© 2022 Subho Sankar Banerjee

A FRAMEWORK FOR INTELLIGENCE AUGMENTED COMPUTING SYSTEMS

BY

SUBHO SANKAR BANERJEE

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2022

Urbana, Illinois

Doctoral Committee:

Professor Ravishankar K. Iyer, Chair
Professor Wen-mei Hwu
Professor Vikram Adve
Professor Subhashis Mitra, Stanford University

ABSTRACT

Large-scale computing systems rely on many control and decision-making algorithms. Classical approaches to designing and optimizing these algorithms are poorly suited to the diverse and demanding requirements of modern systems and emerging applications. The state of the art paradigm for building these control algorithms often devolves into painstakingly built, handcrafted, average-case heuristics. However, as systems and applications have grown in complexity and heterogeneity, designing fixed algorithms that work well across a variety of conditions has become exceedingly difficult and costly. Moreover, we are reaching the limits of conventional approaches of generating heuristics, which involve recurring human-expert-driven engineering efforts. Such an approach will be untenable in the future.

In this thesis, we investigate a new paradigm for solving large scale system management and optimization problems. We develop systems that can learn to optimize the performance on their own using modern machine learning techniques. As a result, in the proposed approach, the system designer need not develop specialized heuristics for low-level design goals. Instead, the designer architects a framework for measurement, estimation, experimentation, and learning that discovers the low-level actions that achieve high-level resource management objectives automatically. We use this approach to build a series of practical intelligent controllers for the management and optimization of large-scale data-parallel and data-processing workloads on heterogeneous computer systems. Our contributions encompass building mathematical models (e.g., for denoising telemetry data), policies (e.g., for scheduling), optimizations to enable real time inference, and the design and implementation of practical software and hardware that provides efficient, scalable, and composable system management solutions.

“To my parents, for their love and support.”

ACKNOWLEDGMENTS

I am thankful to my advisor, Ravishankar K. Iyer, who guided me tirelessly throughout my Ph.D. He was always able to push ideas one level further, provide the input needed to finish a task, and share his experience about the research process, which was invaluable to me. There were several times through numerous paper submissions that his encouragement reenergized me to work harder and exceed my own expectations. I feel very fortunate and privileged to have him as my doctoral advisor.

I would also like to thank my distinguished doctoral dissertation committee members Professor Wen-mei Hwu, Professor Vikram Adve, Professor Subhasish Mitra (Stanford University) for their insightful comments and suggestions on my research work.

The work in this dissertation is the result of collaboration with many other people. Foremost among my collaborators, I am extremely thankful for the help and support of Zbigniew Kalbarcyzk. Zbigniew has been among the first people to hear, critique and mold the ideas presented in this thesis. I count myself especially fortunate to work with him and benefit his perspectives. I would like to thank student-colleagues Saurabh Jha, Phuong Cao, Ching Tan, Haoran Qiu, Archit Patke, Shengkun Cui, Jingde Chen, James Cyriac, and Yuming Wu for contributing to the development and refinement of the ideas in this thesis. I am also extremely thankful for the help, feedback and advice of Mohamed el-Hadedy, Arjun Athreya, Jong Lim, Daniel Chen, Liudmila Mainzer, Steve Lumetta, Deming Chen, and C. Victor Jongeneel, in the earlier part of my doctoral studies, when my work focused designing and implementing efficient accelerators for genomics data-processing workloads.

Beyond direct collaborators on the projects here, many other people contributed to my graduate work and made my time at the University of Illinois an unforgettable experience. I would like to thank my fellow lab-mates and friends from Champaign-Urbana Anand Ramachandran, Anirudh Choudhary, Archit Patke, Arjun Athreya, Atul Bohra, Carmen Cheh, Catello Di Martino, Chang Hu, Ching Tan, Cuong Pham, Daniel Chen, Debjit Pal, Elizabeth Gray, Eric Badger, Erman Gugnoor, Fei Deng, Haiyang Zhang, Haoran Qiu, Haotian Chen, Harshita Sreejith, Homa Alemzadeh, Hui Lin, James Cyriac, Jingde Chen, Key-whan Chung, Krishnakant Saboo, Lavin Devnani, Mihir Iyer, Mohamed el-Hadedy, Mohammad Noureddine, Mosbah Aoudh, Nirupam Roy, Ozgun Numanoglu, Phuong Cao, Pooja Malik, Qingkun Li, Safa Messaoud, Saurabh Jha, Sharon Tang, Shengkun Cui, Trisha Das, Uttam Thakore, Valerio Formicolla, Varun Krishna, Vikram Anjur, Yogatheesan Varatharajah, Yixin Chen, Yuming Wu, Yurui Cao, Zachary Estrada, Zachary Stephens, Zhengping Wang, and

Zhihao Hong for the stimulating discussions on study, research, graduate life, and for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last couple of years. Sincere thanks to Heidi Leerkamp, Kathleen Atchley, Jeni Summers, Carol Bosley, and Carol Wisniewski for their help and administrative support, and Jenny Applequist in helping refine our manuscripts and proposals.

None of this work could have been possible without all the constant unconditional love and endless support from my parents, Sharmila and Saumya. They have supported me during my highs and lows. Without their constant encouragement and sacrifice, I would have never been able to finish my doctoral study. Finally, I would like to thank Dotcom. Watching him do cute and silly things over video calls to my parents was often the most effective stress-relief during my doctoral grind.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1
1.1 Overview 1
1.2 General Approach: Intelligence Augmented Systems 2
1.3 Summary of Contributions: Methods & Systems Developed 4
1.4 Organization 7

CHAPTER 2 BACKGROUND, CHALLENGES AND RELATED WORK 8
2.1 Complexity & Dynamics: The Advent of Heterogeneity 8
2.2 Machine Learning for Systems 11

CHAPTER 3 DEALING WITH NOISY TELEMETRY DATA 15
3.1 Introduction 15
3.2 Background: HPC Errors 17
3.3 Approach Overview 20
3.4 The BayesPerf ML Model 22
3.5 The BayesPerf Implementation 27
3.6 Evaluation & Discussion 31
3.7 Related Work 39
3.8 Summary 40

CHAPTER 4 LEARNING SCHEDULING POLICIES FOR HETEROGENEOUS CLUSTERS 41
4.1 Introduction 41
4.2 Background 43
4.3 Training the POMDP RL-Agent with Back-Propagation 45
4.4 Scheduling Data Center Workloads By Using Reinforcement Learning 48
4.5 Application Specific Features for Scheduling Policies 57
4.6 Evaluation & Discussion 59
4.7 Summary 64

CHAPTER 5 ACCELERATED AND REAL-TIME INFERENCE 65
5.1 Introduction 65
5.2 Background 67
5.3 Approach Overview 70
5.4 Compiler Front-End 71
5.5 Sampling Element Design 73
5.6 Controller Design 81
5.7 Accelerator Synthesis 82

5.8 Evaluation and Discussion 84
5.9 Related Work 91
5.10 Summary 92

CHAPTER 6 CONCLUSIONS 93
6.1 Looking Forward 94

APPENDIX A ACCELERATED LEVENSHTEIN DISTANCE COMPUTATION . 96
A.1 Introduction 96
A.2 Levenshtein Distance Computation and Short-Read Alignment 98
A.3 Design of the ASAP Accelerator 102
A.4 Evaluation and Discussion 115
A.5 Related Work 124
A.6 Summary 126

