hardware and Capo3 software.

3.2 Capo3 Architecture

Capo3 implements the RSM as an extension to the Linux kernel. To record an execution, a driver program initializes a Replay Sphere using the RSM-provided interface. The RSM then logs the input events, sets-up the MRR hardware to log the memory access interleaving, and makes all these logs available to the driver program that is responsible for the persistent storage and management of the logs. Figure 5 shows the high-level architecture of the Capo3 software stack.

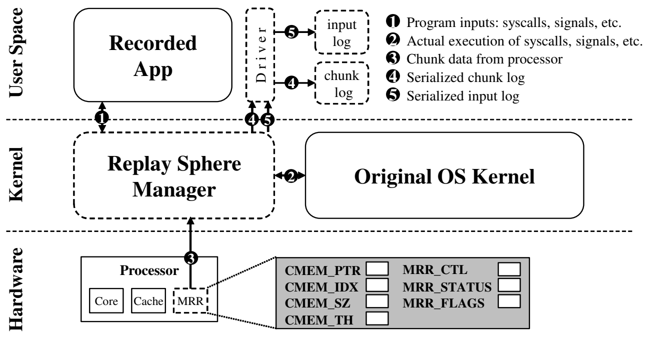


Figure 5: Overall architecture of Capo3. Dashed boxes indicate QuickRec-specific components.

Our decision to use a kernel-based implementation was driven by the observation that the Linux kernel has well-defined places to enable the kernel to interpose on processes. As a result, Capo3 only requires the kernel to be augmented in a few key places, so it can interpose on all system calls, signals, and memory copies between processes and the kernel. These changes also allow Capo3 to virtualize the QuickRec hardware by saving/restoring QuickRec state upon a context switch. Overall, our kernel-based implementation consists of roughly 3.4K lines of code, where the bulk of the code is dedicated to managing the logs, and is well isolated from the rest of the kernel.

There are four different sources of input non-determinism that the RSM captures: system calls, data copied to user-mode address spaces, signals, and non-deterministic processor instructions. To bind these recorded events to their corresponding threads, the RSM assigns a unique R-Thread ID to each recorded thread. During replay, each thread is guaranteed to get the same R-Thread ID. These R-Thread IDs are also used to associate chunks recorded by the QuickRec hardware with their corresponding threads.

3.3 Virtualizing the QuickRec Hardware

To virtualize the QuickRec hardware, the RSM uses the programming interface outlined in Section 2.3. The main components of this interface are the seven registers shown in the lower level of Figure 5. Specifically, the Chunk Memory Pointer (*CMEM_PTR*) points to CMEM, which is the in-memory buffer that contains the logged chunk data. Each thread gets its own CMEM. The Chunk Memory Index (*CMEM_IDX*) indicates the location in CMEM where the next CBUF entry is to be written. This register is updated by hardware as CBUF entries are written to memory. The Size Register (*CMEM_SZ*) indicates the size of CMEM. The Threshold Register (*CMEM_TH*) indicates the threshold at which a CMEM overflow interrupt is generated. The Control Register (*MRR_CTL*) enables and disables chunking under certain conditions, while the Status Register (*MRR_STATUS*) provides the status of the hardware. These last two registers were described in Section 2.3. Finally, the Flags Register (*MRR_FLAGS*) controls kernel-mode recording and is discussed later.

It is the RSM's responsibility to manage the CMEM buffers and virtualize these hardware registers so that different threads can use the hardware without having their chunk data mixed-up. In particular, this involves: (i) ensuring that a valid CMEM pointer is configured before recording begins, (ii) allocating a new CMEM buffer when the previous one fills-up, and (iii) writing to CMEM any contents remaining in the CBUF before a thread is pre-empted.

When a CMEM buffer reaches its capacity, Capo3 writes it to a file. Because there may be multiple full CMEM buffers in the system waiting to be written to the file, the RSM serializes this write operation using a work queue handled by a dedicated thread. This work queue provides an effective back-pressure mechanism when the buffer completion rate of the recorded threads exceeds the speed of the thread that empties the queue. Specifically, when the work queue becomes full, the RSM puts the recorded threads to sleep until the work queue can catch up. This mechanism preserves correctness, although it may negatively impact recording performance.

3.4 Handling Context Switches

On a context switch, the RSM first executes an XFC instruction to ensure that the current chunk terminates, and that all the residual data in the processor's CBUF are flushed to CMEM. This is needed to avoid mixing the log of the current thread with the next thread.

Once this has been performed, the RSM saves and restores the values of the registers in the MRR. Specifically, for the current thread, it saves the registers that the hardware may have modified during execution. They are the CMEM_IDX and MRR_FLAGS registers. Then, before the next thread can execute, the RSM restores the thread's prior CMEM_PTR, CMEM_IDX, CMEM_SZ, CMEM_TH, MRR_CTL, and MRR_FLAGS values, enabling it to correctly resume execution.

3.5 Recording in Kernel Mode

Certain parts of the kernel can interact with a process' address space, creating the potential for the kernel to have races with user-level instructions. The *copy_to_user* family of functions in the Linux kernel is an example of such code. Hence, in order to record all the memory access orderings that can affect the execution of an application during replay, the QuickRec hardware must also capture the execution of these kernel-level memory accesses.

QuickRec provides a flag that, if set, allows the MRR to record kernel instructions as well as user-mode instructions. Hence, to record sections of the kernel such as *copy_to_user()*, our initial approach was to set that flag prior to entering *copy_to_user()* and reset it after returning from *copy_to_user()*. The problem with this approach is that an asynchronous interrupt (e.g., from a hardware device) or a page fault can occur during the execution of *copy_to_user()*. In this case, since the flag is still set, QuickRec would incorrectly record the interrupt or page fault handler code.

Our solution to this problem is to have an MRR_FLAGS register, where the least significant bit (LSB) acts as the previously-mentioned flag. On entry to *copy_to_user()*, we set the LSB, while on returning from it, we reset it. Moreover, the register operates as a shift register. When an exception is taken, the register automatically shifts left with a 0 being inserted into the LSB, which disables recording. Upon returning from the exception handler (as indicated by the *iret* instruction of x86), the register shifts right, restoring the previous value of the LSB. If the exception has happened in the middle of a *copy_to_user()*, this design disables recording as soon as the exception is taken, and resumes it as soon as the execution returns to *copy_to_user()*.

3.6 Handling Input/Chunking Interactions

The RSM component that records the input log and the one that manages the chunking log proceed almost independently from each other, each creating a total order of their events. However, in our initial implementation, we observed a subtle interaction between the two components that resulted in occasional deadlocks.

The problem occurs if a chunk includes instructions from both before and after and input event. In this case, the dependences between chunks and between inputs may intertwine in a way that causes deadlock.

As an example, consider Figure 6a, where chunks C1 and C2 execute on processors P1 and P2. Suppose that C2 first executes an input event that gets ordered in the input log before an input event in C1. Then, due to a data dependence from P1 to P2, C1 is ordered in the chunking log before C2. We have recorded a cyclic dependence, which makes the resulting logs impossible to replay and, therefore, causes deadlock.

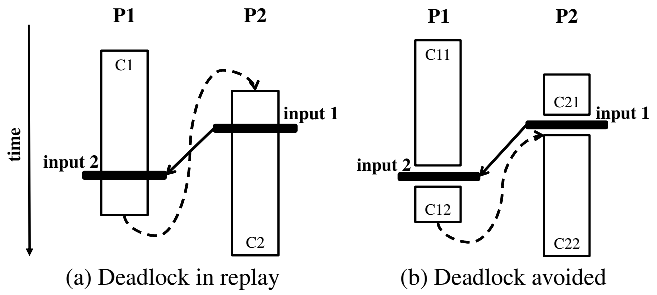


Figure 6: Examples of dependences between input events (solid lines) and between chunks (dashed lines).

To avoid this problem, Capo3 does not let a chunk include instructions from both before and after an input event. Instead, before an input event is recorded, the RSM executes the XTC instruction — therefore terminating the current chunk. With this approach, the situation in Figure 6a transforms into the one in Figure 6b. In this case, there are four chunks and the cyclic dependence has been eliminated. Both input and chunk dependences are satisfied if we replay the chunks in the C11, C21, C12 and C22 order.

Another issue related to the interaction between the two logs is how the replayer can match the input log entries and the chunk log entries generated by the same thread. Fortunately, this is easy, since the RSM assigns a unique R-Thread ID to each thread (Section 3.2). As the logs are generated, they are augmented with the R-Thread ID of the currently-running thread. In particular, as the RSM writes the CMEM buffers to the log, it attaches the current R-Thread ID to the buffer's data.

4. PROTOTYPE CHARACTERIZATION

4.1 Experimental Setup

We evaluate the QuickRec system by collecting and analyzing both log data and performance measurements for a set of SPLASH-2 benchmarks (Table 3). We execute each benchmark to completion, and show results for a default configuration of 4 threads running on 4 cores. In addition, we also assess the scalability of QuickRec by analyzing runs with 1, 2, 4, and 8 threads. For our experiments, we pin each application thread to a particular core. Thus, in the default case, we assign each thread to its own core and, in the 8-threaded case, we assign two threads to each core. We implement Capo3 as a kernel module in Linux 3.0.8.

Benchmark	Input Size	# of Instruc. (B)
Barnes	nbody 8000	3.4
FFT	-m 22	3.7
FMM	-m 30000	5.3
LU	-n 1024	3.0
LU-NC	-n 1200	4.7
Ocean	-n 1026	7.5
Ocean-NC	-e1e-16	2.2
Radix	-n 10000000	2.3
Raytrace	teapot.env	0.3
Water	1000 molecules	5.4

Table 3: Characteristics of the benchmarks. The last column shows the total number of instructions executed in the 4-threaded run in billions. *Water* refers to Water-nsquare.

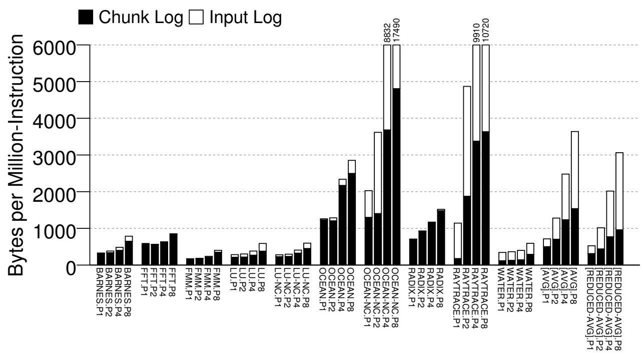
4.2 Log Analysis

In this section, we analyze the size and bandwidth requirements of the logs generated during the recorded execution. In addition, for the chunk log, we perform a detailed characterization. In all cases, we consider logs without data compression.

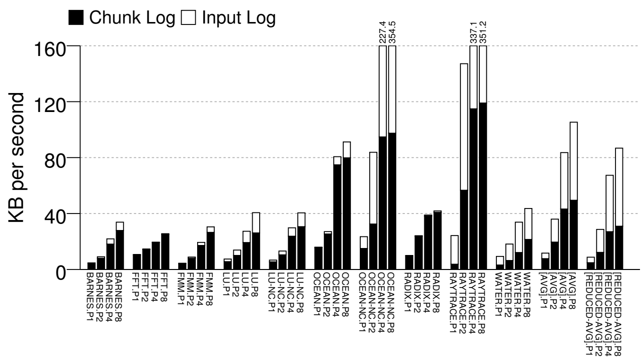
4.2.1 Log Sizes and Bandwidth

Figure 7a shows the *uncompressed* size of the input and chunk logs for each of the benchmarks and for the average case (*AVG*). For each benchmark, we show data for 1-, 2-, 4-, and 8-threaded runs. The size is given in bytes per million instructions. From the bars, we see that the average log size produced by QuickRec for 4 threads is 1,224 and 1,235 bytes per million instructions for input logs and for chunk logs, respectively. These are small numbers. However, the *Ocean-NC* and *Raytrace* benchmarks generate notably larger logs for 4-8 threads. This effect is mainly due to the increased use of synchronization in the benchmarks, which involves frequent calls to the *futex()* system call. As a result, the input log size increases substantially. Also, since Capo3 terminates the running chunk before recording an input event (Section 3.6), the chunk log also grows substantially.

The average log sizes that we measure are in line with sizes reported in previous work. For example, the log sizes reported for



(a) Uncompressed log sizes



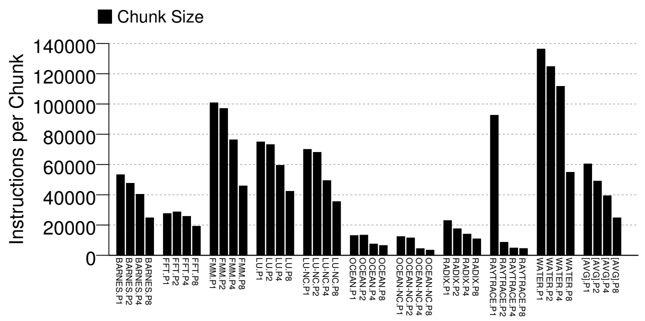
(b) Memory bandwidth requirements

Figure 7: Analyzing the log sizes without data compression and the resulting memory bandwidth requirements.

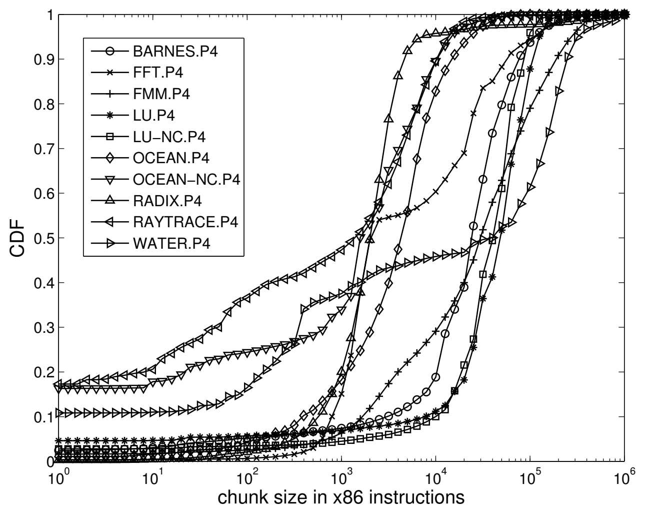
Cyrus [12], DeLorean [23], Rerun [13], and LReplay [7] are all within approximately 0.5x–2x of ours. We also note that our numbers correspond to a simple, unoptimized RnR implementation, and can easily be improved. As a simple example, consider the log entry for a chunk in QuickRec (Figure 4). Of the 128 bits, in most cases, only 80 bits are used for RnR. The remaining bits are mostly used for debugging and characterization of the hardware. If we eliminated them, we would get the average log sizes labeled REDUCED-AVG in Figure 7a. Further log size reductions can be attained with improved bit encoding.

Figure 7b shows the memory bandwidth requirements of logging. The figure is organized as the previous one and shows bandwidth in KB per second. From the average bars, we see that the bandwidth for 4 threads is 40 KB/s and 43 KB/s for input and chunk logs, respectively. These numbers, when combined, represent only 0.3% of the 24 MB/s bandwidth available in our prototype (Table 1). Hence, the effect of logging on bus and memory contention is very small. If we use the 80-bit chunk entries for the log (bars labeled REDUCED-AVG in Figure 7b), the bandwidth requirements are slightly lower.

To reason about the bandwidth requirements of QuickRec's logging on modern computers, consider the following. A modern multicore computer cycles at a higher frequency than our prototype, but it also has higher memory bandwidth. To understand the impact of these changes, we recompiled and ran our benchmarks on a dual socket Xeon server with 2.6 GHz E5-2670 processors. We measured the elapsed time (and speedup over our prototype) of the 4-threaded applications and scale the bandwidth numbers accordingly. Assuming the 80-bit log entry per chunk, we obtained an average bandwidth consumption across the benchmarks of 17.9 MB/s (and 61.1 MB/s for *Ocean-NC*, which is bandwidth-intensive). Given that the E5-2670 processor provides a memory



(a) Average chunk size



(b) Cumulative distribution of chunk size

Figure 8: Chunk size characterization.

bandwidth of up to 6.4 GB/s per core, the logging accounts for only 0.07% on average (and 0.23% in *Ocean-NC*) of the available bandwidth of 4 cores. Based on these estimates, we conclude that the bandwidth usage is negligible and will not have a negative impact on the performance of real systems.

If we compress the logs using gzip's default DEFLATE algorithm, we attain an average compression ratio of 55% for chunk logs and 88% for input logs. Hence, the average 4-threaded benchmark can be recorded for almost three days before filling up a terabyte disk.

Finally, Figure 7a and Figure 7b also suggest that both the log sizes and the bandwidth requirements scale reasonably as the number of threads increases from 1 to 8.

4.2.2 Chunk Characterization

Figure 8a shows the average size of the chunks in terms of retired x86 instructions. Figure 8b shows the distribution of chunk sizes for 4-threaded runs. On average, the size of a chunk for 4-threaded runs is $39K$. However, Figure 8b shows that, while many chunks are large (e.g., more than 80% of the chunks in *Barnes*, *LU*, and *LU-NC* are larger than 10,000), there are many chunks with fewer than 1,000 instructions. For three benchmarks, there is a significant fraction of zero-sized chunks, which mostly result from explicitly terminating a chunk unconditionally at input events. This effect can be avoided by changing Capo3 or the hardware.

Figure 9 details the chunk termination reasons, using the categories shown in Table 2, except that exceptions, chunk-size overflows, and TLB invalidations are grouped together in *Other*. From the figure, we see that the largest contributor to chunk termination is cache line evictions. In the QuickRec hardware, a chunk must be terminated if a line that is evicted from the L2 hits the read set or the write set in the same core. This is because subsequent snoop requests to that line are not delivered to the MRR; they are filtered out by the L2. Techniques to mitigate this behavior will contribute to reducing the number of chunks.

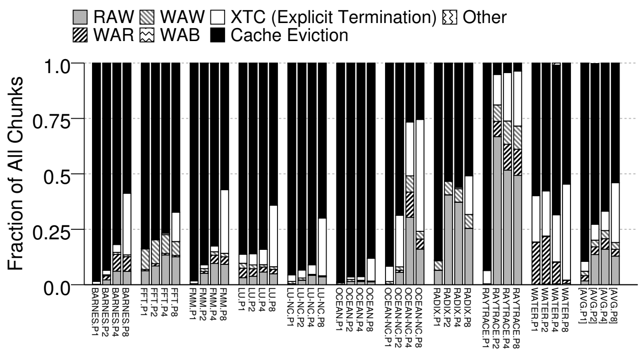


Figure 9: Chunk termination reasons.

Conflicts due to WAR, RAW, WAW and WAB are the second most prevalent reason of chunk terminations. Another frequent reason is explicit chunk termination with XTC. This termination reason is common when we have more threads than processors (i.e., in the 8-threaded runs). In this case, there are many context switches which use XTC. This reason is also common if the benchmark has numerous input events, such as signals or system calls, which require explicit use of XTC to obtain a total order of events. For example, this is the case for *Raytrace* and *Ocean-NC*, which, as shown in Figure 8b, have a large number of zero-sized chunks.

To deal with instruction reordering and instruction atomicity violations, QuickRec appends RSW and IAV information to chunk entries. Figure 10 displays the fraction of chunks that are associated to non-zero RSW and/or IAV values. The figure reveals that such chunks are common. For 4-threaded runs, an average of 16% of the chunks are RSW or IAV chunks. In fact, both RSW-only and IAV-only chunks are common. One interesting case is that of *Radix*, where the fraction of IAV chunks is over 40%. The reason is that *Radix* has a long-running tight loop with several multi-memory-operation instructions. *FFT* has many RSW-only chunks, which result from executions where loads and stores are interleaved. Overall, RnR systems must be designed to handle these cases.

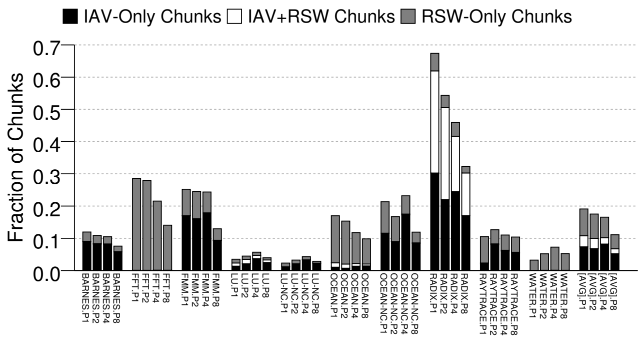


Figure 10: RSW and IAV characterization.

4.3 Performance Measurements

To measure the overhead of QuickRec's different components, we ran each benchmark in five different configurations. First, *na-*

tive is the normal execution with no recording. Second, in *hw-only*, the MRR hardware is enabled and writes chunk data to main memory, but otherwise no other component of the system is enabled. This configuration measures the overhead of the extra memory traffic generated by the MRR. Third, in *input*, the RSM only logs the sources of input non-determinism described in Section 3.2 and the MRR is disabled. Fourth, *chunk* augments the *hw-only* configuration by having the RSM dump the CMEM buffers to a file; no input is recorded. Finally, *combined* is a full recording run where both input and chunk data are processed by the RSM. To reduce the OS-induced noise, each configuration is run five times and the results are averaged. Each run executes with four threads.

Figure 11 shows the execution time of each configuration normalized to the execution time of *native*. The figure shows that, in most benchmarks, recording both input and chunk logs only incurs a 2–4% overhead. The main exceptions are *Ocean-NC* and *Raytrace*, which suffer an overhead close to 50%. As indicated in Figure 7a, these two benchmarks perform substantial synchronization, which involves frequent calls to the *futex()* system call and, often, results in putting threads to sleep. On average across all of the benchmarks, the recording overhead is 13%.

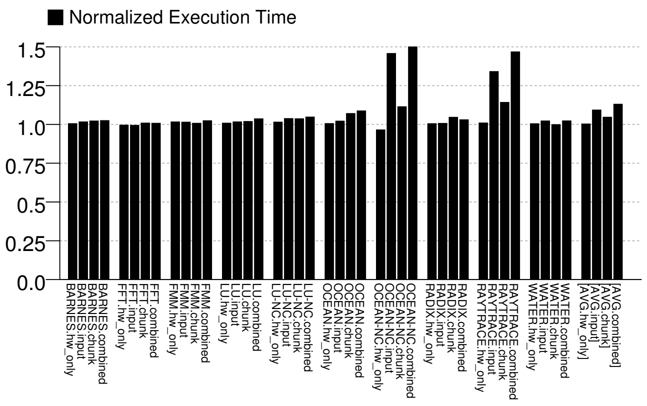


Figure 11: Execution time with each recording configuration for four-threaded executions. The bars are normalized to the execution time of *native*.

Interestingly, the recording overhead is entirely due to the software stack. Indeed, the hardware overhead, as shown in *hw-only*, is negligible. We also see that the software overhead is primarily due to input logging, rather than chunk logging. Overall, future work should focus on optimizing the software stack and, in particular, input logging — specifically, removing the serialization in the recording of input events.

Figure 12 shows the processor time (the time processors spend doing useful work for the applications) separated into user and system time. For each benchmark, we show three bars: one for the recorded application itself (*App*), one for the driver that reads the input log from memory and writes it to disk (*Input*), and one for the driver that reads the chunking log from the memory and writes it to disk (*Chunking*). For each benchmark, the bars are normalized to the processor time of the application.

The figure shows that most of the processor time is spent running the application. On average, the drivers add little overhead. Only the two benchmarks with large logs in Figure 7a spend noticeable time in the drivers. Finally, most of processor time in these applications is user time.

To understand the sources of overhead in QuickRec, Figure 13 breaks down the total processor cycles into four categories. First, *App time* are the cycles spent executing instructions not resulting from Capo3 overhead. Second, *Input overhead (working)* are the

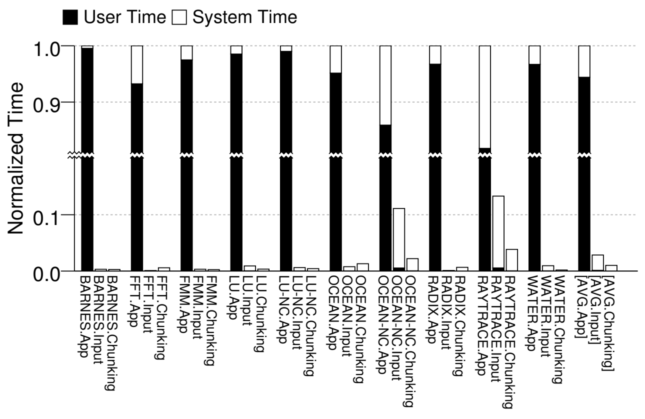


Figure 12: Total time that the processors spend working on the applications divided into user and system time.

cycles spent in Capo3 code managing the input events. Third, *Input overhead (sleeping)* are the cycles spent in Capo3 waiting on synchronization in order to enforce a total order of input events. Finally, *Chunking overhead* are the cycles spent in Capo3 code managing the chunking log. The figure shows the breakdown for different thread counts. As the figure indicates, for 4- and 8-threaded runs, the main overhead of Capo3 is due to enforcing a total order of input events. We are looking into optimizations and/or alternative designs for this component.

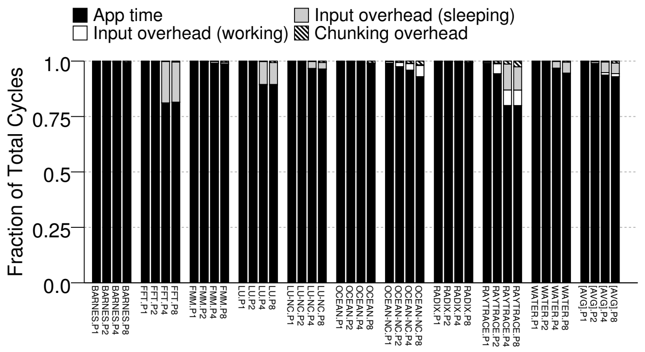


Figure 13: Breakdown of the total processor cycles for different thread counts.

Figures 14 and 15 present detailed breakdowns of the input and chunking overheads, respectively, for different thread counts. In each figure, the overheads are normalized to the overhead of the 1-threaded execution for the given benchmark.

Figure 14 divides the overhead of input recording and management into the contributions of system calls, copy to user (CTU), and other events. In each case, the figure separates working and sleeping overheads. The figure shows that the sleeping overhead resulting from serializing the system calls is by far the largest component for 4- and 8-threaded runs. In particular, *FFT*'s normalized overhead for 4- and 8-threaded runs is high. The reason is that *FFT* has minimal overhead with 1 thread and has many synchronization-induced *futex()* calls with 4 or more threads.

Figure 15 depicts a similar breakdown for the chunk-management overhead. The overhead is divided into execution of XTC instructions (*Chunk term*), execution of XFC instructions (*CBUF flush*), allocation of a new CMEM buffer (*Buffer allocation*), putting a CMEM buffer in the work queue (*To workqueue*) and *Other*. The latter is dominated by the overhead of saving and restoring MRR registers in a context switch. We see that *Buffer allocation* and *Other* dominate.

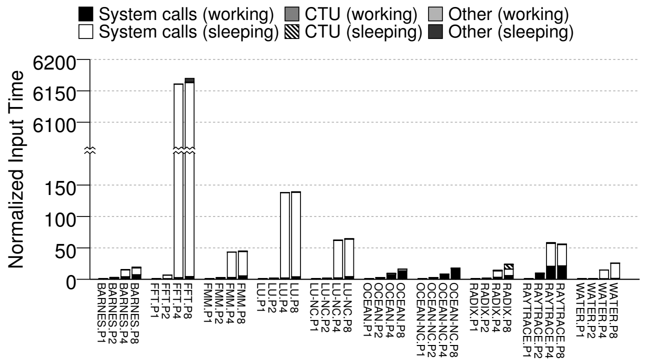


Figure 14: Breakdown of the normalized overhead of input recording and management. CTU stands for Copy To User.

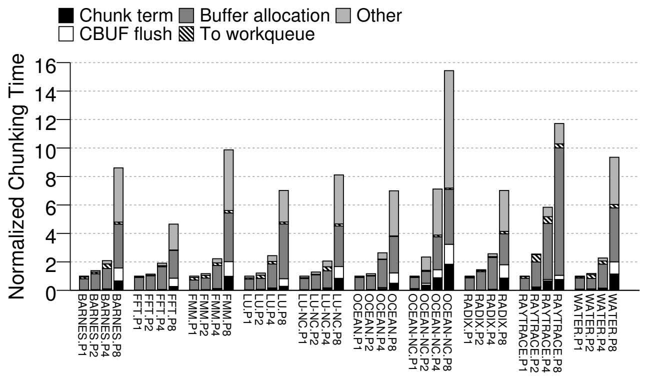


Figure 15: Breakdown of the normalized chunk-management overhead.

5. VALIDATION USING REPLAY

A critical aspect of the design and implementation of a recording system is to validate it with replay. Replaying recorded logs enables full assurance that the recording system captures the correct and complete information. Therefore, in this section we discuss the replayer from the perspective of its validation of QuickRec.

We implemented the replayer using the Pin [20] binary instrumentation framework. We chose this approach for three reasons. First, user-level binary instrumentation is operating-system independent (similar to PinPlay [28]), which enables replay to occur on a machine that is independent from the QuickRec system. Second, Pin operates at speeds faster than existing instruction-set simulators, while maintaining an acceptable level of observability. Third, using Pin, we can extend the replayer by integrating other analysis tools, such as race detectors [2, 33] and debuggers [19].

5.1 High-Level Implementation Description

To correctly replay a recorded execution, the replayer requires the executed code (binary and libraries, including self-modified code), and the program inputs and shared-memory access interleaving experienced during the recorded execution. Prior to replay, the static code is extracted from the log files. Self-modified code, which is not present in the log files, is re-generated by the replayed execution. Non-deterministic inputs are made deterministic by injecting the appropriate recorded data into the replayed execution at appropriate execution points. For most system calls (e.g., *read()*), this operation involves emulating the system call, by: (i) injecting the logged data into the program if there is a logged *copy_to_user()* entry, and (ii) setting the return values as defined in the input log. However, there are a few system calls, such as thread creation and termination, that are re-executed to recreate the proper kernel state.

Chunk ordering is accomplished by counting instructions as they are replayed, and stopping when the counter reaches the logged chunk size. In addition, the replayer enforces the logged chunk order, based on the recorded timestamps.

5.1.1 Chunks with Non-Zero RSW or IAV Counts

To handle the IA memory model correctly, the replayer needs to take into account the values of the RSW and IAV counts. Specifically, to support TSO, the replayer simulates a thread-local store buffer. On a store operation, the replayer writes the address and value of the store to the local store buffer — instead of committing the store to the global memory. On a load operation, the replayer first checks the local store buffer. If the address is not found, it loads the value from the global memory. Then, at the end of the chunk, the replayer drains the stores from the local store buffer, except for a number equal to the RSW count of the chunk, and commits their values to the global memory. The stores remaining in the local store buffer are committed as part of the next chunk.

To handle non-zero IAV counts, the replayer needs to know the number of memory transactions involved in the execution of each instruction. When the replayer finds a chunk whose IAV is nonzero, after executing the chunk, it emulates the execution of the memory transactions of the first instruction after the chunk, one at a time. The replayer stops when the number of memory transactions is equal to the IAV count. The remaining memory transactions of the instruction are emulated at the beginning of the next chunk.

5.2 Validating the Complete System

Prior to full-system tests, we developed multiple levels of system validation. We began with RTL simulations to validate the MRR hardware without software, while we used Simics [21] simulations to validate Capo3. Next, we integrated Capo3 with QuickRec and developed tests to independently exercise the recording functionalities of input non-determinism and shared-memory interleaving. Last, we tested the complete system with our benchmarks.

When bugs were found during full-system tests, the major challenge was pinpointing their origin. In QuickRec, bugs can originate from either the replayer, the recording hardware, or the recording software; distinguishing between the three is usually non-trivial. In our experiments, the most common type of bug manifestation was a divergence between the memory state or the control flow of the recorded and replayed executions. There are many reasons why a divergence can occur, and being able to pinpoint the root cause of such a divergence is critical.

The most obvious location to check for divergent executions is where non-deterministic input events are logged. This is because, during recording, Capo3 saves the contents of the processor registers at the entry of system calls. Hence, the replayer can compare the state of the processor registers before a system call to the recorded state. This provides a clear detection point of divergence. Moreover, a system call should result in a chunk termination and, therefore, should be the last instruction of the chunk it belongs to. This provides another divergence check.

Unfortunately, non-deterministic input events are infrequent and, therefore, insufficient to detect the root cause of most divergences — the source of divergence can be thousands of instructions before the system call. Therefore, a more fine-grained mechanism to detect divergences was needed.

For this purpose, we added a branch-tracing module in the FPGA hardware. It collects the history of branches executed — like the Branch Trace Store of today's IA processors. With this information, the replayer can compare the control flow of the recorded execution with that of the replayed execution. This is a powerful method to detect divergences, since if either the record or replay system has a bug, then the replayed execution typically leads to a different control flow. Also, with branch traces, the detection point of a divergence tends to be close to its source.

5.2.1 Hardware Instruction Counting Bug

With branch tracing, we found one particularly noteworthy hardware bug. In the *water* benchmark, we found that a system call was not aligned with the end of the chunk during replay, indicating a bug in the system. The replayer was encountering a system call two instructions prior to the expected end of the chunk. At first, the problem appeared to be a control-flow divergence manifesting as different instruction counts between the log and replayed execution. However, the branch traces revealed no control-flow divergence. Further investigation showed that the hardware was miscounting instructions when handling floating-point exceptions. Without a confirmation from the branch traces regarding no control-flow divergence, it would have been very difficult to pinpoint this bug.

6. RELATED WORK

RnR systems can be classified into software-only and hardware-assisted. Software-only RnR systems (e.g., [5, 8, 9, 10, 11, 18, 27, 28, 32, 34]) run on commodity hardware and use modified runtime libraries, compilers, operating systems or virtual-machine monitors to capture sources of non-determinism. These software-based approaches are either inherently designed for uniprocessor executions or suffer significant slowdown when applied to multiprocessor executions. DoublePlay [35] attempts to make replay on commodity multiprocessors more efficient. To capture memory non-determinism, it timeslices a multithreaded execution into separate epochs and re-executes each epoch sequentially on a single processor. Hence, for each epoch, it only needs to record the order in which threads are scheduled in the second execution. However, DoublePlay cannot capture all data races and, therefore, cannot be used as a general solution for concurrency debugging. In addition, it requires an extra execution to record thread ordering. Finally, it needs to use modified binaries (in particular, a modified *libc*).

Hardware-assisted solutions use hardware to record memory access order. Some approaches modify coherence transactions in conventional directory-based protocols (e.g., [3, 13, 23, 24, 26, 39, 40]) and some are based on snoopy protocols (e.g., [12, 25, 30, 31]). Some approaches (e.g., [39, 40]) record dependences between pairs of instructions. This strategy can produce large logs and increase associated overhead. To reduce this overhead, chunk-based techniques have been proposed (e.g., [7, 12, 13, 23, 24, 30, 31, 36]). DeLorean [23] and Capo [24] are chunk-based schemes that use speculative multithreading hardware to achieve replay parallelism.

In terms of the hardware, QuickRec resembles CoreRacer [31] the most. While the chunking and the instruction reordering are handled similarly, the main differences are on the implementation of instruction atomicity violation, and on the integration of input recording and chunking. LReplay [7] extends a multiprocessor system with a pending period-based mechanism for recording thread interleaving, and uses large CAM structures to deal with instruction reordering. LReplay is evaluated using RTL simulation and does not discuss issues related to system software.

All of these hardware-assisted approaches have only been modeled using simulation, and often without considering the necessary software support. As such, they have generally ignored practical aspects of RnR systems. The QuickRec system is the first work to evaluate RnR across the entire stack using real hardware.

7. LESSONS LEARNED

The main lessons we learned from this effort are:

• Clearly, to maximize the chance that RnR is considered for adoption, it is critical to minimize the number of touch points that it requires on current processor hardware. QuickRec demonstrates that chunk-based recording can be implemented with low-enough implementation complexity and few-enough touch points to make it attractive to processor vendors.
• By far the biggest challenge of implementing RnR is dealing with the idiosyncrasies of the specific architecture used, as they fundamentally permeate many aspects of the hardware and software. Examples of idiosyncrasies are the memory consistency model and the CISC nature of the architecture.
• The design of the deterministic replayer must account for the micro-architectural details of the system, if it is to reproduce the execution exactly. This was altogether neglected by prior replay work. In fact, such micro-architectural details substantially increase the replayer's complexity, in turn impacting the usage models and potentially the ability to create non-proprietary replay tools.
• A new research direction is to investigate replay techniques that reduce or abstract away the complexity mentioned. Such techniques may hinge on commodity hardware, or may require hardware extensions to enable replay software.
• The design of the recording software stack can considerably impact the hardware design, as well as the overall performance. For instance, to properly record kernel-mode instructions (e.g., *copy_to_user()* calls), we had to make non-trivial changes to the hardware-software interface (Section 3.5). Also, the software stack is responsible for practically all of the QuickRec recording overhead.
• The main performance overhead in QuickRec is in the software layer collecting and managing the input logs. A seemingly unimportant issue such as the serialization of input-event processing has become our most obvious bottleneck. Recording input events very efficiently is an area were further work is needed.
• The performance analysis clearly suggests that, with a slightly-improved software stack, RnR can be used in *always-on* manner, enabling a potentially-large number of new RnR uses. Additional features may need to be added, such as checkpointing and log compression to reduce log file sizes in long-running programs.
• Finally, full-system prototyping is required to understand RnR issues related to architecture idiosyncrasies, hardware-software interaction, and true performance bottlenecks.

8. CONCLUSIONS AND FUTURE WORK

RnR of multithreaded programs on multicores has high potential for several important uses: debugging applications, withstanding machine failures, and improving system security. To make RnR systems practical, this paper has contributed in three ways.

First, we presented the implementation of QuickRec, the first multicore IA-based prototype for RnR of multithreaded programs. The prototype includes an FPGA instantiation of a Pentium multicore and a Linux-based full software stack.

Second, we described several key implementation aspects in QuickRec. We showed how to efficiently handle x86 instructions that produce multiple memory transactions, and detailed the elaborate hardware-software interface required for a working system.

Third, we evaluated QuickRec and demonstrated that RnR can be provided efficiently in real IA multicore machines. We showed that the rate of memory log generation is insignificant, given today's bus and memory bandwidths. Furthermore, the recording hardware had negligible performance overhead. However, the software stack induced an average recording overhead of nearly 13%. Such overhead must come down to ensure always-on use of QuickRec.

Based on this work, we suggest focusing future research on several directions. First, to reduce the software stack overhead, it is important to record input events very efficiently — specifically, in a partially-ordered manner. This will reduce recording overhead, and truly enable always-on RnR.

Second, much emphasis should be placed on the replay aspect of RnR. We need approaches that are tolerant of, and abstract away, the micro-architectural details of the recording platform. Otherwise, proprietary details will stifle the development of replay support. We need creative ways of combining hardware and software support for replay.

Finally, we need to develop and demonstrate many uses of the RnR technology that solve real problems of multicore users. The areas of parallel program development tools and security-checking aids seem particularly ripe for development.

9. REFERENCES

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. An Execution-Backtracking Approach to Debugging. *IEEE Software*, May 1991.

[2] U. Banerjee, B. Bliss, Z. Ma, and P. Petersen. Unraveling Data Race Detection in the Intel Thread Checker. In *STMCS*, March 2006.

[3] A. Basu, J. Bobba, and M. D. Hill. Karma: Scalable Deterministic Record-Replay. In *ICS*, June 2011.

[4] B. Boothe. Efficient Algorithms for Bidirectional Debugging. In *PLDI*, June 2000.

[5] T. Bressoud and F. Schneider. Hypervisor-Based Fault-Tolerance. *ACM Transactions on Computer Systems*, 14(1), February 1996.

[6] S.-K. Chen, W. K. Fuchs, and J.-Y. Chung. Reversible Debugging Using Program Instrumentation. *IEEE Transactions on Software Engineering*, 27(8):715–727, August 2001.

[7] Y. Chen, W. Hu, T. Chen, and R. Wu. LReplay: A Pending Period Based Deterministic Replay Scheme. In *ISCA*, June 2010.

[8] J.-D. Choi and H. Srinivasan. Deterministic Replay of Java Multithreaded Applications. In *SPDT*, August 1998.

[9] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *OSDI*, December 2002.

[10] G. Dunlap, D. Lucchetti, M. Fetterman, and P. Chen. Execution Replay of Multiprocessor Virtual Machines. In *VEE*, March 2008.

[11] A. Forin. Debugging of Heterogeneous Parallel Systems. In *PDD*, May 1988.

[12] N. Honarmand, N. Dautenhahn, J. Torrellas, S. T. King, G. Pokam, and C. Pereira. Cyrus: Unintrusive Application-Level Record-Replay for Replay Parallelism. In *ASPLOS*, March 2013.

[13] D. R. Hower and M. D. Hill. Rerun: Exploiting Episodes for Lightweight Memory Race Recording. In *ISCA*, June 2008.

[14] Intel Corp. *Intel 64 and IA-32 Architectures Software Developer's Manual*. 2002. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[15] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting Past and Present Intrusions Through Vulnerability-Specific Predicates. In *SOSP*, October 2005.

[16] S. T. King and P. M. Chen. Backtracking Intrusions. In *SOSP*, October 2003.

[17] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging Operating Systems with Time-Traveling Virtual Machines. In *USENIX Annual Technical Conference*, April 2005.

[18] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Trans. Comp.*, April 1987.

[19] G. Lueck, H. Patil, and C. Pereira. PinADX: An Interface for Customizable Debugging with Dynamic Instrumentation. In *CGO*, 2012.

[20] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, 2005.

[21] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, February 2002.

[22] J. D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE TCCA Newsletter*, pages 19–25, December 1995.

[23] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In *ISCA*, June 2008.

[24] P. Montesinos, M. Hicks, S. King, and J. Torrellas. Capo: A Software-Hardware Interface for Practical Deterministic Multiprocessor Replay. In *ASPLOS*, March 2009.

[25] S. Narayanasamy, C. Pereira, and B. Calder. Recording Shared Memory Dependencies Using Strata. In *ASPLOS*, October 2006.

[26] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *ISCA*, June 2005.

[27] D. Z. Pan and M. A. Linton. Supporting Reverse Execution for Parallel Programs. In *PDD*, May 1988.

[28] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. PinPlay: A Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs. In *CGO*, April 2010.

[29] C. Pereira, G. Pokam, K. Danne, R. Devarajan, and A.-R. Adl-Tabatabai. Virtues and Obstacles of Hardware-Assisted Multi-Processor Execution Replay. In *HotPAR*, June 2010.

[30] G. Pokam, C. Pereira, K. Danne, R. Kassa, and A.-R. Adl-Tabatabai. Architecting a Chunk-Based Memory Race Recorder in Modern CMPs. In *MICRO*, December 2009.

[31] G. Pokam, C. Pereira, S. Hu, A.-R. Adl-Tabatabai, J. Gottschlich, H. Jungwoo, and Y. Wu. CoreRacer: A Practical Memory Race Recorder for Multicore x86 TSO Processors. In *MICRO*, 2011.

[32] M. Russinovich and B. Cogswell. Replay for Concurrent Non-Deterministic Shared-Memory Applications. In *PLDI*, May 1996.

[33] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer: Data Race Detection in Practice. In *WBIA*, December 2009.

[34] S. Srinivasan, S. Kandula, C. Andrews, and Y. Zhou. Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging. In *USENIX Ann. Tech. Conf.*, June 2004.

[35] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: Parallelizing Sequential Logging and Replay. In *ASPLOS*, March 2011.

[36] G. Voskuilen, F. Ahmad, and T. N. Vijaykumar. Timetraveler: Exploiting Acyclic Races for Optimizing Memory Race Recording. In *ISCA*, June 2010.

[37] Q. Wang, R. Kassa, W. Shen, N. Ijih, B. Chitlur, M. Konow, D. Liu, A. Sheiman, and P. Gupta. An FPGA Based Hybrid Processor Emulation Platform. In *FPL*, August 2010.

[38] XtreamData. http://www.xtreamdata.com.