APPENDIX B ACCELERATED PAIRHMM FORWARD ALGORITHM 128
B.1 Introduction 128
B.2 Background 130
B.3 Accelerator 133
B.4 Algorithmic Optimizations 138
B.5 Experimental Results 141
B.6 Related Work 145
B.7 Summary 146

REFERENCES 148

CHAPTER 1: INTRODUCTION

1.1 OVERVIEW

Emerging applications powered by deep learning demand increased computational capacity to achieve the performance and high resiliency needed to ensure uninterrupted operations in the presence of accidental failures and malicious attacks. Computer systems are rapidly evolving to meet those application demands by incorporating innovations in hardware architecture, operating systems, network interconnects, and storage, leading to increased heterogeneity. The problem of integrating such innovations into a single coherent computing stack has a long history and solutions of drawing upon many areas across computer science and applied mathematics. However, in practice, solutions often devolve into painstakingly built, handcrafted, average-case heuristics (or static policies). For example, hardware makes decisions on voltage scaling and cache prefetching; compilers make decisions about which routines to inline; networking stacks make decisions about when to adjust the congestion window; and operating systems choose which processes and threads to schedule next.

Heuristic generation is already a fundamental challenge. The typical heuristic design flow involves: (i) constructing a simplified model of the system optimization problem; (ii) breaking down high-level optimization objectives (e.g., tail latency of a web service) into low-level design goals (e.g., minimizing resource contention between co-located tasks); (iii) coming up with heuristics to achieve these design goals under the simplified model and extensively tune the heuristics to reach good performance in real-world systems. Variations across machine configurations, workloads, and deployment environments can make the above heuristic generation painful and costly. We are reaching the limits of conventional approaches of generating heuristics, which involve recurring human-expert-driven engineering effort. It is evident that existing approaches that focus on individual pieces of the system stack are untenable.

The state of the art is heavily predicated on solving fixed, well-defined and semantically self-contained problems. Unfortunately, it is becoming increasingly difficult to design highly performant and robust systems using this approach. *First*, the underlying components in many modern applications interact in complex, non-linear ways and are often extremely difficult to model accurately. For example, in cluster scheduling, the runtime of a task varies with data locality, hardware capabilities and characteristics, interactions with other tasks, and interference on shared resources. *Second*, practical optimization algorithms must operate in a wide range of heterogeneous and potentially unknown conditions which are impossible

to captured by fixed and simplified models completely. *Finally*, it is often infeasible to find a correct set of low-level design goals that perfectly add up to the high-level optimization objectives; practical solutions must combine and jointly tune several heuristics to optimize performance. As a result, system designers are often forced to sacrifice performance for simplicity and universality, or be forced to develop point solutions and specialized heuristics for each environment and application.

In this dissertation, we take a step back and ask what is the most natural way for machines to optimize complex computing systems. Rather than explicitly design and tune fixed algorithms for each problem, we seek to enable systems that efficiently learn to optimize the performance on their own. In our proposed approach, the system operator does not design specialized heuristics for low-level design goals using a simplified model of the system. Instead, he architects a framework for *data collection*, *experimentation*, and *learning* to discover the low-level actions that achieve a high-level optimization objective automatically.

1.2 GENERAL APPROACH: INTELLIGENCE AUGMENTED SYSTEMS

This dissertation addresses complexity brought on by heterogeneity (across processing fabrics, interconnection networks) and rapidly changing applications without compromising performance using data-driven optimizations. In particular, we focus on building "system policies"[1]. Given a software or hardware component (e.g., a compiler pass, branch predictor, memory allocate) that makes decisions related to the execution of computer programs, a *system policy* describes how these decisions are made. While the classical paradigm of building such policies has been successful in some homogeneous settings, where we can model the system quite accurately and design algorithms with strong performance (e.g., datacenter transport), we currently lack design techniques for high-performance networked systems across heterogeneous environments and application requirements.

This work augments the computer system with data-driven learning techniques at different levels of the computing stack to overcome the two fundamental challenges of intelligent system management. The first is *complexity*: modern hardware and system software expose diverse configurable parameters whose complicated interactions have surprising effects on performance and reliability. The second is *dynamics*: computing systems must reliably adapt to unpredictable changes in operating environment, input workload, and even user needs. This work combines the use of black-box deep learning techniques—to handle complexity—with domain-knowledge-driven (in this case, knowledge about the system's software and

[1]We refer to these system policies as *policies*.

hardware architecture) Bayesian probabilistic models—to handle dynamics—to produce hybrid machine learning (ML) models that are easier/faster to train and can be more easily debugged, making them more amenable for production deployments. These ML models are incorporated into *agents* that define *policies* (i.e., control algorithms) for the well-defined semantically self-contained tasks (e.g., scheduling, controlling cache allocations). The agent starts knowing nothing about the task at hand and learns by reinforcement-a reward signal that it receives based on how well it is doing on the task. Over time, the agents learn to make decisions directly from the experience of interacting with the environment. As multiple agents are deployed in the system to control different aspects of the computing stack, the underlying policies of the agents are jointly optimized by having multiple agents share a common reward signal. In a sense this enables a kind of cross-stack optimization of control policies of a computer system that simply cannot be done using the traditional heuristic based approach. Further, the data-driven nature of the proposed approach enables the system to achieve optimal results and continuously stay in optimal states, by taking full advantage of contextual information (i.e., telemetry data[2]) collected from multiple layers of the system stack and adapting rapidly to the deployment environment, workloads, and application requirements. Finally, we believe that system management decisions are often highly repetitive, making it easy to collect an abundance of training data to train reinforcement learning (RL) models. This fact largely negates one of the potential drawbacks of RL approaches in practice — their high sample-complexity (i.e., need for a large amount of training data).

At a high level, we believe that several benefits of the proposed approach that are particularly well-suited to for management of future large-scale computing systems. These benefits leverage many inherent system properties and provide a data-driven solution for the underlying optimization problems that traditional approach struggle to tackle:

1. *Tailor policies for a specific environment:* By continuously learning from the real experiences of interacting with a system environment, the agents directly optimize for the actual workload and operating conditions as opposed to relying on inaccurate system models.

2. *Handle hard-to-model system dynamics:* With the use of general purposed and powerful function approximators such as deep neural networks, the agents can incorporate a rich set of latent relationships from raw observations (e.g., interference between workloads on shared resources like CPU caches).

[2]The monitoring data include (i) performance counters from processors and interconnects, (ii) probes and tracing data from operating systems and file systems, (iii) detailed error logs collected across the stack, and (iv) application-level tracing data.

3. *Optimize for high-level system objective directly:* The agents can learn to optimize a variety of high-level optimization objectives (e.g., tail latency) without prior knowledge of how low-level metrics (e.g., cache hit ratio) impact the objective.

4. *Learn data-driven heuristics for hard algorithmic problems:* The underlying problem structures of many systems involve combinatorial optimization problems (e.g., job scheduling), which generally lack a generic optimal solution. The reinforcement driven technique can help improving the optimization solution for individual systems (as demonstrated in other complex combinatorial optimization problems like [1, 2]).

1.3 SUMMARY OF CONTRIBUTIONS: METHODS & SYSTEMS DEVELOPED

This dissertation develops a set of novel, transformative inflight analytics techniques for controlling, managing, and optimizing large-scale heterogeneous computer systems across the hardware and software stack (including processors, interconnects, storage, operating systems, and software infrastructure). The inflight analytics leverage the abundance of monitoring and telemetry data available across the stacks to model different aspects of heterogeneous systems using partially observable Markov decision processes (POMDPs) augmented with domain knowledge. This leads to fundamental improvements to the performance, reliability, and security of large-scale, distributed, and heterogeneous systems. Fig. 1.1 provides an overview of the proposed approach. Broadly, the dissertation will tackle the following research problems.

1.3.1 Dealing with Noisy Telemetry Data

The challenge is to accurately estimate the system state using rich telemetry data from the monitors available across the software and hardware stack. The collected data are often noisy, incomplete and inconsistent in terms of semantics, modalities, time granularities, levels of abstraction, and propagation effects, making the system only partially observable. Hardware performance counters (HPCs) that measure low-level architectural and microarchitectural events are one such category of telemetry data. HPC measurements are error-prone due to non determinism (e.g., undercounting due to event multiplexing, or OS interrupt-handling behaviors). In this thesis, we present BayesPerf (labeled 1 in Fig. 1.1), a system for quantifying uncertainty in HPC measurements by using a domain-driven Bayesian model that captures microarchitectural relationships between HPCs to jointly infer their values as probability distributions. We provide the design and implementation of an accelerator that allows

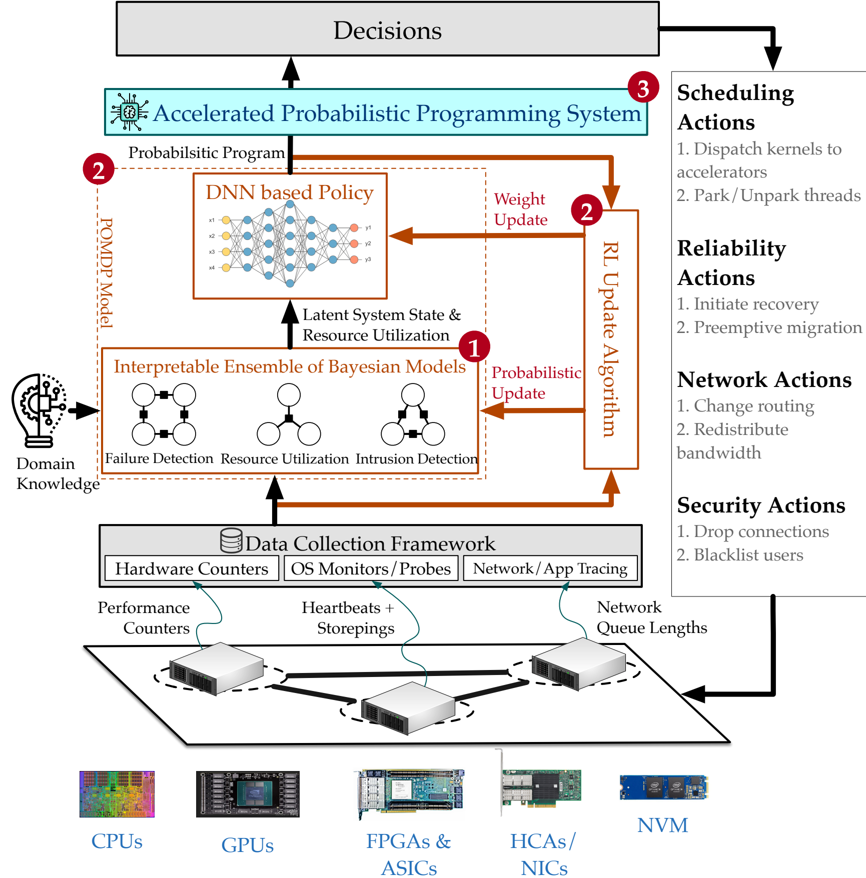


Figure 1.1: The proposed system architecture for integrating inflight analytics in networked systems.

for low-latency and low-power inference of the BayesPerf model for `x86` and `ppc64` CPUs. BayesPerf reduces the average error in HPC measurements from 40.1% to 7.6% when events are being multiplexed.

1.3.2 Learning Scheduling Policies for Heterogeneous Clusters

The challenge is to develop a decision framework that can take actions based on the estimated system state across the system stack to jointly enhance system performance, reliability, and security. The problem of scheduling of workloads onto heterogeneous processors (e.g., CPUs, GPUs, FPGAs) is of one such decision problem that is fundamental importance in modern data centers. Current system schedulers rely on application/system-specific heuristics that have to be built on a case-by-case basis. Recent work has demonstrated ML techniques for automating the heuristic search by using black-box approaches which require significant training data and time, which make them challenging to use in practice.

This thesis presents Symphony, a scheduling framework that addresses the challenge in two ways: (i) a domain-driven Bayesian reinforcement learning (RL) model for scheduling, which inherently models the resource dependencies identified from the system architecture; and (ii) a sampling-based technique to compute the gradients of a Bayesian model without performing full probabilistic inference. Together, these techniques reduce both the amount of training data and the time required to produce scheduling policies that significantly outperform black-box approaches by up to 2.2×. The Symphony system and related training procedures are labeled 2 in Fig. 1.1.

1.3.3 Accelerated and Real-Time Inference

Both training and inference for the proposed ML models are latency-critical and can negatively impact application performance, thereby significantly impacting the practical adoption of inflight analytics. They have been shown to successfully integrate structural prior information about data and effectively quantify uncertainty to enable the development of more powerful, interpretable, and efficient learning algorithms. This thesis presents AcMC^2 (labeled 1 in Fig. 1.1), a compiler that transforms deep Bayesian models into optimized hardware accelerators (for use in FPGAs or ASICs) that utilize Markov chain Monte Carlo methods to infer and query a distribution of posterior samples from the model. The compiler analyzes statistical dependencies in the PM to drive several optimizations to maximally exploit the parallelism and data locality available in the problem. We demonstrate the use of AcMC^2 to implement several learning and inference tasks on a Xilinx Virtex-7 FPGA. AcMC^2-generated accelerators provide a $47 - 100\times$ improvement in runtime performance over a 6-core IBM Power8 CPU and a $8 - 18\times$ improvement over an NVIDIA K80 GPU. This corresponds to a $753 - 1600\times$ improvement over the CPU and $248 - 463\times$ over the GPU in performance-per-watt terms. We will design a domain-specific compiler and runtime framework to automatically synthesize model-specific accelerators that support rapid, low-overhead analytics. The framework will take high-level declarative deep probabilistic programs (i.e., the POMDPs), automatically compile them onto customized accelerators, and deploy them across our customized accelerators (FPGAs and ASICs) and existing CPUs/GPUs. The accelerated probabilistic programming framework is labeled 3 in Fig. 1.1.

1.3.4 Framework for Learning Augmented Computing Systems

Modern large-scale computing systems need to solve several crucial and complex resource management problems in maintaining quality-of-service requirements while sustaining low

performance and cost overheads. Recent research has demonstrated data-driven machine learning (ML) can significantly outperform empirically and hand-tuned policies and heuristics in such problems. In this thesis, we ask a different question: what OS extensions are required to enable ML-driven control at different levels of the computing stack? To answer this question, we build IntelliKernel, a system to integrate reinforcement learning (RL) agents to build optimal resource management policies. The IntelliKernel provides functionality to (i) the RL-agent's lifecycle management, (ii) ensure safety and liveness properties for decisions made by the agents, and (iii) enable cross-agent optimization when multiple agents are deployed in a cooperative or competitive environment. We demonstrate the efficacy of the IntelliKernel in serving requests made to accelerated web-services, where, by using RL to control thread/accelerator scheduling, PCIe transaction scheduling, cache partitioning, and early request dropping, we observe a 12.3× increase in throughput of a Linux+DPDK server with a 12% improvement in tail latency.