[39] M. Xu, R. Bodik, and M. Hill. A "Flight Data Recorder" for Enabling Full-System Multiprocessor Deterministic Replay. In *ISCA*, June 2003.

[40] M. Xu, R. Bodik, and M. D. Hill. A Regulated Transitive Reduction (RTR) for Longer Memory Race Recording. In *ASPLOS*, 2006.

[41] M. V. Zelkowitz. Reversible Execution. *Communications of the ACM*, 16(9):566, September 1973.

SigRace: Signature-Based Data Race Detection

Abdullah Muzahid
University of Illinois at Urbana-Champaign, USA
muzahid2@illinois.edu

Darío Suárez
Universidad de Zaragoza
Zaragoza, Spain
dario@unizar.es

Shanxiang Qi
University of Illinois at Urbana-Champaign, USA
sqi2@illinois.edu

Josep Torrellas
University of Illinois at Urbana-Champaign, USA
torrella@illinois.edu

ABSTRACT

Detecting data races in parallel programs is important for both software development and production-run diagnosis. Recently, there have been several proposals for hardware-assisted data race detection. Such proposals typically modify the L1 cache and cache coherence protocol messages, and largely lose their capability when lines get displaced or invalidated from the cache. To avoid these shortcomings, this paper proposes a novel approach to hardware-assisted data race detection. The approach, called *SigRace*, relies on hardware address signatures. As a processor runs, the addresses of the data that it accesses are automatically encoded in signatures. At certain times, the signatures are automatically passed to a hardware module that intersects them with those of other processors. If the intersection is not null, a data race may have occurred.

This paper presents the architecture of SigRace, an implementation, and its software interface. With SigRace, caches and coherence protocol messages are unmodified. Moreover, cache lines can be displaced and invalidated with no effect. Our experiments show that SigRace is significantly more effective than a state-of-the-art conventional hardware-assisted race detector. SigRace finds on average 29% more static races and 107% more dynamic races. Moreover, if we inject data races, SigRace finds 150% more static races than the conventional scheme.

Categories and Subject Descriptors

B [**Hardware**]: B.3 Memory Structures,B.3.2 Design Styles. **Subjects:** Shared memory; B.3.4 [**Reliability, Testing, and Fault-Tolerance**]: Error checking.

General Terms

Design, Measurement, Reliability.

Keywords

SigRace, Signature, Timestamp, Data Race, Concurrency Defect, Happened-Before.

1. INTRODUCTION

With the widespread use of multicore hardware, parallel programming is likely to become more prevalent. At the same time, concurrency bugs are likely to take on a higher profile and become a very costly problem. Consequently, it is crucial to continue developing more effective techniques to detect and fix them.

An important type of concurrency bug is a data race. A data race occurs when two threads access the same variable without an intervening synchronization and at least one of the accesses is a write. The erroneous program behavior caused by the race may only appear under certain access interleavings, making debugging data races notoriously hard.

For this reason, data race detection has been the subject of much work (e.g., [5, 8, 12, 14, 15, 16, 17, 18, 19, 22, 24, 26, 27, 29, 30]), including the development of commercial software tools for race debugging (e.g., [8, 26]) and even the proposal of special hardware structures in the machine (e.g., [12, 18, 19, 30]). In general, there are two approaches to finding data races, namely the lockset approach, as in Eraser [24], and the happened-before one, as in Thread Checker [8]. The lockset approach is based on the idea that all accesses to a given shared variable should be protected by a common set of locks. Consequently, it tracks the set of locks held while accessing each variable. It reports a violation when the currently-held set of locks (lockset) at two different accesses to the same variable have a null intersection.

The happened-before approach relies on epochs. An epoch is a thread's execution between two consecutive synchronization operations. Each processor has a logical clock, which identifies the epoch that the processor is currently executing. In addition, each variable has a timestamp, which records at which epoch the processor accessed it. When another processor accesses the variable, it compares the variable's timestamp to its own clock, to determine the relationship between the two corresponding epochs: either one logically happened before the other, or the two logically overlap. In the latter case, we have a race.

Race detectors that use these algorithms in software typically induce about 10–50x slowdowns on programs [8, 14, 22, 24]. Such slowdowns can distort the timing of races identified in production runs, and make them hard to find. For this reason, there have been several recent proposals for race detectors with hardware assists [12, 18, 19, 30]. Such schemes should be effective at debugging races in production runs. However, they detect races by augmenting the cache state and the coherence protocol. Specifically, they tag each cache line with a timestamp [12, 18, 19] or a lockset [30], perform additional operations on local/external access to the cache, and piggyback information on cache coherence protocol messages. L1 caches and coherence protocol units are key hardware structures, either time-critical or complicated. In addition, if a line is displaced or invalidated from the cache, these systems typically lose the ability to detect races involving the line.

This paper proposes a novel approach to hardware-assisted data race detection that overcomes these limitations. Our approach, called *SigRace*, relies on hardware address signatures. As a processor runs, the addresses of the data that it accesses are automatically encoded in signatures. At certain times, the signatures are automatically passed to a hardware module that intersects them to those of other processors. If the intersection is not null, a data race may have occurred. With SigRace, there are no changes to the cache or the cache coherence protocol messages, and there are no critical-path operations performed on local/external access to the cache. Moreover, lines can be displaced or invalidated from caches without affecting SigRace's ability to detect data races.

This paper presents the architecture of SigRace, an implementation, and its software interface. Application code is unmodified. Our experiments show that SigRace is significantly more effective than a state-of-the-art conventional hardware-assisted race detector. SigRace finds, on average, 29% more static races and 107% more dynamic races. Moreover, if we inject data races, SigRace finds 150% more static races than the conventional scheme.

This paper is organized as follows: Section 2 gives a background; Sections 3 and 4 describe the SigRace architecture and implementation; Section 5 evaluates SigRace; and Section 6 concludes.

2. BACKGROUND

2.1 Logical Timestamps for Happened-Before

Lamport's happened-before relation [9] in a multithreaded environment states that an event α happened before another β if (i) both are performed by the same thread and α precedes β in program order, or (ii) α is a release and β is an acquire on the same object, or (iii) for some other event γ, α happened before γ and γ happened before β. If α happened before β or vice-versa, the two events are ordered; otherwise, they are concurrent or unordered. The happened-before algorithm for race detection finds out whether two memory accesses to the same location that are performed by different threads are unordered and at least one is a write. This algorithm only detects races that actually occur during execution.

In a typical implementation, each thread maintains a logical vector clock, which has as many components as number of threads [7]. If thread *t* has a vector clock $vc_t[.]$, then the element $vc_t[t]$ contains the time of the thread itself and, given another thread *u*, $vc_t[u]$ contains the latest time of *u* "known" to *t*. When *t* performs a synchronization operation, it starts a new *Epoch* and increments $vc_t[t]$. Suppose that, after *t* performed a release on object *S*, *u* acquires *S*. In this case, *u* increments $vc_u[u]$ and, in addition, updates the rest of $vc_u[.]$ as follows: $vc_u[i] = max(vc_u[i], vc_t[i])$ for every $i \neq u$. Here, $vc_t[.]$ is the vector clock of thread *t after* the release operation. We refer to the value of a thread's vector clock during an epoch as the epoch's *Timestamp*. Figure 1(a) shows an example execution with epoch timestamps.

We determine whether there is a happened-before relation between two epochs by comparing their timestamps. Specifically, if epoch *f* of thread *t* has timestamp $vc_t^f[.]$ and epoch *g* of thread *u* has timestamp $vc_u^g[.]$, then *f* happened before *g* if and only if $vc_t^f[t] < vc_u^g[t]$ and $vc_t^f[u] < vc_u^g[u]$. For example, in Figure 1(a), the epoch after the acquire in Thread 2 happened before the epoch after the second acquire in Thread 0.

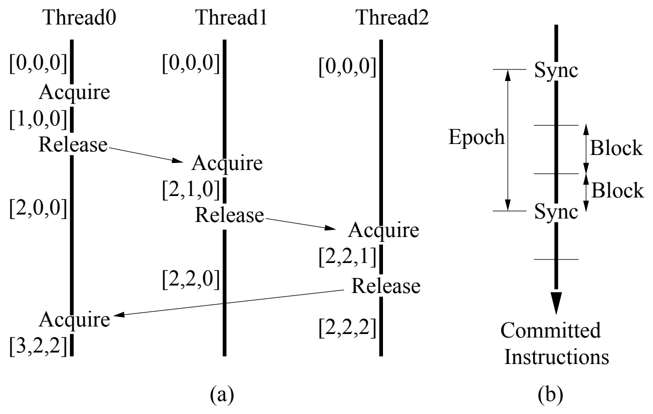


Figure 1: Example of execution of three threads with epoch timestamps in brackets (a), and definitions in a thread's execution (b).

2.2 Hardware Schemes for Race Detection

There are at least four proposals for hardware-assisted data-race detectors, namely Min and Choi's [12], ReEnact [19], CORD [18] and HARD [30]. They all detect races by tagging the state in the caches as it is being accessed, and then piggybacking the tags on cache coherence protocol messages between processors so that they can be compared.

ReEnact and CORD use the happened-before approach. They tag each cache line with timestamp information, and send and compare timestamps at least at every coherence action (invalidation of cached line or external read of a dirty cached line). In ReEnact, the tag is an index into a table of vector-clock timestamps. In CORD, the tag is four scalar timestamps (two for read and two for write), and two sets of read-write bits per word. HARD uses the lockset approach and, therefore, only handles locks properly. It tags each cache line with two special state bits, and a bit vector that represents the lockset for the line. These bits are checked at every access to the line, and are kept coherent by the coherence protocol as if they were data. Finally, Min and Choi use the happened-before approach for only nested doall loops. They tag each cache line with a set of read and write bits for each doall nesting level, and perform tag checking at every cache access.

In all these schemes, the hardware can easily detect an address and an instruction involved in a race on the fly. Then, to reveal the other (or several other) instructions involved in the same race to the programmer, it is necessary to roll back and re-execute the code section. For example, ReEnact [19] executes under thread-level speculation. If a race is detected, it rolls back execution to the most recent checkpoint, places a watchpoint on the racing address, and re-executes. The machine then captures all the accesses to the racing address. In addition, re-execution is also necessary to discard false-positive races. They occur because some of these hardware schemes tag the cache at line-size granularity. Consequently, accesses from different processors to different words of the same line (false sharing) may appear as races. Re-execution disambiguates this case.

Overall, these schemes have two shortcomings. First, they modify the L1 cache, the operations performed on some local/external accesses to L1, and the cache coherence protocol messages. These are key hardware structures, either time-critical or complicated to design and debug. Second, when a line is displaced or invalidated from the cache, the system loses its ability to detect a data race for that line. An exception is CORD, which keeps some timestamp information in memory. We would like a design that decouples cache and coherence protocol from race detection, and has a longer detection window than that provided by cache residence.

2.3 Hardware Address Signatures

A hardware address signature is a long register (e.g., 2Kbits long) where the memory addresses accessed by the processor are automatically hash-encoded and accumulated using a Bloom filter [2]. Signatures have been used in the Bulk system [4] and several subsequent multiprocessor designs (e.g., [3, 13, 28]) to detect data dependences between threads in thread-level speculation and transactional memory. Signatures are efficiently operated on in hardware using simple logic (e.g., bit-wise AND of signatures to find common addresses). From a signature, it is only possible to obtain a superset of the addresses that were originally encoded in the signature. Consequently, operations on signatures may produce false positives, although not false negatives.

In this paper, we use signatures to detect data races. While HARD [30] used a Bloom filter to encode locksets for efficient manipulation, this is the first paper that uses address signatures for happened-before race detection.

3. SIGNATURE-BASED RACE DETECTION

3.1 Overview of the Idea

The idea of SigRace is to automatically record the set of addresses accessed by the processor in a code section in hardware signatures. At appropriate intervals, the signatures and the epoch timestamp are automatically passed to an on-chip hardware module called *Race Detection Module* (RDM). The RDM keeps the signatures and the timestamp in an in-order queue assigned to the initiating processor, and compares them to the entries of queues assigned to other processors using very efficient signature operations. The comparison quickly determines whether there has been a potential data race.

SigRace addresses the two shortcomings of existing hardware-assisted schemes. First, there are no L1 cache modifications, no critical-path operations performed on local/external accesses to L1, and no cache coherence protocol message changes. Signature generation, storage, and comparison are decoupled from caches and coherence protocol. Second, lines can be displaced or invalidated from caches without SigRace losing the ability to detect data races. In practice, the RDM necessarily has limited storage capacity, and old signatures are discarded when room is needed, also limiting the race detection window. We will see, however, that SigRace's race detection capability is higher than that of cache-based systems.

Like all of the currently-proposed hardware schemes (Section 2.2), SigRace needs to rely on rollback and re-execution to provide the full set of racing instructions to the programmer, and to disambiguate false-positive races. However, using signatures introduces two differences. First, since SigRace detects races lazily when signatures are compared, SigRace without re-execution cannot provide any of the racing instructions. In contrast, since the currently-proposed schemes detect the race eagerly, they can plausibly detect one of the racing instructions without re-execution.

The second difference is the source of false-positive races. Unlike currently-proposed schemes, SigRace does not suffer false positives due to false sharing. This is because SigRace encodes *fine-grain* (e.g., word) addresses in signatures. Accesses to different words of the same line do not induce a data race report. However, address aliasing in signatures may induce false positives in SigRace. This is because signatures represent a superset of the addresses that were encoded [4]. False negatives are not possible.

For simplicity, we want SigRace to support the rollback and re-execution largely in software. Consequently, SigRace does not use thread-level speculative execution. Reads and writes commit as usual. We use the ReVive checkpointing/rollback mechanism proposed by Prvulovic *et al.* [20]. After rollback and re-execution to the race, an analysis phase takes place. We envision rollback, re-execution, and analysis to be transparent to the user, who should at worst notice a slight slowdown when many false data races are detected.

Address collection into signatures is disabled and enabled in software at kernel entries and exits, respectively, and, optionally, at library entries and exits. This typically improves race detection. Moreover, the programmer can disable address collection during the execution of certain code sections. Finally, signatures are assigned to software threads rather than to hardware contexts.

In the following, we describe SigRace's operation under three stages: normal execution, re-execution, and race analysis. For simplicity of presentation, this section assumes one thread per processor and no thread migration. The implementation of SigRace is left for Section 4.

3.2 Normal Execution under SigRace

The execution of a thread is logically divided into epochs, which are the dynamic instructions committed between synchronization operations (Figure 1(b)). The latter include, e.g., acquiring a lock, releasing it, waiting on a flag, setting a flag, or crossing a barrier. Under SigRace, each processor keeps the timestamp of the current epoch, which is encoded and updated as per Section 2.1. In addition, the processor has a Read (R) and a Write (W) Signature. When a load or a store commits, a hardware Bloom filter as in [4] automatically hash-encodes and accumulates the address loaded from or stored to, respectively, into the correct signature.

Ideally, a processor can keep its timestamp and R and W signatures to itself until the end of the epoch. At that point, they are made visible to all other processors, to check for data races. In practice, long epochs would cause the signatures to accumulate so much state that any operation on them would likely induce many false positives due to aliasing [4]. Consequently, when the processor has committed a certain number of dynamic instructions that we call a *Block* without finding a synchronization operation, the hardware automatically passes the timestamp and signatures to the RDM. Figure 1(b) shows the resulting execution: a block finishes when either a certain number of dynamic instructions have been committed or a synchronization operation is found.

The exact actions taken when a block in processor *i* finishes for either reason are as follows (Figure 2). First, the hardware automatically dumps the timestamp and R and W signatures into a memory-mapped FIFO queue of registers in the RDM called BlockHistoryQueue[i] (Step *1* in the figure). To save network bandwidth, the data is transferred in compressed format. The R and W signatures are then cleared. Finally, if the block finished because of a synchronization operation, library software updates the epoch timestamp and then saves it in a log in memory to keep a trail of timestamp changes — which is useful if we need to roll back execution.

At the RDM, simple hardware automatically compares the incoming data to entries in all the other BlockHistoryQueue[.] (Step *2* in the figure). Specifically, for a given BlockHistoryQueue[j], the incoming timestamp TS_{i0} gets compared to TS_{j0}, TS_{j1}, etc — in sequence order starting from the latest one available. Such comparisons stop as soon as one of the *j* timestamps is found to precede the incoming timestamp — in this case, due to transitivity, all earlier *j* timestamps will also precede the incoming one. Then, for all timestamp pairs found to be unordered (e.g., TS_{i0} and TS_{jN}), simple

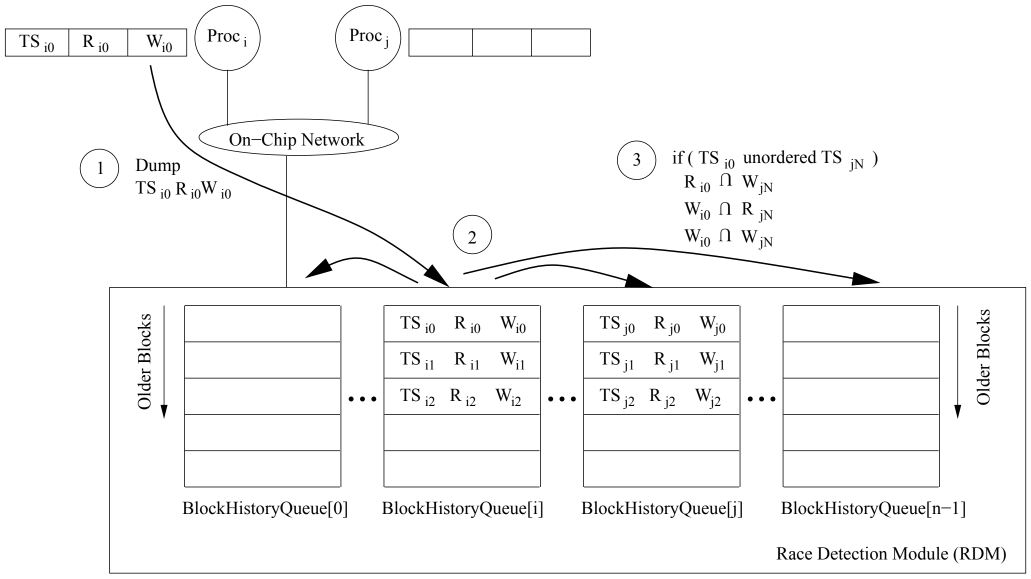


Figure 2: Operations when a block finishes. In the figure, *TS*, *R*, and *W* refer to timestamp and read and write signature, respectively. In any BlockHistoryQueue[k], entries for older blocks have higher subscripts.

signature functional units compute $R_{i0} \cap W_{jN}$, $W_{i0} \cap R_{jN}$, and $W_{i0} \cap W_{jN}$ (Step *3* in the figure). If any of these is not null, the two blocks have accessed the same location(s) without synchronization and at least one wrote. We have detected a data race — or a false positive. We call these two blocks and their corresponding threads the *Conflicting Blocks* and *Threads*.

A BlockHistoryQueue[k] is a FIFO queue. When it overflows, information on the displaced blocks is lost. We have lost the ability to detect data races in those blocks. We accept this limitation to keep overheads to a minimum.

3.3 Re-Execution under SigRace

When a pair of Conflicting Blocks is found, we want to identify for the user the exact instructions and address(es) involved in the race(s), and to weed out any false positive transparently to the user. In our design, an exception forces all processors to roll back to the previous checkpoint and enter the *Re-execution* mode. In this section, we describe the checkpointing support and the re-execution process.

3.3.1 Checkpointing Support

The SigRace design that we present needs a low-overhead checkpointing scheme. Ideally, such a scheme would already be in place for reliability purposes, and SigRace would reuse it. One possible scheme is ReVive [20], which performs incremental memory-state checkpointing. With ReVive, all processors are interrupted at intervals of several milliseconds, at which point, a software handler creates a global light-weight checkpoint. The checkpoint consists of saving the register state of all processors and writing back all the dirty cache lines to memory. Then, during execution, the memory controller logs every first update to a main memory location since the previous checkpoint (i.e., the log saves the value in memory before the first write-back of a dirty line from caches to the location). Rolling back to the previous checkpoint involves undoing the trail of memory updates from this log until the checkpoint, and then restoring the registers. The ReVive design in Prvulovic *et al.* [20] adds a 6.3% execution time overhead.

In addition, the kernel collects and buffers the inputs to the program during Normal execution — such as interrupts, system call returns, and I/O input — and passes them to the re-execution at appropriate times. Support similar to this is provided by Flashback [25] and Rx [21], which require no hardware modifications.

With these two mechanisms, we will now see that SigRace re-executes following the same paths until the first data race is found.

3.3.2 Re-Execution Operation

Re-execution forces the application to follow the same order of epochs as in the original execution, and leaves each thread at the beginning of the *epoch* that the thread was executing when the race was detected. This is shown in Figure 3, where a race was detected at the points shown in Figure 3(a), and re-execution brings the threads to points *A*, *B*, *C*, and *D* in Figure 3(b). Note that re-execution does not bring each thread to the actual block that it was executing when the race was detected. This is because we do not rely on the ability to reproduce block boundaries exactly.

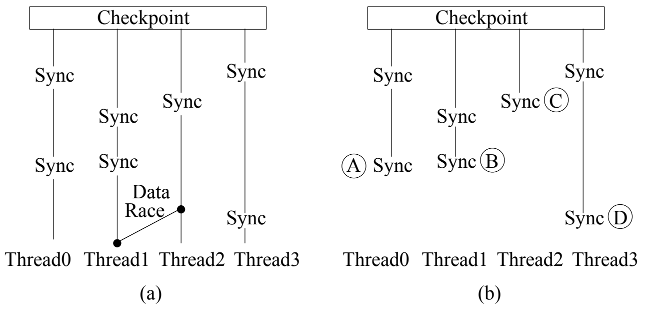


Figure 3: Detection of a data race during Normal (a) and Re-execution (b) modes.

To reproduce the order of epochs, SigRace uses the history of logged timestamps (Section 3.2). They encode the history of synchronization operation orders — i.e., which thread completed a synchronization operation before which other thread. SigRace uses these timestamps to follow the same synchronization orders.

Specifically, each processor has a Thread Re-execution Timestamp (TRT) register into which, as it re-executes, it successively loads the timestamps logged since the checkpoint. Recall that each timestamp was saved *after* the processor went past a synchronization operation. In addition, there is a shared software structure in memory called Global Re-execution Timestamp (GRT) that contains the most up-to-date logical time of each processor during the re-execution. In other words, while the TRT is the "thread view" of the current re-execution time, the GRT is the "true global view". Each processor compares its TRT to the GRT to see when the other processors have executed all the earlier epochs and the processor can proceed. Proceeding means for the processor to perform its next synchronization operation, update its own component of the GRT, execute its next epoch, and read its next logged timestamp into its TRT.

The actual algorithm is as follows. Let us call $grt[.]$ the GRT and $trt_p[.]$ the TRT of processor p. Each i in $grt[i]$ is the latest epoch from processor i that has been executed. For example, Figure 4 repeats the timeline of Figure 1(a) and shows with an arrow the current position of each replaying processor. As a result, $grt[.] = [2, 1, 0]$. All processors are waiting at a synchronization operation and we need to decide which one(s) to execute next. Each processor has loaded into its trt the timestamp it had *after* the synchronization (e.g., $trt_1[.] = [2, 2, 0]$). When a given processor p finds that $grt[i] \geq trt_p[i]$ for all $i \neq p$, then processor p executes the synchronization operation, sets $grt[p] = trt_p[p]$, executes its next epoch, and loads its next logged timestamp into $trt_p[.]$. The last two operations are not performed if there is no next logged timestamp. In the figure, the only processor for which the inequality is true is Processor *1*. Consequently, Processor *1* will execute the release and set GRT to [2,2,0]. Since it has no further timestamp logged, it will wait there.

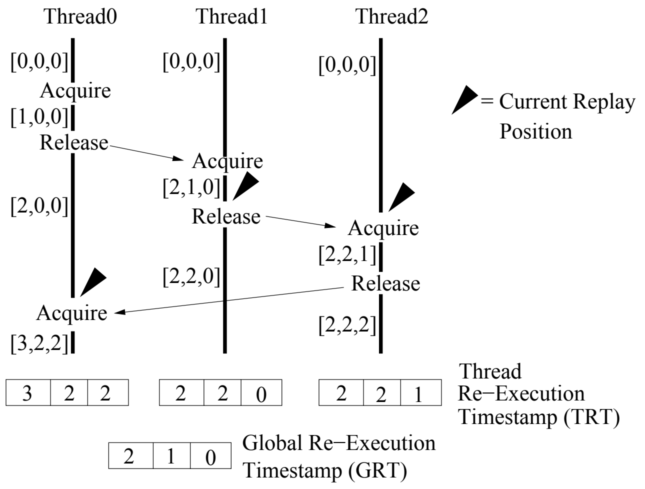


Figure 4: Re-execution using the logged timestamps.

3.4 Race Analysis under SigRace

When all threads have reached their last logged timestamp, execution enters the *Analysis* mode. In this mode, only the threads involved in the data race execute, while the others stall. Specifically, first, the two processors executing the Conflicting threads load into a local register called the *Conflict Signature* the intersection of the two Conflicting blocks' signatures — namely the union of $R_{i0} \cap W_{jN}$, $W_{i0} \cap R_{jN}$, and $W_{i0} \cap W_{jN}$ as per Section 3.2. The Conflict Signature holds the hashed address(es) involved in the race. Then, the two Conflicting threads execute normally up to their next synchronization points, while the hardware automatically intersects their loads and stores against the Conflict Signature. Every time a non-null intersection occurs, a trap is triggered, which records the memory address and the PC. Finally, when both threads have reached their next synchronization points, a software handler compares the record of trapping addresses in both processors, to see if there are common addresses. If so, SigRace has found a data race, which it reports to the user. Otherwise, it was only a false positive and is ignored.

As each Conflicting thread reaches its next synchronization point, it may have executed past its Conflicting block. This is fine, since it enables us to capture as many of the references involved in the data race(s) as possible.

After the Analysis step, execution *seamlessly* returns to the Normal mode of execution. This is enabled by the fact that SigRace continued to perform timestamp/signature logging and signature intersection during Re-execution and Analysis modes — exactly like it did during Normal mode. In this way, the trail of timestamps and signatures is up to date at the point where Analysis completes and all processors resume Normal execution.

Because the Analysis step may push program execution beyond what was executed before the rollback, it is possible that the Analysis step discovers new data races. To address this case, SigRace proceeds as follows. Every time two blocks are found to conflict during Analysis ($R_{i0} \cap W_{jN}$, $W_{i0} \cap R_{jN}$, or $W_{i0} \cap W_{jN}$ are not null), a handler compares their intersection against the contents of the Conflict Signature. If the latter is a superset, no action is taken because this race is already being processed (call it *Race1*). Otherwise, the handler saves the signature intersection and records the need to analyze the new data race (call it *Race2*) later. In this case, after *Race1* is fully analyzed, execution is rolled back, and we proceed to perform Re-execution and Analysis for *Race2*. Note that we cannot analyze the two races concurrently because, by the time we detect the presence of *Race2*, processors have already issued some of the references associated with it.

Overall, to minimize the amount of re-execution, SigRace is designed as follows. When a processor in Normal execution detects a pair of Conflicting blocks, it does not immediately request a rollback. Instead, it continues executing for several more blocks (e.g., 5–10) or until it synchronizes, before interrupting all other processors and requesting rollback. The goal is to collect as many potential races as possible. During Analysis, the Conflict Signature of each processor contains the racing addresses of all the races that the processor is involved in characterizing. In this way, multiple races are analyzed concurrently. Finally, SigRace also saves the Conflict Signatures of the races that it has finished analyzing. In this way, if SigRace has to re-execute the same code a second time, it can ignore the race already analyzed.

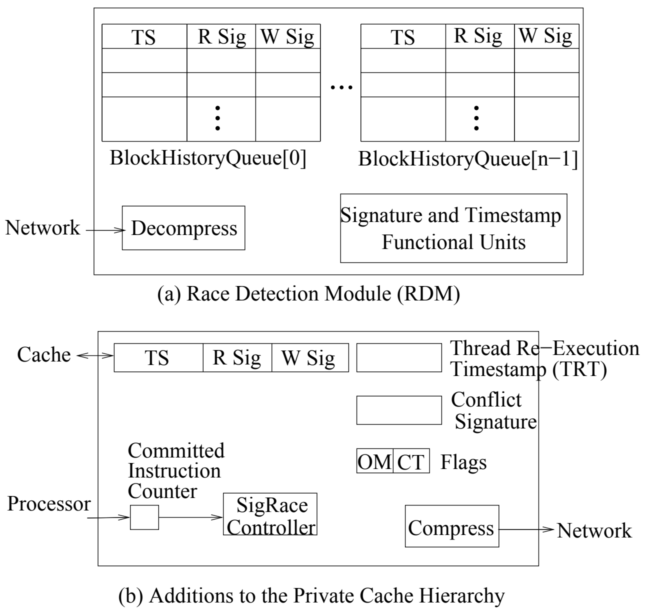
4. SIGRACE IMPLEMENTATION

Our implementation of SigRace requires some hardware and software changes to a chip multiprocessor. The hardware changes are the Race Detection Module (RDM) and some additions to the per-processor cache hierarchy. The cache tag and data arrays are *unmodified*. Also, SigRace does not use speculative multithreading. On the software side, SigRace needs an augmented synchronization library. In this section, we describe the hardware and software components, and then how SigRace is virtualized to make it usable.

4.1 Hardware Modifications

The RDM is a simple on-chip hardware module that is connected to the on-chip network. As shown in Figure 5(a), it contains the

BlockHistoryQueue[.], which stores past timestamps (TS) and signatures for all the processors (Section 3.2). It also includes functional units that operate on signatures (like in Bulk [4]) and timestamps.



(a) Race Detection Module (RDM)

(b) Additions to the Private Cache Hierarchy

Figure 5: Hardware support needed by SigRace.

SigRace also requires some per-processor hardware that is placed in the cache hierarchy in a module that interfaces with the processor, the cache and the network (Figure 5(b)). The module includes storage for the current epoch timestamp and the current block's R and W signatures. The addresses hashed into signatures have a finer granularity than cache line, so that false sharing of a line does not trigger incorrect data race alarms. A good choice is to use word addresses. The module also includes the Thread Re-Execution Timestamp (TRT) for re-execution (Section 3.3) and the Conflict Signature for analysis (Section 3.4). There are two flags, namely the *Operation Mode* (OM) that denotes whether the hardware is in Normal, Re-execution, or Analysis mode, and the *Conflicting Thread* (CT) that denotes whether the thread is a Conflicting one (Section 3.2). There is also a *Committed Instruction Counter*. When the latter reaches the maximum value set for a block — or an approximate value, since there is no need to be exact — it sends a signal to terminate the current block. The SigRace controller then initiates the following actions: dump TS, R and W into the corresponding BlockHistoryQueue[i], and clear R, W, and the Committed Instruction Counter.

The TS and R and W signatures are compressed before being sent to the on-chip network, and decompressed as they get into the RDM. We call these network messages the *Summary* messages. Their compressed size is ≈100 bytes — for 2 signatures of 2 Kbits each and a 160-bit timestamp. This is less than the size of two cache lines, and is sent out every time a block completes (≈2,000 committed instructions). Summary messages from the same processor need to arrive at the RDM in order; messages from different processors can arrive in any order. This centralized RDM design is fine for the small numbers of processors considered in this paper (8). In large, distributed machines, the RDM can be distributed as well.

4.2 Software Interface

High-level synchronization constructs such as M4 macros [11] and OpenMP directives [6] are commonly used by programmers and parallelizing compilers. These constructs can enable SigRace transparently. Specifically, we rewrite such constructs to encapsulate the SigRace operations. As a result, the application code does not need be modified at all, and all we need is to relink it with the new M4 or OpenMP library.

To accomplish this, we start by adding three processor instructions that operate on local SigRace structures (Table 1). Two of the instructions (*collect_on* and *collect_off*) enable and disable the collection of addresses into signatures, and the counting of committed instructions. A variation of these instructions could perform these actions only on a range of addresses. These instructions are used to prevent the signatures from being polluted by unrelated accesses (such as those from the OS or the instrumentation added to the macros) or by obviously-private accesses (e.g., those to the stack). They can also be used to mark a benign data race or an epoch that should *skip the checking*. The other instruction (*sync_reached*) is invoked when execution reaches a synchronization operation. Specifically, it is invoked immediately before performing a release-type operation and immediately after performing a successful acquire-type operation. It tells the SigRace controller to dump TS, R and W into the RDM, clear R, W, and the Committed Instruction Counter, and increment the counter in TS that corresponds to the local thread.

Instruction	Description
collect_on	Collect addresses into R and W, and count committed instructions.
collect_off	Do not collect addresses into R or W, or count committed instructions.
sync_reached	Dump TS, R, and W into the RDM. Clear R, W and the Committed Instruction Counter. Increment the counter in TS that corresponds to the local thread.

Table 1: Instructions to manage SigRace structures.

For simplicity, we assume that these instructions make their side effects visible only when they commit — like the updates of signatures by loads and stores. A design where these actions happen earlier in the pipeline can also be conceived.

With these instructions, we can build new macros for all the synchronization primitives. As an example, we consider the M4 macros for UNLOCK and LOCK. Table 2 shows the conventional implementation and the one adapted for SigRace in Normal execution mode (SN_UNLOCK and SN_LOCK). In the SigRace version, synchronization variables have a *lock* and a *timestamp* field — shown as $1.lock and $1.timestamp, respectively.

In SN_UNLOCK, before unlocking the *lock* field of the variable, the *sync_reached* instruction executes (Table 2). Then, the TS of the processor — which has already been updated by *sync_reached* — is saved in the *timestamp* field of the variable (Line 3). Then, the lock is released. Finally, the updated TS is explicitly saved in a TS log in memory, in case it is needed for re-execution (Line 5). In SN_LOCK, after the lock is acquired and *sync_reached* executed, the new TS is generated. This is done by taking the current TS — which is already updated by *sync_reached* — and the value stored in *timestamp*, and applying the algorithm of Section 2.1 (shown as *GenerateTS*). Finally, the TS is saved in the TS log.

Finally, we need to augment the macros to work for all exe-

Operation	Implementation	Code
Unlock	Conventional	1: UNLOCK(‘{
		2: unlock($1);}’)
	SigRace (Normal Execution Mode)	1: SN_UNLOCK(‘{
		2: sync_reached;
		3: $1.timestamp = TS;
		4: unlock($1.lock);
		5: AppendtoTSLog(TS,TSLog);
		6: }’)
Lock	Conventional	1: LOCK(‘{
		2: lock($1);}’)
	SigRace (Normal Execution Mode)	1: SN_LOCK(‘{
		2: lock($1.lock);
		3: sync_reached;
		4: TS = GenerateTS(TS,
		5: $1.timestamp);
		6: AppendtoTSLog(TS,TSLog);
		7: }’)

Table 2: UNLOCK and LOCK macros: conventional implementation and one for SigRace in Normal execution mode.

```
1:  S_UNLOCK(‘{
2:    collect_off
3:    if (Flags.OM == Re-execution){ /* Re-exec. mode? */
4:      LoadfromTSLog(TRT,OldTSLog);
5:      WhileNotMyTurn(TRT,GRT) {};
6:    }
7:    else if (Flags.OM == Analysis){ /* Analysis mode? */
8:      AnalyzeRecordOfAccesses(); /* Analyze data */
9:      Flags.OM = Normal; /* End of Analysis mode */
10:   }
11:   SN_UNLOCK($1)
12:   if (Flags.OM == Re-execution){ /* Re-exec. mode? */
13:     UpdateGRT(TRT,GRT);
14:     if (OldTSLogEmpty) {
15:       Flags.OM = Analysis; /* Analysis mode */
16:       if (Flags.CT) { /* One of the Conflicting threads? */
17:         LoadConflictSignature();
18:         /* Set up the Conflict Signature. Continue */
19:       }
20:       else { /* Not Conflicting thread */
21:         StallUntilEndAnalysis(); /* Stall */
22:         Flags.OM = Normal; /* End of Analysis */
23:       }
24:     }
25:   }
26:   collect_on
27: }’)
```

Table 3: Resulting UNLOCK macro for SigRace.

cution modes. As an example, Table 3 shows the resulting final S_UNLOCK macro, which builds on top of SN_UNLOCK. The code is surrounded by *collect_off* and *collect_on* to prevent these accesses from polluting the signatures. If the OM flag indicates we are in Re-execution mode, we load the next timestamp from the old timestamp log into the TRT (Line 4), and spin until the GRT reaches the appropriate value (Line 5) (Section 3.3). If, instead, we are in Analysis mode, we have completed the execution of an epoch in this mode in one of the Conflicting threads. It is now time to analyze the record of traps observed (Line 8) (Section 3.4) and, depending on the outcome, proceed in Normal mode.

Irrespective of the mode, we then need to perform the unlock operation (Line 11) as was described in Table 2. Then, if we are in Re-execution mode, we update the GRT with the corresponding counter from the TRT (Line 13) and then check if the old timestamp log is empty. If so, we set the mode to Analysis (Line 15) and check the CT flag to see if this is a Conflicting thread. If so, we set up the Conflict Signature and continue execution (Section 3.4). Otherwise, the thread stalls until the Conflicting threads complete the analysis. After that, we return to Normal mode. Similar code is generated for the other synchronization constructs.

4.3 SigRace Virtualization

Previous discussions have largely used thread and processor interchangeably. In reality, SigRace has to function in an environment where threads migrate across processors and the number of threads and processors may be different. In this section, we consider this environment. We do it in three steps. First, we allow threads to migrate across processors but the number of threads and processors is the same (*Migration* environment). Second, we augment Migration to allow the number of threads to be different (and typically larger) than the number of processors; some threads are waiting for an available processor (*Multiplex* environment). Finally, we augment Multiplex to allow processors that support multiple hardware contexts (*Multithreaded* environment). We discuss each environment under Normal execution, and then consider the Re-execution and Analysis modes.

4.3.1 Enabling Thread Migration

Epoch timestamps and signatures belong to threads rather than processors. Consequently, in the Migration environment, the timestamp is saved when a thread is pre-empted and restored on the processor where the thread runs next. Signatures are not saved and restored because, on thread pre-emption, the currently-running block finishes. At that point, the signatures are sent to the RDM and then cleared.

The threads of a program have a statically-assigned *SigRaceID*, which goes from 0 to the number of threads in the program minus one. They use their SigRaceID to index into vector clocks of processors and array of BlockHistoryQueues in the RDM. Specifically, counter *i* in a vector clock belongs to the thread with SigRaceID = *i*, irrespective of which processor the thread is currently running on. Such thread always updates counter *i* in the vector clock of the processor it is running on. Moreover, signatures from that thread will always be dumped on BlockHistoryQueue[*i*] in the RDM.

In this environment, the hardware in Figure 5 is affected as follows. First, the components in Figure 5(b) belong to a thread. Consequently, the operating system saves and restores them on context switch — except for the signatures and the Committed Instruction Counter, which are cleared. Second, the RDM in Figure 5(a) includes a new hardware structure. It is an indirection table called the *CoreToThread* table. This table has as many entries as cores in the chip. It contains the mapping between core number and SigRaceID of the thread currently running on the core. The operating system updates the table on context switches. During execution, when the RDM receives a message from core *j*, the hardware reads CoreToThread[*j*]. It then uses the value read, say *i*, to store signatures and timestamp in BlockHistoryQueue[*i*].

4.3.2 *Different Thread & Processor Numbers*

The hardware for the Multiplex environment extends the one for Migration by supporting a range of SigRaceID values larger than the number of cores. Specifically, each vector clock in processors and each (software) timestamp field in sychronization variables is sized up to have as many counters as the maximum range of SigRaceID (Figure 6(a)). Similarly, the RDM has as many BlockHistoryQueues as the maximum range of SigRaceID, and the width of the CoreToThread table is increased accordingly (Figure 6(b)).

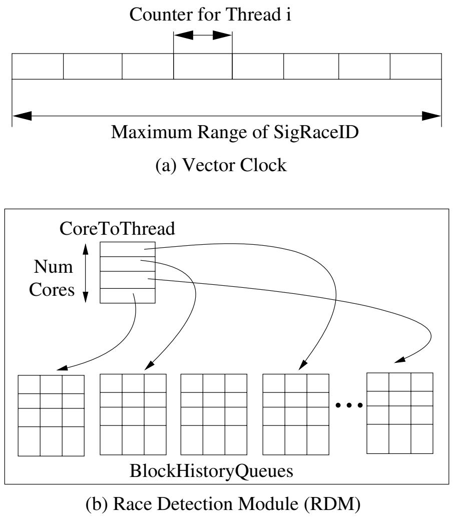


Figure 6: Supporting more threads than cores.

Before a program runs, it declares the number of threads that it will use, and the hardware and software structures mentioned are sized accordingly. While the program runs, the RDM intersects an incoming signature message against all BlockHistoryQueues — even those that belong to threads that are currently not running.

4.3.3 *Enabling Multiple Contexts per Processor*

The Multithreaded environment extends the Multiplex one in that each hardware context in a processor counts as an additional virtual core. This requires increasing the number of counters in the vector clocks and in the timestamp fields of synchronization variables, and the number of BlockHistoryQueues in the RDM.

Each hardware context has a copy of the hardware shown in Figure 5(b). Moreover, the messages that processors send to the RDM have to include both the core ID and the hardware context ID within the core. Only then can the RDM identify the appropriate BlockHistoryQueue to update.

4.3.4 *Re-Execution and Analysis Modes*

In all three environments described, threads are re-executed without any scheduling constraints. Specifically, Re-execution does not need to reproduce the thread schedule followed during the Normal execution. All that is required is that the order of successful synchronization operations be the same as in the Normal execution. This is ensured by reading the timestamp log from memory (Section 3.3.2) and enforcing it. At worst, in the Multiplex environment, performance may suffer because a thread that owns a critical lock may be temporarily not scheduled, preventing other thread from making progress.

In addition, re-execution does not need to reproduce the same block sizes as in the Normal execution. The reason is that Re-execution brings the threads to the beginning of epochs, rather than to specific blocks within epochs.

The checkpointing support described in Section 3.3.1 can still be used. Such support is able to return the memory state of the whole machine to a certain point in the past — without knowing about the number of threads in the program or how they were scheduled. If, however, it is desired to checkpoint only one of several applications that may be running, a different, application-level checkpoint design is needed. Such a design is outside this paper's scope.

As expected from the discussion on the Normal execution mode, there are a few structures used during Re-execution that need to change. First, the TRT (Figure 5(b)) is thread-private, and is saved and restored on context switch. In addition, the TRT and GRT have as many counters as the range of SigRaceIDs in the program. Moreover, threads use their SigRaceID to index into the TRT register, irrespective of what core they are currently running on.

Finally, the Analysis mode requires no change, since only the conflicting threads are participating in the execution. Both the Conflict Signature and the Conflict Thread structures (Figure 5(b)) are thread-private variables and the hardware saves and restores them on context switch.

5. EVALUATION

To evaluate SigRace, we consider four issues: (1) the signature configuration, which determines the number of false positives, (2) the block size and number of entries in each BlockHistoryQueue[i], which determine the window of monitored execution, (3) the effectiveness of SigRace in detecting data races, and (4) the overheads of SigRace. In the following, we first overview the experimental setup and then consider each issue in turn.

5.1 Experimental Setup

Since we are interested in the high-level parameters of SigRace, we use the PIN [10] binary instrumentation tool to design a simulator of the SigRace hardware, and run the applications on a real 8-processor shared-memory machine. This approach has the benefit of execution-driven simulation without incurring the slow speeds of typical cycle-accurate simulators. Table 4 shows the default parameters used in the simulation.

Num. of processors: 8 L1 size: 32 Kbytes L1 line size: 64 bytes Coh. protocol: MESI Checkpt. interval: 1 M committed instr./proc.	Timestamp size: 8 x 20 = 160 bits Sig. size: 2 Kbits each R and W Block size: 2,000 committed instr. BlockHistoryQueue[i] size: 16 entries
Benchmarks: SPLASH2 kernels: FFT, Cholesky, LU SPLASH2 applications: Barnes, Volrend, Ocean, Radiosity, Raytrace, Water-ns, Water-spatial PARSEC kernels: Dedup, Streamcluster PARSEC applic: Blackscholes, Fluidanimate, Swaptions	

Table 4: Default parameters used in the evaluation.

We model an 8-core chip multiprocessor where 32-Kbyte L1 caches are connected in a multistage network and kept coherent with a MESI cache coherence protocol. The timestamp size is very conservatively set to 160 bits. The default values for the size of signatures, block, and BlockHistoryQueue[i] are set according to the sensitivity analyses presented later. We take periodic global checkpoints. A checkpoint is created as soon as a processor has committed 1 M instructions. We use the checkpointed information as a starting point of our Re-execution and Analysis algorithms.

We evaluate SigRace with the SPLASH2 and PARSEC [1] benchmarks. These benchmarks are representative of parallel workloads and exhibit a variety of memory access patterns. For SPLASH2, we use the default inputs, while for PARSEC, we use the *simmedium* input size. We report data for 10 SPLASH2 and 5 PARSEC benchmarks. As shown in Table 4, we separate them into SPLASH2 kernels, SPLASH2 applications, PARSEC kernels, and PARSEC applications.

5.2 Signature Configuration

We test multiple signature configurations, denoted as $B_i\_S_j$. We first partition the address into 2 portions. The possible configurations are the B_i in Table 5. Then, we use multiple Bloom filters in parallel using the *H3* hash function as in [23] — half of them process one portion while the other half the other. The configurations are the S_i in Table 6.

Configuration	Address Partition	
	LSB	USB
B_1	8	24
B_2	10	22
B_3	16	16

Table 5: Address partitions. LSB and USB stand for Lower and Upper Sliced Bits.

Configuration	# of Bloom Filters (*k*)	Bits per Bloom Filter (*n*)	Sig Size (*k* × *n*)
S_1	16	256	4Kbit
S_2	16	128	2Kbit
S_3	16	64	1Kbit
S_4	8	512	4Kbit
S_5	8	256	2Kbit
S_6	8	128	1Kbit

Table 6: Signature organizations.

We run the applications and count the number of signature intersections that indicate a collision while there is none. The ratio of this number over the total number of signature intersections is the false-positive rate. Figure 7(a) shows the average false-positive rate of the applications for our default parameters. In the rest of the paper, we use $B_2\_S_2$, where the false-positive rate is 1.57%.

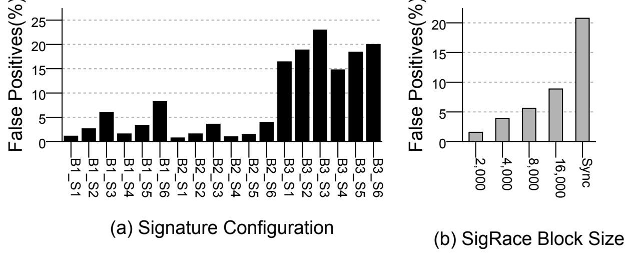


Figure 7: False positive rate versus signature configuration (a) and versus block size (b).

5.3 Block and BlockHistoryQueue[i] Size

If we choose a large SigRace block then, with the same BlockHistoryQueue[i] (BHQ[i]) size, we can monitor a larger instruction window for possible data races. However, as the block size increases, the signature false-positive rate also increases. Figure 7(b) shows the false-positive rate for different block sizes beyond our default of 2,000 committed instructions. *Sync* means terminating a block only at synchronizations. We see that larger blocks induce more false positives.

For a given block size, if we increase the number of entries in BHQ[i], we cover a larger instruction window. However, we have to do more signature operations and the BHQ takes more area.

To evaluate these issues, we run the applications with different numbers of entries in BHQ[i] and different block sizes. When the RDM checks an incoming signature against a BHQ[i], the hardware operates on each of the entries in the BHQ[i] until it finds a block that is a predecessor of the incoming one. If there is such a predecessor, then SigRace does not lose any race detection opportunity. We call this event a *Hit*. Otherwise, SigRace loses race detection opportunity beyond the oldest entry in BHQ[i]. We are interested in the execution window that starts at the previous checkpoint and ends at the block just before the oldest entry in BHQ[i]. We call it the *Lost Detection Window*.

Figure 8(a) shows the lost detection window as a percentage of the checkpoint interval, while Figure 8(b) shows the hit rate of a signature against a BHQ[i], and Figure 8(c) shows the number of timestamp comparisons in a BHQ[i] per signature until hitting in the BHQ[i] or exhausting all full BHQ[i] entries. All figures have the same X axis and share the same legend.

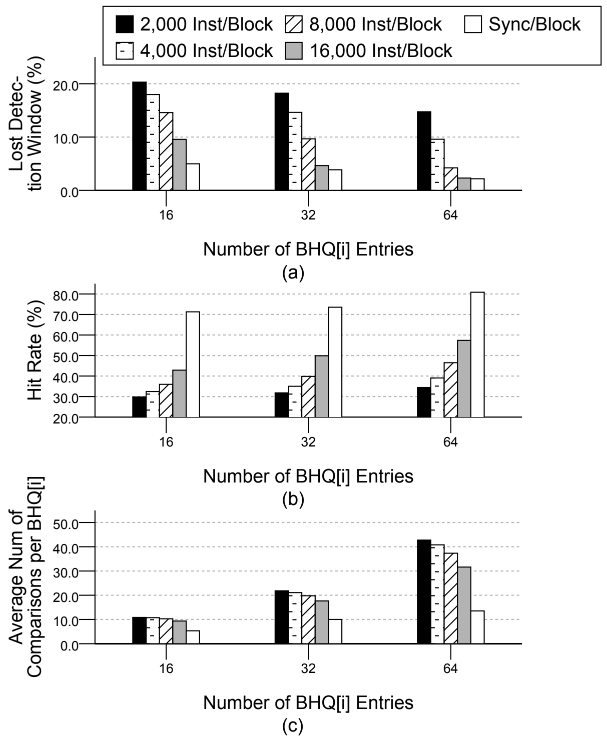


Figure 8: Lost detection window (a), hit rate (b), and number of timestamp comparisons (c) for different numbers of BHQ[i] entries and block size. All figures share the same legend.

Application	Finding Existing Races						Finding Injected Races				
	Ideal SigRace		SigRace		W-ReEnact		Racy	Static Races Found		Runs w/ Races Found	
	Stat	Dyn	Stat	Dyn	Stat	Dyn	Runs	SigRace	W-ReEnact	SigRace	W-ReEnact
FFT	–	–	–	–	–	–	25/25	600	150	25	25
Cholesky	16	19964	16	3539	16	388	3/25	2	2	1	1
LU	–	–	–	–	–	–	25/25	28	75	25	25
Barnes	11	4416	11	719	6	419	1/25	3	1	1	1
Volrend	27	26846	27	11607	18	6858	23/25	345	74	23	21
Ocean	1	29	1	29	1	6	7/25	8	8	7	7
Radiosity	15	59307	15	16951	12	14660	8/25	29	11	8	6
Raytrace	4	30	4	17	3	12	21/25	66	53	21	21
Water-ns	–	–	–	–	–	–	5/25	2	4	1	2
Water-spatial	8	82	4	27	2	3	3/25	6	6	3	3
Dedup	–	–	–	–	–	–	3/25	0	0	0	0
Streamcluster	13	68566	12	14307	12	436	6/25	7	2	5	2
Blackscholes	–	–	–	–	–	–	0/25	0	0	0	0
Fluidanimate	–	–	–	–	–	–	12/25	95	90	12	12
Swaptions	–	–	–	–	–	–	–	–	–	–	–
Total	95	179240	90	47196	70	22782	142/350	1191	476	132	126

Table 7: Effectiveness of SigRace and ReEnact with per-word timestamps in finding existing races and injected races.

We see that, as the number of BHQ[i] entries increases, the lost detection window decreases (Figure 8(a)) and the hit rate increases (Figure 8(b)). However, we have to do more timestamp comparisons until a hit or BHQ[i] exhaustion (Figure 8(c)), and the BHQ takes more area. On the other hand, for a fixed number of BHQ[i] entries, as the block size increases, we lose less window (Figure 8(a)), the hit rate increases (Figure 8(b)) and the number of comparisons decreases (Figure 8(c)) — however, we saw in Figure 7(b) that false positives increase. Overall, we choose as default a block size of 2,000 committed instructions and 16 entries in BHQ[i]. This leads to an average of 20% loss in detection window.

5.4 SigRace Effectiveness

5.4.1 Data Race Detection

To assess SigRace's effectiveness, we use it to find (i) existing data races in our applications and (ii) races that we inject in the applications. We also simulate a cache-based race detector, namely a version of ReEnact [19] with per-word timestamps (*W-ReEnact*). Table 7 shows the results.

Columns 2-7 (*Finding Existing Races*) list the number of races found by *Ideal Sigrace*, SigRace, and W-ReEnact. Ideal SigRace is a SigRace where each BHQ[i] keeps information for *all* the blocks between consecutive checkpoints — rather than for 16 blocks as in SigRace. Races are identified by the two instructions involved in the race and the address accessed. The table counts both static and dynamic races. Dynamic races are the dynamic instances of static races.

The table shows that 8 of the applications have data races. These races include, for example, reads of shared structures outside a critical section before accessing them inside the critical section. They are likely to be all benign races. However, we believe that it is important for any race detector to detect even benign races. This is because, often, benign races are a symptom that the code has a bug or something that the programmer does not understand. In any case, as described in Section 4.2, if the programmer wants SigRace to skip checking for these races, he can mark the code with *collect_off*.

The table shows that SigRace detects 90 static and 47,000 dynamic races. Compared to W-ReEnact, SigRace detects on average 29% more static races and 107% more dynamic races. SigRace's substantially higher effectiveness is due to its ability to monitor a longer window of program at a time. Finally, compared to Ideal SigRace, SigRace detects on average 95% of the static races and 26% of the dynamic ones.

We also inject races. For each application, we perform 25 runs. In each run, we randomly eliminate one dynamic lock-unlock pair or one dynamic barrier. Since the Swaptions code synchronizes with fork/joins, we could not subject it to this experiment. While these are contrived examples, they provide some insight.

Columns 8-12 (*Finding Injected Races*) show the detection capability of SigRace and W-ReEnact. Column 8 (*Racy Runs*) shows the fraction of those 25 runs that actually created races. Then, Columns 9-10 show the number of static races found by SigRace and W-ReEnact, respectively. We see that, on average, SigRace finds 150% more static races than W-ReEnact. This again shows the higher effectiveness of SigRace. Interestingly, there are two applications where W-ReEnact finds more races (LU and Water-ns). This is because, while SigRace typically monitors a longer program window, there are cases when lines remain in the caches for a long time. In this case, W-ReEnact can detect racing accesses that are far apart in the code (over 50,000 instructions apart in these examples). In general, it can be argued that races where the accesses are far apart are least dangerous, since the chances that these accesses appear in reverse order in a different run are lower. Finally, Columns 11-12 show the number of runs in which SigRace and W-ReEnact found at least one race. Again, the number for SigRace is higher.

5.4.2 Opportunity to Detect Data Races

SigRace has an advantage when addresses are in BHQ[.] and not in caches, while W-ReEnact has an edge in the opposite case. In this section, we estimate the frequency of each case. For simplicity, in this experiment only, signatures encode line addresses.

Of all the cache lines with shared data being displaced or invalidated from a cache, Figure 9(a) shows the fraction whose address is strictly present (not just due to aliasing) in the corresponding BHQ[i]. The figure shows the average for different cache sizes and application sets. For the 32KB default cache, the weighted average fraction is ≈59%. Then, Figure 9(b) shows the number of displacements or invalidations of lines with shared data per million instructions executed. For the 32KB default cache, the weighted average can be shown to be ≈2,800. Overall, roughly speaking, compared to SigRace, W-ReEnact loses detection opportunity for 0.59×2,800=1,652 lines per million instructions.

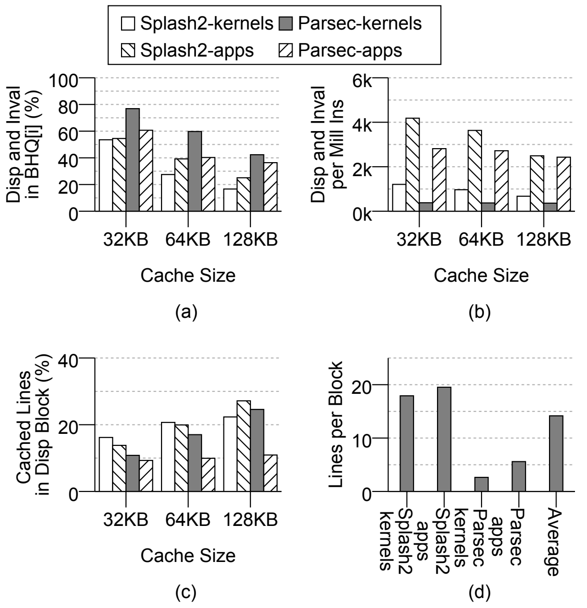


Figure 9: Opportunities for SigRace and W-ReEnact to detect races. Charts (a), (b), and (c) share the same legend.

Given a block being displaced from a BHQ[i], Figure 9(c) shows the fraction of addresses in the block's signatures that are not anywhere else in BHQ[i] and that are in the cache. For the 32KB cache, the weighted average fraction is ≈13%. Figure 9(d) shows the number of addresses of lines with shared data that are encoded in the signatures of one block. This number is on average 14. Overall, since SigRace executes ≈500 blocks per million instructions, compared to W-ReEnact, SigRace loses detection opportunity for 0.13×14×500=910 lines per million instructions. While these numbers give approximate information only, they show W-ReEnact loses more opportunities.

5.5 SigRace Overheads

We estimate the instruction, SRAM memory, bandwidth, and checkpointing overheads of SigRace. To estimate the instruction overhead, we run each application until the first true data race is fully analyzed. In the process, some false positives may occur. We count as instruction overhead all the instructions executed in Re-execution and Analysis modes to characterize the true data race and all the false positives found from the beginning of the program until that point. We stop after analyzing the first true race because then the programmer would stop execution. If the application has no true data race, we insert one in a random location.

Figure 10(a) shows the resulting instruction overhead as a percentage of committed instructions. The average bar is the mean of all the applications. The overhead depends on several things, most notably how far from the previous checkpoint is the conflict detected, and the rate of false positives. We see that, on average, the instruction overhead due to re-execution is 22%. About two thirds of it is caused by false positives.

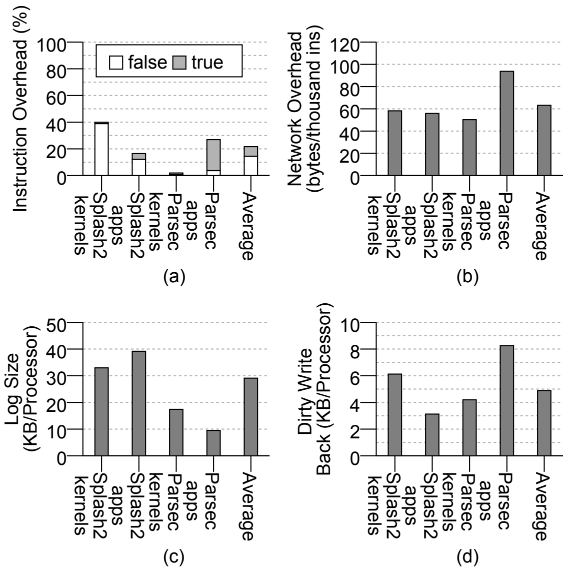


Figure 10: Instruction (a), bandwidth (b), and checkpoint-related (c and d) overheads.

From Figure 5, we see that the main SRAM memory overhead of SigRace per processor includes: a 16-entry BHQ[i] in the RDM (each entry containing a timestamp and a R and W signature), one extra timestamp and R and W signatures, the TRT, and the Conflict signature. Since timestamps are 160 bits and signatures 2K bits, this results in 8512 bytes in the RDM and 808 bytes in the cache hierarchy — independently of the cache size.

To compute the bandwidth overhead of SigRace, we count how many bytes of timestamp-signature messages (compressed) are deposited on the network. Figure 10(b) shows such number per 1,000 instructions committed. We see that, on average, the bandwidth overhead is 63 bytes per thousand committed instructions.

Finally, we measure some overheads of checkpointing every 1M instructions. As per Section 3.3.1, the memory controller saves the value overwritten by every first memory update. Figure 10(c) shows that, on average, this amounts to 29KB of log per processor between checkpoints. Also, at the point of checkpoint, the dirty lines in the cache are written back. As shown in Figure 10(d), this corresponds to, on average, 4.8KB of writebacks per processor.

6. CONCLUSIONS AND FUTURE WORK

This paper proposed SigRace, a novel approach to hardware-assisted data race detection that overcomes shortcomings of previous hardware proposals. To detect races, SigRace does not rely on cache state or coherence protocol messages. Instead, it relies on hardware address signatures. With SigRace, there are no changes to the cache or the cache coherence protocol messages, and there are no critical-path operations performed on local/external access to the cache. Moreover, lines can be displaced or invalidated from caches without affecting SigRace's ability to detect data races.

We presented the architecture of SigRace, an implementation, and its software interface. Application code is unmodified. Our experiments showed that SigRace is significantly more effective than a state-of-the-art conventional hardware-assisted race detector. SigRace found on average 29% more static races and 107% more dynamic races. Moreover, if we inject data races, SigRace found 150% more static races than the conventional scheme. Finally, SigRace had an average instruction overhead due to re-execution of 22%, a bandwidth overhead of 63 bytes per thousand committed instructions, and an SRAM memory overhead of ≈9KB per processor.

We are continuing our work in two main directions. The first one involves eliminating or minimizing the need to perform checkpointing — possibly at the cost of more re-execution. The second one involves improving the scalability of the happened-before clocks and RDM design.

7. ACKNOWLEDGMENTS

We thank the anonymous reviewers and the I-ACOMA group members for their comments. This work was supported in part by the National Science Foundation under grants CNS-0720593 and CCR-0325603; Intel and Microsoft under the Universal Parallel Computing Research Center; and gifts from IBM and Sun Microsystems. Suárez was supported by the Gobierno de Aragón under grant "gaZ: Grupo Consolidado de Investigación"; Spanish Ministry of Education and Science under contracts TIN2007-66423 and Consolider CSD2007-00050; and European Union Network of Excellence HiPEAC-2 (FP7/ICT 217068).

8. REFERENCES

[1] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Comm. of the ACM*, 13(7):422–426, 1970.

[3] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk enforcement of sequential consistency. In *International Symposium on Computer Architecture*, June 2007.

[4] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *International Symposium on Computer Architecture*, June 2006.

[5] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Programming Language Design and Implementation*, June 2002.

[6] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, 1998.

[7] C. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, 1991.

[8] Intel Corporation. Intel Thread Checker. http://www.intel.com, 2008.

[9] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM*, 21(7):558–565, 1978.

[10] C.-K. Luk et al. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, June 2005.

[11] E. Lusk, J. Boyle, R. Butler, T. Disz, B. Glickfeld, R. Overbeek, J. Patterson, and R. Stevens. *Portable programs for parallel processors*. Holt, Rinehart & Winston, 1988.

[12] S. L. Min and J.-D. Choi. An efficient cache-based access anomaly detection scheme. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[13] C. C. Minh et al. An effective hybrid transactional memory system with strong isolation guarantees. In *International Symposium on Computer Architecture*, June 2007.

[14] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *Programming Language Design and Implementation*, June 2007.

[15] R. H. B. Netzer and B. P. Miller. Detecting data races in parallel program executions. In *In Workshop on Advances in Languages and Compilers for Parallel Computing*, 1990.

[16] R. H. B. Netzer and B. P. Miller. Improving the accuracy of data race detection. In *Principles and Practice of Parallel Programming*, April 1991.

[17] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Principles and Practice of Parallel Programming*, June 2003.

[18] M. Prvulovic. CORD: Cost-effective (and nearly overhead-free) order-recording and data race detection. In *International Symposium on High-Performance Computer Architecture*, February 2006.

[19] M. Prvulovic and J. Torrellas. ReEnact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *International Symposium on Computer Architecture*, June 2003.

[20] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *International Symposium on Computer Architecture*, May 2002.

[21] F. Qin, J. Tucek, Y. Zhou, and J. Sundaresan. Rx: Treating bugs as allergies—a safe method to survive software failures. *ACM Transactions on Computer Systems*, 25(3):7, 2007.

[22] M. Ronsse and K. De Bosschere. RecPlay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17(2):133–152, 1999.

[23] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam. Implementing signatures for transactional memory. In *International Symposium on Microarchitecture*, December 2007.

[24] S. Savage et al. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[25] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conference*, June 2004.

[26] Sun Microsystems. Sun Studio Thread Analyzer. http://developers.sun.com/sunstudio, 2007.

[27] C. von Praun and T. R. Gross. Object race detection. In *Object-Oriented Programming, Systems, Languages, and Applications*, October 2001.

[28] L. Yen et al. LogTM-SE: Decoupling hardware transactional memory from caches. In *International Symposium on High Performance Computer Architecture*, February 2007.

[29] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *Symposium on Operating Systems Principles*, October 2005.

[30] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-assisted lockset-based race detection. In *International Symposium on High Performance Computer Architecture*, February 2007.

DOI:10.1145/1610252.1610271

Easing the programmer's burden does not compromise system performance or increase the complexity of hardware implementation.

BY JOSEP TORRELLAS, LUIS CEZE, JAMES TUCK, CALIN CASCAVAL, PABLO MONTESINOS, WONSUN AHN, AND MILOS PRVULOVIC

The Bulk Multicore Architecture for Improved Programmability

MULTICORE CHIPS AS commodity architecture for platforms ranging from handhelds to supercomputers herald an era when parallel programming and computing will be the norm. While the computer science and engineering community has periodically focused on advancing the technology for parallel processing,[8] this time around the stakes are truly high, since there is no obvious route to higher performance other than through parallelism. However, for parallel computing to become widespread, breakthroughs are needed in all layers of the computing stack, including languages, programming models, compilation and runtime software, programming and debugging tools, and hardware architectures.

At the hardware-architecture layer, we need to change the way multicore architectures are designed. In the past, architectures were designed primarily for performance or for energy efficiency. Looking ahead, one of the top priorities must be for the architecture to enable a programmable environment. In practice, programmability is a notoriously difficult metric to define and measure. At the hardware-architecture level, programmability implies two things: First, the architecture is able to attain high efficiency while relieving the programmer from having to manage low-level tasks; second, the architecture helps minimize the chance of (parallel) programming errors.

In this article, we describe a novel, general-purpose multicore architecture—the Bulk Multicore—we designed to enable a highly programmable environment. In it, the programmer and runtime system are relieved of having to manage the sharing of data thanks to novel support for scalable hardware cache coherence. Moreover, to help minimize the chance of parallel-programming errors, the Bulk Multicore provides to the software high-performance sequential memory consistency and also introduces several novel hardware primitives. These primitives can be used to build a sophisticated program-development-and-debugging environment, including low-overhead data-race detection, deterministic replay of parallel programs, and high-speed disambiguation of sets of addresses. The primitives have an overhead low enough to always be "on" during production runs.

The key idea in the Bulk Multicore is twofold: First, the hardware automatically executes all software as a series of atomic blocks of thousands of dynamic instructions called Chunks. Chunk execution is invisible to the software and, therefore, puts no restriction on the programming language or model. Second, the Bulk Multicore introduces the use of Hardware Address Signatures as a low-overhead mechanism to ensure atomic and isolated execution of chunks and help

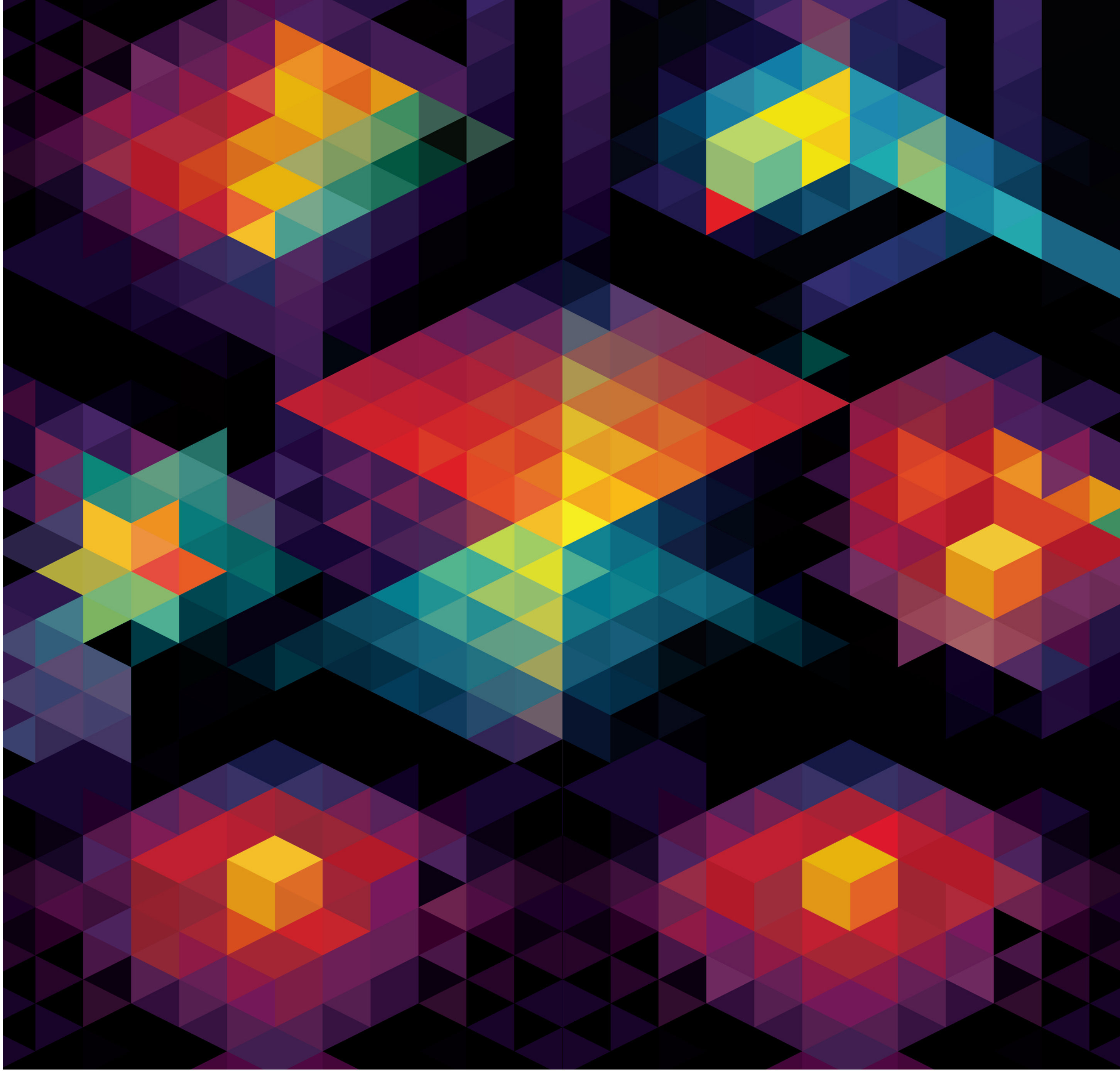


ILLUSTRATION BY ANDY GILMORE

maintain hardware cache coherence.

The programmability advantages of the Bulk Multicore do not come at the expense of performance. On the contrary, the Bulk Multicore enables high performance because the processor hardware is free to aggressively reorder and overlap the memory accesses of a program within chunks without risk of breaking their expected behavior in a multiprocessor environment. Moreover, in an advanced Bulk Multicore design where the compiler observes the chunks, the compiler can further improve performance by heavily optimizing the instructions within each chunk. Finally, the Bulk Multicore organization decreases hardware design complexity by freeing processor designers from having to worry about many corner cases that appear when designing multiprocessors.

Architecture

The Bulk Multicore architecture eliminates one of the traditional tenets of processor architecture, namely the need to commit instructions in order, providing the architectural state of the processor after every single instruction. Having to provide such state in a multiprocessor environment—even if no other processor or unit in the machine needs it—contributes to the complexity of current system designs. This is because, in such an environment, memory-system accesses take many cycles, and multiple loads and stores from both the same and different processors overlap their execution.

In the Bulk Multicore, the default execution mode of a processor is to commit chunks of instructions at a time.[2] A chunk is a group of dynamically contiguous instructions (such as 2,000 instructions). Such a "chunked" mode of execution and commit is a hardware-only mechanism, invisible to the software running on the processor. Moreover, its purpose is not to parallelize a thread, since the chunks in a thread are not distributed to other processors. Rather, the purpose is to improve programmability and performance.

Each chunk executes on the processor atomically and in isolation. Atomic execution means that none of the chunk's actions are made visible to the rest of the system (processors or main memory) until the chunk completes and commits. Execution in isolation means that if the chunk reads a location and (before it commits) a second chunk in another processor that has written to the location commits, then the local chunk is squashed and must re-execute.

To execute chunks atomically and in isolation inexpensively, the Bulk Multicore introduces hardware address signatures.[3] A signature is a register of ≈1,024 bits that accumulates hash-encoded addresses. Figure 1 outlines a simple way to generate a signature (see the sidebar "Signatures and Signature Operations in Hardware" for a deeper discussion). A signature, therefore, represents a set of addresses.

In the Bulk Multicore, the hardware automatically accumulates the addresses read and written by a chunk into a read (R) and a write (W) signature, respectively. These signatures are kept in a module in the cache hierarchy. This module also includes simple functional units that operate on signatures, performing such operations as signature intersection (to find the addresses common to two signatures) and address membership test (to find out whether an address belongs to a signature), as detailed in the sidebar.

Atomic chunk execution is supported by buffering the state generated by the chunk in the L1 cache. No update is propagated outside the cache while the chunk is executing. When the chunk completes or when a dirty cache line with address in the W signature must be displaced from the cache, the hardware proceeds to commit the chunk. A successful commit involves sending the chunk's W signature to the subset of sharer processors indicated by the directory[2] and clearing the local R and W signatures. The latter operation erases any record of the updates made by the chunk, though the written lines remain dirty in the cache.

The W signature carries enough information to both invalidate stale lines from the other coherent caches (using the δ signature operation on W, as discussed in the sidebar) and enforce that all other processors execute their chunks in isolation. Specifically, to enforce that a processor executes a chunk in isolation when the processor receives an incoming signature W_{inc}, its hardware intersects W_{inc} against the local R_{loc} and W_{loc} signatures. If any of the two intersections is not null, it means (conservatively) that the local chunk has accessed a data element written by the committing chunk. Consequently, the local chunk is squashed and then restarted.

Figure 2 outlines atomic and isolated execution. Thread *0* executes a chunk that writes variables *B* and *C*, and no invalidations are sent out. Signature W_0 receives the hashed addresses of *B* and *C*. At the same time, Thread *1* issues reads for *B* and *C*, which (by construction) load the non-

Signatures and Signature Operations in Hardware

Figure 1 in the main text shows a simple implementation of a signature. The bits of an incoming address go through a fixed permutation to reduce collisions and are then separated in bit-fields C_i. Each field is decoded and accumulated into a bit-field V_j in the signature. Much more sophisticated implementations are also possible.

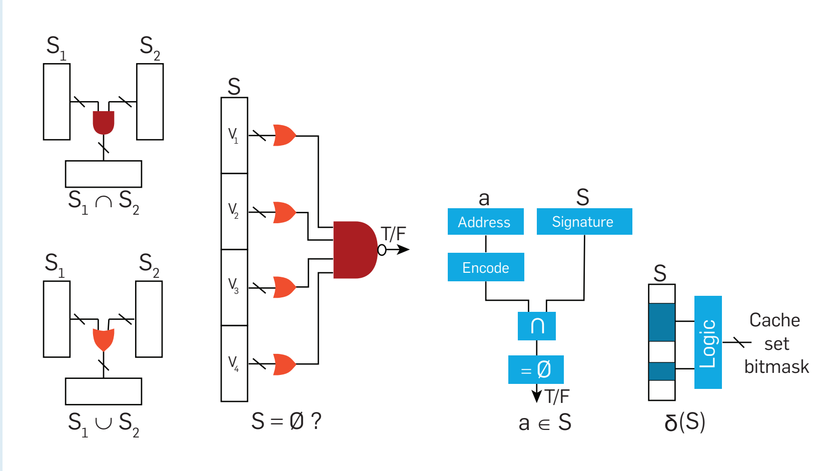
A module called the Bulk Disambiguation Module contains several signature registers and simple functional units that operate efficiently on signatures. These functional units are invisible to the instruction-set architecture. Note that, given a signature, we can recover only a superset of the addresses originally encoded into the signature. Consequently, the operations on signatures produce conservative results.

The figure here outlines five signature functional units: intersection, union, test for null signature, test for address membership, and decoding (δ). Intersection finds the addresses common to two signatures by performing a bit-wise AND of the two signatures. The resulting signature is empty if, as shown in the figure, any of its bit-fields contains all zeros. Union finds all addresses present in at least one signature through a bit-wise OR of the two signatures. Testing whether an address *a* is present (conservatively) in a signature involves encoding *a* into a signature, intersecting the latter with the original signature and then testing the result for a null signature.

Decoding (δ) a signature determines which cache sets can contain addresses belonging to the signature. The set bitmask produced by this operation is then passed to a finite-state machine that successively reads individual lines from the sets in the bitmask and checks for membership to the signature. This process is used to identify and invalidate all the addresses in a signature that are present in the cache.

Overall, the support described here enables low-overhead operations on sets of addresses.[3]

Operations on signatures.



speculative values of the variables—namely, the values before Thread *0*'s updates. When Thread *0*'s chunk commits, the hardware sends signature W_0 to Thread *1*, and W_0 and R_0 are cleared. At the processor where Thread *1* runs, the hardware intersects W_0 with the ongoing chunk's R_1 and W_1. Since $W_0 \cap R_1$ is not null, the chunk in Thread *1* is squashed.

The commit of chunks is serialized globally. In a bus-based machine, serialization is given by the order in which *W* signatures are placed on the bus. With a general interconnect, serialization is enforced by a (potentially distributed) arbiter module.[2] W signatures are sent to the arbiter, which quickly acknowledges whether the chunk can be considered committed.

Since chunks execute atomically and in isolation, commit in program order in each processor, and there is a global commit order of chunks, the Bulk Multicore supports sequential consistency (SC)[9] at the chunk level. As a consequence, the machine also supports SC at the instruction level. More important, it supports high-performance SC at low hardware complexity.

The performance of this SC implementation is high because (within a chunk) the Bulk Multicore allows memory access reordering and overlap and instruction optimization. As we discuss later, synchronization instructions induce no reordering constraint within a chunk.

Meanwhile, hardware-implementation complexity is low because memory-consistency enforcement is largely decoupled from processor structures. In a conventional processor that issues memory accesses out of order, supporting SC requires intrusive processor modifications. For example, from the time the processor executes a load to line L out of order until the load reaches its commit time, the hardware must check for writes to L by other processors—in case an inconsistent state was observed. Such checking typically requires sending, for each external coherence event, a signal up the cache hierarchy. The signal snoops the load queue to check for an address match. Additional modifications involve preventing cache displacements that could risk missing a coherence event. Consequently, load queues, L1 caches, and other critical processor components must be augmented with extra hardware.

Figure 1. A simple way to generate a signature.

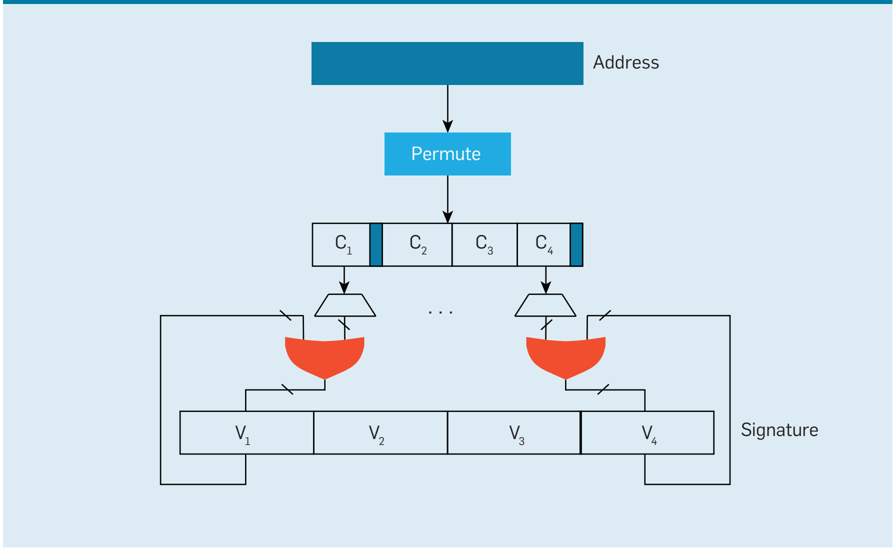
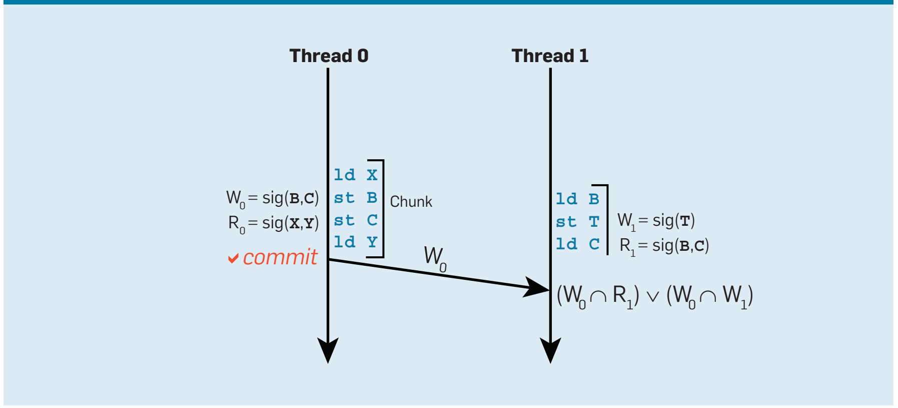


Figure 2. Executing chunks atomically and in isolation with signatures.



In the Bulk Multicore, SC enforcement and violation detection are performed with simple signature intersections outside the processor core. Additionally, caches are oblivious to what data is speculative, and their tag and data arrays are unmodified.

Finally, note that the Bulk Multicore's execution mode is not like transactional memory.[6] While one could intuitively view the Bulk Multicore as an environment with transactions occurring all the time, the key difference is that chunks are dynamic entities, rather than static, and invisible to the software.

High Programmability

Since chunked execution is invisible to the software, it places no restriction on programming model, language, or runtime system. However, it does enable a highly programmable environment by virtue of providing two features: high-performance SC at the hardware level and several novel hardware primitives that can be used to build a sophisticated program-development-and-debugging environment.

Unlike current architectures, the Bulk Multicore supports high-performance SC at the hardware level. If we generate code for the Bulk Multicore using an SC compiler (such as the BulkCompiler[1]), we attain a high-performance, fully SC platform. The resulting platform is highly programmable for several reasons. The first is that debugging concurrent programs with data races would be much easier. This is because the possible outcomes of the memory accesses involved in the bug would be easier to reason about, and the debugger would in fact be able to reproduce the buggy interleaving. Second, most existing software correctness tools (such as Microsoft's CHESS[14]) assume SC. Verifying software correctness under SC is already difficult, and the state space balloons if non-SC interleavings need to be verified as well. In the next few years, we expect that correctness-verification tools will play a larger role as more parallel software is developed. Using them in combination with an SC platform would make them most effective.

A final reason for the programmability of an SC platform is that it would make the memory model of safe languages (such as Java) easier to understand and verify. The need to provide safety guarantees and enable performance at the same time has resulted in an increasingly complex and unintuitive memory model over the years. A high-performance SC memory model would trivially ensure Java's safety properties related to memory ordering, improving its security and usability.

The Bulk Multicore's second feature is a set of hardware primitives that can be used to engineer a sophisticated program-development-and-debugging environment that is always "on," even during production runs. The key insight is that chunks and signatures free development and debugging tools from having to record or be concerned with individual loads and stores. As a result, the amount of bookkeeping and state required by the tools is substantially reduced, as is the time overhead. Here, we give three examples of this benefit in the areas of deterministic replay of parallel programs, data-race detection, and high-speed disambiguation of sets of addresses.

Note, too, that chunks provide an excellent primitive for supporting popular atomic-section-based techniques for programmability (such as thread-level speculation[17] and transactional memory[6]).

Deterministic replay of parallel programs with practically no log. Hardware-assisted deterministic replay of parallel programs is a promising technique for debugging parallel programs. It involves a two-step process.[20] In the recording step, while the parallel program executes, special hardware records into a log the order of data dependences observed among the multiple threads. The log effectively captures the "interleaving" of the program's threads. Then, in the replay step, while the parallel program is re-executed, the system enforces the interleaving orders encoded in the log.

The Bulk Multicore supports high-performance sequential memory consistency at low hardware complexity.

In most proposals of deterministic replay schemes, the log stores individual data dependences between threads or groups of dependences bundled together. In the Bulk Multicore, the log must store only the total order of chunk commits, an approach we call DeLorean.[13] The logged information can be as minimalist as a list of committing-processor IDs, assuming the chunking is performed in a deterministic manner; therefore, the chunk sizes can be deterministically reproduced on replay. This design, which we call OrderOnly, reduces the log size by nearly an order of magnitude over previous proposals.

The Bulk Multicore can further reduce the log size if, during the recording step, the arbiter enforces a certain order of chunk commit interleaving among the different threads (such as by committing one chunk from each processor round robin). In this case of enforced chunk-commit order, the log practically disappears. During the replay step, the arbiter reinforces the original commit algorithm, forcing the same order of chunk commits as in the recording step. This design, which we call PicoLog, typically incurs a performance cost because it can force some processors to wait during recording.

Figure 3a outlines a parallel execution in which the boxes are chunks and the arrows are the observed cross-thread data dependences. Figure 3b shows a possible resulting execution log in OrderOnly, while Figure 3c shows the log in PicoLog.

Data-race detection at production-run speed. The Bulk Multicore can support an efficient data-race detector based on the "happens-before" method[10] if it cuts the chunks at synchronization points, rather than at arbitrary dynamic points. Synchronization points are easily recognized by hardware or software, since synchronization operations are executed by special instructions. This approach is described in ReEnact[16]; Figure 4 includes examples with a lock, flag, and barrier.

Each chunk is given a counter value called ChunkID following the happens-before ordering. Specifically, chunks in a given thread receive ChunkIDs that increase in program order. Moreover, a synchronization between two threads orders the ChunkIDs of the chunks involved in the synchronization. For example, in Figure 4a, the chunk in Thread *2* following the lock acquire (Chunk *5*) sets its ChunkID to be a successor of both the previous chunk in Thread *2* (Chunk *4*) and the chunk in Thread *1* that released the lock (Chunk *2*). For the other synchronization primitives, the algorithm is similar. For example, for the barrier in Figure 4c, each chunk immediately following the barrier is given a ChunkID that makes it a successor of all the chunks leading to the barrier.

Using ChunkIDs, we've given a partial ordering to the chunks. For example, in Figure 4a, Chunks *1* and *6* are ordered, but Chunks *3* and *4* are not. Such ordering helps detect data races that occur in a particular execution. Specifically, when two chunks from different threads are found to have a data-dependence at runtime, their two ChunkIDs are compared. If the ChunkIDs are ordered, this is not a data race because there is an intervening synchronization between the chunks. Otherwise, a data race has been found.

A simple way to determine when two chunks have a data-dependence is to use the Bulk Multicore signatures to tell when the data footprints of two chunks overlap. This operation, together with the comparison and maintenance of ChunkIDs, can be done with low overhead with hardware support. Consequently, the Bulk Multicore can detect data races without significantly slowing the program, making it ideal for debugging production runs.

Enhancing programmability by making signatures visible to software. Finally, a technique that improves programmability further is to make additional signatures visible to the software. This support enables inexpensive monitoring of memory accesses, as well as

Making Signatures Visible to Software

We propose that the software interact with some additional signatures through three main primitives:[18]

The first is to explicitly encode into a signature either one address (Figure 1a) or all addresses accessed in a code region (Figure 1b). The latter is enabled by the *bcollect* (begin collect) and *ecollect* (end collect) instructions, which can be set to collect only reads, only writes, or both.