1.4 ORGANIZATION

The remainder of the document is organized as follows. First, we describe *background and related work* in this area in Chapter 2. Then, Chapter 3 describes the models, analytics, and optimizations to deal with *noisy and incomplete telemetry data* which forms the input based on which the proposed systems can make control decisions. Chapter 4 describes a model, design, training and validation of a scheduling policy which makes used of the demonised telemetry data to dispatch data-parallel data-processing tasks to a heterogeneous cluster of processors. Chapter 5 describes methods to enable near real-time training and inference on the probabilistic and deep learning models. In particular, we demonstrate our prior work in using sampling based approximation techniques to accelerate and enable backpropagation based training of deep Bayesian ML models, and our prior work in building a high-level synthesis compiler for generating FPGA and ASIC implementations of Markov chain Monte-Carlo-based inference procedures on probabilistic programs. Chapter 4 describes how the parts of the proposed system described in Chapters 3 to 5 are brought together into a unified framework for *intelligence augmented computing systems*, that can jointly disparate high-level objectives across the computing stack. Finally, in Chapter 6, we conclude this dissertation by summarizing our contributions and discussing exciting open problems and promising research directions in this field.

CHAPTER 2: BACKGROUND, CHALLENGES AND RELATED WORK

This dissertation takes a first step towards building computer systems that can learn to efficiently optimize performance and resilience on their own through modern data-driven machine learning techniques. The implementation and deployment of such intelligent system management has the potential to make the process of programming complicated systems much easier, leading to reduced development time and greater efficiency. However, in order to build such systems, one has to tackle the two fundamental challenges of intelligent system management.

1. *Complexity*: Modern hardware and system software expose diverse configurable parameters whose complicated interactions have surprising effects on performance and reliability.

2. *Dynamics*: Computing systems must reliably adapt to unpredictable changes in operating environment, input workload, and even user needs.

In this chapter, we first discuss in §2.1 the role of heterogeneity in exasperating the two challenges described above. Then, in §2.2, we describe the opportunities for innovations in data-driven optimizations and machine learning (ML) techniques in addressing the challenge of complexity and dynamics. We reflect on the related work and describe the research gap addressed in this dissertation. We briefly introduce several multidisciplinary concepts with the intention of making this work accessible to readers from multiple disciplines.

2.1 COMPLEXITY & DYNAMICS: THE ADVENT OF HETEROGENEITY

The unprecedented scale and complexity of today's datacenter applications present tremendous challenges to the design of systems infrastructure. A primary driver for this increasing complexity has been the advent of statistical- and machine-learning applications. Traditionally homogeneous datacenters are progressively shifting to heterogeneous designs to accommodate the needs of these emerging applications. For example, GPGPU, special purpose chips/ASICs, like Google's TPUs [3], reconfigurable fabrics using FPGAs, like Microsoft's Catapult [4], or a combination of the two are becoming increasingly common. Performance variability has been widely reported in such contexts from hardware-induced variability [5], to stragglers in batch analytics [6], variability in VM network performance [7, 8], and tail request latencies in microservices [9].

Below we discuss the reasons behind this increasing heterogeneity, and how its increased prevalence is handled by the state of the art systems. We define the shortcomings of these systems and how how our approach deals with these shortcomings.

Homogeneous Datacenters. Datacenters have achieved their prevalence in large enterprise and high-performance computing settings by offering resource flexibility and cost efficiency. Cost efficiency, specifically, comes from leveraging the economies of scale of buying thousands of the same type of servers. The benefits of this homogeneity, however, go beyond cost efficiency. Managing a homogeneous system is also much easier from the perspective of the operating system, the cluster manager (the datacenter-wide resource manager/scheduler), the compiler, and the application design itself. Further, large scale application deployment also becomes much simpler when all servers look the same, and the main placement decision has to do with resource availability.

Application Demands & Hardware Heterogeneity. This increasing popularity of large datacenters has coincided with the slowdown of technology scaling (i.e., end of Moore's law), and the significant increase in on-chip power densities (i.e., end of Dennard scaling) [10]. Hence emerging applications which require significantly more compute have required exploration of special-purpose hardware designs, to maintain the same power and/or cost trade-offs. The specific implementations of such processors has varied across applications depending on the maturity of the target application (i.e., lack of algorithmic churn), the user's emphasis on programmability, and the cost of fabricating a new chip. ASICs work better for stable applications, whose core computation does not fundamentally change and the chip fabrication cost is tolerable, while FPGAs are better suited for applications where accommodating algorithm/logic changes is important or where the overheads of chip fabrication may be undesirable. For example, the advent of large-scale genomic datasets and their enormous applicability to drive research breakthroughs in the clinical setting [11] has led to proliferation of custom accelerators to quickly, efficiently, and cheaply process this data (e.g., [12, 13, 14, 15, 16, 17, 18]). In this dissertation, we use large scale genome processing (and the custom accelerators used therein) as an benchmark application to validate our results (see **??**).

Heterogeneity in Application Architecture. Further, the (software) architecture of datacenter applications has also changed. In place of large monolithic services, modern applications have increasingly adopted fine-grained, modular designs with thousands of short-lived functions, each of which has strict latency requirements (often in microseconds) [19]. Missing these requirements does not only affect the offending function itself, but, can cause cascading performance issues across a cluster [20, 21]. By default, traditional general-purpose servers are not designed to achieve such low and predictable latency. Instead, they are

designed for compatibility to any application a user may launch on them thereby leading to unpredictable performance jitters.

Complexity and Dynamics. As a result of the increased heterogeneity in the hardware and software architectures of emerging datacenters, these systems becoming more complicated. The increasing diversity of hardware and software configurations has led to an increase in the number of configuration parameters for these components. Further, interactions between changes in such configuration parameters becomes incredibly complex, and modeling those interactions is increasingly difficult. Even a few different resources can present a large search space which is infeasible to explore exhaustively at run-time. This has not only resulted in reduced performance, but also in reliability issues like hard-to-replicate bugs [22, 23, 24, 25]. At the same time, evolving computing systems must also reliably adapt system dynamics, i.e., to unpredictable changes in operating environment, input workload, and even user needs. Such changes arise out of the need to efficiently run an ever changing mix of traditional and emerging applications (as highlighted above), or due to the emergence of new classes of applications that in turn drives heterogeneity and complexity [26]. In the context of efficient and intelligent system policies, this implies that policies must not be set in stone after they have been designed, rather, be constructed to evolve over time to capture important aspects of the system dynamics. Hence *complexity* and *dynamics* has deep implications across the system stack. Below we discuss a few of them in the context of this dissertation.