The second primitive is to disambiguate the addresses accessed by the processor in a code region against a given signature. It is enabled by the *bdisamb.loc* (begin disambiguate local) and *edisamb.loc* (end disambiguate local) instructions (Figure 1c), and can disambiguate reads, writes, or both.

The third primitive is to disambiguate the addresses of incoming coherence messages (invalidations or downgrades) against a given local signature. It is enabled by the *bdisamb.rem* (begin disambiguate remote) and *edisamb.rem* (end disambiguate remote) instructions (Figure 1d) and can disambiguate reads, writes, or both. When disambiguation finds a match, the system can deliver an interrupt or set a bit.

Figure 2 includes three examples of what can be done with these primitives: Figure 2a shows how the machine inexpensively supports many watchpoints. The processor encodes into signature *Sig2* the address of variable *y* and all the addresses accessed in function *foo()*. It then watches all these addresses by executing *bdisamb.loc* on *Sig2*.

Figure 2b shows how a second call to a function that reads and writes memory in its body can be skipped. In the figure, the code calls function *foo()* twice with the same input value of *x*. To see if the second call can be skipped, the program first collects all addresses accessed by *foo()* in *Sig2*. It then disambiguates all subsequent accesses against *Sig2*. When execution reaches the second call to *foo()*, it can skip the call if two conditions hold: the first is that the disambiguation did not find a conflict; the second (not shown in the figure) is that the read and write footprints of the first *foo()* call do not overlap. This possible overlap is checked by separately collecting the addresses read in *foo()* and those written in *foo()* in separate signatures and intersecting the resulting signatures.

Finally, Figure 2c shows a way to detect data dependences between threads running on different processors. In the figure, *collect* encodes all addresses accessed in a code section into *Sig2*. Surrounding the collect instructions, the code places *disamb.rem* instructions to monitor if any remotely initiated coherence-action conflicts with addresses accessed locally. To disregard read-read conflicts, the programmer can collect the reads in a separate signature and perform remote disambiguation of only writes against that signature.

Figure 1. Primitives enabling software to interact with additional signatures: collection (a and b), local disambiguation (c), and remote disambiguation (d).

```
                      bcollect Sig1      bdisamb.loc Sig1      bdisamb.rem Sig1
                      x = ...            x = ...               x = ...
                      ... = y            ... = y               ... = y
Encode Addr, Sig1     ecollect Sig1      edisamb.loc Sig1      edisamb.rem Sig1
      (a)                  (b)                 (c)                   (d)
```

Figure 2. Using signatures to support data watchpoints (a), skip execution of functions (b), and detect data dependencies between threads running on different processors (c).

```
                                    bcollect Sig2
                                    foo(x)
                                    ecollect Sig2
encode &y, Sig2                     bdisamb.loc Sig2
bcollect Sig2        foo(x)         ... = y            bdisamb.rem Sig2
foo()                ... = y        z = ...            bcollect Sig2
ecollect Sig2        z = ...   →    edisamb.loc Sig2   ... = y
bdisamb.loc Sig2     foo(x)         if (conflict) {    z = ...
      .                             foo(x)             ecollect Sig2
      .                             }                  edisamb.rem Sig2
     (a)                   (b)                               (c)
```

novel compiler optimizations that require dynamic disambiguation of sets of addresses (see the sidebar "Making Signatures Visible to Software").

Reduced Implementation Complexity

The Bulk Multicore also has advantages in performance and in hardware simplicity. It delivers high performance because the processor hardware can reorder and overlap all memory accesses within a chunk—except, of course, those that participate in single-thread dependences. In particular, in the Bulk Multicore, synchronization instructions do not constrain memory access reordering or overlap. Indeed, fences inside a chunk are transformed into null instructions. Fences' traditional functionality of delaying execution until certain references are performed is useless; by construction, no other processor observes the actual order of instruction execution within a chunk.

Moreover, a processor can concurrently execute multiple chunks from the same thread, and memory accesses from these chunks can also overlap. Each concurrently executing chunk in the processor has its own R and W signatures, and individual accesses update the corresponding chunk's signatures. As long as chunks within a processor commit in program order (if a chunk is squashed, its successors are also squashed), correctness is guaranteed. Such concurrent chunk execution in a processor hides the chunk-commit overhead.

Bulk Multicore performance increases further if the compiler generates the chunks, as in the BulkCompiler.[1] In this case, the compiler can aggressively optimize the code within each chunk, recognizing that no other processor sees intermediate states within a chunk.

Finally, the Bulk Multicore needs simpler processor hardware than current machines. As discussed earlier, much of the responsibility for memory-consistency enforcement is taken away from critical structures in the core (such as the load queue and L1 cache) and moved to the cache hierarchy where signatures detect violations of SC.[2] For example, this property could enable a new environment in which cores and accelerators are designed without concern for how to satisfy a particular set of access-ordering constraints. This ability allows hardware designers to focus on the novel aspects of their design, rather than on the interaction with the target machine's legacy memory-consistency model. It also motivates the development of commodity accelerators.

Figure 3. Parallel execution in the Bulk Multicore (a), with a possible OrderOnly execution log (b) and PicoLog execution log (c).

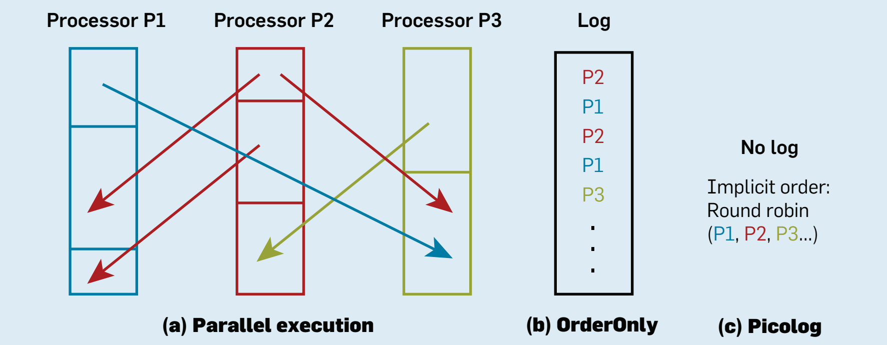
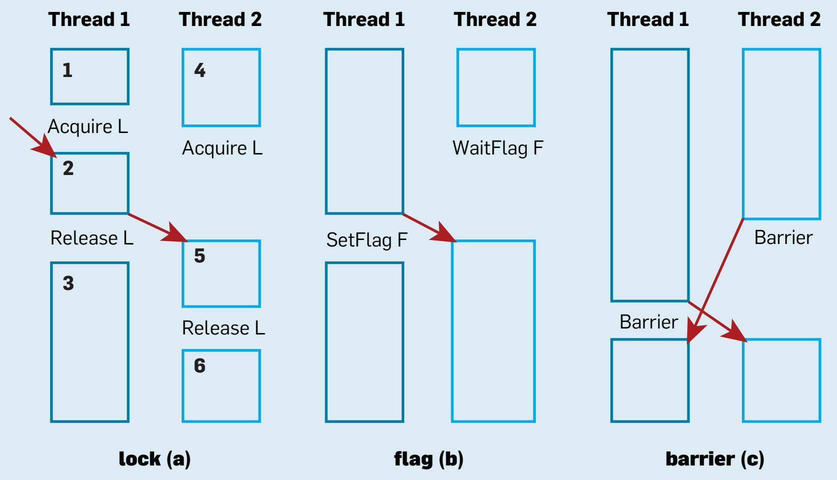


Figure 4. Forming chunks for data-race detection in the presence of a lock (a), flag (b), and barrier (c).



Related Work

Numerous proposals for multiprocessor architecture designs focus on improving programmability. In particular, architectures for thread-level speculation (TLS)[17] and transactional memory (TM)[6] have received significant attention over the past 15 years. These techniques share key primitive mechanisms with the Bulk Multicore, notably speculative state buffering and undo and detection of cross-thread conflicts. However, they also have a different goal, namely simplify code parallelization by parallelizing the code transparently to the user software in TLS or by annotating the user code with constructs for mutual exclusion in TM. On the other hand, the Bulk Multicore aims to provide a broadly usable architectural platform that is easier to program for while delivering advantages in performance and hardware simplicity.

Two architecture proposals involve processors continuously executing blocks of instructions atomically and in isolation. One of them, called Transactional Memory Coherence and Consistency (TCC),[5] is a TM environment with transactions occurring all the time. TCC mainly differs from the Bulk Multicore in that its transactions are statically specified in the code, while chunks are created dynamically by the hardware. The second proposal, called Implicit Transactions,[19] is a multiprocessor environment with checkpointed processors that regularly take checkpoints. The instructions executed between checkpoints constitute the equivalent of a chunk. No detailed implementation of the scheme is presented.

Automatic Mutual Exclusion (AME)[7] is a programming model in which a program is written as a group of atomic fragments that serialize in some manner. As in TCC, atomic sections in AME are statically specified in the code, while the Bulk Multicore chunks are hardware-generated dynamic entities.

The signature hardware we've introduced here has been adapted for use in TM (such as in transaction-footprint collection and in address disambiguation[12,21]).

Several proposals implement data-race detection, deterministic replay of multiprocessor programs, and other debugging techniques discussed here without operating in chunks.[4,11,15,20] Comparing their operation to chunk operation is the subject of future work.

Future Directions

The Bulk Multicore architecture is a novel approach to building shared-memory multiprocessors, where the whole execution operates in atomic chunks of instructions. This approach can enable significant improvements in the productivity of parallel programmers while imposing no restriction on the programming model or language used.

At the architecture level, we are examining the scalability of this organization. While chunk commit requires arbitration in a (potentially distributed) arbiter, the operation in chunks is inherently latency tolerant. At the programming level, we are examining how chunk operation enables efficient support for new program-development and debugging tools, aggressive autotuners and compilers, and even novel programming models.

Acknowledgments

We would like to thank the many present and past members of the I-acoma group at the University of Illinois who contributed through many discussions, seminars, and brainstorming sessions. This work is supported by the U.S. National Science Foundation, Defense Advanced Research Projects Agency, and Department of Energy and by Intel and Microsoft under the Universal Parallel Computing Research Center, Sun Microsystems under the University of Illinois OpenSPARC Center of Excellence, and IBM.

References

1. Ahn, W., Qi, S., Lee, J.W., Nicolaides, M., Fang, X., Torrellas, J., Wong, D., and Midkiff, S. BulkCompiler: High-performance sequential consistency through cooperative compiler and hardware support. In *Proceedings of the International Symposium on Microarchitecture* (New York City, Dec. 12–16). IEEE Press, 2009.
2. Ceze, L., Tuck, J., Montesinos, P., and Torrellas, J. BulkSC: Bulk enforcement of sequential consistency. In *Proceedings of the International Symposium on Computer Architecture* (San Diego, CA, June 9–13). ACM Press, New York, 2007, 278–289.
3. Ceze, L., Tuck, J., Cascaval, C., and Torrellas, J. Bulk disambiguation of speculative threads in multiprocessors. In *Proceedings of the International Symposium on Computer Architecture* (Boston, MA, June 17–21). IEEE Press, 2006, 227–238.
4 Choi, J., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., and Sridharan, M. Efficient and precise data-race detection for multithreaded object-oriented programs. In *Proceedings of the Conference on Programming Language Design and Implementation* (Berlin, Germany, June 17-19). ACM Press, New York, 2002, 258–269.
5 Hammond, L., Wong, V., Chen, M., Carlstrom, B.D., Davis, J.D., Hertzberg, B., Prabhu, M.K., Wijaya, H., Kozyrakis, C., and Olukotun, K. Transactional memory coherence and consistency. In *Proceedings of the International Symposium on Computer Architecture* (München, Germany, June 19–23). IEEE Press, 2004, 102–113.
6. Herlihy M. and Moss, J.E.B. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the International Symposium on Computer Architecture* (San Diego, CA, May 16–19). IEEE Press, 1993, 289–300.
7 Isard, M. and Birrell, A. Automatic mutual exclusion. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (San Diego, CA, May 7–9). USENIX, 2007.
8. Kuck, D. Facing up to software's greatest challenge: Practical parallel processing. *Computers in Physics 11*, 3 (1997).
9. Lamport, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers C-28*, 9 (Sept. 1979), 690–691.
10. Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558–565.
11. Lu, S., Tucek, J., Qin, F., and Zhou, Y. AVIO: Detecting atomicity violations via access interleaving invariants. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, Oct. 21–25). ACM Press, New York, 2006, 37–48.
12. Minh, C., Trautmann, M., Chung, J., McDonald, A., Bronson, N., Casper, J., Kozyrakis, C., and Olukotun, K. An effective hybrid transactional memory with strong isolation guarantees. In *Proceedings of the International Symposium on Computer Architecture* (San Diego, CA, June 9–13). ACM Press, New York, 2007, 69–80.
13. Montesinos, P., Ceze, L., and Torrellas, J. DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proceedings of the International Symposium on Computer Architecture* (Beijing, June 21–25). IEEE Press, 2008, 289–300.
14. Musuvathi, M. and Qadeer, S. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the Conference on Programming Language Design and Implementation* (San Diego, CA, June 10–13). ACM Press, New York, 2007, 446–455.
15. Narayanasamy, S., Pereira, C., and Calder, B. Recording shared memory dependencies using strata. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, Oct. 21–25). ACM Press, New York, 2006, 229–240.
16. Prvulovic, M. and Torrellas, J. ReEnact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Proceedings of the International Symposium on Computer Architecture* (San Diego, CA, June 9–11). IEEE Press, 2003, 110–121.
17. Sohi, G., Breach, S., and Vijayakumar, T. Multiscalar processors. In *Proceedings of the International Symposium on Computer Architecture* (Santa Margherita Ligure, Italy, June 22–24). ACM Press, New York, 1995, 414–425.
18. Tuck, J., Ahn, W., Ceze, L., and Torrellas, J. SoftSig: Software-exposed hardware signatures for code analysis and optimization. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (Seattle, WA, Mar. 1–5). ACM Press, New York, 2008, 145–156.
19. Vallejo, E., Galluzzi, M., Cristal, A., Vallejo, F., Beivide, R., Stenstrom, P., Smith, J.E., and Valero, M. Implementing kilo-instruction multiprocessors. In *Proceedings of the International Conference on Pervasive Services* (Santorini, Greece, July 11–14). IEEE Press, 2005, 325–336.
20. Xu, M., Bodik, R., and Hill, M.D. A 'flight data recorder' for enabling full-system multiprocessor deterministic replay. In *Proceedings of the International Symposium on Computer Architecture* (San Diego, CA, June 9–11). IEEE Press, 2003, 122–133.
21. Yen, L., Bobba, J., Marty, M., Moore, K., Volos, H., Hill, M., Swift, M., and Wood, D. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proceedings of the International Symposium on High Performance Computer Architecture* (Phoenix, AZ, Feb. 10–14). IEEE Press, 2007, 261–272.

Josep Torrellas (torrellas@cs.uiuc.edu) is a professor and Willett Faculty Scholar in the Department of Computer Science at the University of Illinois at Urbana-Champaign.

Luis Ceze (luisceze@cs.washington.edu) is an assistant professor in the Department of Computer Science and Engineering at the University of Washington, Seattle, WA.

James Tuck (jtuck@ncsu.edu) is an assistant professor in the Department of Electrical and Computer Engineering at North Carolina State University, Raleigh, NC.

Calin Cascaval (cascaval@us.ibm.com) is a research staff member and manager of programming models and tools for scalable systems at the IBM T.J. Watson Research Center, Yorktown Heights, NY.

Pablo Montesinos (pmontesi@samsung.com) is a staff engineer in the Multicore Research Group at Samsung Information Systems America, San Jose, CA.

Wonsun Ahn (dahn2@uiuc.edu) is a graduate student in the Department of Computer Science at the University of Illinois at Urbana-Champaign.

Milos Prvulovic (milos@cc.gatech.edu) is an associate professor in the School of Computer Science, College of Computing, Georgia Institute of Technology, Atlanta, GA.

BulkCompiler: High-Performance Sequential Consistency through Cooperative Compiler and Hardware Support

W. Ahn, S. Qi, M. Nicolaides, J. Torrellas
University of Illinois at Urbana-Champaign
dahn2,sqi2,nicolai1,torrella@illinois.edu

J.-W. Lee, X. Fang, S. Midkiff
Purdue University
jaewoolee,xfang,smidkiff@purdue.edu

David Wong
Intel Corporation
david.c.wong@intel.com

ABSTRACT

A platform that supported Sequential Consistency (SC) for *all* codes — not only the well-synchronized ones — would simplify the task of programmers. Recently, several hardware architectures that support high-performance SC by committing groups of instructions at a time have been proposed. However, for a platform to support SC, it is insufficient that the hardware does; the compiler has to support SC as well.

This paper presents the hardware-compiler interface, and the main compiler ideas for *BulkCompiler*, a simple compiler layer that works with the group-committing hardware to provide a *whole-system high-performance* SC platform. We introduce ISA primitives and software algorithms for BulkCompiler to drive instruction-group formation, and to transform code to exploit the groups. Our simulation results show that BulkCompiler not only enables a whole-system SC environment, but also one that actually outperforms a conventional platform that uses the more relaxed Java Memory Model by an average of 37%. The speedups come from code optimization inside software-assembled instruction groups.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures — MIMD processors; D.3.2 [**Programming Languages**]: Language Classifications — Concurrent, distributed, and parallel languages; D.3.4 [**Programming Languages**]: Processors — Compilers, Optimization

General Terms

Algorithms, Design, Performance.

Keywords

Sequential Consistency, Atomic Region, Chunk-Based Architecture, Compiler Optimization.

1. INTRODUCTION

The arrival of multicore chips as the commodity architecture for many platforms has highlighted the need to make parallel programming easier. While this endeavor necessitates advances in all layers of the computing stack, at the hardware architecture layer it requires that multicores be designed to support programmer-friendly models of concurrency and memory consistency efficiently.

The memory consistency model specifies what values a load can return in a shared-memory multithreaded program [1]. One such model is Sequential Consistency (SC). SC mandates that the result of any execution of the program be the same as if the memory operations of all the processors were executed in some total sequential order, and those of each individual processor appear in this sequence in the order specified by its thread [15]. There is consensus that software writers prefer that the platform support SC because it offers the same simple memory interface as a multitasking uniprocessor.

For software that is well synchronized (i.e., one that does not contain data races), most systems used today support SC with high performance. This is because synchronization operations totally order those accesses from different threads that, if overlapped, could result in non-intuitive return values for loads. Unfortunately, much current software, ranging from user applications to libraries, virtual machine monitors, and OS, has data races — either by accident or by design. For these codes, SC is not provided. Moreover, as more beginner programmers attempt parallel programming on multicores in the next few years, the number of codes with data races may well increase.

1.1 Benefits of Supporting SC

Devising a platform that supports SC with high performance for *all* codes — including those with data races — would have four key benefits. The first one is that debugging concurrent programs would be easier. This is because the possible outcomes of the memory accesses involved in the bug would be easier to reason about, and the debugger could in fact *reproduce* the buggy interleaving.

A second benefit stems from the fact that existing software correctness tools almost always assume SC — for example, Microsoft's CHESS [19]. Verifying software correctness under SC is already hard, and the state space balloons if non-SC interleavings need to be inspected as well. In the next few years, software correctness verification tools are expected to play a larger role. Using them in combination with an SC machine would make them most effective.

A third benefit of SC is that it would make the memory model of safe languages such as Java easier to understand and verify. The need to provide safety guarantees and enable performance at the same time has resulted in an increasingly complex and unintuitive memory model over the years. A high-performance SC memory model would trivially ensure Java's safety properties related to memory ordering, and improve its security and usability.

Finally, some programmers want to program with data races to obtain high performance. This includes, for instance, writers of OS and virtual machine monitors. If the machine provided SC, the risk of introducing bugs would be reduced and the code portability enhanced.

1.2 Goal of the Paper and Contributions

From this discussion, we argue that supporting SC is a worthy goal. Recently, there have been several proposals for hardware architectures that support high-performance SC [3, 4, 6, 11, 12, 29, 32]. Some of these architectures support SC all the time by repeatedly committing groups of instructions atomically — called chunks in BulkSC [6], transactions in TCC [12], or implicit transactions in checkpointed multiprocessors [29]. Each instruction group executes atomically and in isolation, generating a total commit order of chunks and, therefore, instructions, in the machine. Such properties guarantee SC. Moreover, thanks to operating in large instruction groups, the overheads of supporting SC are small. Conceivably, a similar environment can be attained with a primitive for atomic region execution such as that of Sun's Rock [7], if it is invoked continuously.

Unfortunately, for a platform to support SC, it is *not enough* that the hardware support SC; the software — in particular, the compiler for programs written in high-level languages — *has to support SC as well*. For this reason, there have been several research efforts on compilation for SC [13, 28, 31]. Such efforts have sought to transform the code to satisfy SC on conventional multiprocessor hardware. The results have been slowdowns — often significant — relative to the relaxed memory models of current machines.

Remarkably, with the group-commit architectures, we have an opportunity to develop a high-performance SC compiler layer. Since the hardware already supports high-performance SC, all we need is for the compiler to drive the group-formation operation, and adapt code transformations to it. With the combination of hardware and compiler, the result is a *whole-system high-performance SC platform*. Furthermore, since the hardware guarantees atomic group execution, the compiler can attempt more aggressive optimizations than in conventional, relaxed-consistent platforms. The result is even *higher performance* than current aggressive platforms.

This paper presents the hardware-compiler interface and the main ideas for a compiler layer that works in the BulkSC architecture (as a representative of the group-commit architectures) to provide whole-system high-performance SC. We call our compiler algorithm *BulkCompiler*. Our specific contributions include: (i) ISA primitives for BulkCompiler to interface to the chunking hardware, (ii) compiler algorithms to drive chunking and code transformations to exploit chunks, and (iii) initial results of our algorithms with Java programs on a simulated BulkSC architecture.

Our results use Java applications modified with our compiler algorithms and compiled with Sun's Hotspot server compiler [22]. A whole-system SC environment with BulkCompiler and simulated BulkSC architecture outperforms a simulated conventional hardware platform that uses the more relaxed Java Memory Model by an average of 37%. The speedups come from code optimization inside software-assembled instruction chunks.

This paper is organized as follows: Section 2 gives a background; Sections 3 and 4 describe BulkCompiler and how it manages the chunks; Sections 5 and 6 evaluate the system; Section 7 assesses the results, and Section 8 discusses related work.

2. BACKGROUND

We describe the BulkSC architecture and the current approaches for compiler-driven enforcement of SC.

2.1 BulkSC: High-Performance SC Hardware

In the BulkSC multiprocessor [6], as a processor executes a thread, it automatically breaks the instruction stream into chunks and commits each chunk atomically. A *Chunk* is a group of *dynamically* contiguous instructions — 2,000 in the current implementation. This "chunked" mode of execution and commit is a hardware-only mechanism, which is invisible to the software running on the processor.

Each chunk executes on the processor *atomically* and *in isolation*. This means that none of the actions of the chunk are made visible to the rest of the system (other processors and main memory) until when the chunk commits. Moreover, if the chunk reads a location and, before it commits, a second chunk in another processor that has written to the same location commits, then the local chunk gets squashed and has to re-execute. Atomic chunk execution is supported by buffering in the L1 cache the state that the chunk is generating. Moreover, as the chunk executes, a Bloom filter automatically encodes in a *R* and *W* signature, the memory addresses read and written, respectively. After the chunk completes, the hardware sends *W* to an arbiter, which forwards it to other processors. In the other processors, *W* is intersected with the local signatures. A non-null result indicates an overlap of addresses, which causes the chunk in that processor to get squashed and restarted.

Since chunks execute atomically and in isolation, commit in program order in each processor, and the arbiter globally orders their commit, BulkSC supports SC at the chunk level — and, as a consequence, SC at the instruction level.

This is a high-performance SC implementation because the hardware can reorder and overlap all memory accesses within a chunk — except, of course, those that participate in single-thread dependences. In particular, synchronization instructions induce no reordering constraint. Indeed, *fences* inside a chunk are *transformed into no-ops* by the hardware. Their functionality — to delay execution until certain references are performed — is useless since, by construction, no other processor will observe the actual order of instruction execution within a chunk. Moreover, a processor can also overlap the execution of consecutive chunks [6].

2.2 Algorithm for Generating Chunks

In BulkSC, the hardware finishes the current chunk and starts a new one when the number of dynamic instructions executed exceeds a certain threshold that we call *maxChunkSize* (e.g., 2,000 instructions). There are, however, some events that affect the regular generation of chunks. Table 1 lists these events and, under *Actions in BulkSC*, the actions taken [6]. For example, when the write set of the chunk is about to overflow the cache, the hardware commits the current chunk at this point and starts a new chunk. The last column of the table will be discussed later.

2.3 Compiler-Driven Enforcement of SC

A compiler can take programs with potential data races and transform them to enforce SC even on a machine that implements a relaxed memory consistency model [13, 28, 31]. The general idea is to identify the minimal set of ordered pairs of memory accesses that should not be re-ordered, and then (1) insert a fence along every path between the first and second access in each pair, and (2) prohibit the compiler from performing any transformation that reorders any such pair.

The compiler analysis needed involves first performing Escape analysis [28], which determines which loads and stores may refer to memory locations accessed by multiple threads. Then, May-happen-parallel (or Thread-structure) analysis [20, 28] determines which memory accesses can happen in parallel. Based on these, Delay Set analysis [26] determines which of the shared accesses should not be reordered within a thread.

Event	Actions in BulkSC	Actions with BulkCompiler Inside Atomic Region
maxChunkSize instructions executed	The hardware commits the current chunk and starts a new chunk	No action
Cache overflow	The hardware commits the current chunk at this point, and starts a new chunk	The hardware squashes the current chunk and restarts it at the Safe Version point
Data collision with remote chunk	The hardware squashes the chunk and re-executes it. If the chunk is squashed *M* times, then the chunk also reduces its size to minimize collisions	Same as in under BulkSC. However, if the chunk size has to be reduced, restart the chunk at the Safe Version point
Exceptions (including system calls)	When the code wants to perform an uncacheable access, the hardware commits the current chunk at this point, performs the uncached operation, and starts a new chunk	When the code wants to perform an uncacheable access, squash the chunk and restart it at the Safe Version point. Do not set up an atomic region to include uncacheable accesses
Interrupts	The hardware completes the current chunk and then processes the interrupt in a new chunk(s)	The hardware squashes the current chunk, processes the interrupt, and then restarts the initial chunk under an atomic region again

Table 1: Events that affect chunk generation.

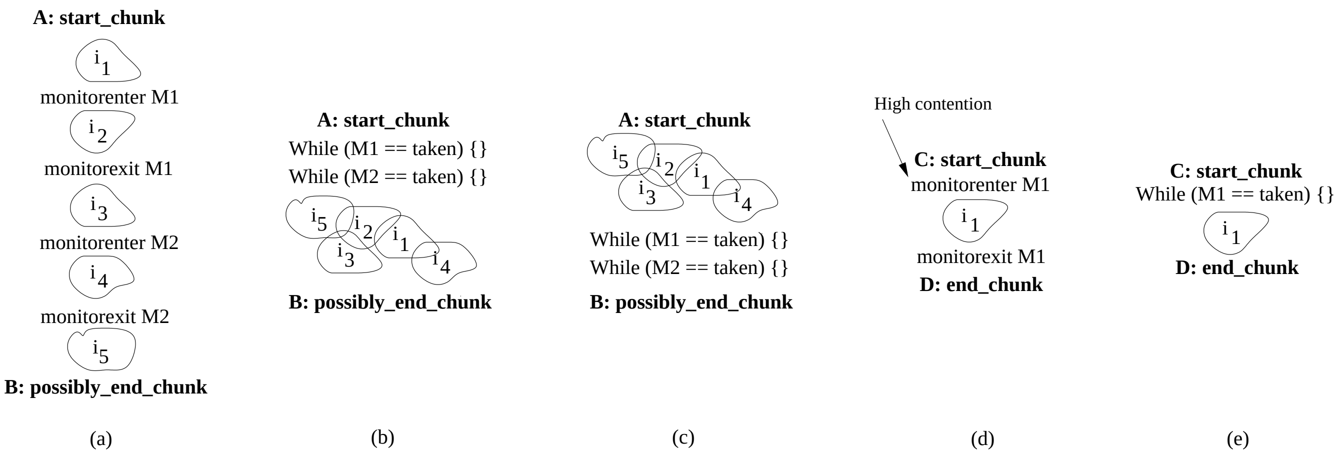


Figure 1: Compiler-driven chunking for high performance SC. In the figure, each i_j represents a set of instructions.

Unfortunately, the compiler analysis required is very costly both in runtime and in implementation effort — in part because every step needs interprocedural analysis. Moreover, all three existing implementations [13, 28, 31] report noticeable slowdowns relative to execution of the application under the relaxed model — in some cases, applications become several times slower. Our paper's goal is to deliver SC with even higher performance than current relaxed models.

3. BULKCOMPILER FOR SC

We submit that an architecture with continuous group commit such as BulkSC [6], TCC [12], or Checkpointed Multiprocessors [29] can potentially deliver whole-system (hardware plus software) SC at a higher performance than conventional machines deliver a relaxed memory consistency model. This is because, if the compiler drives chunk formation appropriately, the atomicity guarantee of chunks can enable many compiler optimizations inside the chunk.

In particular, we focus on multiprocessor-related issues. We observe that synchronization and fences can substantially hurt the performance of conventional relaxed-consistency machines. At the same time, synchronization-aware chunk formation can eliminate some of these problems, and further enable conventional compiler optimizations that improve performance.

In this section, we discuss the main ideas, the new instructions added, and the basics of the algorithms in *BulkCompiler* — our compilation layer for group-commit architectures. In a later section (Section 7), we briefly discuss how we can also improve the performance of relaxed memory consistency models in these architectures and enable new compiler optimizations.

3.1 Main Ideas

A compiler for a group-commit architecture should select the chunk boundaries so that they (1) maximize the potential for compiler optimization and (2) minimize the chance of chunk squash. Since the design space is large, this paper focuses on the multiprocessor related issues of synchronization and fences. In this area, BulkCompiler relies on one idea to maximize compiler optimization and one to minimize squashes.

3.1.1 Maximizing Compiler Optimization

To maximize compiler optimization, BulkCompiler identifies *low contention* critical sections (which are mostly in the form of synchronized blocks in Java). Then, it includes one or several of them and their surrounding code in the same chunk (Figure 1(a)). After this, each acquire operation (*monitorenter* instruction in Java bytecode) is replaced with a spinning loop, which checks if the synchronization variable is taken using *plain loads*. Moreover, all the release operations (*monitorexit* in Java bytecode) are removed. Next, we move the spinning on the locks with plain loads to the top of the chunk — subject to data and control dependences — to prepare the code for compiler optimization better. Finally, with the synchronizations removed, we let the compiler aggressively reorder and optimize the code inside the chunk. The resulting code is shown in Figure 1(b), where the overlapping sets of instruction denote the

Instruction	Functionality
beginAtomic PC	Finishes the current chunk, triggers a register checkpoint in hardware, and starts a new chunk. It takes as argument the program counter (PC) of the entry point to the *Safe Version* of the code, which will be executed if the chunk needs to be chopped into smaller chunks.
endAtomic&Cut	Finishes the current chunk and changes the mode of chunking from software-driven to hardware-driven. The hardware will start a new chunk next.
endAtomic	Changes the mode of chunking from software-driven to hardware-driven, enabling the hardware to finish the current chunk when it wants to (e.g., when the chunk size reaches *maxChunkSize*).
squashChunk	Squashes the current chunk and restarts it at the Safe Version. It involves clearing the BulkSC signatures, invalidating the cache lines written by the chunk, and restoring the checkpointed register file.
cutChunk	Finishes the current hardware-driven chunk, inducing the hardware to start a new one. It has no effect if found inside a *beginAtomic* to *endAtomic&Cut* (or *endAtomic*) region.

Table 2: Instructions added so that the compiler manages the chunking.

effect of compiler optimization. Note that checking all the locks at the beginning of the chunk may slightly reduce concurrency. However, since we apply this transformation to low-contention critical sections, such effect is insignificant.

Since the chunk will be executed atomically, there is no need to acquire and release a lock. However, the chunk still needs to read the locks with plain loads, to check if any lock is taken. A lock can be taken if another thread, after failed attempt(s) to execute its own chunk atomically, reverted to a (non-speculative) *Safe Version* of the code, where it grabbed the lock. We will see in Section 3.4 that every atomic region has a corresponding Safe Version, where any locks are acquired and released explicitly. This is the same approach followed by the Speculative Lock Elision (SLE) algorithm [23] and its implementation in the Sun Rock [9].

If any of the locks is taken, the code spins. When the owner of the lock commits the lock release, the spinning chunk will observe a data collision on the spinning variable. At that point, it will be squashed and re-started.

By eliminating the synchronization operations, this transformation improves performance in two ways. First, the processor avoids performing the costly synchronization operations, replacing acquires with the much cheaper loads. More importantly, however, is that this transformation eliminates the constraints on instruction reordering imposed by synchronization instructions. Indeed, even under current relaxed memory models, compilers neither move instructions across synchronization operations nor allocate shared data in registers across them. This disables many instances of conventional optimizations such as register allocation, common subexpression elimination, loop invariant code motion, or redundant code motion, to name a few. After we remove the synchronization operations, a conventional compiler can reorder instructions and perform all of these optimizations.

We can place the spinning on the locks with plain loads at the end of the chunk, after all the work is done (Figure 1(c)). This approach makes a difference when one or more locks are taken by other processors and, therefore, the chunk will eventually be squashed. In this case, having the spinning at the end of the chunk can enable prefetching of read-only data for the chunk re-execution. However, it may also cause exceptions resulting from accessing data of a critical section while another processor is also accessing it. Overall, since we apply this transformation to low-contention critical sections, these effects are not very significant.

Finally, this transformation is especially attractive in Java programs, which is the environment examined in this paper. This is because Java programs have many low-contention critical sections in the form of synchronized methods — often in thread-safe Java libraries. The synchronized blocks in these methods are compiled into Java bytecode using the *monitorenter* and *monitorexit* bytecode instructions surrounding the code in the block.

3.1.2 Minimizing Squashes

The second idea in BulkCompiler is to minimize squashes by identifying *high-contention* critical sections and tight-fitting a chunk around it (Figure 1(d)). As in the previous transformation, *monitorenter* is replaced with a loop that checks if the lock is taken using plain loads. *Monitorexit* is removed (Figure 1(e)). Tight-fitting the chunk reduces the chances that different processors collide on this critical section, and also reduces the number of wasted instructions per squash. It also enables processors to hand over access to popular critical sections to other processors sooner, since chunks commit sooner.

Even after all these transformations, chunks created by the compiler can collide at runtime — either on the synchronization variable or on another variable. In this case, they retry as per the default BulkSC execution. However, there are events that require reducing the size of the chunk, such as a cache overflow or performing an uncached memory access. Reducing the chunk size could lead to non-SC executions if the broken chunk exposes reordered references to shared data. To prevent this, BulkCompiler also creates the Safe Version of the code mentioned before. The Safe Version does not reorder references to shared variables and includes the *monitorenter* and *monitorexit* instructions.

Overall, with these changes on top of high-performance SC hardware, we target a performance higher than that attained with the relaxed Java Memory Model on conventional hardware, while providing whole-system SC.

3.2 New Instructions Added

Table 2 shows the instructions added to enable the compiler to manage the chunking. The principal ones are *beginAtomic*, which marks the beginning of an atomic region, and *endAtomic&Cut* or *endAtomic*, which mark the end. *BeginAtomic* causes the BulkSC hardware to finish the current chunk and start a new one. It also creates a register checkpoint to revert to if the chunk is squashed. The instruction takes the program counter (PC) of the entry point to the Safe Version of the code for the chunk. When the atomic region is squashed, depending on the reason for the squash, the hardware returns execution to either the *beginAtomic* instruction or the entry point to the Safe Version.

EndAtomic&Cut terminates the current chunk and then lets the BulkSC hardware take over the chunking — the hardware will start a new chunk next. *EndAtomic* simply lets the BulkSC hardware take over the chunking. This means that the current chunk may continue executing until a total of *maxChunkSize* instructions since *beginAtomic* have been executed. When a chunk is executing within the *beginAtomic* to *endAtomic&Cut* (or *endAtomic*) instruction pairs, reaching the *maxChunkSize* instruction count does not cause chunk termination. Overall, with these primitives, we surround the groups of low-contention synchronized blocks as in Fig-

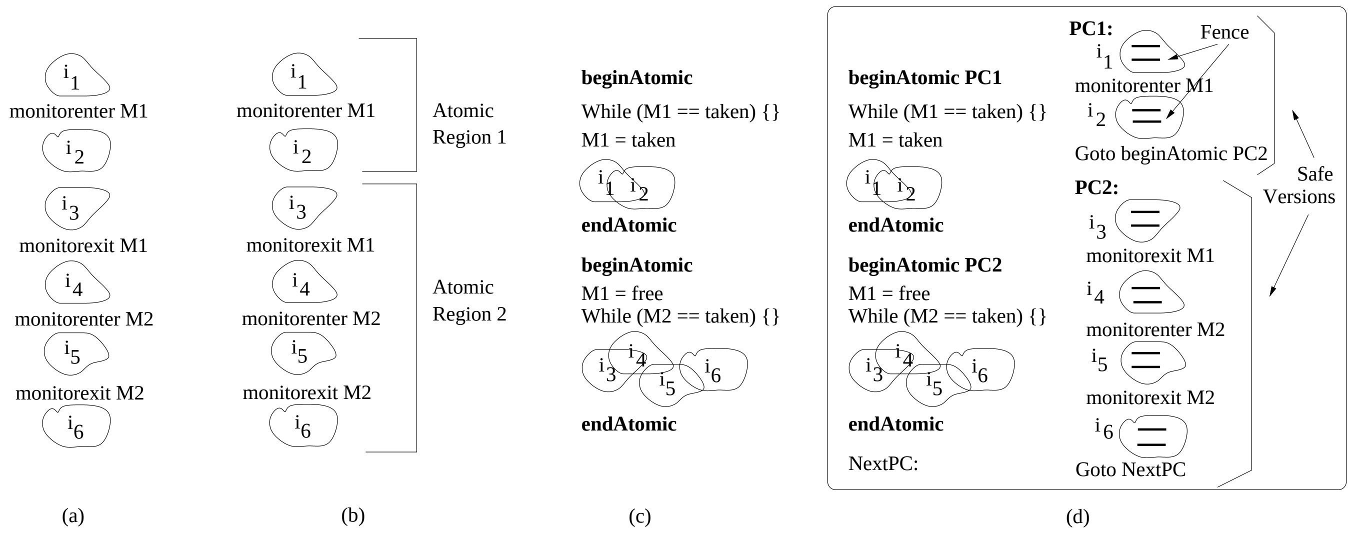


Figure 2: Transforming a large code section. In the figure, each i_j represents a set of instructions.

ure 1(b) with *beginAtomic* at point *A* and *endAtomic* at point *B*; we surround the high-contention synchronized blocks as in Figure 1(e) with *beginAtomic* at point *C* and *endAtomic&Cut* at point *D* in the figure.

To the compiler, *beginAtomic* has acquire semantics, which means that it cannot move any escaping reference (i.e., reference to shared data) that follows *beginAtomic* to before it. *EndAtomic&Cut* and *endAtomic* have release semantics, and the compiler cannot move any escaping reference that precedes them to after them.

The table shows two more instructions, called *squashChunk* and *cutChunk*. The former squashes the current chunk and restarts at the Safe Version. It can be used for speculative compiler optimizations, which sometimes require a rollback after discovering that they have performed an illegal transformation (e.g., [21]). The *cutChunk* instruction simply finishes the current hardware-driven chunk, inducing the hardware to start a new chunk. It has no effect if found inside a *beginAtomic* to *endAtomic&Cut* (or *endAtomic*) region. Note that if, dynamically, *endAtomic&Cut*, *cutChunk*, or potentially *endAtomic* are immediately followed by *beginAtomic*, the latter does not start a second chunk beyond the one that the hardware is starting.

3.3 Difference to Transactional Memory

To understand our transformations, it is useful to compare them to Transactional Memory (TM). The main goal of TM is enhancing concurrency; the main goal of our transformations is enhancing the performance of each thread through compiler optimization while preserving SC. However, since we focus on optimization opportunities afforded by synchronizations, our use of an SLE-like algorithm also enhances concurrency, especially in high-contention critical sections.

To see the difference between the two goals, consider a synchronized block that is too large for the hardware to provide atomicity. Unlike TM, BulkCompiler still benefits from splitting the code into two atomic regions. This is seen in Figure 2(a), which shows code with two synchronized blocks protected by locks M1 and M2. Assume that BulkCompiler estimates that the code in the M1 block has a footprint that amply overflows the cache. Further, assume that it estimates that the code before the M1 block (i_1) could be optimized together with the code inside the block. In this case, it partitions the code into two atomic regions that it estimates fit in the cache (Figure 2(b)): one that executes i_1 and the beginning of the first block, and another that executes the rest of the code.

BulkCompiler relies on the hardware guarantee that each region executes atomically. It transforms the code as shown in Figure 2(c): synchronization operations become plain accesses and the code is aggressively reordered and optimized. In particular, in the first atomic region, *monitorenter* is replaced with a spinning loop, which checks if the lock is taken using plain loads. If the lock is free, the code sets it to taken. If the chunk eventually finishes and commits, this lock update will be made visible; however, the chunk may be squashed before committing by the commit of another chunk that also set the lock. On the other hand, if the lock was not free, the code spins and will not commit. The chunk will eventually get squashed, either when the thread is pre-empted from the processor or when the chunk that releases the lock commits.

In the second atomic region, *monitorexit M1* simply becomes a plain write to the lock variable to release it. If the chunk commits, the write will be visible to the rest of the processors. Note that lock variable M1 has to be explicitly written as taken or freed, although the writes can be plain stores. This is because, since the synchronized block is now split into two regions, atomicity is no longer guaranteed and we have to rely on the value of the variable to prevent illegal interleavings. Finally, the accesses to *M2* are replaced with a spinning loop on *M2* with plain loads as described before. Overall, in all cases, the rest of the code is heavily optimized and the system satisfies SC.

3.4 Safe Version of the Atomic Region Code

It is possible that an atomic region gets squashed. Recall that Column 2 of Table 1 showed the events that affect chunks in the original BulkSC architecture. The last column of the table shows how we slightly change the BulkSC hardware so that it guarantees the atomicity of atomic regions.

First, inside an atomic region, the chunk is prevented from finishing when the number of instructions reaches past *maxChunkSize*, to guarantee that the entire atomic region does in fact commit atomically. Second, since this requirement can result in long atomic regions, we want to process interrupts as soon as they are received — rather than waiting until the current chunk completes. Consequently, on reception of an interrupt, the current chunk is squashed, the interrupt is processed, and then the initial chunk is restarted from the beginning — using the checkpoint from *beginAtomic*.

Finally, to guarantee the atomicity of atomic regions, events that previously triggered a chunk squash may need to be handled differently. These events include (i) cache overflows, (ii) uncacheable accesses in exceptions (which include system calls), and (iii) data collisions with a remote chunk. How we handle these events largely depends on whether the event will (likely) repeat after the chunk is squashed and restarted.

The events that are unlikely to repeat are most data collisions. In this case, the atomic region is squashed and then re-executed from the beginning. The events that repeat are cache overflow, uncacheable accesses in exceptions, and repeated data collisions on the same chunk in pathological cases. Some cases of uncacheable accesses can be avoided by not including problematic system calls inside atomic regions. However, the rest of the events are largely unpredictable and hard to avoid. The atomic region cannot be simply squashed and re-executed since it will be squashed again.

To make progress in these cases, we would have to commit a downsized chunk —- i.e., the code up until we cause the cache overflow, or reach the uncacheable access or the access that causes the collision. However, this would break the atomicity of the chunk and, potentially, expose inconsistent or non-SC state. Consequently, to address these cases, a Safe Version of the code is generated for each atomic region. This safe code does not rely on atomic execution to preserve SC. If the atomic region needs to be truncated for any of the "repeatable" reasons, the chunk is squashed and execution is transferred to the PC of the Safe Version entry point — as given in the *beginAtomic* instruction.

The Safe Version of the code acquires and releases locks explicitly. Moreover, it also has to satisfy SC. Therefore, BulkCompiler conservatively identifies all the escaping references in the code using the algorithm in [16]. Then, it adds a fence at the beginning of the Safe Version code, and after every escaping reference. The fences prevent the compiler from reordering the escaping accesses — and hence ensure SC at a performance cost. The analysis of Section 2.3 could keep the overheads to a minimum. Figure 2(d) shows the final code for the example.

Fortunately, part of this performance loss is transparently recovered by the chunking hardware. Specifically, as the BulkSC hardware executes the Safe Version code with hardware-driven chunks, fences are *no-ops* (Section 2.1). The accesses that fall in the same chunk will be overlapped and reordered by the hardware, irrespective of the presence of the fences. Note also that, since Safe Versions are rarely executed, they will not hurt the instruction cache through code bloat noticeably.

4. ALGORITHM DESIGN

In this section, we describe the algorithms that we use and some of the corner cases encountered.

4.1 Inserting Atomic Regions

At the highest level, our algorithm desires to have all escaping references contained in atomic regions, and for each region to be as large as possible to expose the maximum number of optimization opportunities. Doing this naively, however, will lead to excessive squashing of atomic regions due to conflicts or cache overflow, and difficulty in generating code for the Safe Versions of the regions.

The algorithm that we use is shown in Figure 3. This algorithm is applied to each method in turn. Prior to actually selecting atomic regions, the algorithm performs aggressive inlining, escape analysis [16], and loop blocking. Inlining reduces the impact of using an intraprocedural algorithm for selecting atomic regions. Escape analysis identifies the escaping references in the method, namely the references to objects that may be accessed by two or more threads. These references should be enclosed in atomic regions. Finally, loop blocking transforms inner-most loops into a loop nest, with a constant bound on the iteration count of the innermost loop. This allows the innermost loop to be enclosed in an atomic region that fits in the cache. Loops not containing any escaping references need not be blocked.

```
1. Perform aggressive inlining.
2. Perform escape analysis and mark escaping references.
3. Block inner-most loops that have escaping references.
4. Traverse code while enclosing each escaping reference in
   an atomic region.
5. Expand each atomic region $r$ that is immediately control
   dependent on statement $c$. We enclose adjacent statements
   $s$ while all the following hold:
   a. $s$ is control equivalent to $r$. If $s$ is not control equivalent
      to $r$, then:
      i.  if $s$ is inside the $c$ control structure, expand $r$ to
          contain the code from $s$ to $P_s$ (the post-dominator
          of $s$). The same applies if $P_s$ is encountered first.
      ii. if $s = c$, first expand $r$ downwards until $P_c$ (the
          post-dominator of $c$), and then also add $c$ to $r$. The
          same applies if $P_c$ is encountered first.
   b. the estimated footprint of $r$ fits in the cache.
   c. $s$ is not in a highly-contended synchronized block that
      does not contain $r$.
6. Generate the Safe Version for all the atomic regions.
```

Figure 3: Algorithm that inserts atomic regions in a method.

The algorithm then begins a traversal of the code, and each escaping reference is placed into an atomic region. After all escaping references are enclosed in atomic regions, a second pass is made to expand atomic regions and merge them where necessary. In the second pass, each atomic region is visited in turn. When an atomic region is visited, it is expanded to enclose code before and after the atomic region, with limits on this expansion as described shortly. If the expansion of an atomic region r_i encounters another atomic region r_j, r_j is merged into r_i, forming a single atomic region.

Three conditions need to hold during this expansion process. The first one is that the atomic region must begin and end at control equivalent points. Let c be the statement on which region r containing escaping reference e is immediately control dependent. This condition is easily satisfied when the statement s encountered while expanding region r is control equivalent to r. However, if it is not control equivalent, care must be taken. Specifically, (1) if s is inside the c control structure and the code from s to P_s (the post-dominator of s) is small enough so that s to P_s fits in the region, then s to P_s is added to r. The same applies if P_s is encountered instead of s. Moreover, (2) if $s = c$, then r is first expanded to cover all statements between c to P_c (the post-dominator of c) such that all statements control-dependent on c are inside r, and then c is also added to r. The same applies if P_c is encountered instead of c.

The second condition is that the estimated footprint of the atomic region fits in the cache. A model is used to estimate the contribution of each statement to the footprint. However, the available footprint is assumed exceeded if the algorithm attempts to (1) expand the atomic region into a loop other than the innermost loop around the

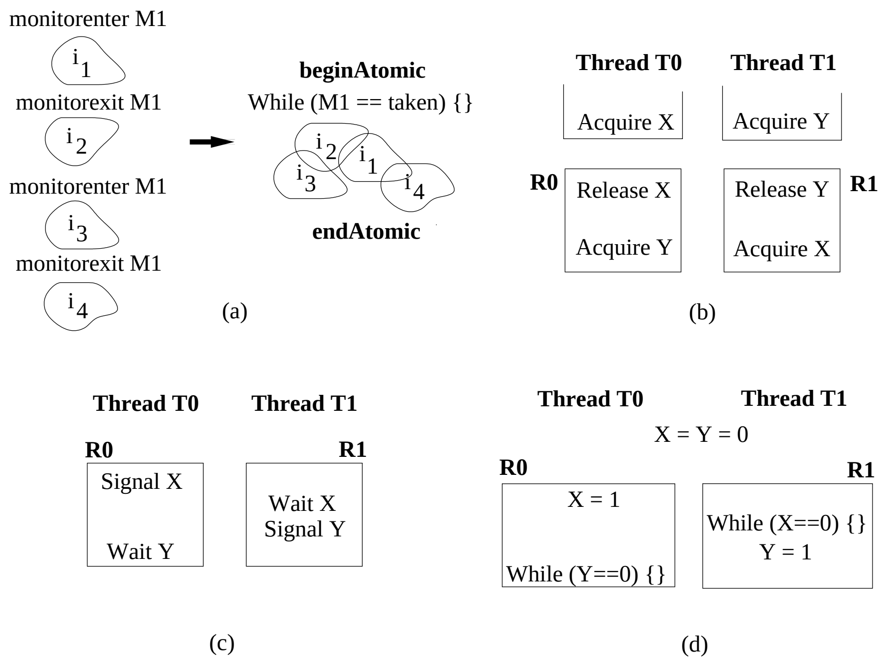


Figure 4: Examples of chunks with synchronization operations.

escaping reference e, or (2) include in the atomic region a non-inlined method call.

The third condition is that atomic regions will not expand to contain statements within a highly-contended synchronized block unless the escaping reference e is in that block. If e is in a highly-contended block, the region will at most be expanded to cover the highly-contended synchronized block.

The resulting atomic regions start and end at control equivalent parts of the program. This ensures that all atomic region starts have a corresponding atomic region end, regardless of the path taken by the program when executing. It also simplifies the generation of the code for Safe Versions.

Finally, Safe Versions of the regions are formed by duplicating the block of code in the atomic regions. A fence is placed at the beginning of the Safe Version code and after every escaping reference, to ensure that the compiler does not reorder escaping references.

4.2 Lock Compression and Region Nesting

When an atomic region contains multiple synchronized blocks protected by the same lock, the algorithm introduces a single check for the lock variable (Figure 4(a)). We call this scheme *Lock Compression*.

It is possible that the code contains nested atomic regions. At runtime, our chunking hardware flattens them out, and considers them just one large atomic region. To do this, the hardware keeps a nesting-level counter, and the chunk ends only at the outermost *endAtomic&Cut* or *endAtomic*. Moreover, when a squash is triggered, the outermost atomic region is squashed and, if appropriate, its Safe Version is invoked.

4.3 Visibility with Synchronizations

When our algorithm produces an atomic region with accesses to multiple synchronization variables, there may be interactions between threads that cause problems of *Visibility*. As an example, the problem occurs when an atomic region in Thread *T0* releases variable *X* and acquires variable *Y*, while an atomic region in Thread *T1* releases variable *Y* and acquires variable *X*. This is shown for regions *R0* and *R1* in Figure 4(b). For simplicity, the figure shows acquire and release operations — in practice, our algorithm will have replaced them with plain memory accesses to the variables. In the figure, Region *R0* cannot complete and make its release of *X* visible to *R1* because it is spinning on *Y*, which *T1* holds. *R1* is in a symmetrical situation. The result is deadlock, as both threads are spinning on the acquires.

The problem is not limited to a pattern where both threads first release a variable and then acquire a second one. It also occurs when there is a *handshake* pattern between two threads. This pattern is shown in Figure 4(c), using Signal and Wait synchronization operations. Again, we show these operations for simplicity, although our algorithm uses plain accesses. In the figure, Region *R0* signals synchronization variable *X* (effectively a release) and then waits on *Y* (effectively an acquire), while *R1* waits on *X* and then signals *Y*. Both threads end up spinning on the waits, unable to complete the regions.

These visibility problems do not occur with the hardware-driven chunks of BulkSC [6]. This is because such chunks complete as soon as *maxChunkSize* instructions are executed, rather than when a certain static instruction is reached. In both examples, the threads would spin in the acquires (or in the waits) until they reach *maxChunkSize* instructions. At that point, they would finish the chunks, making the two releases (in Figure 4(b)) or the signal to *X* (in Figure 4(c)) visible.

Similar visibility problems have been observed by proposals that integrate locks and transactions [24, 33] and by discussions of transactional memory atomicity semantics [18]. Ziarek *et al* [33] propose to solve the deadlock problem by detecting that two transactions are not completing, squashing them, and executing lock-based versions of the code. The authors state that these cases happen rarely.

BulkCompiler uses a similar approach, which is detailed in Section 4.4 and is simpler to implement. However, the problem with "unpaired" synchronization shown in Figure 4(b) cannot occur for high-contention critical sections because BulkCompiler tight-fits the atomic region around the section. For low-contention critical sections, an unpaired synchronization may be lumped with other access(es) to synchronization variable(s) within a single atomic region. However, because of the low contention for the synchronization variables, the probability of an interleaving that causes deadlock is very low. An alternative design is to have the compiler disable the creation of such atomic regions.

4.4 Visibility with Data Races

If the code is not properly synchronized, data races may produce the deadlock-prone access patterns discussed above. For example, Figure 4(d) shows data races that create the handshake pattern. In this case, the compiler may be unable to detect the possibility of deadlock — except, perhaps, at the cost of expensive and conservative *Must-alias* analysis.

To handle this case and other deadlocks at runtime, BulkCompiler relies on detecting that two chunks are not completing, squashing them, and then triggering the execution of their Safe Versions. Note that, in our environment, detecting that chunks are not completing is easy. Rather than measuring wall-clock time, we count the number of completed instructions — which is needed by the BulkSC hardware anyway. If this number is very high, the processor is likely spinning on a tight loop. At that point, the spinning chunks are squashed and the Safe Versions executed. One option for chunks that suffer frequent timeouts is recompilation.

5. EXPERIMENTAL SETUP

5.1 Compiler and Simulator Infrastructure

Our evaluation infrastructure uses two main components: the Hotspot Java Virtual Machine (JVM) for servers [22] from Sun Microsystems and a Simics-based [30] simulator of the BulkSC architecture [6]. Hotspot is an aggressive commercial-grade compiler with extensive support for just-in-time compilation and adaptive optimization. It is included in OpenJDK7 [27]. We use Hotspot to compile both the unmodified applications for a conventional architecture, and the applications modified with the BulkCompiler algorithms for a BulkSC architecture. We report the difference in performance.

We apply the algorithm described in Section 4.1 to Java source code using a profile-driven infrastructure that currently requires substantial hand-holding. We are in the process of automating the infrastructure. Since we are instrumenting at the Java source code level, we cannot directly insert our assembly instructions of Section 3.2. Instead, we use the JNI (Java Native Interface) to wrap the instructions in Java methods — at the cost of some overhead.

The resulting modified source is compiled to bytecode, and then run on the Hotspot JVM. The Hotspot JVM executes on top of a full-system execution-driven simulator built using Simics [30]. The simulator uses the x86 ISA extended with the BulkCompiler instructions. The simulator models a BulkSC multiprocessor [6], including the chunk-based speculative execution, checkpointing, chunk squash and rollback, signature operation, and the extensions needed for BulkCompiler. For comparison, we also model a plain, non-chunk-based multiprocessor.

We model a multicore with 4 single-issue processors running at 4 GHz. Each processor has a 4-way, 64-Kbyte L1 data cache with 64-byte lines. If the cache overflows while executing an atomic region, the chunk gets squashed. Given that the processor model is simple, we report performance in number of cycles taken by the program assuming a constant CPI of 1, irrespective of the instruction type, or whether an access hits or misses in the cache. In some of the experiments, we will assign a fixed cost in cycles to each CAS (Compare-And-Swap) operation. CAS is used to implement synchronization in the Hotspot JVM. In all cases, the results are measured after the application has run a sufficient number of instructions to warm up the code cache.

5.2 Experiments and Applications

We start by identifying which synchronization variables in the application have high contention and which have low contention. For this, we use Hotspot, which provides options to profile dynamic locking behavior. It is as simple as running with an additional Hotspot argument. This information enables the targeting of the atomic regions. In addition, our infrastructure uses a simple model of the data footprint of each code section, which is used to decide when the atomic region should terminate, to minimize cache overflow. We often chop loops into multiple blocks of appropriate sizes in order to put each block inside an atomic region.

For the evaluation, we use the SPECJBB2005 and SPECJVM98 benchmark suites. In addition, we also evaluate two additional applications with substantial synchronization, namely *MonteCarlo* from SPECJVM2008 and *JLex* from [2]. Of these applications, SPECJBB2005 and *MonteCarlo* run with 4 threads, and *Mtrt* of SPECJVM98 runs with 2 threads. The rest of SPECJVM98 and *JLex* run with a single thread, although they have many synchronized blocks. These synchronizations are in the Java library code, which includes synchronization because it has to be thread safe. Each application runs for at least 1B instructions before being measured.

Finally, among the SPECJVM98 applications, we could not evaluate *Javac* or *MpegAudio* because they are commercial applications with no source code, which we need for source level instrumentation. However, we were able to include *Jack* (another SPECJVM98 commercial application) because it has become open source under the name of JavaCC. The JavaCC source distribution includes an input set which is an identical copy of the input set for *Jack* with a few syntactic modifications.

6. EVALUATION

In this evaluation, we first describe the optimizations that we enable, then present the simulated speedups, and finally characterize the transformations performed.

6.1 Understanding the Optimizations Enabled

To understand the way in which BulkCompiler's transformations enable Hotspot to generate faster code, we analyzed the intermediate representation of the code generated by Hotspot with and without the BulkCompilerchanges. We did not add any new compiler optimization to take advantage of chunk-based execution; conventional Hotspot optimizations perform significantly better once Hotspot is given control of the chunks. The following are some common patterns seen:

Loop unswitching. This transformation involves moving a loop-invariant test out of a loop, and then producing two versions of the loop, one in the if-branch of the test, and the other in the else-branch. With the removal of the test, the two loop bodies have a more streamlined control flow and, therefore, the compiler can optimize them, creating better-quality code. The presence of synchronization within the loop had prevented this optimization, since it would have been in violation of the Java Memory Model. However, after BulkCompiler has wrapped the loop inside an atomic region and replaced the synchronizations with plain accesses, Hotspot performs this optimization automatically. The Java Memory Model will not be violated because the hardware guarantees that there are no intervening conflicting accesses until the atomic region runs to completion.

Null check elimination. In order to satisfy Java safety guarantees, the compiler needs to insert null checks before every object reference — unless it is able to prove that the reference is non-null. If the compiler can prove that two references point to the same object, it can safely remove the checks on the second reference. This situation occurs often inside a loop, where a reference remains invariant through all the iterations. In this case, the compiler peels off the first iteration of the loop, where it inserts all the checks, and removes the checks from the main body of the loop. Hotspot could not do this optimization if there were intervening synchronizations between the references, since it would be illegal. After BulkCompiler's transformations, Hotspot performs this optimization.

Range check elimination. In addition to performing null checks, the compiler is also required to check that an array reference does not exceed the boundaries of the array. Like for null checks, if the compiler is able to prove that an earlier range check subsumes a later range check, the later check can be removed. Once again, however, the presence of intervening synchronizations prevented Hotspot to perform the same loop-peeling optimization in the code described above. With BulkCompiler's transformations, Hotspot performs the optimization.

Loop invariant code motion. Often, the same expression is computed at every iteration of a loop. A common example is when the range of an array which does not change in size needs to be computed repeatedly within a loop. This transformation involves moving the computation outside the loop. If the loop has synchronizations, Hotspot cannot move the computation. With BulkCompiler's transformations, Hotspot can perform the optimization without violating the Java or SC memory models.

Register allocation. Memory locations that were allocated in registers cannot survive synchronization boundaries. The data needs to be stored to memory and loaded back from it, or the Java Memory Model would be violated. BulkCompiler's transformations result in the removal of many register allocation restrictions, which often result in much more efficient code.

Besides these types of optimizations, the removal of memory fences done by BulkCompiler gives Hotspot much more room for code scheduling. Scheduling is especially important for potentially long delay loads and stores. However, this effect is not evaluated in our results due to the simplistic timing model used in our simulator.

6.2 Simulated Speedups

To estimate the performance gains enabled by BulkCompiler, we simulate two environments. The first one (*Baseline*) is unmodified Java running on a conventional (i.e., without chunks) multiprocessor. The second one (*BulkCompiler*) is code transformed by BulkCompiler running on a BulkSC multiprocessor.

As indicated before, because of the model used in our simulator, we report performance in number of cycles taken by the programs assuming a constant CPI of 1, irrespective of the instruction type. For this reason, we call the two environments above *Baseline_1* and *BulkCompiler_1*. However, it is well known that an important source of overhead in implementations of Java is the actual read-modify-write operations (e.g., CAS) performed in the frequent synchronizations — in the case of Hotspot, potentially two read-modify-write operations for each synchronized block, one at the beginning and one the end. BulkCompiler's transformations replace these operations with plain accesses. Consequently, in our simulations, we also report results for a second scenario, namely one where each instruction takes 1 cycle except for the read-modify-write operations, which take 20 cycles each. The latter is the overhead measured in our workstations for a read-modify-write operation. We call the two environments *Baseline_20* and *BulkCompiler_20* for the two architectures.

Since these environments do not include a high-fidelity architectural model, they do not capture how different memory models use microarchitectures for access overlapping. However, they capture how the compiler can re-order and transform the code under different models, changing the number of instructions executed.

Figure 5(a) shows, for each application, the speedup of *BulkCompiler_1* over *Baseline_1*, while Figure 5(b) shows the speedup of *BulkCompiler_20* over *Baseline_20*. The bars also include the average for the SPECJVM98 applications, and the average for all the applications. Recall that *BulkCompiler* delivers SC execution, while *Baseline* executes with the relaxed Java Memory Model.

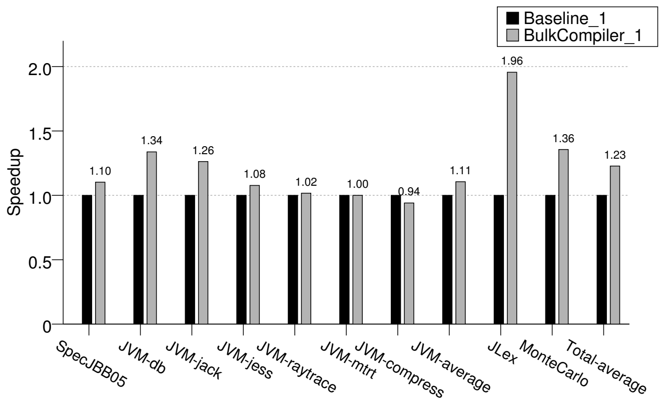
The figures show that *BulkCompiler* delivers substantial speedups over *Baseline*. In the environment where all the instructions have the same cost, the average speedup of *BulkCompiler_1* across all the applications is 1.23 (or 1.11 if we only consider SPECJVM98). In the environment where the read-modify-write instructions are more costly, which we consider to be more realistic, the speedups are higher. Specifically, the average speedup is 1.37 (or 1.23 if we only consider SPECJVM98). These results show that a *whole-system SC platform*, which guarantees SC at both the compiler and hardware levels, can deliver higher performance than a state-of-the art platform that supports the relaxed Java Memory Model (*Baseline*).

An analysis of the applications shows that most of them get speedups, sometimes quite high. The exceptions are *JVM-raytrace*, *JVM-mtrt*, and *JVM-compress*. We did not get speedups for these applications largely because they do not contain much synchronization in the first place. However, also notice that instrumenting with atomic regions and enforcing SC did not cause them to slow down significantly, either. This is despite the fact that we wrap the *BulkCompiler* assembly instructions in JNI calls (Section 5.1), which introduce some overhead. Such overhead would not be present in an implementation that works on the Hotspot intermediate representation. Finally, we note that the speedups of *JLex* are the same for *BulkCompiler_1* and *BulkCompiler_20*. This is because the locks in *JLex* were mostly in the biased [25] state, which does not use any read-modify-write operations.

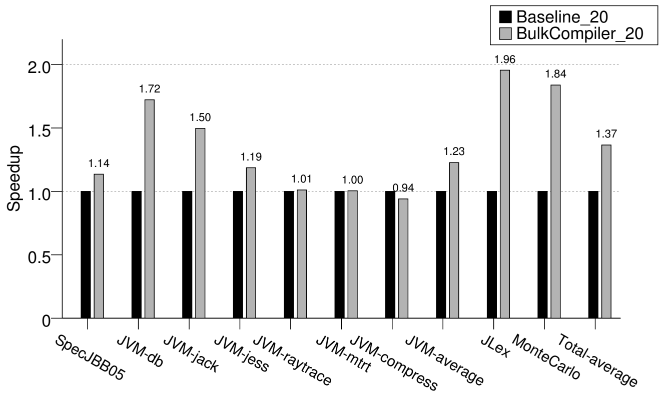
6.3 Characterizing the Transformations

In this section, we characterize the dynamic behavior of the code transformed by BulkCompiler as it runs on the BulkSC architecture. The data is shown in Table 3, where AR stands for Atomic Region. In the table, Columns 2–4 show the percentage of dynamic instructions inside atomic regions in the program, the number of dynamic atomic regions, and the average dynamic size of an atomic region in instructions, respectively. We can see from this data that our atomic regions cover the great majority of the execution (74% of the dynamic instructions on average). The remaining execution largely contains private references. We also see that there are many dynamic atomic regions and that they are very large — about 52,000 dynamic instructions on average. These atomic regions are largely loops with small to modest write footprints. The average atomic region size for *JVM-compress* is smaller than the others. This is because system calls interspersed across this application force the creation of smaller atomic regions. At this size, the overhead of our JNI calls becomes more significant and, hence, we suffer a 6% overhead as can be seen in Figure 5, even with a negligible squash rate.

Columns 5–7 give more information about these atomic regions, namely the number of synchronized blocks per region in the origi-



(a)



(b)

Figure 5: Speedups of *BulkCompiler_1* over *Baseline_1* (a), and of *BulkCompiler_20* over *Baseline_20* (b).

nal code, and their write and read footprints in number of 64-byte lines, respectively. We can see that, on average, each atomic region used to contain about 600 synchronized blocks. By transforming their synchronization operations into plain memory accesses, we enable many optimizations in Hotspot. As mentioned in Section 6.2, *JVM-raytrace*, *JVM-mtrt*, and *JVM-compress* do not have much synchronization and, therefore, show no speedups.

We also see that the atomic regions have a small write footprint (184 lines on average). This allows them to fit inside the cache without overflows. The read footprint is larger, but recall that, in BulkSC the read footprint *does not need to remain in the cache* — signatures keep a record of the lines read [6].

For example, *JVM-db* has a large read footprint but a tiny write footprint compared to the size of its atomic regions. This is due to the fact that *JVM-db* spends the bulk of its time sorting its database index, which involves string comparisons of index entries and swaps when entries are out of order. The index is only updated on swaps, which are much less frequent than the number of read accesses required for the string comparisons. This is the reason for the small write footprint. However, each access to the index is protected by a synchronized block, giving BulkCompiler ample optimization opportunities. Other applications follow a similar pattern.

Finally, the last column shows the fraction of dynamic instructions in atomic regions that get squashed. We see that, on average, only 0.48% of the instructions in atomic regions get squashed. This represents a tolerable fraction of work lost.

7. DISCUSSION

These experiments are only an initial estimation of the potential of exposing an architecture with all-the-time group commits to the compiler. Indeed, we need a high-fidelity model of the microarchitecture to assess whether BulkCompiler's higher freedom to schedule long-latency memory accesses within a large atomic region translates into performance impact.

Moreover, this paper has focused only on (i) synchronization-related issues and (ii) enabling *conventional* compiler optimizations that already exist in Hotspot — such as register allocation or loop-invariant code motion. BulkCompiler can be augmented with *novel* compiler optimizations enabled by the all-the-time group-commit hardware. Some of these optimizations could focus on speeding-up single-thread execution — an area explored by Neelakantam *et al* [21] (Section 8). Other optimizations could specifically focus on other multithreaded issues such as load imbalance.

Another avenue of research is to apply the memory ordering relaxation provided by all-the-time group commits to improve the performance of other memory consistency models beyond SC. We

Application	% of Dyn Instructions in ARs	# of Dynamic ARs	Dyn AR Size	# Sync Blocks per AR	Write Footprint (Lines)	Read Footprint (Lines)	% Instructions in AR Squashed
SPECJBB05	44.5	323086	19117.2	212	489.4	865.6	0.79
JVM-db	75.8	22451	119176.0	2000	84.4	3123.0	0.40
JVM-jack	29.5	2382	30105.2	792	119.7	229.4	1.31
JVM-jess	62.6	33995	43475.6	102	141.1	449.7	0.27
JVM-raytrace	85.8	61419	19771.1	0	51.7	613.9	0.10
JVM-mtrt	77.5	61627	19589.0	0	305.5	1297.0	0.14
JVM-compress	92.7	1632082	5418.6	0	28.1	144.5	0.04
JLex	97.4	45846	131474.0	317	426.9	705.7	0.91
MonteCarlo	99.9	16778	82535.1	2000	11.0	13.0	0.34
Average	74.0	244407	52295.8	602	184.2	826.9	0.48

Table 3: Characterizing the dynamic behavior of the code transformed by BulkCompiler. AR stands for Atomic Region.

are confident that our techniques can improve the performance of relaxed memory models as well. Work by Wenisch *et al* [32] and Blundell *et al* [3] point to the potential of these ideas.

Finally, this work is applicable beyond BulkSC to all all-the-time group-commit architectures and, with some extensions, to conventional architectures that support hardware TM.

8. RELATED WORK

8.1 Software-Only Sequential Consistency

There have been three major software-only efforts to enforce SC in programs that are not well synchronized. The most sophisticated one is the Pensieve Project [28], which provides SC for Java. Their SC compiler uses a combination of escape analysis [28], thread-structure analysis [28], delay set analysis [26, 28], and an optimized fence-insertion algorithm [10]. All but the fence-insertion algorithm are interprocedural analyses that are fairly complex. Overall, their method induces slowdowns of over 10% on average over the relaxed Java Memory Model.

Liblit *et al* [17] developed an SC version of Titanium [13]. In the same project, Krishnamurthy and Yelick [14] showed how the regular structure of SPMD programs could be exploited to reduce the complexity of delay set analysis in those programs. Finally, Von Praun and Gross [31] used an object-based analysis for delay set analysis to determine reference orders that needed to be enforced because of inter-thread conflicts. Overall, none of these methods reported speedups for applications, and some reported significant slowdowns in one or more applications. In contrast, our combined hardware-software SC scheme delivers speedups over the relaxed Java Memory Model.

8.2 Exploiting Support for Atomicity

There has been substantial recent work on exploiting hardware support for atomicity. The Transmeta Code Morphing concept involved aggressively optimizing the code with speculative transformations [8]. It appears that most of the optimizations were for single-thread execution. Neelakantam *et al* [21] sped-up hot sections of the code by developing an optimized, speculative “trace” of the code and running it under hardware atomicity. If the code takes an unexpected control path, the section is squashed and the full version of the code is executed. They largely focus on optimizing single-thread execution, typically in loop iterations, although they mention the application of SLE to critical sections. BulkCompiler differs in its emphasis on grouping many low-contention critical sections in a large atomic region to enable conventional optimizations. It also differs in its goal to support SC.

Carlstrom *et al* [5] take lock-based Java programs and convert them into transactions. They describe how critical sections and other constructs are converted into transactions. However, they neither mention whether this change enables compiler optimizations nor are they focused on SC. Other authors such as Ziarek *et al* [33] and Rossbach *et al* [24] have studied environments that integrate locks and transactions, finding some of the problems we faced.

Re-writing a critical section with a synchronization-free fast path executing under atomic hardware, and a slow path with the complete code has been proposed in SLE [23] and used in TM libraries [9].

9. CONCLUSIONS

A platform that provides high-performance SC at the hardware and software levels for all codes, including those with data races, will substantially simplify the task of programmers. This paper presented the hardware-compiler interface, and the main ideas for *BulkCompiler*, a compiler layer that works with the BulkSC chunking hardware to provide a *whole-system high-performance SC platform*. Our specific contributions included: (i) ISA primitives for BulkCompiler to interface to the chunking hardware, (ii) compiler algorithms to drive chunking and code transformations to exploit chunks, and (iii) initial results of our algorithms on Java programs.

Our results used Java application suites modified with our compiler algorithms and compiled with Sun’s Hotspot server compiler. A whole-system SC environment with BulkCompiler and simulated BulkSC hardware outperformed a simulated conventional hardware platform that used the more relaxed Java Memory Model by an average of 37%. The speedups came from code optimization inside software-assembled instruction chunks.

This work is applicable beyond BulkSC to all group-commit architectures and, with some extensions, to conventional architectures that support hardware TM. We are now extending BulkCompiler to drive novel compiler optimizations for single- and multi-threading, and to apply them to relaxed memory models as well.

10. ACKNOWLEDGMENTS

We thank the anonymous reviewers and the I-ACOMA group members for their comments. This work was supported in part by the National Science Foundation under grants CNS 07-20593 and CCR 03-25603; Intel and Microsoft under the Universal Parallel Computing Research Center (UPCRC); Sun Microsystems under the University of Illinois OpenSPARC Center of Excellence; and a gift from IBM.

11. REFERENCES

[1] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *Western Reseach Laboratory-Compaq. Research Report 95/7*, September 1995.

[2] E. Berk. JLex: A Lexical Analyzer Generator for Java. `http://www.cs.princeton.edu/~appel/modern/java/JLex/`.

[3] C. Blundell, M. M. Martin, and T. F. Wenisch. InvisiFence: Performance-Transparent Memory Ordering in Conventional Multiprocessors. In *International Symposium on Computer Architecture*, June 2009.

[4] H. Cain and M. Lipasti. Memory Ordering: A Value-Based Approach. In *International Symposium on Computer Architecture*, June 2004.

[5] B. Carlstrom et al. Transactional Execution of Java Programs. In *Workshop on Synchronization and Concurrency in Object-Oriented Languages*, October 2005.

[6] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *International Symposium on Computer Architecture*, June 2007.

[7] S. Chaudhry et al. Simultaneous Speculative Threading: A Novel Pipeline Architecture Implemented in Sun's ROCK Processor. In *International Symposium on Computer Architecture*, June 2009.

[8] J. Dehnert et al. The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges. In *International Symposium on Code Generation and Optimization*, March 2003.

[9] D. Dice et al. Applications of the Adaptive Transactional Memory Test Platform. In *Workshop on Transactional Computing*, February 2008.

[10] X. Fang, J. Lee, and S. P. Midkiff. Automatic Fence Insertion for Shared Memory Multiprocessing. In *International Conference on Supercomputing*, June 2003.

[11] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *International Symposium on Computer Architecture*, May 1999.

[12] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *International Symposium on Computer Architecture*, June 2004.

[13] A. Kamil, J. Su, and K. A. Yelick. Making Sequential Consistency Practical in Titanium. In *International Conference on Supercomputing*, November 2005.

[14] A. Krishnamurthy and K. A. Yelick. Analyses and Optimizations for Shared Address Space Programs. *Journal of Parallel and Distributed Computing*, November 1996.

[15] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, September 1979.

[16] K. Lee and S. P. Midkiff. A Two-Phase Escape Analysis for Parallel Java Programs. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2006.

[17] B. Liblit, A. Aiken, and K. A. Yelick. Type Systems for Distributed Data Sharing. In *International Static Analysis Symposium*, June 2003.

[18] M. Martin, C. Blundell, and E. Lewis. Subtleties of Transactional Memory Atomicity Semantics. *IEEE Computer Architecture Letters*, July 2006.

[19] M. Musuvathi and S. Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *International Symposium on Programming Language Design and Implementation*, June 2007.

[20] G. Naumovich and G. Avrunin. A Conservative Data Flow Algorithm for Detecting All Pairs of Statements that May Happen in Parallel. In *International Symposium on Foundations of Software Engineering*, November 1998.

[21] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware Atomicity for Reliable Software Speculation. In *International Symposium on Computer Architecture*, June 2007.

[22] M. Paleczny, C. Vick, and C. Click. The Java HotspotTM Server Compiler. In *Symposium on JavaTM Virtual Machine Research and Technology Symposium*, April 2001.

[23] R. Rajwar and J. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *International Symposium on Microarchitecture*, December 2001.

[24] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, B. Aditya, and E. Witchel. TxLinux: Using and Managing Hardware Transactional Memory in an Operating System. In *Symposium on Operating Systems Principles*, October 2007.

[25] K. Russell and D. Detlefs. Eliminating Synchronization-Related Atomic Operations with Biased Locking and Bulk Rebiasing. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 2006.

[26] D. Shasha and M. Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *Transactions on Programming Languages and Systems*, April 1988.

[27] Sun Microsystems. OpenJDK. `http://openjdk.java.net/`.

[28] Z. Sura, X. Fang, C.-L. Wong, S. P. Midkiff, J. Lee, and D. Padua. Compiler Techniques for High Performance Sequentially Consistent Java Programs. In *Symposium on Principles and Practice of Parallel Programming*, June 2005.

[29] E. Vallejo, M. Galluzzi, A. Cristal, F. Vallejo, R. Beivide, P. Stenstrom, J. Smith, and M. Valero. Implementing Kilo-Instruction Multiprocessors. *International Conference on Pervasive Services*, July 2005.

[30] Virtutech. Simics. `http://www.simics.net/`.

[31] C. von Praun and T. R. Gross. Static Conflict Analysis for Multi-Threaded Object-Oriented Programs. In *Conference on Programming Language Design and Implementation*, June 2003.

[32] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for Store-Wait-Free Multiprocessors. In *International Symposium on Computer Architecture*, June 2007.

[33] L. Ziarek et al. A Uniform Transactional Execution Environment for Java. In *European Conference on Object-Oriented Programming*, July 2008.

2011 International Conference on Parallel Architectures and Compilation Techniques

DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism*

Byn Choi,† Rakesh Komuravelli,† Hyojin Sung,† Robert Smolinski,† Nima Honarmand,†
Sarita V. Adve,† Vikram S. Adve,† Nicholas P. Carter,‡ and Ching-Tsun Chou‡

†Department of Computer Science
University of Illinois at Urbana-Champaign
denovo@cs.illinois.edu

‡Intel
Santa Clara, CA
{nicholas.p.carter, ching-tsun.chou}@intel.com

Abstract—**For parallelism to become tractable for mass programmers, shared-memory languages and environments must evolve to enforce disciplined practices that ban "wild shared-memory behaviors;" e.g., unstructured parallelism, arbitrary data races, and ubiquitous non-determinism. This software evolution is a rare opportunity for hardware designers to rethink hardware from the ground up to exploit opportunities exposed by such disciplined software models. Such a co-designed effort is more likely to achieve many-core scalability than a software-oblivious hardware evolution.**

This paper presents DeNovo, a hardware architecture motivated by these observations. We show how a disciplined parallel programming model greatly simplifies cache coherence and consistency, while enabling a more efficient communication and cache architecture. The DeNovo coherence protocol is simple because it eliminates transient states – verification using model checking shows 15X fewer reachable states than a state-of-the-art implementation of the conventional MESI protocol. The DeNovo protocol is also more extensible. Adding two sophisticated optimizations, flexible communication granularity and direct cache-to-cache transfers, did not introduce additional protocol states (unlike MESI). Finally, DeNovo shows better cache hit rates and network traffic, translating to better performance and energy. Overall, a disciplined shared-memory programming model allows DeNovo to seamlessly integrate message passing-like interactions within a global address space for improved design complexity, performance, and efficiency.

I. Introduction

Achieving the promise of Moore's law will require harnessing increasing amounts of parallelism using multicores, with predictions of hundreds of cores per chip. Shared-memory is arguably the most widely used general-purpose multicore parallel programming model. While shared-memory provides the advantage of a global address space, it is known to be difficult to program, debug, and maintain [52]. Specifically, unstructured parallel control, data races, and ubiquitous non-determinism make programs difficult to understand, and sacrifice safety, modularity, and composability. At the same time, designing performance-, power-, and complexity-scalable hardware for such a software model remains a major challenge; e.g., directory-based cache coherence protocols are notoriously complex [3] and hard to scale and an active area of research [72, 37, 54, 63, 57]. More fundamentally, a satisfactory definition of memory consistency semantics (i.e., specification of what value a shared-memory read should return) for such a model has proven elusive, and a recent paper makes the case for rethinking programming languages and hardware to enable usable memory consistency semantics [4].

*This work is supported in part by Intel and Microsoft through the Universal Parallel Computing Research Center (UPCRC) at Illinois and by the National Science Foundation under grant number CCF-1018796. We thank Craig Zilles for discussions in the initial phase of this project and for the project name.

The above problems have led some researchers to promote abandoning shared-memory altogether (e.g., [52]). Some projects do away with coherent caches, most notably the 48 core Intel Single-Chip Cloud Computer [43], pushing significant complexity into the programming model. An alternative view is that the above problems are not inherent to a global address space paradigm, but instead occur due to undisciplined programming models that allow arbitrary reads and writes for implicit and unstructured communication and synchronization. This results in "wild shared-memory" behaviors with unintended data races and non-determinism and implicit side effects. The same phenomena result in complex hardware that must assume that any memory access may trigger communication, and performance- and power-inefficient hardware that is unable to exploit communication patterns known to the programmer but obfuscated by the programming model.

There is much recent software work on more *disciplined* shared-memory programming models to address the above problems (Section I-B). This paper concerns the first step of a hardware project, DeNovo, that asks the question: *if software becomes more disciplined, can we build more performance-, power-, and complexity-scalable hardware?* Specifically, this paper focuses on the impact of disciplined software on the cache coherence protocol.

A. Hardware Coherence Scaling Issues

Shared-memory systems typically implement coherence with snooping or directory-based protocols. Although current directory-based protocols are more scalable than snooping protocols, they suffer from several limitations:

Performance and power overhead: They incur several sources of latency and traffic overhead, impacting performance and power; e.g., they require invalidation and acknowledgment messages (which are strictly overhead) and indirection through the directory for cache-to-cache transfers.

Verification complexity and extensibility: They are notoriously complex and difficult to verify since they require dealing with subtle races and many transient states [60, 34]. Furthermore, their fragility often discourages implementors from adding optimizations to previously verified protocols – additions usually require re-verification due to even more states and races.

State overhead: Directory protocols incur high directory storage overhead to track sharer lists. Several optimized directory organizations have been proposed, but also require considerable overhead and/or excessive network traffic and/or complexity. These protocols also require several coherence state bits due to the large number of protocol states (e.g., ten bits in [63]). This state overhead is amortized by tracking coherence at the granularity of cache lines. This can result in performance/power anomalies and inefficiencies when the granularity of sharing is different from a contiguous cache line (e.g., false sharing).

Researchers continue to propose new hardware directory organizations and protocol optimizations to address one or more of the above limitations [72, 37, 54, 50, 1, 62, 67]; however, all of these approaches incur one or more of complexity, performance, power, or storage overhead. Recently, Kaxiras and Keramidas [45] exploited the data-race-free property of current memory consistency models to address the performance and power costs of the directory-based MESI protocol. DeNovo is a hardware-software co-designed approach that exploits emerging disciplined software properties in addition to data-race-freedom to target all the above mentioned limitations of directory protocols for large core counts.

B. Software Scope

There has been much recent research on disciplined shared-memory programming models with explicit and structured communication and synchronization for both deterministic and non-deterministic algorithms [7]; e.g., Ct [33], CnC [22], Cilk++ [18], Galois [49], SharC [10], Kendo [61], Prometheus [9], Grace [14], Axum [35], and Deterministic Parallel Java (DPJ) [21, 20].

Although the targeted disciplined programming models are still under active research, many of them guarantee determinism. We focus this paper on deterministic codes for three reasons. (1) There is a growing view that deterministic algorithms will be common, at least for client-side computing [7]. (2) Focusing on these codes allows us to investigate the "best case;" i.e., the potential gain from exploiting strong discipline. (3) These investigations will form a basis on which we develop the extensions needed for other classes of codes in the future; in particular, disciplined non-determinism, legacy software, and programming models using "wild shared-memory." Synchronization mechanisms involve races and are used in all classes of codes; here, we assume special techniques to implement them (e.g., hardware barriers, queue based locks, etc.) and postpone their detailed handling to future work. We also postpone OS redesign work, and hope to leverage recent work on "disciplined" OS; e.g., [13, 42] and [25].

We use Deterministic Parallel Java (DPJ) [21] as an exemplar of the emerging class of deterministic-by-default languages (Section II), and use it to explore how hardware can take advantage of strong disciplined programming features. Specifically, we use three features of DPJ that are also common to several other projects: (1) structured parallel control; (2) data-race-freedom, and guaranteed deterministic semantics unless the programmer explicitly requests non-determinism (called determinism-by-default); and (3) explicit specification of the side effects of parallel sections; e.g., which (possibly non-contiguous) regions of shared-memory will be read or written in a parallel section.

Most of the disciplined models projects cited above also enforce a requirement of structured parallel control (e.g., a nested fork join model, pipelining, etc.), which is much easier to reason about than arbitrary (unstructured) thread synchronization. Most of these, including all but one of the commercial systems, guarantee the absence of data races for programs that type-check. Coupled with structured parallel control, the data-race-freedom property guarantees determinism for several of these systems. We also note that data races are prohibited (although not checked) by existing popular languages as well; the emerging C++ and C memory models do not provide *any* semantics with any data race (benign or otherwise) and Java provides extremely complex and weak semantics for data races only for the purposes of ensuring safety. The information about side effects of concurrent tasks is also available in other disciplined languages, but in widely varying (and sometimes indirect) ways. Once we understand the types of information that is most valuable, our future work includes exploring how the information can be extracted from programs in other languages.

C. Contributions

DeNovo starts from language-level annotations designed for concurrency safety, and shows that they can be efficiently represented and used in hardware for better complexity and scalability. Two key insights underlie our design. First, structured parallel control and knowing which memory regions will be read or written enable a cache to take responsibility for invalidating its own stale data. Such self-invalidations remove the need for a hardware directory to track sharer lists and to send invalidations and acknowledgements on writes. Second, data-race-freedom eliminates concurrent conflicting accesses and corresponding transient states in coherence protocols, eliminating a major source of complexity. Our specific results are as follows, applied to a range of array and complex pointer-based applications.

Simplicity: To provide quantitative evidence of the simplicity of the DeNovo protocol, we compared it with a conventional MESI protocol by implementing both in the Murphi model checking tool [29]. For MESI, we used the implementation in the Wisconsin GEMS simulation suite [56] as an example of a (publicly available) state-of-the-art, mature implementation. We found several bugs in MESI that involved subtle data races and took several days to debug and fix. The debugged MESI showed 15X more reachable states compared to DeNovo, with a verification time difference of 173 seconds vs 8.66 seconds.

Extensibility: To demonstrate the extensibility of the DeNovo protocol, we implemented two optimizations: (1) Direct cache-to-cache transfer: Data in a remote cache may directly be sent to another cache without indirection to the shared lower level cache (or directory). (2) Flexible communication granularity: Instead of always sending a fixed cache line in response to a demand read, we send a programmer directed set of data associated with the region information of the demand read. Neither optimization required adding any new protocol states to DeNovo; since there are no sharer lists, valid data can be freely transferred from one cache to another.