1. *Datacenter Resource Management:* Despite the performance and power benefits of heterogeneity, when it comes to resource management in modern datacenters, it can also introduce complexity. Where previously every server could be expected to yield similar levels of performance, as heterogeneity becomes more prevalent, the range of performance and power behaviors across hardware platforms becomes more diverse. OS and cluster schedulers need to be aware of the different profiles of heterogeneous machines when allocating resources to applications, to avoid exacerbating performance unpredictability, especially in latency-critical services. Today, system schedulers generally rely on application- and system-specific heuristics with extensive domain-expert driven tuning of scheduling policies, e.g., [27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42]. Such heuristics are difficult to generate as variation across applications and system configurations lead to significant time and money being spent in painstaking heuristic search. That is, these techniques do not satisfactorily handle complexity nor dynamics. Recent work has demonstrated ML techniques [39, 40, 41, 42] for automating heuristic searches by using black-box approaches which require significant training data and time, making them challenging to use in practice. These ML-based systems are able to

handle complexity to a degree, but are not able to handle dynamics as they require enormous amounts of training data and time to achieve their goals. In this dissertation (in Chapter 4) we will show how complexity and dynamics can be jointly addressed using hybrid Bayesian-deep reinforcement learning (RL) agents that are trained end-to-end using back-propagation.

2. *Programmability:* The lack of programmability in hardware accelerators has been one of the main roadblocks towards a more widespread adoption of heterogeneous platforms. So far, programming accelerators has been limited to expert developers, with deep understanding of both the application and hardware platform. There have been efforts to provide higher-level constructs for that can be used to declare and annotate the parallelism available in a program thereby lowering the expertise required to build accelerators (e.g., [43, 44, 45, 46, 47]). However, these innovations are still not for the end-user, they are meant to increase productivity of the previously mentioned expert developers. The techniques and systems presented in this dissertation creates new class of user (akin to, but not the same as the system operator) of the system: one who collects telemetry data, experiments and learns/constructs system policies using the RL agents described above. This new user has stringent latency constraints and often requires access to the hardware accelerators in order to execute and train their system policies. This dissertation demonstrates (in Chapters 3 and 5) the design and implementation of high-level synthesis compilers for the generation of efficient hardware accelerators for the RL agents described above.

2.2 MACHINE LEARNING FOR SYSTEMS

One promising way forward is abstracting away the complexity that hardware heterogeneity introduces by leveraging automated, data-driven techniques in the design and management of future datacenter systems. Computer systems are full of sequential decision-making tasks that can naturally be expressed as bandits or Markov decision processes (MDP) and hence RL. ML has long been used in areas such as branch prediction [48]. Recent work has shown promising results of using ML in automated hardware design [26, 49, 50, 51, 52], code generation [53, 54, 55], network-level QoS and congestion management [56, 57, 58, 59], database indexing and querying [60, 61, 62, 63], distributed scheduling [39, 40, 41, 56], and software engineering [21, 64, 65] among many others. This has become possible because (i) advances in tracing/monitoring systems have increased the availability of large datasets, and (ii) recent surge in ML model research has improved their quality and practicality.

However, it is important to note that the use of ML for systems is not a departure from traditional systems research. Rather, ML only provides a new set of automated tools for systems research. One interpretation is that systems researchers have used a "real-life version of gradient descent" to move the system to a local optimum, not unlike what ML would do.

From a ML perspective, computer systems present many challenging problems. The landscape of decision-making problems in systems is vast, ranging from centralized control problems (e.g., a scheduling agent responsible for an entire computer cluster) to distributed multi-agent problems where multiple entities with partial information collaborate to optimize system performance (e.g., network congestion control with multiple connections sharing bottleneck links). Further, the control tasks manifest at a variety of timescales, from fast, reactive control systems with sub-second response-time requirements (e.g., admission/eviction algorithms for caching objects in memory) to longer term planning problems that consider a wide range of signals to make decisions (e.g., VM allocation/placement in cloud computing).

Pitfalls In Applying ML to Systems. The successes described above often hide the fact that ML does not always lead to the immediate wins that its popularity promises. This aspect is often obscured in the discussion of ML for Systems, in part due to the popularity of end-to-end learning. Modern ML techniques can learn complex behavior, and it is therefore possible to train models that learn complex policies end-to-end. While these approaches often work, they are

1. not always data efficient,
2. consume large amounts of resources, and
3. sometimes do not conclusively outperform strong baselines.

Further, there is an operational *trade-off is often between accuracy and interpretability*, i.e., at the end of training we are left with large deep learning models that produce good results, but it is difficult to elucidate their internal mechanisms. As a result, these models can fail in unintuitive and embarrassing ways, or worse, *fail silently*. As a result, *the models require significant safety harnesses.*

Many systems problems have a known structure. However, end-to-end learning has to learn this structure from scratch and may re-learn known facts, at the cost of maximizing performance on the otherwise intractable part. In this dissertation, we therefore argue that effectively applying ML to systems requires

1. incorporating our knowledge of the systems' hardware and software architecture into the ML model as an inductive bias to prevent it from having to re-learn known details about a system (i.e., well-understood models and rules of execution).

2. intelligently identifying which part of a systems policy requires ML, and developing specific ML techniques for this part.

Hence, this dissertation advocates the use of ML to solve fixed, well-defined and semantically self-contained problems as opposed to build systems with end-to-end learned policies. We demonstrate that such models and rules of execution can be encoded into the proposed Bayesian deep learning-based RL agents, in essence making them *white-box* models that are sample-efficient, scalable, interpretable (debuggable), transferable between deployments, and generalizable to large numbers of workloads.

Resource Constraints for Learned Policies. Another key challenge in evaluating the the benefit of learned policies is that even if ML improves a particular performance metric, its resource cost does not always justify the improvement in production. For example, OSs and runtimes often need to make decisions in micro/nanoseconds. In contrast, even small neural networks often take hundreds of microseconds. Further, models can often also can consume significant resources like the memory required to store model parameters, and as a result significantly affect application performance. A common pitfall in other related work, is that the proposed ML models are trained, tested and validated offline, or in simulators (e.g., [66]). As a result, these techniques often fail to consider the real-time nature of the problems and hence fail to provide benefits in production deployments [67].

This dissertation addresses the above challenge in a fundamental way, starting with mathematical models and ending with real software and hardware (i.e., that is deployed in situ) that provides efficient, scalable, and composable system management solutions. In particular, we address the latency criticality of the proposed Bayesian-deep RL agents by providing algorithmic approximations for inference using sampling techniques (in Chapter 4), and by building custom accelerators to rapidly sampling the models for both training and inference (in Chapter 5). Further, in **??**, we show how these custom accelerators can be pooled/multiplexed between multiple machines (at the rack-scale) to mitigate the cost of deploying a ML for systems accelerator across an entire fleet of servers in a datacenter.

Challenges in Applying RL to Systems Policies. Computer systems give rise to new challenges for learning algorithms that are not common in other domains. For example, time-varying state or action spaces (e.g., dynamically varying number of jobs and machines in a computer cluster), structured data sources (e.g., graphs to represent data flow of jobs). Such complexities often prevent off-the-shelf RL methods from achieving strong performance in different computer system problems. Here, we primarily focus on the common challenges that arise across many systems in different stages of the RL design pipeline.

1. *Needle-in-the-haystack Problem:* In some computer systems, the majority of the state-

action space presents little difference in reward feedback for exploration. This provides no meaningful gradient during RL training, especially in the beginning, when policies are randomly initialized. Random exploration is not effective at learning here because any random action can easily overshadow several good actions, making it difficult to distinguish good action sequences from bad ones. Here using domain-knowledge to confine the search space helps to train a strong policy. In this dissertation we use the Bayesian component of the proposed RL-agents to incorporate this domain knowledge: effectively reducing the probability of taking certain random steps thereby confining the search space.