Storage overhead: Our protocol incurs no storage overhead for directory information. On the other hand, we need to maintain information about regions and coherence state bits at the granularity at which we guarantee data-race freedom, which can be less than a cache line. For low core counts, this overhead is higher than with conventional directory schemes, but it pays off after a few tens of cores and is scalable (constant per cache line). A positive side effect is that it is easy to eliminate the requirement of inclusivity in a shared last level cache (since we no longer track sharer lists). Thus, DeNovo allows more effective use of shared cache space.

Performance and power: In our evaluations, the base DeNovo protocol showed about the same or better memory behavior than the MESI protocol. With the optimizations described, DeNovo saw a reduction in memory stall time of up to 81% compared to MESI. In most cases, these stall time reductions came from commensurate reductions in miss rate and were accompanied with significant reductions in network traffic, thereby benefiting not only execution time but also power.

II. Background: Deterministic Parallel Java

DPJ is an extension to Java that enforces *deterministic-by-default* semantics via compile-time type checking [21, 20]. DPJ provides a new type and effect system for expressing important patterns of deterministic and non-deterministic parallelism in imperative, object-oriented programs. Non-deterministic behavior can only be obtained via certain explicit constructs. For a program that does not use such constructs, DPJ guarantees that if the program is well-typed, any two parallel tasks are *non-interfering*, i.e., do not have conflicting accesses.

DPJ's parallel tasks are iterations of an explicitly parallel `foreach` loop or statements within a `cobegin` block; they synchronize through an implicit barrier at the end of the loop or block. Parallel control flow thus follows a scoped, nested, fork-join structure, which simplifies the use of explicit coherence actions in DeNovo at fork/join points. This structure defines a natural ordering of the tasks, as well as an obvious definition (omitted here) of when two tasks are "concurrent". It implies an obvious sequential equivalent of the parallel program (`for` replaces `foreach` and `cobegin` is simply ignored). DPJ guarantees that the result of a parallel execution is the same as the sequential equivalent.

In a DPJ program, the programmer assigns every object field or array element to a named "*region*" and annotates every method with read or write "*effects*" summarizing the regions read or written by that method. The compiler checks that (i) all program operations are type safe in the region type system; (ii) a method's effect summaries are a superset of the actual effects in the method body; and (iii) that no two parallel statements interfere. The effect summaries on method interfaces allow all these checks to be performed without interprocedural analysis.

For DeNovo, the effect information tells the hardware what fields will be read or written in each parallel "phase" (`foreach` or `cobegin`). This enables efficient software-controlled coherence mechanisms and powerful communication management, discussed in the following sections.

DPJ has been evaluated on a wide range of deterministic parallel programs. The results show that DPJ can express a wide range of realistic parallel algorithms, and that well-tuned DPJ programs exhibit good performance [21].

III. DeNovo Coherence and Consistency

A shared-memory design must first and foremost ensure that a read returns the correct value, where the definition of "correct" comes from the memory consistency model. Modern systems divide this responsibility between two parts: (i) cache coherence, and (ii) various memory ordering constraints. These are arguably among the most complex and hard to scale aspects of shared-memory hierarchy design. Disciplined models enable mechanisms that are potentially simpler and more efficient to achieve this function.

The deterministic parts of our software have semantics corresponding to those of the equivalent sequential program. A read should therefore simply return the value of the last write to the same location that is before it in the deterministic sequential program order. This write either comes from the reader's own task (if such a write exists) or from a task preceding the reader's task, since there can be no conflicting accesses concurrent with the reader (two accesses are concurrent if they are from concurrent tasks). In contrast, conventional (software-oblivious) cache coherence protocols assume that writes and reads to the same location can happen concurrently, resulting in significant complexity and inefficiency.

To describe the DeNovo protocol, we first assume that the coherence granularity and address/communication granularity are the same. That is, the data size for which coherence state is maintained is the same as the data size corresponding to an address tag in the cache and the size communicated on a demand miss. This is typically the case for MESI protocols, where the cache line size (e.g., 64 bytes) serves as the address, communication, and coherence granularity. For DeNovo, the coherence granularity is dictated by the granularity at which data-race-freedom is ensured – a word for our applications. Thus, this assumption constrains the cache line size. We henceforth refer to this as the word based version of our protocol. We relax this assumption in Section III-B, where we decouple the address/communication and coherence granularities and also enable sub-word coherence granularity.

Without loss of generality, throughout we assume private and writeback L1 caches, a shared last-level on-chip L2 cache inclusive of only the modified lines in any L1, a single (multicore) processor chip system, and no task migration. The ideas here extend in an obvious way to deeper hierarchies with multiple private and/or cluster caches and multichip multiprocessors, and task migration can be accommodated with appropriate self-invalidations before migration. Below, we use the term *phase* to refer to the execution of all tasks created by a single parallel construct (foreach or cobegin).

A. DeNovo with Equal Address/Communication and Coherence Granularity

DeNovo eliminates the drawbacks of conventional directory protocols as follows.

No directory storage or write invalidation overhead: In conventional directory protocols, a write acquires ownership of a line by invalidating all other copies, to ensure later reads get the updated value. The directory achieves this by tracking all current sharers and invalidating them on a write, incurring significant storage and invalidation traffic overhead. In particular, straightforward bit vector implementations of sharer lists are not scalable. Several techniques have been proposed to reduce this overhead, but typically pay a price in significant increase in complexity and/or incurring unnecessary invalidations when the directory overflows. DeNovo eliminates these overheads by removing the need for ownership on a write. Data-race-freedom ensures there is no other writer or reader for that line in this parallel phase. DeNovo need only ensure that (i) outdated cache copies are invalidated before the next phase, and (ii) readers in later phases know where to get the new data.

For (i), each cache simply uses the known write effects of the current phase to invalidate its outdated data before the next phase begins. The compiler inserts self-invalidation instructions for each region with these write effects (we describe how regions are conveyed and represented below). Each L1 cache invalidates its data that belongs to these regions with the following exception. Any data that the cache has read or written in this phase is known to be up-to-date since there cannot be concurrent writers. We therefore augment each line with a "touched" bit that is set on a read. A self-invalidation instruction does not invalidate a line with a set touched bit or that was last written by this core (indicated by the `registered` state as discussed below); the instruction resets the touched bit in preparation for the next phase.

For (ii), DeNovo requires that on a write, a core register itself at (i.e., inform) the shared L2 cache. The L2 data banks serve as the registry. An entry in the L2 data bank either keeps the identity of an L1 that has the up-to-date data (`registered` state) or the data itself (`valid` state) – a data bank entry is never required to keep both pieces of information since an L1 cache registers itself in precisely the case where the L2 data bank does not have the up-to-date data. Thus, DeNovo entails *zero overhead for directory (registry) storage*. Henceforth, we use the term L2 cache and registry interchangeably.

We also note that because the L2 does not need sharer lists, it is natural to not maintain inclusion in the L2 for lines that are not registered by another L1 cache – the registered lines do need space in the L2 to track the L1 id that registered them.

No transient states: The DeNovo protocol has three states in the L1 and L2 – `registered`, `valid`, and `invalid` – with obvious meaning. (The touched bit mentioned above is local to its cache and irrelevant to external coherence transactions.) Although textbook descriptions of conventional directory protocols also describe 3 to 5 states (e.g., MSI) [40], it is well-known that they contain many hidden transient states due to races, making them notoriously complex and difficult to verify [3, 65, 70]. For example, considering a simple MSI protocol, a cache may request ownership, the directory may forward the request to the current owner, and another cache may request ownership while all of these messages are still outstanding. Proper handling of such a race requires introduction of transient states into the cache and/or directory transition tables.

DeNovo, in contrast, is a true 3-state protocol with *no transient states*, since it assumes race-free software. The only possible races are related to writebacks. As discussed below, these races either have limited scope or are similar to those that occur in uniprocessors. They can be handled in straightforward ways, without transient protocol states (described below).

The full protocol: Table I shows the L1 and L2 state transitions and events for the full protocol. Note the lack of transient states in the caches.

Read requests to the L1 (from L1's core) are straightforward – accesses to valid and registered state are hits and accesses to invalid state generate miss requests to the L2. A read miss does not have to leave the L1 cache in a pending or transient state – since there are no concurrent conflicting accesses (and hence no invalidation requests), the L1 state simply stays invalid for the line until the response comes back.

For a write request to the L1, unlike a conventional protocol, there is no need to get a "permission-to-write" since this permission is implicitly given by the software race-free guarantee. If the cache does not already have the line registered, it must issue a registration request to the L2 to notify that it has the current up-to-date copy of the line and set the registry state appropriately. Since there are no races, the write can *immediately* set the state of the cache to registered, without waiting for the registration request to complete. Thus, *there is no transient or pending state for writes either*.

The pending read miss and registration requests are simply monitored in the processor's request buffer, just like those of other reads and writes for a single core system. Thus, although the request buffer technically has transient states, these are not visible to external requests – external requests only see stable cache states. The request buffer also ensures that its core's requests to the same location are serialized to respect uniprocessor data dependencies, similar to a single core implementation (e.g., with MSHRs). The memory model requirements are met by ensuring that all pending requests from the core complete by the end of this parallel phase (or at least before the next conflicting access in the next parallel phase).

The L2 transitions are also straightforward except for writebacks which require some care. A read or registration request to data that is invalid or valid at the L2 invokes the obvious response. For a request for data that is registered by an L1, the L2 forwards the request to that L1 and updates its registration id if needed. For a forwarded registration request, the L1 always acknowledges the requestor and invalidates its own copy. If the copy is already invalid due to a concurrent writeback by the L1, the L1 simply acknowledges the original requestor and the L2 ensures that the writeback is not accepted (by noting that it is not from the current registrant). For a forwarded read request, the L1 supplies the data if it has it. If it no longer has the data (because it issued a concurrent writeback), then it sends a negative acknowledgement (nack) to the original requestor, which simply resends the request to the L2. Because of race-freedom, there cannot be another concurrent write, and so no other concurrent writeback, to the line. Thus, the nack eventually finds the line in the L2, without danger of any deadlock or livelock. The only somewhat less straightforward interaction is when both the L1 and L2 caches want to writeback the same line concurrently, but this race also occurs in uniprocessors.

Conveying and representing regions in hardware: A key research question is how to represent regions in hardware for self-invalidations. Language-level regions are usually much more fine-grain than may be practical to support in hardware. For example, when a parallel loop traverses an array of objects, the compiler may need to identify (a field of) *each object* as being in a distinct region in order to prove the absence of conflicts. For the hardware, however, such fine distinctions would be expensive to maintain. Fortunately, we can coarsen language-level regions to a much smaller set without losing functionality in hardware. The key insight is as follows. For self-invalidations, we need regions to identify which data could have been written in the current phase. It is not important to distinguish which core wrote which data. In the above example, we can thus treat the entire array of

	$Read_i$	$Write_i$	$Read_k$	$Register_k$	Response for $Read_i$	Writeback
$Invalid$	Update tag; Read miss to L2; Writeback if needed	Go to $Registered$; Reply to core i; Register request to L2; Write data; Writeback if needed	Nack to core k	Reply to core k	If tag match, go to $Valid$ and load data; Reply to core i	Ignore
$Valid$	Reply to core i	Go to $Registered$; Reply to core i; Register request to L2	Send data to core k	Go to $Invalid$; Reply to core k	Reply to core i	Ignore
$Registered$	Reply to core i	Reply to core i	Reply to core k	Go to *Invalid*; Reply to core k	Reply to core i	Go to *Valid*; Writeback

(a) **L1 cache of core i. $Read_i$ = read from core i, $Read_k$ = read from another core k (forwarded by the registry).**

	Read miss from core i	Register request from core i	Read response from memory for core i	Writeback from core i
$Invalid$	Update tag; Read miss to memory; Writeback if needed	Go to $Registered_i$; Reply to core i; Writeback if needed	If tag match, go to $Valid$ and load data; Send data to core i	Reply to core i; Generate reply for pending writeback to core i
$Valid$	Data to core i	Go to $Registered_i$; Reply to core i	X	X
$Registered_j$	Forward to core j; Done	Forward to core j; Done	X	if i==j go to $Valid$ and load data; Reply to core i; Cancel any pending Writeback to core i

(b) **L2 cache**

TABLE I: Baseline DeNovo cache coherence protocol for (a) private L1 and (b) shared L2 caches. Self-invalidation and touched bits are not shown here since these are local operations as described in the text. Request buffers (MSHRs) are not shown since they are similar to single core systems.

objects as one region.

Alternately, if only a subset of the fields in each object in the above array is written, then this subset aggregated over all the objects collectively forms a hardware region. Thus, just like software regions, hardware regions need not be contiguous in memory – they are essentially an assignment of a color to each heap location (with orders of magnitude fewer colors in hardware than software). Hardware regions are not restricted to arrays either. For example, in a traversal of the spatial tree in an n-body problem, the compiler distinguishes different tree nodes (or subsets of their fields) as separate regions; the hardware can treat the entire tree (or a subset of fields in the entire tree) as an aggregate region. Similarly, hardware regions may also combine field regions from different aggregate objects (e.g., fields from an array and a tree may be combined into one region).

The compiler can easily *summarize* program regions into coarser hardware regions as above and insert appropriate self-invalidation instructions. The only correctness requirement is that the self-invalidated regions must cover all write effects for the phase. For performance, these regions should be as precise as possible. For example, fields that are not accessed or read-only in the phase should not be part of these regions. Similarly, multiple field regions written in a phase may be combined into one hardware region for that phase, but if they are not written together in other phases, they will incur unnecessary invalidations.

During final code generation, the memory instructions generated can convey the region name of the address being accessed to the hardware; since DPJ regions are parameterizable, the instruction needs to point to a hardware register that is set at runtime (through the compiler) with the actual region number. When the memory instruction is executed, it conveys the region number to the core's cache. A straightforward approach is to store the region number with the accessed data line in the cache. Now a self-invalidate instruction invalidates all data in the cache with the specified regions that is not `touched` or `registered`.

The above implementation requires storing region bits along with data in the L1 cache and matching region numbers for self-invalidation. A more conservative implementation can reduce this overhead. At the beginning of a phase, the compiler conveys to the hardware the set of regions that need to be invalidated in the *next* phase – this set can be conservative, and in the worst case, represent all regions. Additionally, we replace the region bits in the cache with one bit: `keepValid`. indicating that the corresponding data need not be invalidated until the end of the *next* phase. On a miss, the hardware compares the region for the accessed data (as indicated by the memory instruction) and the regions to be invalidated in the next phase. If there is no match, then `keepValid` is set. At the end of the phase, all data not `touched` or `registered` are invalidated and the `touched` bits reset as before. Further, the identities of the `touched` and `keepValid` bits are swapped for the next phase. This technique allows valid data to stay in cache through a phase even if it is not `touched` or `registered` in that phase, without keeping track of regions in the cache. The concept can be extended to more than one such phase by adding more bits and if the compiler can predict the self-invalidation regions for those phases.

Example: Figure 1 illustrates the above concepts. Figure 1(a) shows a code fragment with parallel phases accessing an array, S, of structs with three fields each, X, Y, and Z. The X (respectively, Y and Z) fields from all array elements form one DeNovo region. The first phase writes the region of X and self-invalidates that region at the end. Figure 1(b) shows, for a two core system, the L1 and L2 cache states at the end of Phase 1, assuming each core computed one contiguous half of the array. The computed X fields are `registered` and the others are invalid in the L1's while the L2 shows all X fields registered to the appropriate cores. (The direct communication is explained in the next section.)

B. DeNovo with Address/Communication Granularity > Coherence Granularity

To decouple the address/communication and coherence granularity, our key insight is that any data marked `touched` or `registered` can be copied over to any other cache in `valid` state (but not as `touched`). Additionally, for even further optimization (Section III-D1), we make the observation that this transfer can happen without going through the registry/L2 at all (because the registry does not track sharers). Thus, no serialization at a directory is required. When (if) this copy of data is accessed through a demand read, it can be immediately marked `touched`. The above copy does not incur false sharing (nobody loses ownership) and, if the source is the non-home node, it does not require extra hops to a directory.

With the above insight, we can easily enhance the baseline word-based DeNovo protocol from the previous section to operate on a larger communication and address granularity; e.g., a typical cache line size from conventional protocols. However, we still maintain coherence state at the granularity at which the program guarantees data race freedom; e.g., a word. On a demand request, the cache servicing the request can send an entire cache line worth of data, albeit with some of the data marked invalid (those that it does not have as `touched` or `registered`). The requestor then merges the valid words in the response message (that it does not already have `valid` or `registered`) with its copy of the cache line (if it has one), marking all of those words as `valid` (but not `touched`).

Note that if the L2 has a line `valid` in the cache, then an element of that line can be either `valid` (and hence sent to the requestor) or `registered` (and hence not sent). Thus, for the L2, it suffices to keep just one coherence state bit at the finer (e.g., word) granularity with a line-wide valid bit at the line granularity.[1] As before, the id of the registered core is stored in the data array of the registered location.

This is analogous to sector caches – cache space allocation (i.e., address tags) is at the granularity of a line but there may be some data within the line that is not valid. This combination effectively allows exploiting spatial locality without any false sharing, similar to multiple writer protocols of software distributed shared memory systems [46].

C. Flexible Coherence Granularity

Although the applications we studied did not have any data races at word granularity, this is not necessarily true of all applications. Data may be shared at byte granularity, and two cores may incur conflicting concurrent accesses to the same word, but for different bytes. A straightforward implementation would require coherence state at the granularity of a byte, which would be significant storage overhead. [2] Although previous work has suggested using byte based granularity for state bits in other contexts [53], we would like to minimize the overhead.

[1]This requires that if a registration request misses in the L2, then the L2 obtain the full line from main memory.

[2]The upcoming C and C++ memory models and the Java memory model do not allow data races at byte granularity; therefore, we also do not consider a coherence granularity lower than that of a byte.

We focus on the overhead in the L2 cache since it is typically much larger (e.g., 4X to 8X times larger) than the L1. We observe that byte granularity coherence state is needed only if two cores incur conflicting accesses to different bytes in the same word in the same phase. Our approach is to make this an infrequent case, and then handle the case correctly albeit at potentially lower performance.

In disciplined languages, the compiler/runtime can use the region information to allocate tasks to cores so that byte granularity regions are allocated to tasks at word granularities when possible. For cases where the compiler (or programmer) cannot avoid byte granularity data races, we require the compiler to indicate such regions to the hardware. Hardware uses word granularity coherence state. For byte-shared data such as the above, it "clones" the cache line containing it in four places: place i contains the ith byte of each word in the original cache line. If we have at least four way associativity in the L2 cache (usually the case), then we can do the cloning in the same cache set. The tag values for all the clones will be the same but each clone will have a different byte from each word, and each byte will have its own coherence state bit to use (essentially the state bit of the corresponding word in that clone). This allows hardware to pay for coherence state at word granularity while still accommodating byte granularity coherence when needed, albeit with potentially poorer cache utilization in those cases.

D. Protocol Optimizations

1) Eliminating indirection: Our protocol so far suffers from the fact that even L1 misses that are eventually serviced by another L1 cache (cache-to-cache transfer) must go through the registry/L2 (directory in conventional protocols), incurring an additional latency due to the indirection.

However, as observed in Section III-B, `touched/registered` data can always be transferred for reading without going through the registry/L2. optimization). Thus, a reader can send read requests directly to another cache that is predicted to have the data. If the prediction is wrong, a Nack is sent (as usual) and the request reissued as a usual request to the directory. Such a request could be a demand load or it could be a prefetch. Conversely, it could also be a producer-initiated communication or remote write [2, 48]. The prediction could be made in several ways; e.g., through the compiler or through the hardware by keeping track of who serviced the last set of reads to the same region. The key point is that there is no impact on the coherence protocol – no new states, races, or message types. The requestor simply sends the request to a different supplier. This is in sharp contrast to adding such an enhancement to MESI.

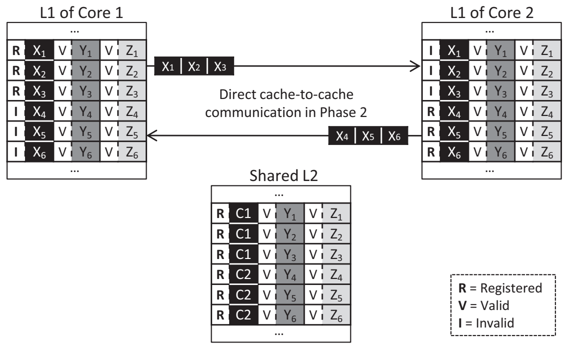
This ability essentially allows DeNovo to seamlessly integrate a message passing like interaction within its shared-memory model. Figure 1 shows such an interaction for our example code.

2) Flexible communication granularity: Cache-line based communication transfers data from a set of contiguous addresses, which is ideal for programs with perfect spatial locality and no false sharing. However, it is common for programs to access only a few data elements from each line, resulting in significant waste. This is particularly common in modern object-oriented programming styles where data

```
class S_type {
        X in DeNovo-region ■ ;
        Y in DeNovo-region ▨ ;
        Z in DeNovo-region □ ;
}
S _type S = new S_type[size];
...
Phase1 writes■          // DeNovo effect
        foreach  i in 0, size {
                S[i].X = ...;
        }
        self_invalidate( ■);
}

Phase2 reads■ , ... { ... }
...
```

(a)



(b)

Fig. 1: (a) Code with DeNovo regions and self-invalidations and (b) cache state after phase 1 self-invalidations and direct cache-to-cache communication with flexible granularity at the beginning of phase 2. X_i represents $S[i].X$. Ci in L2 cache means the word is registered with Core i. Initially, all lines in the caches are in `valid` state.

structures are often in the form of arrays of structs (AoS) rather than structs of arrays (SoA). It is well-known that converting from AoS to SoA form often gives a significant performance boost due to better spatial locality. Unfortunately, manual conversion is tedious, error-prone, and results in code that is much harder to understand and maintain, while automatic (compiler) conversion is impractical except in limited cases because it requires complex whole-program analysis and transformations [28, 44]. We exploit information about regions to reduce such communication waste, without changing the software's view.

We have knowledge of which regions will be accessed in the current phase. Thus, when servicing a remote read request, a cache could send `touched` or `registered` data only from such regions (recall these are at field granularity within structures), potentially reducing network bandwidth and power. More generally, the compiler may associate a default prefetch granularity attribute with each region that defines the size of each contiguous region element, other regions in the object likely to be accessed along with this region (along with their offset and size), and the number of such elements to transfer at a time. This information can be kept as a table in hardware which is accessed through the region identifier and an entry provides the above information; we call the table the *communication region table*. The information for the table itself may be partly obtained directly through the programmer, deduced by the compiler, or deduced by a runtime tool. Figure 1 shows an example of the use of flexible communication granularity – the caches communicate multiple (non-contiguous) fields of region X rather than the contiguous X, Y, and Z regions that would fall in a conventional cache line. Again, in contrast to MESI, the additional support required for this enhancement in DeNovo does not entail any changes to the coherence protocol states or introduce new protocol races.

This flexible communication granularity coupled with the ability to remove indirection through the registry/L2 (directory) effectively brings the system closer to the efficiency of message passing while still retaining the advantages of a coherent global address space. It combines the benefits of various previously proposed shared-memory techniques such as bulk data transfer, prefetching, and producer-initiated communication, but in a more software-aware fashion that potentially results in a simpler and more effective system.

E. Storage Overhead

We next compare the storage overhead of DeNovo to other common directory configurations.

DeNovo overhead: At the L1, DeNovo needs state bits at the word granularity. We have three states and one touched bit (total of 3 bits). We also need region related information. In our applications, we need at most 20 hardware regions – 5 bits. These can be replaced with 1 bit by using the optimization of the `keepValid` bit discussed in Section III-A. Thus, we need a total of 4 to 8 bits per 32 bits or 64 to 128 bits per L1 cache line. At the L2, we just need one valid and one dirty bit per line (per 64 bytes) and one bit per word, for a total of 18 bits per 64 byte L2 cache line or 3.4%. If we assume L2 cache size of 8X that of L1, then the L1 overhead is 1.56% to 3.12% of the L2 cache size.

In-cache full map directory. We conservatively assume 5 bits for protocol state (assuming more than 16 stable+transient states). This gives 5 bits per 64 byte cache line at the L1. With full map directories, each L2 line needs a bit per core for the sharer list. This implies that DeNovo overhead for just the L2 is better for more than a 13 core system. If the L2 cache size is 8X that of L1, then the total L1+L2 overhead of DeNovo is better at greater than about 21 (with `keepValid`) to 30 cores.

Duplicate tag directories. L1 tags can be duplicated at the L2 to reduce directory overhead. However, this requires a very high associative lookup; e.g., 64 cores with 4 way L1 requires a 256 way associative lookup. As discussed in [72], this design is not scalable to even low tens of cores system.

Tagless directories and sparse directories. The tagless directories work uses Bloom filter based directory organization [72]. Their directory storage requirement appears to be about 3% to over 5% of L1 storage for core counts ranging from 64 to 1K cores. This does not include any coherence state overhead which we include in our calculation for DeNovo above. Further, this organization is lossy in that larger core counts require extra invalidations and protocol complexity.

Many sparse directory organizations have been proposed that can drastically cut directory overhead at the cost of sharer list precision, and so come at a significant performance cost especially at higher core counts [72].

Processor Parameters	
Frequency	2GHz
Number of cores	64
Memory Hierarchy Parameters	
L1 (Data cache)	128KB
L2 (16 banks, NUCA)	32MB
Memory	4GB, 4 on-chip controllers
L1 hit latency	1 cycle
L2 hit latency	29 $\sim$ 61 cycles
Remote L1 hit latency	35 $\sim$ 83 cycles
Memory latency	197 $\sim$ 261 cycles

TABLE II: Parameters of the simulated processor.

IV. Methodology

A. Simulation Environment

Our simulation environment consists of the Simics full-system functional simulator that drives the Wisconsin GEMS memory timing simulator [56] which implements the simulated protocols. We also use the Princeton Garnet [8] interconnection network simulator to accurately model network traffic. We chose not to employ a detailed core timing model due to an already excessive simulation time. Instead, we assume a simple, single-issue, in-order core with blocking loads and 1 CPI for all non-memory instructions. We also assume 1 CPI for all instructions executed in the OS and in synchronization constructs.

Table II summarizes the key common parameters of our simulated systems. Each core has a 128KB private L1 Dcache (we do not model an Icache). L2 cache is shared and banked (512KB per core). The latencies in Table II are chosen to be similar to those of Nehalem [36], and then adjusted to take some properties of the simulated processor (in-order core, two-level cache) into account.

B. Simulated Protocols

We compared the following 8 systems:

MESI word (MW) and line (ML): MESI with single-word (4 byte) and 64-byte cache lines, respectively. The original implementation of MESI shipped with GEMS [56] does not support non-blocking stores. Since stores are non-blocking in DeNovo, we modified the MESI implementation to support non-blocking stores for a fair comparison. Our tests show that MESI with non-blocking stores outperforms the original MESI by 28% to 50% (for different applications).

DeNovo word (DW) and line (DL): DeNovo with single-word (Section III) and 64-byte cache lines, respectively.

For DL, we do not charge any additional cycles for gathering/scattering valid-only packets. We charge network bandwidth for only the valid part of the cache line plus the valid-word bit vector.

DL with direct cache-to-cache transfer (DD): Line-based DeNovo with direct cache-to-cache transfer (Section III-D1). We use oracular knowledge to determine the cache that has the data. This provides an upper-bound on achievable performance improvement.

DL with flexible communication granularity (DF): Line-based DeNovo with flexible communication granularity (Section III-D2). Here, on a demand load, the communication region table is indexed by the region of the demand load to obtain the set of addresses that are associated with that load, referred to as the *communication space*. We fix the maximum data communicated to be 64 bytes for DF. If the communication space is smaller than 64 bytes, then we choose the rest of the words from the 64-byte cache line containing the demand load address. We optimistically do not charge any additional cycles for determining the communication space and gathering/scattering that data.

DL and DW with both direct cache-to-cache transfer and flexible communication granularity (DDF and DDFW respectively): Line-based and word-based DeNovo with the above two optimizations, direct cache-to-cache transfer and flexible communication granularity, combined in the obvious way.

We do not show word based DeNovo augmented with just direct cache-to-cache transfer or just flexible communication granularity because of lack of space, the results were as expected and did not lend new insights, and the DeNovo word based implementations have too much tag overhead compared to the line based implementations.

C. Conveying Regions and Communication Space

Regions for self-invalidation: In a real system, the compiler would convey the region of a data through memory instructions (Section III). For this study, we created an API to manually instrument the program to convey this information for every allocated object. This information is maintained in a table in the simulator. At every load or store, the table is queried to find the region for that address (which is then stored with the data in the L1 cache).

Self invalidation: This API call invalidates all the data in the cache associated with the given region, if the data is not `touched` or `registered`. For the applications studied in this paper (see below), the total number of regions ranged from 2 to about 20. These could be coalesced by the compiler, but we did not explore that here.

Communication space: To convey communication granularity information, we again use a special API call that controls the communication region table of the simulator. On a demand load, the table is accessed to determine the communication space of the requested word. In an AoS program, this set can be simply defined by specifying 1) what object fields, and 2) how many objects to include in the set. For six of our benchmarks, these API calls are manually inserted. The seventh, kdTree, is more complex, so we use an automated correlation analysis tool to determine the communication spaces. We omit the details for lack of space.

D. Protocol Verification

We used the widely used Murphi model checking tool [29] to formally compare the verification complexity of DeNovo and MESI. We model checked the word-based protocol of DeNovo and MESI. We derived the MESI model from the GEMS implementation (the SLICC files) and the DeNovo model directly from our implementation. To keep the number of explored states tractable, as is common practice, we used a single address / single region (only for DeNovo), two data values, two cores with private L1 cache and a unified L2 with in-cache directory (for MESI). We modeled an unordered full network with separate request and reply links. Both models allow only one request per L1 in the rest of the memory hierarchy. For DeNovo, we modeled the data-race-free guarantee by limiting conflicting accesses. We also introduced the notion of phase boundary to provide a realistic model to both protocols by modeling it as a sense reversing barrier. This enables cross phase interactions in both protocols. As we modeled only one address to reduce the number of states explored, we modeled replacements as unconditional events that can be triggered at any time.

E. Workloads

We use seven benchmarks to evaluate the effectiveness of DeNovo features for a range of dense-array, array-of-struct, and irregular pointer-based applications. FFT (with input size m=16), LU (with 512x512 array and 16-byte blocks), Radix (with 4M integers and 1024 radix), and Barnes-Hut (16K particles) are from the SPLASH-2 benchmark suite [69]. kdTree [27] is a program for construction of k-D trees which are well studied acceleration data structures for ray tracing in the increasingly important area of graphics and visualization. We run it with the well known bunny input. We use two versions of kdTree: kdTree-false which has false sharing in an auxiliary data structure and kdTree-padded which uses padding to eliminate this false sharing. We use these two versions to analyze the effect of application-level false sharing on the DeNovo protocols. We also use fluidanimate (with simmedium input) and bodytrack (with simsmall input) from the PARSEC benchmark suite [16]. To fit into the fork-join programming model, fluidanimate was modified to use the ghost cell pattern instead of mutexes, and radix was modified to perform a parallel prefix with barriers instead of condition variables. For bodytrack, we use its pthread version unmodified.

V. Results

We focus our discussion on the time spent on memory stalls and on network traffic since DeNovo targets these components. Figures 2a, 2b, and 2c respectively show the memory stall time, read miss counts, and network traffic for all eight protocols described in Section IV-B for each application. Each bar (protocol) is normalized to the corresponding (state-of-the-art) MESI-line (ML) bar.

The memory stall time bars (Figure 2a) are divided into four components. The bottommost indicates time spent by a memory instruction stalled due to a blocked L1 cache related resource (e.g., the 64 entry buffer for non-blocking stores is full). The upper three indicate additional time spent stalled on an L1 miss that gets resolved at the L2, a remote L1 cache, or main memory respectively. The miss count bars (Figure 2b) are divided analogously. The network traffic bars (Figure 2c) show the number of flit crossings through on-chip network routers due to reads, writes, writebacks, and invalidations respectively.

For reference, Figure 2d shows the overall execution time for all the protocols and applications, divided into time spent in compute cycles, memory stalls, and synchronization stalls respectively.

LU and bodytrack show considerably large synchronization times. LU has inherent load imbalance. Using larger input sizes would reduce synchronization time, but prohibitively long simulation times made that impractical for this paper. Bodytrack has several sequential phases and a limited amount of parallelism for the input used (only up to 60 threads in some phases [17]). The idle cores in these phases result in the high synchronization time.

MESI vs. DeNovo word protocols (MW vs. DW): MW and DW are not practical protocols because of their excessive tag overhead. A comparison is instructive, however, to understand the efficacy of selective self-invalidation, independent of line-based effects such as false sharing. In all cases, DW's performance is competitive with MW. For the cases where it is slightly worse (LU, Barnes and Bodytrack), the cause is higher remote L1 hits in DW than in MW. This is because in MW, the first reader forces the last writer to writeback to L2. Thus, subsequent readers get their data from L2 for MW but need to go to the remote L1 (via L2) for DW, slightly increasing the memory stall time for DW. However, in terms of network traffic, DW always significantly outperforms MW.

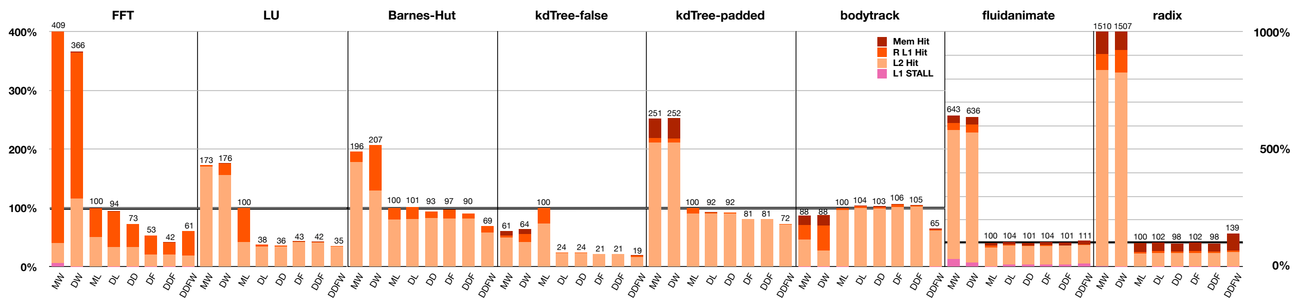
MESI vs. DeNovo line protocols (ML vs. DL): DL shows about the same or better memory stall times as ML. For LU and kdTree-false, DL shows 62% and 76% reduction in memory stall time over ML, respectively. Here, DL enjoys one major advantage over ML: DL incurs no false sharing due to its per-word coherence state. Both LU and kdTree-false contain some false sharing, as indicated by the significantly higher remote L1 hit component in the miss rate count and memory stall time graphs for ML. In terms of network traffic, DL outperforms ML except for fluidanimate and radix. Here, DL incurs more network traffic because registration (write-traffic) is still at word-granularity (shown in 2c). This can be potentially mitigated with a "write-combining" optimization that aggregates individual registration requests similar to a combining write buffer.

Effectiveness of cache lines for MESI: Comparing MW and ML, we see that the memory stall time reduction resulting from transferring a contiguous cache line instead of just a word is highly application dependent. The reduction is largest for radix (a large 93%), which has dense arrays and no false sharing. Most interestingly, for kdTree-false (object-oriented AoS style with false sharing), the word based MESI does better than the line based MESI by 39%. This is due to the combination of false sharing and less than perfect spatial locality. Bodytrack is similar in that it exhibits little spatial locality due to its irregular access pattern. Consequently, ML shows higher miss counts and memory stall times than MW (due to cache pollution from the useless words in a cache line).

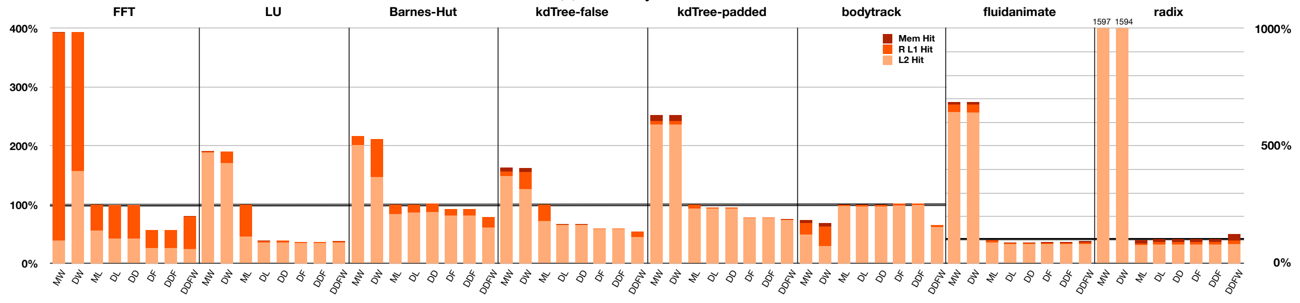
Effectiveness of cache lines for DeNovo: Comparing DW with DL, we see again the strong application dependence of the effectiveness of cache lines. However, because false sharing is not an issue with DeNovo, both LU and kdTree-false enjoy larger benefits from cache lines than in the case of MESI (78% and 63% reduction in memory stalls). Analogous to MESI, Bodytrack sees larger memory stalls with DL than with DW because of little spatial locality.

Effectiveness of direct cache-to-cache transfer with DL: FFT and barnes exhibit much opportunity for direct cache-to-cache transfer. For these applications, DD is able to significantly reduce the remote L1 hit latencies when compared to DL.

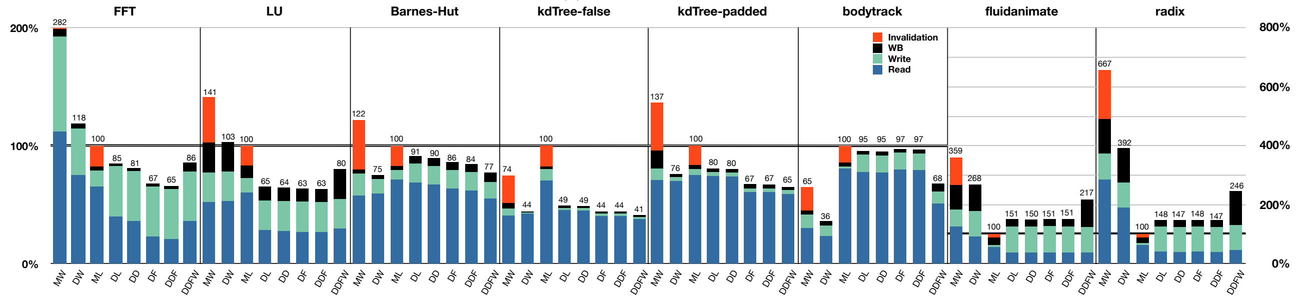
Effectiveness of flexible communication granularity with DL: DF performs about as well or better than ML and DL for all cases, except for LU. LU does not do as well because of the line granularity for cache allocation (addresses). DF can bring in data from multiple cache lines; although this data is likely to be useful, it can potentially replace a lot of allocated data. Bodytrack shows a similar phenomenon, although to a much lesser extent. As we see later, flexible communication at word



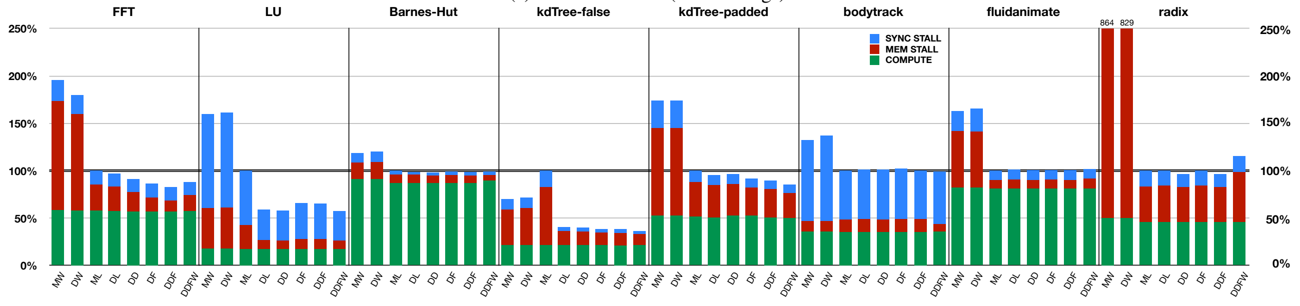
(a) Memory stall time.



(b) Read miss counts.



(c) Network traffic (flit-crossings).



(d) Execution time.

Fig. 2: Comparison of MESI vs. DeNovo protocols. All bars are normalized to the corresponding ML protocol.

address granularity does much better for LU and Bodytrack. Overall, DF shows up to 79% reduction in memory stall time over ML and up to 44% over DL. These results are pessimistic since we did not transfer more than 64 bytes of data at a time.

Effectiveness of combined optimizations with DL: DDF combines the benefits of both DD and DF to show either about the same or better performance than all the other line based protocols (except for LU for reasons described above).

Effectiveness of combined optimizations with DW: For applications like LU and bodytrack with low spatial locality, word-based protocols have the advantage over line based protocols by not bringing in potentially useless data and/or not replacing potentially useless data. We find that DW with our two optimizations (DDFW) does indeed perform better than DDF for these two applications. In fact, DDFW does better for 5 out of the 8 applications. This motivates our future work on using a more software-aware (region based) address granularity to get the best benefit of our optimizations.

Effectiveness of regions and touched bits: To evaluate the effectiveness of regions and touched bits, we ran DL without them. This resulted in all the valid words in the cache being invalidated by the self-invalidation instruction. Our results (not shown in detail) show 0% to 25% degradation for different applications, which indicates that these techniques are beneficial for some applications.

Protocol verification results: Through model checking, we found three bugs in DeNovo and six bugs including two deadlock scenarios in MESI. Note that DeNovo is much less mature than the GEMS MESI protocol which has been used by many researchers. In DeNovo, all bugs were simple to fix and showed mistakes in translating our internal high level specification into the implementation (i.e., their solutions were already present in our internal high level description of the protocol). In MESI, all the bugs except one of the deadlocks are caused by protocol races between L1 writebacks and other cache events. These involved subtle races and took several days to track, debug and fix. After fixing all the bugs, the model for MESI explores 1,257,500 states in 173 seconds whereas the model for DeNovo explores 85,012 states in 8.66 seconds. Our experience clearly indicates the simplicity and reduced verification overhead for DeNovo compared to MESI.

VI. Related Work

There is a vast body of work on improving the shared-memory hierarchy, including coherence protocol optimizations (e.g., [51, 55, 54, 63, 66]), relaxed consistency models [30, 32], using coarse-grained (multiple *contiguous* cache lines, also referred to as regions) cache state tracking (e.g., [23, 59, 71]), smart spatial and temporal prefetching (e.g., [64, 68]), bulk transfers (e.g., [11, 26, 38, 39], producer-initiated communication [2, 48]), recent work specifically for multicore hierarchies (e.g., [12, 37, 72]), and many more. Our work is inspired by much of this literature, but our focus is on a holistic rethinking of the cache hierarchy driven by disciplined software programming models to benefit hardware complexity, performance, and power. Below we elaborate on work that is the most closely related.

The recent SARC coherence protocol [45] exploits the data-race-free programming model [6], but is based on the conventional directory-based MESI protocol. SARC introduces "tear-off, read-only" (TRO) copies of cache lines for self-invalidation and also uses direct cache-to-cache communication with writer prediction to improve power and performance. Their results, like ours, prove the usefulness of disciplined software for hardware. Unlike DeNovo, SARC does not reduce the directory storage overhead (the sharer list) or reduce protocol complexity. Also, in SARC, all the TRO copies are invalidated at synchronization points while in DeNovo, as shown in Section V, region information and touched bits provide an effective means for selective self-invalidation. Finally, SARC does not explore flexible communication granularity since it does not have the concept of regions and also it is susceptible to false sharing.

Other efforts target one or more of the cache coherence design goals at the expense of other goals. For example, the work in [51] uses self-invalidations but introduces a much more complex protocol. The work in [47] does not incur complexity but requires traffic-heavy flushing of all dirty lines to the global shared cache at the end of each phase with some assumptions about the programming model. Another compiler-hardware coherence approach [58] does not support remote cache hits, instead they require writes to a shared-level cache if there is a potential inter-phase dependency. The SWEL protocol [62] and Atomic Coherence [67] work to simplify the protocol at the expense of relying on limited interconnect substrates. SWEL dynamically places read-write shared data in the lowest common level of shared cache and uses a bus for invalidation. Atomic Coherence uses nanophotonics to guard each coherence action with a mutex. Both protocols eliminate transient states, but limit the network.

Philosophically, the software distributed shared memory literature is also similar, where the system exploits data-race-freedom to allow large granularity communication (virtual pages) without false sharing (e.g., [5, 15, 24, 19]). These techniques mostly rely on heavyweight mechanisms like virtual memory management, and have struggled to find an appropriate high-level programming model. Recent work [31] reduces performance overheads through hardware support.

Some work has also abandoned cache coherence altogether [41] at the cost of significant programming complexity.

VII. Conclusions and Future Work

This paper takes the stance that disciplined programming models will be essential for software programmability and clearly specifiable hardware/software semantics, and asks how such models impact hardware. The paper shows that race-freedom, structured parallel control, and the knowledge of regions and effects in deterministic codes enable much simpler, more extensible, and more efficient cache coherence protocols than the state-of-the-art. This paper is the first step in exploiting what appears to be a tremendous opportunity to rethink multicore memory hierarchies driven by disciplined software models. There are several avenues of future work: extending the ideas here to the main memory system; extending to handle other forms of disciplined and non-disciplined codes (e.g., disciplined non-deterministic codes, synchronization, and legacy codes); using regions to drive address (cache allocation) and coherence granularity; more realistic implementations of the optimizations explored here; and automating the generation of hardware regions and communication spaces through a compiler/runtime implementation.

References

[1] OpenSPARC^{TM} T2 system-on-chip (soc) microarchitecture specification, May 2008.

[2] H. Abdel-Shafi et al. An Evaluation of Fine-Grain Producer-Initiated Communication in Cache-Coherent Multiprocessors. In *HPCA*, 1997.

[3] D. Abts et al. So Many States, So Little Time: Verifying Memory Coherence in the Cray X1. In *IPDPS*, 2003.

[4] S. V. Adve and H.-J. Boehm. Memory Models: A Case for Rethinking Parallel Languages and Hardware. *CACM*, Aug. 2010.

[5] S. V. Adve et al. A Comparison of Entry Consistency and Lazy Release Consistency. In *HPCA*, pages 26–37, February 1996.

[6] S. V. Adve and M. D. Hill. Weak Ordering - A New Definition. In *Proc. 17th Intl. Symp. on Computer Architecture*, pages 2–14, May 1990.

[7] V. S. Adve and L. Ceze. *Workshop on Deterministic Multiprocessing and Parallel Programming, U-Washington*, 2009.

[8] N. Agarwal et al. Garnet: A detailed interconnection network model inside a full-system simulation framework. Technical Report CE-P08-001, Princeton University, 2008.

[9] M. D. Allen, S. Sridharan, and G. S. Sohi. Serialization Sets: A Dynamic Dependence-based Parallel Execution Model. In *PPoPP*, pages 85–96, 2009.

[10] Z. Anderson et al. SharC: Checking Data Sharing Strategies for Multithreaded C. In *PLDI*, pages 149–158, 2008.

[11] R. H. Arpaci et al. Empirical Evaluation of the CRAY-T3D: A Compiler Perspective. In *ISCA*, pages 320–331, June 1995.

[12] A. Basu et al. Scavenger: A New Last Level Cache Architecture with Global Block Priority. In *MICRO*, 2007.

[13] A. Baumann et al. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *SOSP*, 2009.

[14] E. D. Berger et al. Grace: Safe Multithreaded Programming for C/C++. In *OOPSLA*, pages 81–96, 2009.
[15] B. N. Bershad and M. J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report TR CMU-CS-91-170, CMU, 1991.
[16] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, Jan. 2011.
[17] C. Bienia et al. Fidelity and scaling of the parsec benchmark inputs. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, 2010.
[18] R. D. Blumofe et al. Cilk: An Efficient Multithreaded Runtime System. In *PPoPP*, pages 207–216, 1995.
[19] M. A. Blumrich et al. Virtual memory mapped network interface for the shrimp multicomputer. In *ISCA*, pages 142–153, 1994.
[20] R. Bocchino et al. Safe Nondeterminism in a Deterministic-by-Default Parallel Language. In *POPL*, 2011. To appear.
[21] R. L. Bocchino, Jr. et al. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA*, pages 97–116, 2009.
[22] Z. Budimlic et al. Multi-core Implementations of the Concurrent Collections Programming Model. In *IWCPC*, 2009.
[23] J. Cantin et al. Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking. In *ISCA*, pages 246–257, June 2005.
[24] M. Castro et al. Efficient and flexible object sharing. Technical report, IST - INESC, Portugal, July 1995.
[25] K. Chakraborty et al. Computation Spreading: Employing hardware migration to specialize CMP cores on-the-fly. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 283–292, New York, NY, USA, 2006. ACM.
[26] R. Chandra et al. Performance Evaluation of Hybrid Hardware and Software Distributed Shared Memory Protocols. In *ICS*, 1994.
[27] B. Choi et al. Parallel SAH k-D Tree Construction. In *High Performance Graphics (HPG)*, 2010.
[28] S. Curial et al. Mpads: memory-pooling-assisted data splitting. In *ISMM*, pages 101–110, 2008.
[29] D. L. Dill et al. Protocol Verification as a Hardware Design Aid. In *ICCD '92*, pages 522–525, Washington, DC, USA, 1992. IEEE Computer Society.
[30] M. Dubois et al. Delayed Consistency and its Effects on the Miss Rate of Parallel Programs. In *SC*, pages 197–206, 1991.
[31] C. Fensch and M. Cintra. An OS-based alternative to full hardware coherence on tiled CMPs. In *HPCA*, 2008.
[32] K. Gharachorloo et al. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *ISCA*, pages 15–26, May 1990.
[33] A. Ghuloum et al. Ct: A Flexible Parallel Programming Model for Tera-Scale Architectures. Intel White Paper, 2007.
[34] S. Gjessing et al. Formal specification and verification of sci cache coherence: The top layers. October 1989.
[35] N. Gustafsson. Axum: Language Overview. Microsoft Language Specification, 2009.
[36] D. Hackenberg et al. Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems. In *MICRO*, pages 413–422. IEEE, 2009.
[37] N. Hardavellas et al. Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches. In *ISCA*, pages 184–195, 2009.
[38] K. Hayashi et al. AP1000+: Architectural Support of PUT/GET Interface for Parallelizing Compiler. In *ASPLOS*, pages 196–207, 1994.
[39] J. Heinlein et al. Coherent Block Data Transfer in the FLASH Multiprocessor. In *ISPP*, pages 18–27, 1997.
[40] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4th edition, 2007.
[41] J. Howard et al. A 48-core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In *ISSCC*, pages 108–109, 2010.
[42] G. C. Hunt and J. R. Larus. Singularity: Rethinking the Software Stack. In *ACM SIGOPS Operating Systems Review*, 2007.
[43] Intel. The SCC Platform Overview. http://techresearch.intel.com/spaw2/uploads/files/SCC_Platform_Overview.pdf.
[44] T. E. Jeremiassen and S. J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *PPOPP*, pages 179–188, 1995.
[45] S. Kaxiras and G. Keramidas. SARC Coherence: Scaling Directory Cache Coherence in Performance and Power. *IEEE Micro*, 30(5):54 –65, Sept.-Oct. 2010.
[46] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *ISCA*, pages 13–21, 1992.
[47] J. H. Kelm et al. Rigel: An Architecture and Scalable Programming Interface for a 1000-core Accelerator. In *ISCA*, 2009.
[48] D. A. Koufaty et al. Data Forwarding in Scalable Shared-Memory Multiprocessors. In *SC*, pages 255–264, 1995.
[49] M. Kulkarni et al. Optimistic Parallelism Requires Abstractions. In *PLDI*, pages 211–222, 2007.
[50] A. Kumar et al. Efficient and scalable cache coherence schemes for shared memory hypercube multiprocessors. In *SC*, New York, NY, USA, 1994. ACM.
[51] A. R. Lebeck and D. A. Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. In *ISCA*, pages 48–59, Jun 1995.
[52] E. A. Lee. The Problem with Threads. *IEEE Computer*, 39(5):33–42, May 2006.
[53] B. Lucia et al. Conflict Exceptions: Simplifying Concurrent Language Semantics with Precise Hardware Exceptions for Data-Races. In *ISCA*, 2010.
[54] M. M. Martin et al. Token coherence: Decoupling performance and correctness. In *ISCA*, 2003.
[55] M. M. Martin et al. Using Destination-Set Prediction to Improve the Latency/Bandwidth Tradeoff in Shared-Memory Multiprocessors. In *ISCA*, 2003.
[56] M. M. K. Martin et al. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Computer Architecture News*, 33(4):92–99, 2005.
[57] M. R. Marty et al. Improving Multiple-CMP Systems Using Token Coherence. In *HPCA*, pages 328–339, 2005.
[58] S. L. Min and J.-L. Baer. Design and analysis of a scalable cache coherence scheme based on clocks and timestamps. *IEEE Trans. on Parallel and Distributed Systems*, 3(2):25–44, January 1992.
[59] A. Moshovos. RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence. In *ISCA*, 2005.
[60] A. Nanda and L. Bhuyan. A formal specification and verification technique for cache coherence protocols. In *ICPP*, pages I22–I26, 1992.
[61] M. Olszewski et al. Kendo: Efficient Deterministic Multithreading in Software. In *ASPLOS*, pages 97–108, 2009.
[62] S. H. Pugsley et al. SWEL: Hardware Cache Coherence Protocols to Map Shared Data onto Shared Caches. In *PACT*, 2010.
[63] A. Raghavan et al. Token Tenure: PATCHing Token Counting using Directory-Based Cache Coherence. In *MICRO*, 2008.
[64] S. Somogyi et al. Spatial Memory Streaming. In *ISCA*, pages 252–263, 2006.
[65] D. J. Sorin et al. Specifying and verifying a broadcast and a multicast snooping cache coherence protocol. *IEEE Trans. Parallel Distrib. Syst.*, 13(6):556–578, 2002.
[66] K. Strauss et al. Flexible Snooping: Adaptive Forwarding and Filtering of Snoops in Embedded-Ring Multiprocessors. In *ISCA*, pages 327–338, 2006.
[67] D. Vantrease et al. Atomic Coherence: Leveraging Nanophotonics to Build Race-Free Cache Coherence Protocols. In *HPCA*, 2011.
[68] T. Wenisch et al. Temporal Streaming of Shared Memory. In *ISCA*, pages 222–233, 2005.
[69] S. C. Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA*, 1995.
[70] D. A. Wood et al. Verifying a multiprocessor cache controller using random case generation. *IEEE DToC*, 7(4), 1990.
[71] J. Zebchuk et al. A Framework for Coarse-Grain Optimizations in the On-Chip Memory Hierarchy. In *MICRO*, pages 314–327, 2007.
[72] J. Zebchuk et al. A Tagless Coherence Directory. In *MICRO*, 2009.

DeNovoND: Efficient Hardware Support for Disciplined Non-Determinism

Hyojin Sung, Rakesh Komuravelli, and Sarita V. Adve

Department of Computer Science
University of Illinois at Urbana-Champaign
{sung12, komurav1, sadve}@illinois.edu

Abstract

Recent work has shown that disciplined shared-memory programming models that provide deterministic-by-default semantics can simplify both parallel software and hardware. Specifically, the DeNovo hardware system has shown that the software guarantees of such models (e.g., data-race-freedom and explicit side-effects) can enable simpler, higher performance, and more energy-efficient hardware than the current state-of-the-art *for deterministic programs.* Many applications, however, contain non-deterministic parts; e.g., using lock synchronization. For commercial hardware to exploit the benefits of DeNovo, it is therefore necessary to extend DeNovo to support non-deterministic applications.

This paper proposes DeNovoND, a system that supports lock-based, disciplined non-determinism, with the simplicity, performance, and energy benefits of DeNovo. We use a combination of distributed queue-based locks and access signatures to implement simple memory consistency semantics for safe non-determinism, with a coherence protocol that does not require transient states, invalidation traffic, or directories, and does not incur false sharing. The resulting system is simpler, shows comparable or better execution time, and has 33% less network traffic on average (translating directly into energy savings) relative to a state-of-the-art invalidation-based protocol for 8 applications designed for lock synchronization.

Categories and Subject Descriptors B.3.2 [*Hardware*]: Memory Structures – Cache memories; Shared memory; C.1.2 [*Processor Architectures*]: Multiple Data Stream Architectures (Multiprocessors) – Parallel processors

Keywords shared memory, cache coherence, disciplined parallelism, memory consistency, non-determinism

1. Introduction

Shared-memory remains a popular programming model among multicore programmers and is the de facto model provided by multicore hardware. It is, however, increasingly evident that unbridled "wild" shared-memory programming environments that allow data races, ubiquitous non-determinism, unstructured parallelism, and complex memory consistency models make programming, debugging, testing, and maintaining software difficult [1, 32]. Recent software research has therefore proposed more *disciplined* shared-memory programming models that retain the advantage of a global address space, but make it easier to write safe parallel programs that are easier to debug, test, and maintain [4–6, 10–12, 14, 18, 19, 30, 39].

At the same time, providing hardware cache coherence and consistency that can scale in a power-efficient manner to hundreds of cores is also a significant challenge. There has recently been a surge in research by academics (see Section 7) and hardware companies [23, 26] to address this challenge in unconventional ways. In particular, the DeNovo hardware project observes that disciplined shared-memory programming models such as mentioned above can drive a holistic rethinking of the multicore memory hierarchy, providing more complexity-, performance-, and power-efficient hardware than the state-of-the-art *for deterministic programs* [17]. This paper shows that the benefits of disciplined programming and DeNovo can be extended to non-deterministic programs as well.

DeNovo has used Deterministic Parallel Java (DPJ) as an example disciplined programming model [11] to drive its design. DPJ provides the programmer with a novel region-based type and effects system to convey the read and write side-effects on shared-memory for every method. A type-checked DPJ program is guaranteed deterministic-by-default semantics. That is, unless non-determinism is explicitly requested, DPJ programs appear deterministic and with sequential semantics (the programmer can debug and test such a program as if it were sequential). Even when non-determinism is explicitly requested, DPJ provides strong safety guarantees; e.g., data-race-freedom, strong isolation, and sequential composition of deterministic code sections [12]. The DPJ compiler enforces these guarantees by checking that conflicting accesses from two concurrent tasks – the root cause of non-determinism – are always identified (as atomic) and always occur within explicitly marked atomic sections.

DeNovo has so far focused on deterministic programs, and shown that DPJ's information and guarantees can be exploited to provide a simpler and more efficient cache coherence protocol than the state-of-the-art MESI for such programs [17]. Specifically, DeNovo's protocol has the following advantages: (1) The implementation has no transient states and so is much easier to verify (verification is an order of magnitude faster) and much easier to extend (incorporating optimizations did not introduce any protocol state changes). (2) DeNovo does not rely on writer-induced invalidations; it therefore eliminates invalidation message traffic and does not require storage overhead for sharer lists in directories, removing a key source of unscalability. (3) DeNovo keeps coherence state at the granularity at which data is shared and so does not suffer from false sharing (the added state overhead is much less than the reduced directory state). Overall, compared to MESI, DeNovo is much simpler,

ASPLOS '13, March 16–20, 2013, Houston, Texas, USA.
Copyright © 2013 ACM 978-1-4503-1870-9/13/03. . . $15.00

performs comparably or better than MESI, and is more energy-efficient (since it reduces cache misses and network traffic) for a range of deterministic codes.

Although determinism is considered desirable for many application classes, there are many common codes that are non-deterministic or contain parts that are non-deterministic, most commonly through lock synchronization. For example, 21 out of 25 of the PARSEC and SPLASH-2 benchmarks contain locks in some parts. DeNovo currently cannot run such codes.[1] For commercial hardware to be able to exploit the benefits of DeNovo, it is imperative that we develop techniques to support non-deterministic codes with at least as much performance as conventional systems, without losing the benefits of DeNovo.

This paper explores exploiting disciplined programming models to develop simpler and more efficient hardware even for programs that contain non-determinism. We use DPJ's safe non-determinism model (with atomic sections replaced with locks), and show that simple additions to the DeNovo coherence protocol can support such non-determinism without giving up on DeNovo's previous advantages. We call the resulting system DeNovoND.

For deterministic programs, DeNovo achieves its benefits primarily by recognizing that DPJ explicitly provides the regions that could be potentially written in a parallel phase (e.g., DPJ's foreach or cobegin constructs) through its explicit effects. At the start of a new phase, DeNovo's cores execute compiler-inserted self-invalidations to all regions that could have write effects in the previous phase. Their caches therefore now have only valid data. If any of this data is updated in the next phase, DPJ's data-race-freedom guarantee ensures that only the writing core will read that data, ensuring up-to-date values for all reads. These observations eliminate the need for writer-induced invalidations, directories, and false sharing due to cache line driven protocols.

Unlike DeNovo, DeNovoND cannot assume that a parallel phase will have no conflicting accesses among concurrent tasks any more, but it knows that such accesses will be protected by the same lock (this lock may change in a different parallel phase). Further, such accesses are explicitly identified as atomic accesses in DPJ programs. Within a critical section, DeNovoND therefore tracks atomic writes through a signature which is conveyed to the next acquirer of the lock. The acquirer uses the signature to determine which data to invalidate in its cache. The strong guarantees given by DPJ enable an efficient implementation, while still providing freedom to express a variety of non-deterministic algorithms. Underlying the above is an implementation for a lock that does not require directories and a full-fledged MESI protocol – we use a distributed queue based implementation modeled after the Queue-on-Sync-Bit (QOSB) lock [20].

Overall, our system retains the advantages of DeNovo while significantly expanding the class of programs it supports without compromising performance. Specifically, for lock accesses, although DeNovo's coherence protocol state machine is extended to handle the distributed queue, it reuses the state bits from DeNovo's data accesses. For data accesses, again, no new externally visible states are added; the only support needed is a signature per core, the ability to transfer it to the next acquirer, and to use it for self-invalidation at subsequent reads. A bit per word at the L1 cache is used as an optimization. We continue to not have any directories, not have invalidations, and not incur false sharing.

We compared DeNovoND with a state-of-the-art MESI protocol for 11 benchmarks with lock synchronization. 3 of these spent more than 70% of their time in lock acquires, clearly requiring alternate synchronization techniques for reasonable parallel efficiency that are out of the scope of this work. We therefore focus on the remaining 8 benchmarks here, although we report results for the above 3 as well for completeness. We found that DeNovoND performs comparably or slightly better than MESI in terms of execution time. DeNovoND also shows 33% lower network traffic than MESI on average, which directly translates into energy savings. Performance optimizations previously proposed for DeNovo (for cache to cache and flexible granularity data transfer) [17] are applicable to DeNovoND as well without any additional changes, but are orthogonal to this work and not reported here. Thus, DeNovoND allows us to extend the benefits of DeNovo to include lock-based (safe) non-deterministic applications.

Our system shares commonalities with previous software distributed shared memory consistency models such as lazy release consistency [27], entry consistency [7], and scope consistency [22] as well as recent hardware shared-memory work that exploits data-race-freedom such as SARC [25]. However, none of those systems distinguish between deterministic and non-deterministic accesses in a way that is possible with our hardware/software co-designed approach, and so those systems cannot exploit the corresponding optimizations. Section 7 discusses the relationship of our work to prior work in more detail.

While DeNovoND takes a major step in exploiting software discipline in hardware for a larger class of programs, there is still much left to future work and outside the scope of one paper. Section 8 discusses future work to explore how to incorporate other key constructs (e.g., pipelined parallelism), and support more complex codes such as legacy codes and operating systems within this vision.

2. Background

2.1 Deterministic Parallel Java (DPJ)

DPJ is an extension to Java that enforces deterministic-by-default semantics via compile-time type checking [11, 12]. We first discuss DPJ without non-deterministic constructs [11]. DPJ provides parallel constructs of foreach and cobegin to express parallelism in a structured way as in many current languages (we refer to an iteration of a foreach loop or a parallel statement of a cobegin as a task). DPJ provides a new type and effect system for expressing common patterns of imperative, object-oriented programs. The DPJ programmer assigns every object field or array element to a named "region" and annotates every method with read and write "effects" summarizing the regions read and written by that method (a region can be non-contiguous in memory). The compiler uses this information to (i) type-check program operations in the region type system and (ii) ensure that no two parallel tasks interfere (conflict).

DPJ also provides parallel constructs that are potentially non-deterministic; i.e., foreach_nd and cobegin_nd [12]. These constructs allow conflicting accesses between their tasks, but require that such accesses be enclosed within atomic sections, their read and write effect declarations also include the atomic keyword, and their region types be declared as atomic. Note that there continue to be no conflicts allowed between a task from a deterministic parallel construct and any other concurrent (non-deterministic or deterministic) task. The compiler checks that all of the above constraints are satisfied by any type-checked program, again using a simple, modular type checking algorithm.

With the above constraints, DPJ is able to provide the following guarantees: (1) Data-race freedom. (2) Strong isolation of accesses in atomic section constructs and all deterministic parallel constructs; i.e., these constructs appear to execute atomically. (3) Sequential composition for deterministic constructs; i.e., tasks of a deterministic construct appear to occur in the sequential order

[1] The DeNovo work reports results for some of these benchmarks, but the parts with locks were either run sequentially or rewritten or not simulated [17].

implied by the program (even if they contain or are contained within non-deterministic constructs). (4) Determinism-by-default; i.e., any parallel construct that does not contain an explicit non-deterministic construct provides deterministic heap output for a given heap input. The above guarantees are strong – they not only ensure sequential consistency but also allow programmers to reason with very high-level strongly isolated and composable components such as complete foreach constructs and all atomic sections.

Although DPJ supports atomic sections, this paper assumes we can convert them to locks. This is possible because by default we can associate each atomic region with its own lock. For each atomic section, we can acquire locks for each atomic region that it accesses in a predefined order. This can be optimized in several ways; e.g., by coarsening the locks. An implementation of this algorithm is outside the scope of this paper. We therefore use hand inserted locks – for the applications we used, these locks were as provided in the original application.

2.2 DeNovo for Deterministic Codes

DeNovo divides the coherence problem into two parts:

(1) *No stale data:* A read should never see stale data in its private cache(s).

(2) *Locatable up-to-date data:* When a read misses in its private cache(s), it should know where to get an up-to-date copy of the data.

Above, *stale* and *up-to-date* are defined by the memory consistency model (sequential semantics, in our case). For (1), DeNovo recognizes that DPJ explicitly provides the regions that could be potentially written in a parallel phase (each DPJ parallel construct such as cobegin and foreach forms a phase, with an implicit barrier at the join). Before starting a new phase, a core issues compiler-inserted self-invalidations for all regions that could have write effects in the previous phase, eliminating all stale data from its private cache(s).[2] For data updated in the current phase, DPJ's data-race-freedom guarantee ensures that only the writing core will read that data, ensuring up-to-date values for all (private) cache hits. For (2), DeNovo uses a structure called the *registry* to keep track of one up-to-date copy of each cache line. This is analogous to a conventional directory, but unlike the latter, it does not track all sharers of a cache line (eliminating a source of unscalability). With systems with a shared last level cache, the data bank of the cache doubles as the registry storing the data or a pointer to it.

The DeNovo protocol has three states, *Registered*, *Valid*, and *Invalid*. These states are analogous to those in a conventional MSI directory protocol; *Registered* is similar to *M* with the line modified in a private cache and *Valid* is similar to *S*. The DeNovo protocol state transition diagram also resembles typical textbook pictures for MSI. A key difference, however, is that real implementations of MSI have tens of transient states to handle protocol races, introducing significant complexity and making verification difficult. In contrast, DeNovo has no transient states since it assumes race-free software, which eliminates virtually all races from the protocol hardware.

Next we describe the key aspects of the protocol's operation and refer to [17] for more details. For easier exposition, we assume a two level cache hierarchy with a shared L2 without loss of generality, and a line size of one word (this is relaxed below). A read hits in the L1 if the line is *Valid* or *Registered*. A read miss request goes to the registry (the shared L2) and either finds the data there or a pointer to the L1 that contains the data in *Registered* state. In the latter case, the request is routed to the registered data for service.

[2] This requires the cache to store region information as described in [17].

A write to data in *Registered* state at the L1 updates the data. A write to data in *Valid* or *Invalid* state at the L1 immediately transitions the data to *Registered* and updates it (no transients) and generates a registration request (and a writeback if needed). If the data is not registered elsewhere, the L2 immediately registers it and sends an acknowledgment. Otherwise, the L2 records the new registration and forwards the request to the previously registered core to relinquish its registration. Due to the data-race-free guarantee, registration transfer occurs only once in a phase (assuming no task migration, which can also be easily handled [17]), without any danger of protocol races.

Additionally, as an optimization, L1 contains *touched* bits that are set when the corresponding data is read. Due to data-race-freedom, it is guaranteed that no other core will write such data in that phase. Thus, "touched" data is up-to-date and does not need to be invalidated for the next phase. All self-invalidations occur at the end of the phase – regions with write effects in that phase are invalidated unless the data is registered or touched. Touched bits are reset after the invalidation, in preparation for the next phase.

The baseline word-based DeNovo protocol assumes equal address/tag allocation, communication, and coherence granularity, which is the granularity at which data-race-freedom is ensured. This granularity is a word for the applications evaluated. (Details about supporting sub-word (byte) granularity can be found in [17].) DeNovo further observes that any data that is marked *touched* or *Registered* is always up-to-date and can be freely copied from one cache to another without informing anyone (there is no directory tracking sharer lists). Thus, the word-based DeNovo protocol is easily enhanced to operate on larger communication and address/tag allocation granularities, while still maintaining coherence state at the word granularity.

A natural granularity for communication and allocation is a conventional cache line (e.g., 64 bytes), and the corresponding DeNovo protocol is referred to as the line based protocol. Here, a responding cache for a demand request sends a cache line worth of data (potentially with some words marked as invalid) and the valid words in the response message are merged with the local copy of the cache line of the requestor. These words are marked as *Valid*, but not *touched* (the *touched* bit is set when those words are actually read). DeNovoND is designed on top of this line-based protocol.

DeNovo has also explored more flexible communication granularities (more or less than one cache line) and direct L1 to L1 data transfers. These optimizations are simple with DeNovo and do not require any new states, but are difficult to incorporate in conventional protocols because they introduce even more transient states. The same optimizations can be directly applied to DeNovoND as well, again with no new states for DeNovoND. We do not study them here since they are orthogonal to the goal of this paper.

The DeNovo protocol we study additionally implements the optimization of write combining where multiple registration requests to words in a given cache line are combined into one request. This optimization was mentioned, but not implemented, in [17] to reduce write traffic. This optimization is not meaningful for conventional protocols since conventional store requests always operate on a full line while DeNovo registrations are for a word.

3. DeNovoND Design Overview

3.1 Basic Assumptions and Definitions

We assume all synchronization occurs through DPJ's parallel constructs (foreach, cobegin, and their nd versions) and through locks. We assume a barrier at the implicit join associated with the parallel constructs. We say all concurrent tasks of a given parallel construct – loop iterations in a foreach and parallel statements in a cobegin – form a *phase*.

For locks, we assume that an atomic section does not call a parallel construct, as is the case with all our applications. Thus, all operations of an atomic section occur within a single task and are enclosed within a lock acquire and release to the same lock variable (there may be nested locks to different lock variables). We refer to memory operations within such a lock acquire/release pair as occurring in a critical section protected by that lock variable.

For data accesses, we assume the ISA provides a mechanism by which loads and stores can be tagged as accessing atomic regions with atomic effects (e.g., with a bit in the op-code). The DPJ compiler has this information and can generate code with the bit set for such accesses. We refer to such accesses below as *atomic* accesses and to others as *non-atomic* accesses. Note that the former are regular data accesses from atomic sections and are not to be confused with atomic read-modify-writes or the C++ atomic keyword used for synchronization races.

Without loss of generality, we assume a two level cache hierarchy. We also assume a shared L2 cache. DeNovoND can be extended to deeper hierarchies and private last level caches in a straightforward way (similar to DeNovo [17]).

3.2 Memory Consistency Model

For a correct design, we must first understand the constraints imposed by the memory consistency model which specifies what value a read must return.

Informal model: DPJ provides a very strong consistency model. It guarantees sequential consistency and hence a total order over all memory operations (that is consistent with program order). A read must return the value of the last write to its location as defined by this total order. DPJ also enforces additional rules that further constrain this last write for data operations, simplifying reasoning for software and implementation for hardware as follows.

Non-atomic accesses: DPJ ensures that for a non-atomic access, there cannot be a conflicting access by another concurrent task in the same phase. Thus, for a non-atomic read, the last conflicting write is either from its own task or from a task in a previous phase. This is identical to DeNovo and we can use the identical implementation.

Atomic accesses: For atomic accesses as defined above, DPJ allows conflicting accesses among concurrent tasks, but ensures that all such accesses to a given location are in critical sections protected with the same lock. These critical sections must execute atomically, imposing a total order on all conflicting atomic accesses within a phase. A read therefore must return the value from the (unique) last conflicting write from a critical section in the current phase; if such a write does not exist, then the read must return the (unique) last conflicting write from the previous phase.

Formal model: We now state the model more formally. Note that this model is motivated as a specification for hardware and is therefore at a low level, in terms of individual reads and writes. DPJ programmers work at a higher level in terms of composition and serialization of higher level constructs (cobegin, atomic section, etc.) as described in Section 2.1. Our model can be stated in two parts for synchronization and data accesses respectively:

(1) Synchronization accesses are sequentially consistent. This implies a total order between phases and between critical sections to a given lock variable within a phase; this total order is consistent with program order.

(2) For conflicting data accesses, X and Y, we define a *happens-before* relation, denoted $\rightarrow_{hb}$ such that $X \rightarrow_{hb} Y$ iff

Type 1 edge: X's phase precedes Y's phase (by the total order in (1)), or

Type 2 edge: X and Y are in the same task, and X is before Y by program order, or

Type 3 edge: X and Y are atomic accesses in critical sections protected by the same lock variable, and X's critical section precedes Y's critical section (by the total order in (1)).

Then DPJ's guarantees ensure that $\rightarrow_{hb}$ orders all conflicting accesses, and hardware should ensure that a data read returns the value of the last conflicting write in $\rightarrow_{hb}$ order. For a non-atomic read, the last write is always ordered before it by a *type 1* or *type 2* $\rightarrow_{hb}$ edge. For an atomic read, the last write may be ordered before it by a *type 2* or *type 3* edge if such a write exists; otherwise, it is ordered by a *type 1* edge.

3.3 Data Coherence Mechanism

The coherence mechanism must simply ensure that a read returns the value from the write as defined by the consistency model. As with DeNovo, we divide the coherence mechanism into two components:

(1) *No stale data:* A read should never see *non-last* (stale) data in its private cache(s).

(2) *Locatable up-to-date data:* When a read misses in its private cache(s), it should know where to get the *last* (up-to-date) copy of the data.

Above, *last* is precisely defined by the happens-before order. For non-atomic accesses, both components above remain identical to DeNovo since the consistency model requirements are identical. For atomic accesses, the requirements are met as follows.

No stale data: For the first requirement of no stale data, we use self-invalidations as with DeNovo, thereby precluding the need for adding invalidation messages and directories with sharer lists. Additional self-invalidations are needed with DeNovoND only if there are conflicting atomic accesses among concurrent tasks in a phase (otherwise, DeNovo's self-invalidations at the start of a phase suffice). In the case of conflicting atomic accesses among concurrent tasks, we use the happens-before relation to determine *when* and *what* to self-invalidate as follows.

To determine when to self-invalidate, we note that a concurrent conflicting read must be in a critical section itself and must return the value of the last write also in a critical section protected by the same lock in the same phase (type 2 or 3 edge). Thus, it is sufficient to self-invalidate any time between the start of a critical section and an atomic read in that section.

To determine what to self-invalidate, we have several choices. We could invalidate the entire cache (which seems excessive) or only the atomic regions (for which we would need to keep extra state to identify in the cache). An alternative is for each core to update a signature that records all writes to atomic regions, and then to transfer this signature when the lock is acquired by another core. On a first atomic read to a location, the acquiring core needs to check the signature and self-invalidate the location if it is present in the signature. The acquiring core must forward the union of its signature and the signatures it has received to the next acquirer.

Locatable up-to-date data: For the second requirement of finding the value of the last write on a miss, we use ideas similar to DeNovo. On a write to valid or invalid data, the L1 cache sends a registration request to the L2. The registrations are required to complete before the lock release so that conflicting writes from critical sections are serialized in the right order (it is possible to postpone the registration completion until the next lock acquire). A read that misses in the cache simply goes to the registry (L2) to find the up-to-date value.

Thus we continue with only three states in the protocol as before: *Valid*, *Invalid*, and *Registered*. The extra work over DeNovo is to update the signature on atomic writes, send the signature on a lock transfer, and invalidate appropriately on atomic reads. Section 4.1 discusses each of these steps in more detail.

3.4 Distributed Queue-based Locks

Our distributed queue-based lock design is modeled after QOSB [20, 24], where the identities of the cores waiting for a lock are maintained in a queue of pointers distributed across the waiting cores' L1 caches and the L2 cache. All requests to a given lock are serialized at the corresponding shared L2 cache bank. The data portion of the L2 cache entry for a contended lock tracks the last requestor (i.e., the tail of the queue of waiters), referred to as *tailPtr*. When the L2 receives the next request for the lock, it forwards it to the current tail's L1. On receiving such a forwarded request, the L1 checks a bit in its copy of the lock word, called the *Locked* bit, to determine if the lock is still held or was unlocked. In the former case, the L1 stores the requestor's ID in another field of the lock word, referred to as *nextPtr*. In the latter case, the L1 responds to the requestor with its signature and transfers the lock, marking its own lock word *Invalid*. When a core releases a lock, its L1 checks its *nextPtr* – if not null, it transfers the lock (with the signature) to the *nextPtr* core; otherwise, it unsets its *Locked* bit. We allow eviction of lock words from the L1 and L2 caches by reusing the data portion of the lock words in the next level of the memory hierarchy to store lock queue information. This approach relies on using L2 data banks to store (non-data) metadata, which is similar to DeNovo's tracking of registration information for the *Registered* state. Section 4.2 discusses our implementation in more detail.

4. Implementation

This section discusses in detail how DeNovoND implements the memory consistency model and the coherence mechanism described in Section 3 using *access signatures* and the distributed queue-based lock mechanism. We also qualitatively discuss the hardware and performance overheads of the implementation.

4.1 Access Signatures for Coherence of Atomic Accesses

DeNovoND's memory consistency model requires that a read return the value of the last write preceding it, as ordered by the three types of *happens-before* edges described in Section 3. DeNovo already guarantees that a write ordered by a type 1 or type 2 edge is seen at a read (the former through self-invalidations at the start of a new phase and the latter through single core semantics). For a non-atomic read, a write is ordered only through the above two edge types; therefore, DeNovo already provides consistency for such reads. For atomic reads where a previous (atomic) write is ordered by a type 3 edge, however, DeNovoND must provide a new mechanism – it needs to track which data in atomic regions has been modified in a critical section in the current phase, as well as a mechanism to efficiently represent and transfer this information on a successful lock acquire.

We use an "access signature" for the purpose of tracking atomic writes. A signature is a compact representation of a set at the expense of precision. Its main functionality includes element insertion, membership query, and flash clear functions. DeNovoND implements the access signature as a small Bloom filter in hardware [9]. Due to its storage efficiency, simplicity, and low access latency, a hardware Bloom filter has been a popular solution for many areas including networking and transactional memory [13, 16].

For our Bloom filters, the keys are addresses accessed (i.e., atomic regions that have atomic effects in this phase), since we are interested only in modifications made to those addresses. The key domain dynamically changes between cores and phases, as a new set of atomic accesses occurs. To keep the false positive rate of Bloom filter reasonably low, the size of each Bloom filter should be determined based on the average size of the key domain. This turns out to be quite small in our case (256 bits suffice) since we only track atomic accesses in a given phase (later sections discuss

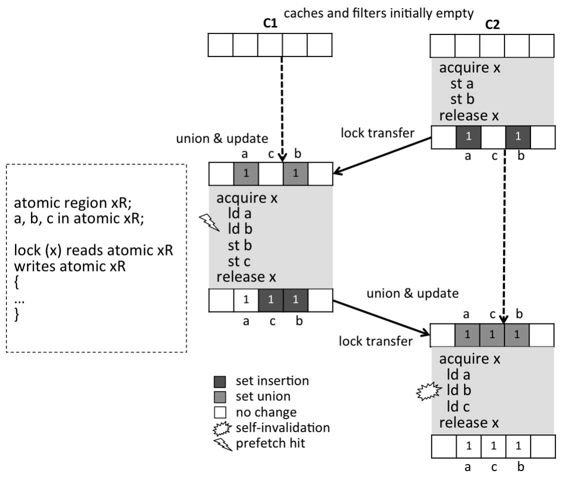


Figure 1: An example of propagating atomic writes using access signatures. Assume *a* and *b* are in the same cache line.

the size in more detail). We conservatively keep one filter per core to track all modifications across different critical sections (with different locks) on the same core. Thus, for a system with *n* cores, we have a total of *n* Bloom filters in the system.

The following uses Figure 1 as a running example to show how DeNovoND uses the Bloom filters. On the left, the figure shows DPJ style code depicting three variables, *a*, *b*, and *c* in atomic region *xR*. It then shows a critical section protected by lock *x* with atomic read and write effects on region *xR*. The right side of the figure shows an execution with two cores, *C1* and *C2*. *C2* acquires the lock for the critical section first, followed by *C1* and then *C2* again. The figure also shows the signatures at each core, assuming a perfect hash function.

On atomic writes: An atomic write (as determined by the op-code of the store instruction as discussed in Section 3) invokes the same cache protocol operations as in DeNovo. That is, if the word is not in *Registered* state at the L1, a registration request is sent to the L2. Additionally, the word is updated right away and any required writeback is sent to the L2 as well.

For DeNovoND, an atomic write additionally inserts the accessed address into its core's Bloom filter. To avoid repeating insertion of the same address to the Bloom filter, we can add an additional bit, called the "dirty bit," to mark a memory location already updated in a given phase. The "dirty bit" is set on the first atomic store request to a word in a phase, and all dirty bits get unset at the end of a phase. If a store finds the dirty bit already set, it means the word is already inserted into the core's Bloom filter and does not need to be inserted again. Since this is purely an optimization, we can piggyback the functionality of a dirty bit on other state bits described below (e.g., the *touched-atomic* bit) – this may result in some extraneous resets, but does not affect correctness and reduces extra state.

Thus, at the end of a critical section, all addresses modified in the section are recorded in the core's filter; i.e., their entries are nonzero. From Figure 1, every store request to *a*, *b*, and *c* in the lightly shaded critical sections updates the Bloom filter on *C1* and *C2*. The second critical section phase on *C2* does not update the Bloom filter since it does not have atomic writes.

On acquire/release: On an acquire, all modifications preceding the release associated with the acquire are made visible to the acquirer by transferring the access signature at the releaser. The releaser compresses and sends the Bloom filter at its core to the acquirer, when transferring the lock. The acquirer, on receiving the Bloom filter, updates its own Bloom filter by making a *union* of its local Bloom filter and the releaser's Bloom filter. Figure 1 shows the resulting Bloom filters at the beginning of each critical section, of which the lightly shaded entries come from the union operation. Note that we only send the signature, not the actual data. On acquire and release points, we also reset the "touched-atomic" and "prefetch" bits (as will be explained in detail below).

On atomic reads: Atomic reads need to conceptually consult the signatures obtained from remote releasers to determine if cached data is valid or stale. If the read is to a word in *Registered* state in the L1, then regardless of the signature state, the word is up-to-date in the cache and the read is a cache hit. If the word is *Invalid* in L1, then a normal read request is sent to L2. If the word is in *Valid* state, then it is also up-to-date if its address does not appear in the access signature. If the word is in *Valid* state and its address hits in the access signature, then it may or may not be up-to-date depending on whether it has been previously read in this critical section.

Specifically, if the word has already been read in this critical section, the previous read brought up-to-date data that is still valid (since no other core can write to the word during the same critical section). We identify this situation by using a *touched-atomic* bit that is set on the first read of the word in a critical section and reset at the release – more precisely, it needs to be reset only when the lock is handed off for another core's acquire (lock hand-off). Thus, a read to a word in *Valid* state with *touched-atomic* bit set is a cache hit.

Another case where a valid word may be up-to-date is when it is obtained as part of a cache line transfer for a demand access to another word in that line. We would like to take advantage of such a prefetch as with conventional cache lines and with DeNovo. If the word comes directly from the L2 or from memory, then it is definitely valid. If it comes from a remote cache, then it is valid if that word was marked as *touched-atomic* or *Registered* in the remote cache. In this case, we can conceptually add another bit called the "prefetch bit" which can be set for prefetched words with the above properties. These bits must be reset on the next lock hand-off or the next acquire, whichever happens first. A read that accesses a valid word with *prefetch* bit set is considered a cache hit. Although the *touched-atomic* and *prefetch* bits are separately motivated, both functions can be achieved by a single bit that we collectively refer to as the *touched-atomic* bit.

In summary, the *touched-atomic* bit of a word is set on the first read of the word in a critical section or for a word prefetched from L2/memory or from a remote L1 in *touched-atomic* or *Registered* state. The bit is reset on an acquire or lock hand-off, including the end of the phase. A read to *Valid* data with *touched-atomic* bit set or with an address that misses in the access signature is considered a hit. Otherwise, the *Valid* data is no longer up-to-date and must be marked invalid and a read miss request is issued.

In Figure 1, assume that variables *a* and *b* are in the same cache line. Then *C1*'s `load b` will be a hit since *C1*'s `load a` will bring in *b* as well and set its *touched-atomic* bit. On the other hand, `load b` in *C2*'s second critical section is a miss. This is because the preceding `load a` will read *a* in its own cache in *Registered* state and so will not prefetch *b* which is registered at *C1*.

Finally, we note that using a single, plain Bloom filter at each core to determine what to invalidate is inherently conservative. For example, it is possible that an address may have been updated before it had been last seen by a core but not updated again since then; our system will still invalidate the address on a read (in the same phase) from that core. In addition, false positives in a finite Bloom filter cause valid addresses to be invalidated if the filter entry is updated by another address mapped to the same entry. Another source of imprecision occurs when the signature is transferred well after the lock release occurs. Such a signature may include addresses to accesses after the release and before the subsequent acquire – these do not precede the acquire by happens-before and may lead to false positives and unnecessary invalidations. Our evaluation, however, showed that such cases did not occur often for applications with reasonable lock synchronization; nevertheless, we later discuss some approaches to mitigate such effects (Section 6).

End of phase actions: At the end of a phase, as with DeNovo, we insert self-invalidation instructions for all regions with writable effects in that phase. This includes atomic and non-atomic regions. Analogous to DeNovo, all data in such regions is invalidated unless it is registered or its *touched* bit (for non-atomic regions) is set or its *touched-atomic* bit (for atomic regions) is set. All *touched* and *touched-atomic* bits are reset at the end of the phase and all Bloom filters are cleared.

4.2 Lock Implementation

Tables 1a and 1b describe the state transitions for the L1 and L2 caches respectively for lock words, building on top of the DeNovo line protocol (as with DeNovo, the coherence states are at word granularity). We next discuss these in detail.

L1 transitions: There are two states at L1 for a lock word: *LockQ* and *Invalid*. The lock word transitions to *LockQ* on receiving a lock request from its core, and stays there until it transfers the lock (along with the access signature) to *nextPtr* or until the line is evicted. While in *LockQ* state, a bit in the data portion of the lock entry, called *Locked*, indicates whether the lock is held or released. Figure 2 shows the lock word layout at the L1 with a lock queue.

On a lock request by a core, its L1 sets the *Locked* bit for the corresponding word. If the word was already in *LockQ* state, the L1 informs the core of a successful lock acquire. If the previous state was *Invalid*, a lock request is sent to the L2 and the core is stalled (the cache does not service any further requests from the core) until the response is received.