2. *State-Action Space Encoding:* In some systems, the action space grows exponentially large as the problem size increases, e.g., in scheduling to map tasks to processors. In this case, the number of possible mappings and thus, actions increases with the number of new tasks in the system. Encoding such a large action space is challenging and makes it hard to use off-the-shelf RL agents. Here, using domain specific representations that capture inherent structure in the state-action space can significantly improve training efficiency and generalization. In this dissertation we deal with such issues leveraging lower dimensional graph embeddings and Graph network layers to significantly improve generalization [68].

CHAPTER 3: DEALING WITH NOISY TELEMETRY DATA

3.1 INTRODUCTION

Hardware performance counters (HPCs) are widely used in profiling applications to characterize and find bottlenecks in application performance. Even though HPCs can count hundreds of different types of architectural and microarchitectural events, they are limited because those events are collected (i.e., multiplexed) on a fixed number of hardware registers (usually 4–10 per core). As a result, they are error prone because of application, sampling, and asynchronous collection behaviors borne out of multiplexing. Such behavior in HPC measurements is not a new problem, and has been known for the better part of a decade [69, 70, 71, 72, 73, 74, 75].

Targeted Need. Traditional approaches of tackling HPC errors have relied on collecting measurements across several application runs, and then performing offline computations to (i) impute missing or errored measurements with new values (e.g., [71]); or (ii) dropping outlier values to reduce overall error (e.g., [74]). Both of these require time and compute resources for collecting training data and inference, thus are suitable for offline analysis (like profiling). These techniques are untenable in emergent applications that use HPCs as inputs to complete a feedback loop and make dynamic real-time decisions that affect system resources using a variety of machine learning (ML) methods. Examples include online performance hotspot identification (e.g., [21]), userspace or runtime-level scheduling (e.g., [29, 39, 70, 76, 77]), and power and energy management (e.g., [78, 79, 80, 81]), as well as attack detectors and system integrity monitors [82]. In such cases, the HPC measurement errors propagate, get exaggerated, and can lead to *longer training time* and *poor decision quality* (as illustrated in §3.6.3). This is not surprising because ML systems are known to be sensitive to small changes in their inputs (e.g., in adversarial ML) [83, 84, 85]. As we will show in §3.2, HPC measurement errors can be large (as much as 58%); hence they must be explicitly handled.

This chapter presents BayesPerf, a system for quantifying uncertainty and correcting errors in HPC measurements using a domain-driven Bayesian model that captures microarchitectural relationships between HPCs. BayesPerf corrects HPC measurement errors at the system (i.e., CPU and OS) level, thereby allowing the down-stream control and decisions models that use HPCs to be simpler, faster and use less training data (if used with ML). The proposed model is based on the insight that even though individual HPC measurements might be in error, groups of different HPC measurements that are related to one

another can be jointly considered—to reduce the measurement errors—using the underlying statistical relationships between the HPC measurements. We derive such relationships by using design and implementation knowledge of the microarchitectural resources provided by CPU vendors [86, 87]. For example, the number of LLC misses, the size of DMA transactions, and the DRAM bandwidth utilization are related quantities,[1] and can be used to reduce measurement errors in each other.

Approach & Contributions. The key contributions are:

1. *The BayesPerf ML Model.* We present a probabilistic ML model that incorporates microarchitectural relationships to combine measurements from several noisy HPCs to infer their true values, as well as quantify the uncertainty in the inferred value due to noise. Hence allowing:
   (a) improving decision-making with explicit quantification of HPC measurement uncertainty.
   (b) reduced need for aggressive (high-frequency) HPC sampling (which negatively impacts application performance) to capture high-fidelity measurements, thereby increasing our observability into the system.

2. *The BayesPerf Accelerator.* To enable the use of BayesPerf ML model in latency-critical, real-time decision-making tasks, this chapter presents the design and implementation of an accelerator for Monte Carlo-based training and inference of the BayesPerf model. The accelerator exploits
   (a) high-throughput random-number generators.
   (b) maximal parallelism based on the statistical relationships mentioned above, to rapidly sample multiple parts of the BayesPerf model in parallel.

3. *A Prototype Implementation.* We describe an FPGA-based prototype implementation of the BayesPerf system (on a Xilinx Virtex 7 FPGA) for Linux running on Intel x86_64 (Sky Lake) and IBM ppc64 (Power9) processors. The BayesPerf system is designed to provide API-compatibility with Linux's `perf` subsystem [88], allowing it to be used by *any* userspace performance monitoring tool for both `x86_64` and `ppc64` systems. Our experiments demonstrated that BayesPerf reduces the average error in HPC measurements from 40.1% to 7.6% when events are being multiplexed, which is an overall 5.28× error reduction. Further, the BayesPerf accelerator provides an 11.8×

[1] In a simple processor, DRAM Bandwidth = (LLC misses × Cache line size+ # DMA Transactions × Transaction size)/Clocks.

reduction in power consumption, while adding less than 2% read latency overhead over native HPC sampling.

4. *Increasing training and model efficiency of decision-making tasks.* We demonstrate the generality of the BayesPerf system by integrating it with a high-level ML-based IO scheduler that controls transfers over a PCIe interconnect. We observed that the training time for the scheduler was reduced by 37% ($\sim$ 52 hr reduction) and the average makespan of scheduled workloads decreased by 19%.

The remainder of the chapter is organized as follows. First in §3.2, we discuss the sources of HPC measurement errors. Then in §3.3 we provide an overview of the design of the BayesPerf system. §3.4 describes the formulation, training and inference of the ML model used to correct errors. §3.5 describes the accelerator that allows inference on the ML model in real-time. Then in §3.6 we discuss a prototype implementation and it's evaluation. Finally, in §3.7 and §3.8, we put BayesPerf in perspective of traditional methods, and describe future challenges, respectively.

3.2 BACKGROUND: HPC ERRORS

Every modern processor has a logical unit called the Performance Monitoring Unit (PMU), which consists of a set of HPCs. An HPC counts how many times a certain event occurs during a time interval of a program's execution. The number and configurability of the HPCs vary across processor vendors and microarchitectures. For example, modern Intel processors have three fixed HPCs (which measure ISA-related events) and eight programmable HPCs per core (which measure microarchitectural events and are split between the SMT threads on the core) [89]. The events measured by an HPC are vendor-specific and microarchitecture-dependent, and vary with processor models within the same microarchitecture. For example, an Intel Haswell CPU has 400 programmable events, compared to the 1623 events on a HaswellX CPU; both have the same number of HPC registers per core (three + eight) [70]. Therefore, one must carefully pick and configure which events to monitor with the available registers.

Reading HPCs. Performance counters can be read using:

1. *Polling:* The HPCs can be read at any instant by using specific instructions to write (to configure the HPC) and read (to poll the value of an HPC) model-specific registers (MSRs) that represent HPCs. For example, `x86_64` uses specific instructions to read (i.e., `rdmsr`) from and write (i.e., `wrmsr`) to MSRs, respectively; both instructions

require OS-level access privilege, and hence are performed by the OS on behalf of a user. Here, one HPC is programmed to count only one event during the execution of a program. Hence, polling is ineffective, as the number of events that can be simultaneously measured is limited by the number of available hardware registers.

2. *Sampling:* HPCs also support sampling of counters based on the occurrence of events, thereby letting multiple events timeshare a single HPC [72, 90]. This feature is enabled through a specific interrupt, called the Performance Monitoring Interrupt (PMI), which can be generated after the occurrence of a certain number of events (i.e., a predetermined threshold). The interrupt handler then polls (i.e., samples) the HPC. The multiplexing of events occurs through a separate scheduling interrupt that is triggered periodically to change the configuration of the HPCs and swap events in and out. The collected measurements are generally scaled to account for the time they were not scheduled to a HPC [73], and that can lead to making erroneous measurements. Sampling is necessary due to the severe disparity between the numbers of events types and the number of counters.

Sources of Errors. In addition to the errors due to event multiplexing, HPCs demonstrate other modalities of measurement error. For example, HPC measurements can vary across runs because of OS activity, scheduling of programs in multitasking environments, memory-layout, and memory-pressure, and varied multi-processor interactions may change between different runs. Nondeterminism in OS behavior (e.g., servicing of hardware interrupts) also plays a significant role in HPC measurement errors [75]. Performance counters have also been shown to over count certain events on some processors [75]. Finally, the implementation of userspace and OS-kernel-level tools can cause different tools to provide different measurements for the same HPCs in strictly controlled environments for the same application. The variations in measurements may result from the techniques involved in acquiring them, e.g., the point at which they start the counters, the reading technique (polling or sampling), the measurement level (thread, process, core, multiple cores), and the noise-filtering approach used.

Measurement Errors. As a result of this non-determinism, quantifying error in HPCs is difficult as there is no way to get "ground truth" measurements because of inherent variations in measurements. In this chapter, we define HPC error as magnitude of difference between corresponding HPC measurements made in two runs of a workload, one in polling and other in sampling mode. The correspondence between the two HPC traces (time-series) is established by *dynamic time warping* [91] that calculates an "alignment" between the two time series datasets using edit-distance.[2]

[2]This definition of error is based on prior work on HPC errors [74].

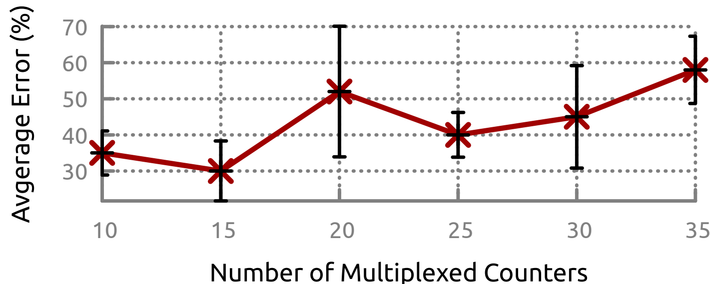


Figure 3.1: Errors due to event multiplexing in HPC measurements across ten application runs.

Fig. 3.1 illustrates the net effect of measurement errors on the fidelity of an HPC counter using Linux's `perf` subsystem. In this case, the baseline dataset is collected using polling, and the target dataset is collected using sampling, each on 10 independent application runs capturing both variations in a single run, and variations across runs. We observe a $58 \pm 9.3\%$ average error in HPC measurements when 35 on-core events are being multiplexed on an Intel processor, compared to the baseline of polling 4 events at a time.[3]

Errors in Derived Events. Such high error is particularly troubling, as it is quite conceivable to count 35 events simultaneously, particularly for measuring *derived events*. Derived events are obtained by combining individual HPC measurements in a mathematical expression. Consider for example, the "`Backend_Bound_SMT`" derived event on Intel BroadwellX processor. It measures the fraction of µops issue slots utilized in a core, and alone takes measurements from 16 HPCs to compute [86]. This information might be valuable in a OS-level scheduler that controls an SMT processor, with the objective of minimizing interference between CPU-bound processes/threads. Often such information would be conflated with other derived metrics like "`Memory_Bound`" and "`Frontend_Bound_SMT`", which together would require the use of 29 unique counters. That according to Fig. 3.1 would incur an average error of $\sim 45\%$. This is further exasperated by the fact that the HPCs need to be counted per-SMT thread, per-core, and per-socket. For example, in an average 2-socket server system this would imply collecting thousands of counters (i.e., 2784 HPCs = 29 counters $\times$ 24 cores $\times$ 2 sockets).

Adding More Registers? A relevant question to ask is whether the HPC-error problem will disappear if more HPC registers are added into future CPUs. The short answer is that it will not, because as we continue to add more monitors, the system complexity increases which is untenable in commercial CPUs that are often driven by other practical considerations. Hence, HPC counters will eventually always end up introducing the sampling-based error.

[3]The experimental setup is described in detail in §3.6.1.

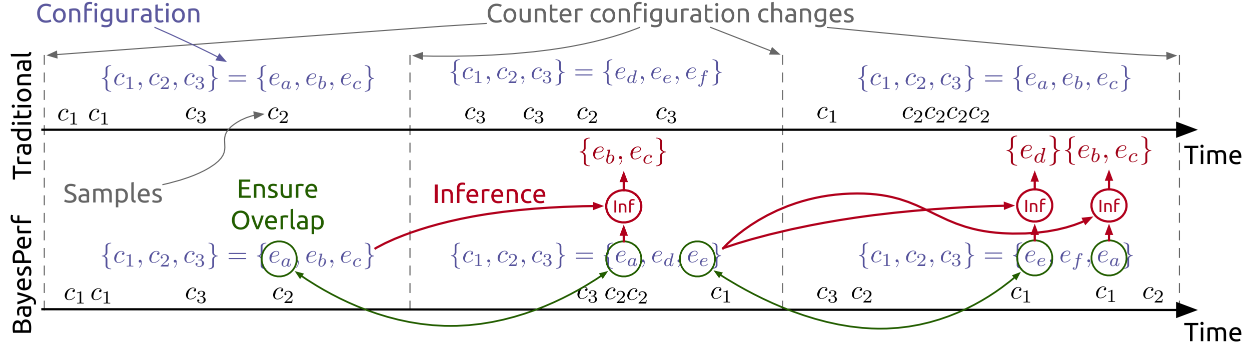


Figure 3.2: Overview of the BayesPerf ML model.

3.3 APPROACH OVERVIEW

Key Insight. The key insight that drives this work is that microarchitectural invariants (e.g. [86, 87, 92]) can be applied to measured HPC data to estimate whether it is, in fact, in error (i.e., a detector). Further, we can quantify the "uncertainty" of an HPC measurement by quantifying the probability of deviation from that invariant (i.e., its egregiousness). When the above is applied to a group of HPC measurements, each targeting different microarchitectural units, the underlying invariants can be composed, encoded as statistical relationships, i.e., joint probability distributions, which can then be composed into larger *probabilistic graphical models*. We then use a Bayesian inference approach to integrate the data and prior knowledge of the system to effectively attenuate the high error measurements and significantly amplify correct measurements, all in real-time. This works in practice as the number of HPCs with lower errors are generally more numerous than those with higher error (also verified by our observations), hence they bias the aggregate results to the lower errored values. As a result, BayesPerf significantly outperforms traditional purely data-driven statistical approaches for outlier detection.

BayesPerf ML Model. Below, we provide a high-level description of the model, using the example illustrated in Fig. 3.2. In this example, the goal is to measure (by multiplexing) a set of events $\{e_a, \ldots, e_f\}$, on a set of HPCs $\{c_1, c_2, c_3\}$.