On an unlock request to *LockQ* state, if *nextPtr* is not null, the L1 transfers the lock to the *nextPtr* core and transitions to *Invalid*. Otherwise, it unsets *Locked*. An unlock request to *Invalid* state generates a request to the L2. This request is simply a notification and does not bring back the cache line (the state stays *Invalid*).

An L1 in *LockQ* state may receive a remote lock request forwarded by the L2. If the *Locked* bit is set, the request is queued in *nextPtr*; otherwise, it is serviced immediately by transferring the lock and changing the state to *Invalid*. The L1 may also receive a remote lock request in *Invalid* state due to a previous writeback. If this request is only for the signature, it transfers the signature (along with an implicit lock transfer) to the remote requestor. If the request is for the lock as well, then it signifies a race between the L1's writeback and the remote request at the L2. In this case, L1 returns a Nack to the L2 – we discuss how the L2 responds to the Nack in detail below.

Eviction of lines with lock words at the L1 is similar to DeNovo's L1 evictions (not shown in Table 1a). The main difference is that the writeback message needs to indicate which words are in *LockQ* state so that the L2 can perform appropriate action as discussed below. Table 1a does not show any action for writeback requests generated by L2 for L1. This is because the L2 does not need to maintain inclusion with the L1 for lock words (similar to *Valid* data in DeNovo). The distributed lock queue constructed in the L1s stays valid and does not need to be rebuilt on an L2 writeback.

L2 transitions without L1 writebacks: The L2 has two states – *Invalid* and *Valid*. The main source of complexity at the L2 comes from L1 writebacks of *LockQ* words; we therefore first discuss L2 transitions without L1 writebacks, indicated by *WB*=0 in Table 1.

On a lock request in *Valid* state, the L2 forwards the request to its *tailPtr* core and updates the *tailPtr* with the requesting core's ID. A lock request in *Invalid* state allocates the line for the lock word, triggers a fetch from memory, and keeps the L2 in *Invalid* state. When the response returns, the L2 transitions to *Valid* and applies the actions for the *Valid* state to the lock request (i.e., forwards the

	Lock request from core *i*	Unlock request from core *i*	Response for lock request from core *i*	Remote lock request from core *k*
LockQ	set *Locked*	**if** *nextPtr* != null send response to *nextPtr*; go to *Invalid* **else** unset *Locked*	unstall core *i*; merge received signature	**if** *Locked* is set *nextPtr* := *k* **else** send response to core *k*; go to *Invalid*
Invalid	stall core *i*; update tag; go to *LockQ*; set *Locked*; send lock request to L2 (writeback if needed)	send unlock request to L2	X	**if** sig-only request send response to core *k* **else** send *Nack* to L2

(a) L1 cache for core *i*

	Lock request from core *i*	Unlock request from core *i*	Lock/Unlock/WB/Nack response from memory for core *i*	Lock writeback from core *i*	Nack from core *i* for core *k*
Valid	**if** *WB* == 0 fwd req to *tailPtr*; **else** // *WB* == 1 **if** *Locked* is not set send sig-only req to *lastAcquirer* for *i*; *WB* := 0 **else** // *Locked* is set **if** *firstWaiter* != null fwd req to *tailPtr* **else** *firstWaiter* := *i*; *tailPtr* := *i*	**if** *firstWaiter* != null send sig-only req to *i* for *firstWaiter*; *WB* := 0 **else** unset *Locked*	X	**if** *firstWaiter* == null copy *Locked* from WB message; *lastAcquirer* := *i*; *firstWaiter* := *nextPtr*; *WB* := 1; **else** // race **if** *Locked* is not set send sig-only req to *i* for *firstWaiter*	**if** *WB* == 0 *firstWaiter* := *k* **else** **if** *Locked* is not set send sig-only req to *lastAcquirer* for *k*; *lastAcquirer* := null **else** *firstWaiter* := *k*
Invalid	update tag; send data req to memory; (writeback if needed)	update tag; send data req to memory; (writeback if needed)	**if not** tag match allocate line; update tag; (writeback if needed) go to *Valid*; apply actions for Lock/Unlock/WB/Nack as specified in *Valid*	update tag; send data req to memory; (writeback if needed)	update tag; send data req to memory; (writeback if needed)

(b) L2 cache

Table 1: State transitions for a lock word. *X* indicates unreachable states.

request to *tailPtr*). If the line was deallocated between the request and the response due to eviction, another line is allocated and the above action taken.

An unlock request in *Valid* state can only occur if the unlocking L1 previously performed a writeback on the lock (i.e., *WB=1*), and so is discussed below.

Writebacks generated by the L2 to memory are similar to DeNovo. As we see below, all the lock queue related information needed at the L2 is maintained as part of the lock word in the L2 – on an L2 writeback, this information is simply preserved at memory and made available to the L2 for later use.

Handling L1 lock writeback at the L2: When the L2 receives a writeback from an L1, it must ensure that it stores all information needed to construct the lock queue that was stored at the L1. This information is stored in the data portion of the L2 along with the *tailPtr*. An L1 writeback containing a lock word can originate only from the head of the lock queue in *LockQ* state because other cores are either stalled on their lock request or invalidated after transferring the lock. The L2, therefore, stores the following information in its data portion on an L1 writeback from core *i* (Figure 2 illustrates the L2 data layout with example values before and after the writeback):[3]

WB: The *WB* bit is set to 1 to indicate that the lock has been evicted from the L1 of the head of the lock queue.

Locked: The *Locked* bit from the writeback message is copied into the L2 to indicate whether the lock was released (*Locked*=0) at the time of the writeback.

lastAcquirer: L2 sets *lastAcquirer* as *i*. This is used to forward the next lock requestor to core *i* to obtain the access signature.

firstWaiter: L2 copies *nextPtr* from the writeback message into its *firstWaiter* field to indicate the first element in the queue after the head. On a subsequent unlock, the lock must be transferred to the *firstWaiter* core if it is not null.

Next we revisit the transitions for various messages at the L2 when the *Valid* state has *WB*=1. On a lock request, if *Locked* is not set (writeback occurred after lock release), L2 forwards the request to the *lastAcquirer* core. This request is for the access signature only since we already know that the lock has been released. If *Locked* is set (writeback before release), then L2 checks if *firstWaiter* is null. If it is not null, then L2 queues the request by forwarding it to *tailPtr*. Otherwise, it sets *firstWaiter* to *i* since there is no other waiter in the queue.

Similarly for unlock requests, if *firstWaiter* is not null, L2 forwards the request of *firstWaiter* to *lastAcquirer* for the signature (and implicit lock transfer). Otherwise, the queue is empty. L2 resets *Locked*, indicating that the evicted head is unlocked now and is ready to transfer the lock.

Handling races: There can be a race between an L1 lock writeback from core *i* and a request for the same lock from another core *k*. Thus, before getting the writeback, the L2 can forward core *k*'s request to L1. In this case, L1 nacks the request back to L2, which takes the following actions depending on whether it has already received the writeback (last column of Table 1b):

[3] Storing these fields in the data bank of the L2 does not limit the number of cores that can be supported as we can increase the data size of a lock variable as needed.

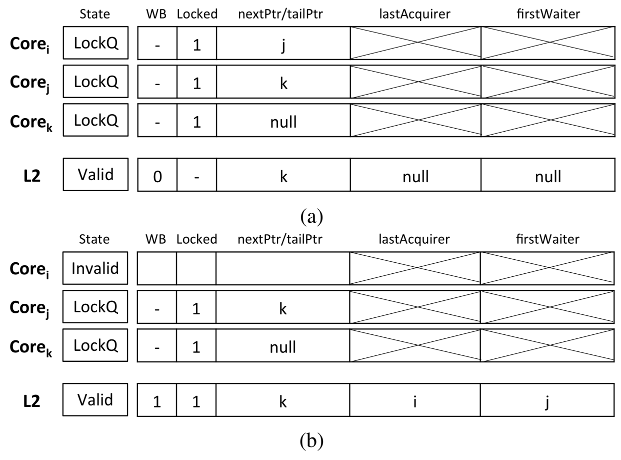


Figure 2: Example showing L1 and L2 data layout for the distributed queue-based lock (a) before writeback and (b) after writeback.

The Nack arrives before the writeback (WB=0): L2 simply sets *firstWaiter* to core *k*. When the writeback arrives, L2 finds its *firstWaiter* is not null and its request must be handled. If the *Locked* bit in the writeback is unset, L2 knows the lock was released and so can forward *firstWaiter*'s request to core *i* for signature transfer. If the *Locked* bit is set, then nothing needs to be done; the lock transfer to core *k* will occur when the *Unlock* arrives.
The Nack arrives after the writeback (WB=1): L2 services core *k*'s request using the information stored in the writeback; if *Locked* is not set, the request is forwarded to *lastAcquirer*. Otherwise, *k* is stored as the *firstWaiter*.

The above race is the only one that occurs in the lock protocol. It involves at most two cores and results in exactly one possible Nack message that the L2 immediately handles, with no deadlock or livelock causing actions.

4.3 Overheads

DeNovoND incurs the following overheads over DeNovo.
Hardware Bloom filter: There is one Bloom filter per core. A conservative upper bound for its size is the virtual memory size. In practice, an effective size can be empirically determined by measuring the number of atomic writes to distinct addresses in various applications. The size must also be large enough to have tolerable false positive rates. In our system, a relatively small size Bloom filter of only 256 bits worked well and provided performance similar to an infinite size Bloom filter for most cases. This is because the size of the key domain is restricted only to the addresses in atomic regions, and the filter is flash cleared at the end of a phase.

The quality of the hash function also impacts the efficiency of Bloom filters [42]. We experimented with two hash functions, multi-bit selection (similar to the one used in [16]) and H_3 (universal hash function that provides uniformly distributed hash values [15]), which showed consistent performance across applications. For our evaluation, we used H_3 which worked better with applications with high false positive rates. Finally, [16] has shown that Bloom filter operations of element insertion, membership query, and flash clear can be implemented very efficiently in hardware.
Storage overhead: Our distributed queue-based lock protocol reuses the L1 and L2 cache data banks to store the waiter queue information, incurring zero storage overhead for that purpose. It requires one additional state *LockQ* at L1 to distinguish between lock and data words. This does not result in any added storage overhead for L1 state as DeNovo already requires two bits per word for storing three states (*Invalid, Valid, and Registered*). With an additional *LockQ* state, we now have four states stored in two bits. The two L2 states for lock words can reuse the L2 per-word state bit of the baseline DeNovo protocol – lock words simply add new transitions to the existing L2 states, triggered by lock related messages. Thus, the lock protocol does not incur any additional storage overhead. The externally visible protocol states for data accesses also stay the same as for DeNovo. For efficient tracking of atomic writes, however, we added a *touched-atomic* bit per word in the L1 as an additional state bit (used only by the local core).
Communication and computation overhead: On acquire/release, the Bloom filter of the releaser is piggybacked on the lock transfer message. In order to minimize impact on network traffic, we can compress the Bloom filter using run-length encoding as in [16] or a Bloom-filter specific compression technique [38]. In our evaluations, we conservatively do not model such compression and charge the full 256 bits (32 bytes) of network traffic for the Bloom filter at a lock transfer. When a core receives a lock transfer message along with the signature, it needs to merge the received Bloom filter with its own before executing memory instructions in the critical section. The time for merging can be partially hidden by not blocking the execution until the first write/read instruction to an atomic region is issued.

For the distributed queue-based lock, there is an additional overhead for writeback messages which need to include an additional bit per word to indicate if the word is in *LockQ* state so that the L2 can perform appropriate lock related actions for this word. This overhead, however, can be compensated by observing that the writeback message does not have to contain full lock words, but only the *Locked* and *nextPtr* parts. The queue-based lock protocol also requires new state transitions in response to lock related messages; however, these do not introduce any new transient states or interact with the data protocol and can be separately verified.

5. Evaluation Methodology

5.1 Simulation Environment

For our evaluations, we use the Wind River Simics [34] full-system functional simulator to drive the Wisconsin GEMS detailed memory timing simulator [35] that we modified to implement our protocols. We also use the Princeton Garnet [3] interconnection network simulator to model network communication. To keep simulation times reasonable, as is common practice, we employ a simple, single-issue, in-order core model with blocking loads and 1 CPI for all non-memory instructions. (Note that DeNovoND does not require simple cores, but detailed timing simulation of a complex core would take an inordinate amount of time and we believe would not qualitatively affect our results.) We also assume 1 CPI for instructions executed inside the OS.

Table 2 shows the key parameters of our simulated systems. We simulate a multicore with 16 cores, a 64KB private L1 data cache per core (we do not model an Icache), a 16MB shared, NUCA L2 cache, and 4 memory controllers, all connected by a 2D mesh network. We configured the miss latencies to approximate those of the Nehalem processors [21]; e.g., a last-level shared cache miss (memory hit) costs 190 to 309 cycles on Nehalem (several of the latencies specify a range, depending on which L2 bank, remote L1 cache, or memory controller is accessed). We use the Bloom filter implementation shipped with GEMS [35] with the H_3 hashing function and 256 single-bit entries. We also simulated configurations with infinite Bloom filter entries for reference.

5.2 Simulated Systems

Our distributed queue-based lock is specifically designed for DeNovoND, reusing the coherence states of DeNovo, with no added transient states and limited race interactions. Implementing it on a conventional MESI-like protocol is possible, but will involve far more complexity to deal with interactions with the already existing numerous transient states and race conditions. On the other hand, comparing DeNovoND with distributed queue-based locks and MESI with conventional locking may not be fair to MESI. We therefore implemented simplified (idealized) queue-based locks that work for both MESI and DeNovo to isolate the effectiveness of access signatures. This idealized implementation maintains a "lock table" which is keyed by a lock variable address and maintains the waiter queue for each lock. Accesses to this table – creating an entry and grabbing the lock, adding a core to the waiter queue, waking up the first waiter in the queue, etc. – do not incur extra cycles. We also do not charge traffic overhead for lock and signature transfer for the idealized lock. Once a core is ready to release the idealized lock, lock transfer is instant and the next requestor wakes up immediately. Hence we evaluated the following systems:

MESI: We simulated MESI using idealized queue-based locks (MIL) and the `POSIX pthreads` mutex library (MPL). We modified the original implementation of MESI in GEMS [35] to support non-blocking writes for a fair comparison with DeNovoND where writes are non-blocking by default. Atomic instructions used in `pthreads` mutex codes are simulated using blocking store fences for correct execution.

DeNovoND: We simulated DeNovoND with idealized queue-based locks (DIL) and with distributed queue-based locks (DQL), both with a 256 bit Bloom filter (DIL-256 and DQL-256)) and, for reference, an infinite size Bloom filter (DIL-inf and DQL-inf). For DQL, operations on the lock incur latency consistent with table 2. For the signature transfer, we add a 256 bit (32 byte) payload to the lock transfer message and simulate network traffic and latency accordingly. This is conservative for DQL-256 since the signature could be compressed. It is aggressive but reasonable for DQL-inf since DQL-inf is intended to be a best case reference model.

5.3 Workloads

We evaluated 11 benchmarks with lock synchronization, taken from various suites to represent a range of behavior such as lock frequency, lock granularity, contention, critical section length, and shared working-set size. We evaluated *barnes* (16K particles), *ocean* (258×258), and *water* (512 molecules) from SPLASH-2 [45]; *fluidanimate* (35K particles) and *streamcluster* (8,192 points) from PARSEC 2.1 [8]; *tsp* (17 cities) as used in [12]; and *kmeans* (8,192 points, 24 dimensions, 16 centers), *ssca2* (2^{13} nodes), *genome* (256 nucleotides), *intruder* (1,024 traffic flows), and *vacation* (16,384 records) from STAMP [37].

The benchmarks from SPLASH-2 and PARSEC represent traditional applications designed and optimized to scale well with lock synchronization. The benchmarks from STAMP and *tsp*, however, were originally designed for hardware and software transactional memory. We ported them to use locks for our simulated systems. For short transactions, we directly replaced them with critical sections (*tsp*, *kmeans*, *ssca2*, and *intruder*). For longer transactions, we used finer-grained locks (*genome*, *vacation*).

We found that 3 out of the 6 transactional applications (*genome*, *intruder*, and *vacation*) spent $> 70\%$ of their execution time on lock acquire for all studied configurations. Clearly, parallelization using lock synchronization is inappropriate for these applications, for both MESI and DeNovoND. We therefore focus our results on the other 8 applications, referring to them as "lock-efficient" applications (Section 6.1). For completeness, we separately report results for the above three lock-inefficient applications (Section 6.2). We discuss optimizations to improve the performance of DeNovoND for the lock-inefficient applications, but fundamentally, these must be parallelized using different techniques for reasonable parallel speedups. Such techniques (including possibly transactional memory) are outside the scope of this work.

Core frequency	2GHz
# of cores	16
L1 data cache	64KB, 64 bytes (16 words) line size
L2 (16 banks, NUCA)	16MB, 64 bytes line
Memory	4GB, 4 on-chip controllers
L1 hit latency	1 cycle
L2 hit latency	29 to 61 cycles (bank-dependent)
Remote L1 hit latency	35 to 83 cycles
Memory hit latency	197 to 261 cycles
Network parameters	2D mesh, 16 bit flits
Bloom filter size	256 bits (infinite for reference)
hash function	4 H_3

Table 2: Simulated system parameters.

Finally, the lock-inefficient applications showed significant nondeterminism in execution time. Although our timing simulations are deterministic, they depend on the state of the system when the application is started (the Simics checkpoint at the start of the application). For different state, the lock-inefficient applications showed varying results. We therefore ran each such application with five different checkpoints for each system and averaged the results (the same five checkpoints are used for all systems). We also report the results for the lock-efficient applications averaged across three different checkpoints, but these applications did not show much variability across their checkpoints.

6. Performance Results

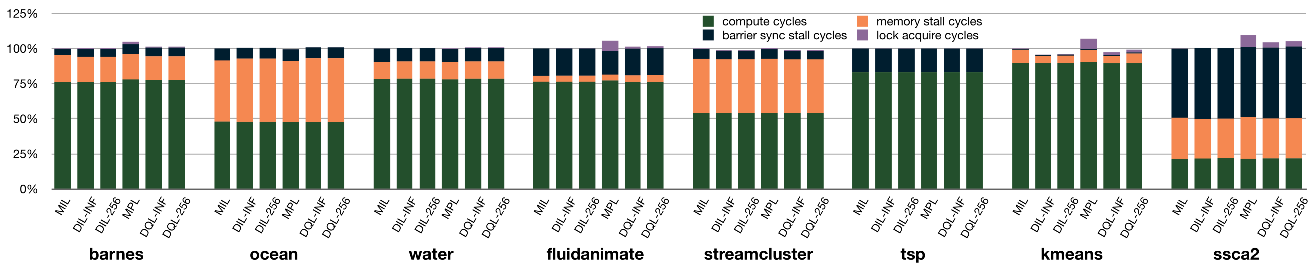
6.1 Lock-Efficient Applications

Figure 3a shows the execution time for our 8 lock-efficient applications for the 6 configurations described in Section 5.2. All bars are normalized to MIL. Each bar is divided into compute time, stall time due to data memory accesses (henceforth referred to as *memory time*), barrier time, and lock acquire time. Since we model non-blocking lock releases, lock release time is negligible. Since our focus is on the memory system, Figure 3b blows up the memory time in each bar of Figure 3a, divided into stalls for L1 misses resolved at L2, a remote L1, or main memory. Since all modeled systems implement non-blocking stores, virtually all memory stalls are due to loads. Figure 4a presents network traffic for the same applications on MPL and DQL-256 (normalized to MPL), classified by the message type: load, store, queue lock/unlock, writeback, and invalidation. The queue lock/unlock traffic exists only in DQL-256 for transferring distributed queue-based locks with signatures. For MPL, the lock traffic is aggregated with the data load and store traffic. Note that only MPL incurs invalidation traffic. We do not show network numbers with other configurations because they are idealized, but we confirmed that the network results for DQL-256 stay qualitatively similar even when compared to MIL.

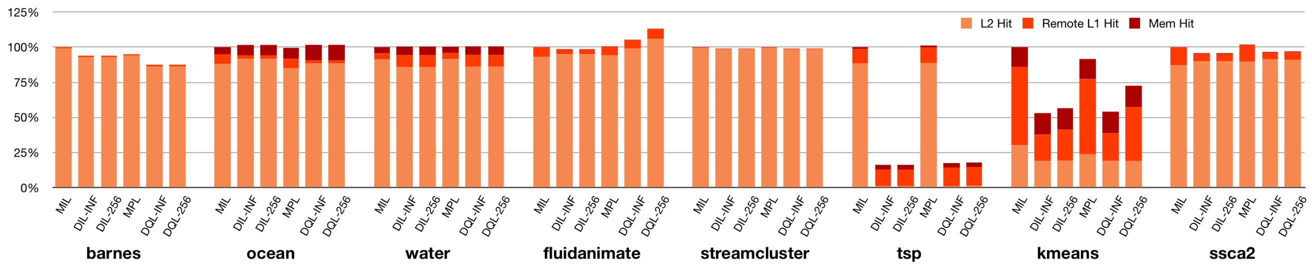
MIL vs. DIL-inf: For all 8 applications, DeNovoND shows the same or slightly better (up to 5%) execution time compared to MESI with idealized locks and infinite length Bloom filter. Focusing on memory time, again DIL-inf is either the same or better than MIL. For some applications, DIL is much better than MIL; e.g., 47% and 84% better for *kmeans* and *tsp* respectively. This is because MIL suffers from false sharing while DIL does not due to its per-word coherence state.

MPL vs. DQL-inf: Comparing the realistic lock implementations (but still with infinite Bloom filter size), we find that for all 8 applications, DQL-inf shows comparable or slightly better execution time than MPL. In fact, even compared to the idealized lock implementation in MIL, the execution time for DQL-inf is about the same or better in 7 of 8 cases and only 4% worse in the remaining case (*ssca2*). In terms of memory time, again DQL-inf is either comparable or sees large benefits due to the lack of false sharing relative to both MPL and MIL.

Impact of finite signatures: We next evaluate the impact of restricting the Bloom filter size: DIL-inf vs. DIL-256 and DQL-inf

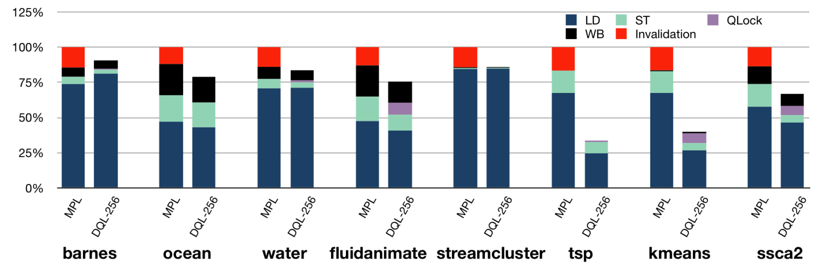


(a) Execution time.

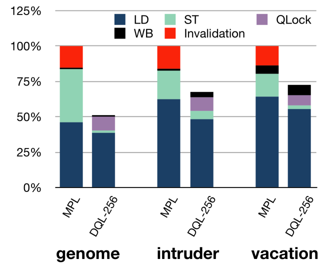


(b) Memory stall time.

Figure 3: Total execution time (a) and memory stall time (b) of lock-efficient applications on 6 configurations, normalized to MIL.



(a) Network traffic (lock-efficient).



(b) Network traffic (lock-inefficient).

Figure 4: Network traffic of all applications on MPL and DQL-256, normalized to MPL.

vs. DQL-256. The 256 bit Bloom filters show virtually the same execution times as the infinite length filters. In terms of memory time, the two Bloom filter sizes are similar for 6 of the 8 applications. For *fluidanimate* and *kmeans*, however, the 256 bit filter shows a degradation. For *kmeans*, memory time for DQL-256 continues to remain significantly better than for both MESI configurations (20% or more better), but for *fluidanimate*, it is worse by 13% (the only application where this is the case).

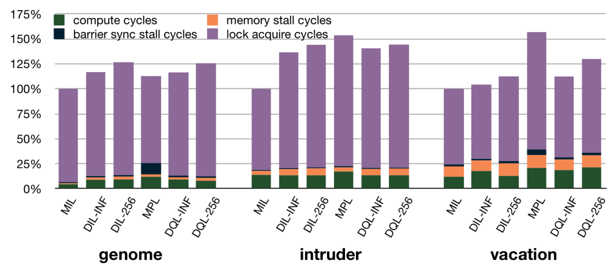
Fluidanimate and *kmeans* show the above behavior due to a confluence of a few subtle effects. First, both use critical sections where an atomic region address that is read is also written. Often an atomic region address read by a core was also last written by the same core (either in the previous phase or in a previous critical section). If this address is still in the core's cache in modified (for MESI) or registered (for DeNovoND) state, then the read will be a hit for both MESI and DeNovoND. Otherwise, if the address was written back, the read will be a miss for both MESI and DeNovoND. The difference between the protocols arises for any other atomic region addresses that come along with such a read miss as part of the same cache line. If the same core reads such an address in a subsequent critical section without an intervening write by another core, then MESI will still hit in the cache but DeNovoND will have to check against the Bloom filter. This could require a self-invalidation since the corresponding Bloom filter bit may be set, resulting in an extra miss over MESI. A smaller Bloom filter exacerbates this problem since it also results in false positives on the key domain. Further, the effect is more noticeable in DQL than in DIL because *fluidanimate* and *kmeans* have fine-grained locks – these locks pollute the cache and cause more replacements, exacerbating the above effect.

Network traffic: Figure 4a shows that for all the applications, DQL-256 has much lower traffic than MPL (33% on average, 67% maximum). This directly translates into energy reduction.

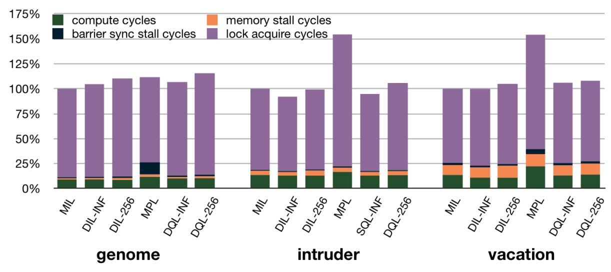
The primary sources of these savings in DeNovoND are as follows: (1) DeNovoND does not incur any traffic for invalidations, a significant effect in all applications. (2) Store traffic is reduced in some applications because store requests in DeNovoND do not bring in the cache line – they directly write into the L1 word and only send out a registration request for that word (multiple registrations for a given line are combined and sent on the network as mentioned in Section 2.2). (3) The net reduction in load misses (memory time) due to the lack of false sharing (Figure 3b) directly leads to lower load traffic in several applications. (4) Load traffic is further reduced because a load response only contains valid or registered words of a cache line. Since coherence state is preserved per word, some words may be invalid at the servicing cache.

A source for increased network traffic in DeNovoND is the 32 byte signature with all lock transfers. Figure 4a shows that this is small in all our applications. It can be further reduced through compression techniques mentioned in Section 4.3.

Summary: Overall, our results show that for these applications, the access signature mechanism allows DeNovoND to enjoy all the benefits of DeNovo even in the presence of lock-based synchronization. Further, the signature size needed is small (32 bytes).



(a) Baseline.



(b) "Write-Once" Atomic Region and Signature Clearing.

Figure 5: Total execution time of lock-inefficient applications on six configurations: (a) baseline, (b) with "write-once" atomic region optimization and signature clearing (threshold=99%) applied, normalized to the MESI with idealized locks (MIL) configuration.

6.2 Lock-Inefficient Applications

The lock-inefficient applications spend more than 70% of their time on lock acquires, but are presented here for completeness. Figure 5a shows their execution times analogous to Figure 3a. There are several ways in which these applications differ from the lock-efficient ones. First, as mentioned earlier, they are dominated by lock acquire time and so need a significantly different algorithm for parallelization and/or synchronization. These applications were originally designed to study transactional memory. Some of them use patterns for which lock-free synchronization is commonly used. Supporting such forms of parallelism and synchronization is outside the scope of this paper, but forms a key part of our future work.

Second, as discussed in Section 5.3, these applications show significant non-determinism. Although we report results averaged over five runs starting from five different Simics checkpoints (the same five checkpoints for each system), the variability makes comparing different systems difficult.

Third, we find that compute time varies across different systems for each of these applications. Although not shown here, a significant fraction of compute time comes from the OS (e.g., due to frequent memory allocations), forming the main source of the compute time variation. (The lock-efficient applications have negligible OS compute time.) Our results must therefore be understood in the context of the above caveats.

MIL vs. DIL-inf: For all three applications, DIL-inf shows observably worse performance than MIL (16% for *genome*, 36% for *intruder*, and 5% for *vacation*). A large part of the performance difference appears to come from acquire time; e.g., DIL-inf spends 40% more cycles waiting for lock acquisition than MIL with *intruder*. Though memory time is a very small portion, it affects acquire time by increasing the time spent within critical sections. Our detailed results show that DIL-inf suffers from higher memory time than MIL, especially for *genome* and *intruder*.

The higher memory time above occurs due to an access pattern where an address is written only once in a phase and then read several times. Specifically, *genome* and *intruder* use list and hash table data structures that store "data" or "key-data" pairs of each entry as a field of the entry object – in these programs, the data is initialized when a new element is inserted (within a critical section) but never modified afterwards. A core may read this data later in different critical sections – DeNovoND will self-invalidate on such reads since it does not know if there was an intervening write since the last read. MESI, on the other hand, will hit on such reads if they happen close enough to exploit temporal locality.

Section 6.2.1 discusses how we can use software information to remedy the above situation. We believe, however, that a better solution to this problem is a better synchronization construct – using locks for such reads is overkill. Such constructs in the context of DeNovo and DeNovoND are a key part of our future work.

MPL vs. DQL-inf: DQL-inf performs slightly worse than MPL with *genome* for the same reason as the comparison between MIL and DIL-inf. DQL-inf outperforms MPL with *intruder* and *vacation* – for these applications, MPL has significantly higher acquire time than MIL. MPL's pthread locks, however, are inherently inefficient with high lock contention; therefore, this is not a fair comparison for MESI. Thus, little can be deduced here except perhaps that DeNovoND performance seems to be in the same range as MESI (this inability to draw a conclusion is an inherent artifact of the problem studied).

Impact of finite signatures: With smaller Bloom filter sizes, false positives exacerbate the impact of the conservative invalidations described above; for *genome* and *intruder* – DIL-256 and DQL-256 perform worse than DIL-inf and DQL-inf by 4% to 10%.

Vacation does not suffer from the conservative invalidations of *genome* and *intruder*, but reveals a different source of inefficiency with smaller signatures. Figure 5a shows DIL-256 is 8% worse than DIL-inf, while DQL-256 is 17% worse than DQL-inf for this application. This is mainly due to its large working set of atomic data, which can increase the false positive rate if a Bloom filter is too small. In addition, *vacation* has only one phase without any barriers in between; thus the Bloom filters get filled up for a long period without clearing. This further exacerbates the false positive rate, resulting in unnecessary self-invalidations and higher memory times. Section 6.2.1 describes an optimization technique called signature clearing to deal with this issue.

Network traffic: Figure 4b shows network traffic of the lock-inefficient applications on MPL and DQL-256. DQL-256 generates less network traffic (up to 48%) than MPL for all three applications for reasons similar to that for the lock-efficient applications. In addition, with relatively high lock contention, repeated accesses to lock variables can generate increasingly higher network traffic in MPL. In contrast, distributed queue-based lock request/response traffic scales in proportion to the number of lock transfers.

6.2.1 Optimizations

Handling "write-once" atomic data: As with the case with *intruder* and *genome*, once a new entry is created and then inserted into a data structure (list, hash table, etc.), the "data" portion of the entry may remain read-only for the entire execution while other fields of the entry are modified as the structure grows or shrinks. In this case, classifying the "data" as atomic makes every self-invalidation after the very first one (the memory location may have been used and freed before) unnecessary.

DeNovoND can safely get rid of these invalidations by identifying such atomic accesses as made to a "write-once" atomic region. In addition to general information about atomic regions and effects, software can allow such "write-once" atomic data to be marked differently by using a special region ID or a special op-code for the write. Then DeNovoND can exploit it to prevent such data from being self-invalidated as follows. If the data is known to be in a "write-once" atomic region, DeNovoND does not reset its *touched-atomic* bit on lock transfer; therefore, when the data is accessed (read) again later, it is treated as if it has been already accessed in the same critical section (with *touched-atomic* bit set) and will not be self-invalidated, thereby eliminating several subsequent misses.

The write-once annotation can be considered to be a generalization of *final* variables in Java; a final variable can only be initialized once, either at the time of declaration or by the constructor of the class in which it is declared [40]. Our write-once variables must be written (at most) once per parallel phase.

Signature clearing: Depending on the atomic write-set size in a phase, the fixed-size hardware Bloom filter may get saturated (all bits set) before the phase is over. This drives the false positive rate very high, resulting in many unnecessary self-invalidations. Saturated Bloom filters can be flash-cleared by a simple hardware operation, but it also requires flushing out atomic words in the cache. Also, the fact that a signature has been cleared in the releaser should be propagated to the acquirer so that the acquirer can update its cache according to the new version of the Bloom filter. We implemented a signature clearing algorithm that carries a vector of clearing counters per core. When signature clearing is triggered on a core, its counter is incremented. The vector of clearing counters is transferred on a lock transfer along with the access signature. The acquirer compares the received vector with its own, and performs signature clearing if there exists an element in the received vector that has a larger counter than the corresponding element in its own vector. Before the lock is transferred again, the vector is updated to have up-to-date values.

Performance impact: Figure 5b presents execution times analogous to figure 5a, but with the above optimizations applied.

For *genome*, all DeNovoND protocols now perform comparable to the MESI counterpart. Our detailed results show large reductions in memory time from the write-once optimization (118% to 151%). Since this reduction mainly comes from atomic accesses within critical sections, lock contention also improved. *Intruder* shows similarly dramatic results in memory time improvement with consequently large improvements in execution time for the DeNovoND configurations; acquire time is reduced by 36 to 42%, memory time by 56 to 76%, and overall execution time by 43% on average.

For *vacation*, DIL-256 and DQL-256 (protocols with finite Bloom filters) show performance benefits from signature clearing; DIL-256 and DQL-256 were 17% and 8% worse than DIL-inf and DQL-inf respectively without signature clearing. With signature clearing, with 99% filter saturation percentage as the trigger for clearing, the difference is reduced to 5% and 2%.

Overall, the optimizations are quite effective, making the DeNovoND protocols comparable or better than the corresponding MESI protocols even for the lock-inefficient applications.

7. Related Work

There has been much research on improving the performance of memory consistency models by guaranteeing consistency only at synchronization points. Our work is closest to that of lazy release consistency (LRC) [27], entry consistency (EC) [7], and scope consistency (ScC) [22]. A key focus of these models is saving invalidation network traffic by postponing propagation of modified data until an acquire. LRC maintains consistency of all shared data at every lock transfer. EC attempts to reduce traffic by requiring programmers to bind every shared object with a lock, and transferring only the bound data objects on a lock transfer. ScC attempts to relax the strict and explicit bindings between data and lock in EC; instead, it uses “consistency scope” to implicitly associate data and the acquire/release pair protecting the data. DeNovoND is similar in that it also assumes a software guarantee for data-race freedom and association of atomic regions and sections. However, a key difference between DeNovoND and the above models is that the latter are designed for software distributed shared memory, keeping coherence information at a coarse-grained page granularity and storing information about modified data in data structures in user space. DeNovoND focuses on tightly coupled multicores with different trade-offs. In particular, DeNovoND implements a much simplified yet effective scheme for tracking modified atomic locked data in hardware, while leveraging the feature of the baseline system (no invalidation traffic) for non-atomic data.

REFLEX [33] employs software distributed shared memory with release consistency to make it easier to program low-power smartphones. It uses either eager or lazy update propagation depending on the initiating core’s power profile. While REFLEX concentrates on adapting release consistency for low power on heterogeneous systems, DeNovoND is a more general solution that addresses complexity, performance, and power.

The recent SARC coherence protocol [25] also exploits the data-race-free programming model, but their goal is to improve the conventional directory-based protocol [2]. SARC self-invalidates “tear-off, read-only” (TRO) copies of data to save power. However, SARC does not eliminate directory storage overhead or reduce protocol complexity like DeNovoND and its baseline system. Also, the concept of touched bit, which plays an important role in DeNovo and DeNovoND is not present in SARC.

Other efforts to improve coherence achieve one or more of our goals at the expense of other goals. [31] introduces more complexity for self-invalidations and [36] requires writes to go to a shared cache if there are potential conflicts. SWEL [41] and Atomic Coherence [44] rely on specific interconnect substrates to simplify their protocols. Rigel [28] and Cohesion [29] propose systems with accelerators using a hybrid memory model based on shared memory, and employ software-driven invalidation for coherence. However, Rigel eagerly writes back all dirty lines to the global shared cache at phase boundaries, causing potentially unnecessary and bursty network traffic. DeNovoND self-invalidates potentially stale blocks only, avoiding this unnecessary traffic. Cohesion does not address existing limitations of software and directory-based hardware coherence mechanisms. Its software coherence issues extra coherence instructions wasting cycles and network bandwidth since its coherence tracking is conservative and coarse-grained, while the hardware directory-based protocol has the same current complexity and scalability issues. In contrast, DeNovoND starts from a simple protocol and makes it easy to add various optimizations to improve performance and energy further without complicating the protocol.

We leverage much prior work on Bloom filters, which have recently been widely used for access tracking [16, 43, 46]. Typical prior such usage, however, uses filters in the range of 1K to 2K bits. DeNovoND is able to achieve competitive performance with 256 bits, with commensurately lower space and computation overheads, since its key domain is limited to atomic addresses.

8. Conclusion

This paper takes a significant step towards a vision for complexity-, performance-, and energy-efficient multicores enabled by disciplined shared-memory programming practices. Prior work on DeNovo showed how this vision could be achieved for deterministic programs. This paper develops DeNovoND, a system that additionally supports disciplined non-determinism with minimal additional overheads and complexity relative to DeNovo.

DeNovoND exploits a previously developed software-level guarantee that non-deterministic (atomic) data accesses are distinguishable and protected by a lock. The key insight is to use small and simple hardware Bloom filters to track and communicate such accesses across lock transfers, preserving DeNovo’s previous advantages of no transient states, directory overhead, invalidation messages, or false sharing. Underlying the data transfer mechanism is a distributed queue-based lock mechanism that uses the cache data banks to construct a lock-waiter queue, without additional state bits or directory storage.

DeNovoND provides comparable or better performance than MESI with the lock-efficient programs studied here. Further, network traffic is significantly reduced, impacting energy. We also identified some patterns in lock-inefficient code that did not work as well with DeNovoND – we showed optimizations to mitigate those effects, but believe the correct solution lies in alternate forms of synchronization for such codes.

As future work, we plan to explore broadening the scope of our vision of hardware-software co-design rooted in disciplined programming to embrace further programming patterns such as pipelined parallelism and "lock-free" data structures, as well as support complex codes such as legacy codes and operating systems. Our ability to easily extend DeNovo to embrace lock based codes gives us further confidence in generalizing this vision.

Acknowledgments

This work was supported in part by Intel Corp. and Microsoft Corp. through the Universal Parallel Computing Research Center at Illinois, Intel Corp. through the Illinois-Intel Parallelism Center, and the National Science Foundation under grant CCF-1018796.

References

[1] S. Adve and H.-J. Boehm. Memory Models: A Case for Rethinking Parallel Languages and Hardware. *CACM*, Aug. 2010.

[2] S. Adve and M. Hill. Weak Ordering - A New Definition. In *ISCA*, 1990.

[3] N. Agarwal, T. Krishna, L.-S. Peh, and N. Jha. GARNET: A Detailed On-Chip Network Model inside a Full-System Simulator. In *ISPASS*, 2009.

[4] M. Allen, S. Sridharan, and G. Sohi. Serialization Sets: A Dynamic Dependence-based Parallel Execution Model. In *PPoPP*, 2009.

[5] Z. Anderson, D. Gay, R. Ennals, and E. Brewer. SharC: Checking Data Sharing Strategies for Multithreaded C. In *PLDI*, 2008.

[6] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe Multithreaded Programming for C/C++. In *OOPSLA*, 2009.

[7] B. Bershad, M. Zekauskas, and W. Sawdon. The Midway Distributed Shared Memory System. In *Compcon Digest of Papers.*, 1993.

[8] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.

[9] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *CACM*, 13:422–426, 1970.

[10] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *PPoPP*, 1995.

[11] R. Bocchino, Jr., V. Adve, D. Dig, S. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA*, 2009.

[12] R. L. Bocchino, Jr., S. Heumann, N. Honarmand, S. V. Adve, V. S. Adve, A. Welc, and T. Shpeisman. Safe Nondeterminism in a Deterministic-by-Default Parallel Language. In *POPL*, 2011.

[13] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An Improved Construction for Counting Bloom Filters. In *ESA*, 2006.

[14] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Taşirlar. Concurrent Collections. *Sci. Program.*, 18(3-4), Aug. 2010.

[15] J. L. Carter and M. N. Wegman. Universal classes of hash functions (extended abstract). In *STOC*, 1977.

[16] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *ISCA*, 2006.

[17] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *PACT*, 2011.

[18] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic Shared Memory Multiprocessing. In *ASPLOS*, 2009.

[19] A. Ghuloum, E. Sprangle, J. Fang, G. Wu, and X. Zhou. Ct: A Flexible Parallel Programming Model for Tera-Scale Architectures, 2007.

[20] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. In *ASPLOS*, 1989.

[21] D. Hackenberg, D. Molka, and W. E. Nagel. Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems. In *MICRO*. IEEE, 2009.

[22] L. Iftode, J. P. Singh, and K. Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. In *SPAA*, 1996.

[23] Intel. The SCC Platform Overview, 2010.

[24] A. Kägi, D. Burger, and J. R. Goodman. Efficient Synchronization: Let Them Eat QOLB. In *ISCA*, 1997.

[25] S. Kaxiras and G. Keramidas. SARC Coherence: Scaling Directory Cache Coherence in Performance and Power. *IEEE Micro*, 2010.

[26] S. Keckler, W. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the Future of Parallel Computing. *IEEE Micro*, 31:7–17, 2011.

[27] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *ISCA*, 1992.

[28] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel. Rigel: An Architecture and Scalable Programming Interface for a 1000-core Accelerator. In *ISCA*, 2009.

[29] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel. Cohesion: A Hybrid Memory Model for Accelerators. In *ISCA*, 2010.

[30] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic Parallelism Requires Abstractions. In *PLDI*, 2007.

[31] A. Lebeck and D. Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. In *ISCA*, 1995.

[32] E. A. Lee. The Problem with Threads. *IEEE Computer*, 39(5), 2006.

[33] F. X. Lin, Z. Wang, R. LiKamWa, and L. Zhong. Reflex: Using Low-Power Processors in Smartphones without Knowing Them. In *ASPLOS*, 2012.

[34] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *Computer*, 35:50–58, 2002.

[35] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Computer Architecture News*, 2005.

[36] S. L. Min and J.-L. Baer. Design and Analysis of a Scalable Cache Coherence Scheme Based on Clocks and Timestamps. *TPDS*, 1992.

[37] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *IISWC*, 2008.

[38] M. Mitzenmacher. Compressed Bloom Filters. In *PODC*, 2001.

[39] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *ASPLOS*, 2009.

[40] Oracle. Java Language and Virtual Machine Specifications.

[41] S. H. Pugsley, J. B. Spjut, D. W. Nellans, and R. Balasubramonian. SWEL: Hardware Cache Coherence Protocols to Map Shared Data onto Shared Caches. In *PACT*, 2010.

[42] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam. Implementing Signatures for Transactional Memory. In *MICRO*, 2007.

[43] A. Shriraman, S. Dwarkadas, and M. L. Scott. Flexible Decoupled Transactional Memory Support. In *ISCA*, 2008.

[44] D. Vantrease, M. H. Lipasti, and N. Binkert. Atomic Coherence: Leveraging Nanophotonics to Build Race-Free Cache Coherence Protocols. In *HPCA*, 2011.

[45] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA*, 1995.

[46] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *HPCA*, 2007.