Deciding Schedules of HPCs: BayesPerf first determines a schedule of how the events are multiplexed on the HPCs. The schedule consists of a set of *HPC configurations* that are collected over time. We define an HPC configuration as a mapping between counters and events, that defines which counters are collected at an instant of time. The notation $\{c_1, c_2, c_3\} = \{e_a, e_b, e_c\}$ is used to define such a configuration, and imply that c_1 counts e_a. The scheduling process is driven primarily by the microarchitectural considerations of the available HPCs and the types of events that each one can measure, i.e., as not all HPCs

can measure all events. Traditional HPC measurement tools, like the Linux `perf` subsystem trigger HPC configuration changes in a round-robin manner, based on a periodic hardware timer-driven interrupt (see Fig. 3.2). BayesPerf uses a similar interrupt driven approach, but does not use round-robin to build a schedule of configurations. *It creates configurations of overlapping counters, such that each set of counters have "statistical relationships" to other events in preceding and subsequently scheduled configurations.* For example, in Fig. 3.2, e_a and e_e are such overlapping events. As we will show in §3.4, these "statistical relationships" can be derived based on microarchitectural invariants (i.e., domain knowledge) that tie together the resources underlying the measurements. BayesPerf encodes those invariants as generative *joint-* and *conditional-probability distributions* for the processors used in our experiments.

Inferring Unscheduled Events: At each instant of time, BayesPerf then uses sampled data from the overlapping events to compute a full posterior distribution (i.e., the likely values and their associated uncertainties) of the unscheduled events using a Bayesian inference approach. Consider e_b in the second time slice of Fig. 3.2. It is calculated using its' own samples from the previous time slice and the samples of e_a (which is the event repeated across time slice one and two) in the current time slice. The result of the Bayesian inference using the sampled data is a probability distribution $\Pr(e_b^t|e_b^{t-1}, e_a^t)$ at time t; this distribution not only gives us an estimate of e_b (i.e., by finding the most likely value of e_b under the distribution), but also quantifies uncertainty (i.e., using the probability value $\Pr(e_b|\dots)$) in that estimate. The compositional nature of Bayesian inference allows chain events across multiple time slices, if the overall set of events to be measured is large, albeit at the cost of larger uncertainty in the estimate. For example, in Fig. 3.2 the chain of events $(e_b \rightarrow e_a) \rightsquigarrow (e_a \rightarrow e_e) \rightsquigarrow (e_e \rightarrow e_d)$ can be used directly estimate e_b from samples of e_a, but also transitively estimate it from samples of e_e. Here "$\rightarrow$" describes the above statistical relationships between events in a configuration (i.e., in a single time slice), and "$\rightsquigarrow$" describes data collected between overlapping events across time slices.

The BayesPerf system then allows an user to poll the posterior probability distributions of any of the events being collected. These distributions can be passed along (i.e., integrated) into higher-level ML/control frameworks or used directly to compute error bounds of HPC measurements.

BayesPerf Accelerator. Though the BayesPerf ML model is able to provide significantly higher-quality samples from the raw HPC measurements, it introduces the additional runtime overhead of performing Bayesian inference on every new measurement polled by the user. Consider Fig. 3.3; it shows the average overhead (over 100 reads) of reading a HPC value using the Linux kernel's (perf subsystem) `read()` system call (i.e., polling), the x86_64 `rdpmc` instruction to read HPCs in userspace, a purely CPU implementation of the BayesPerf ML

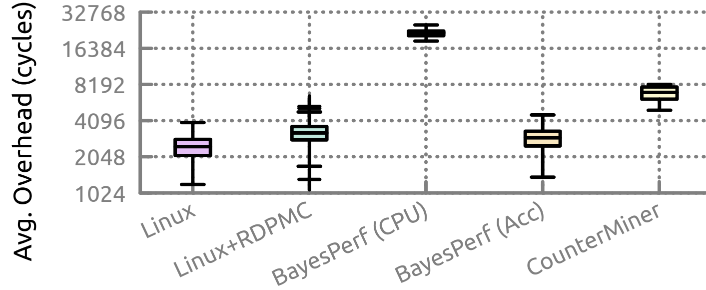


Figure 3.3: Latency overhead of reading counters with BayesPerf compared to traditional methods on an x86 CPU.

model (using TensorFlow Probability [93, 94]), an FPGA accelerated version of BayesPerf (described later in §3.5), and CounterMiner[74] (described later in §3.6 and used as a baseline in our evaluation). We observe that a single HPC read when the CPU implementation of BayesPerf is being used has approximately 9× longer latency than native polling of the HPC. In order to reduce the latency, we introduce an accelerator that parallelizes the process of computing posterior inference on the BayesPerf ML model. The accelerator largely builds upon our prior work [95] in building MCMC accelerators that treats a lack of statistical dependencies between variables as a scope for parallel execution. Using the accelerator, BayesPerf adds less than 2% overhead in read latency compared to the native solution. Our implementation of the accelerator on a PCIe-attached FPGA device can take advantage of modern cache-coherent accelerator-processor communication protocols like CAPI [96], and essentially provide users with the same interface and same performance characteristics they could get if they were natively polling the OS for HPC measurements.

3.4 THE BAYESPERF ML MODEL

In this section we first discuss formalization of the HPCs and events for a generic CPU. Then, in §3.4.1, we discuss the problem of scheduling sets of performance counters onto available HPCs. Finally, in §3.4.3, we discuss an inference strategy to compute the posterior distribution of a single event based on generated schedule and HPC measurement samples.

Formalism. We assume that every processor has a pre-determined number of fixed and programmable HPCs. We refer to them as n_f and n_p, respectively. The HPCs themselves are indexed and referred to as $f_1 \dots f_{n_f}$ for the fixed HPCs and $c_1 \dots c_{n_p}$ for the programmable HPCs. The processor as a whole has a set $E = \{e_1, \dots, e_{n_e}\}$ of n_e architectural and microarchitectural events that are measured using f_* and c_*. At any point in time, the programmable HPCs are configured to count any one of the events in E. The instantaneous

mapping between counters and events is called a *configuration*. Fixed HPCs are not considered in a configuration, as they cannot be programmed. Not all programmable HPCs will be able to count all events (i.e., all configurations might not be valid), depending on microarchitectural and implementation considerations. For example, an Intel off-core response event requires one HPC and one MSR register, and the `L1D_PEND_MISS.PENDING` event can be only counted on the third HPC on Haswell/Broadwell systems. Configuration validity constraints are known ahead of time, can be dynamically checked, and must always be satisfied. BayesPerf uses the Linux's builtin validity checker.

A sample s_j is generated from an HPC c_i (i.e., an interrupt is fired to read the value of a counter and store it in memory) when a particular threshold $\tau_{i,k}$ is reached on one of the fixed HPCs f_k.[4] That process is denoted by $s_j \sim c_i$ if $f_k \geq \tau_{i,k}$. In addition to the value of the counter, the sampling process also records two time measurements, t_r^i and t_e^i, where $t_e^i \leq t_r^i$. They correspond to the total time the application has been running, and the total time for which an event has been sampled (i.e., it has been enabled), respectively. Traditional approaches (e.g., one that is used in Linux) use these times to correct HPC undercounting errors and assume that the true value of a performance counter is scaled according to $s_j \mapsto s_j \times {}^{t_r^i}/_{t_e^i}$.

Statistical Dependencies. Some subsets of events in E have statistical relationships between them. Those statistical relationships are described by *joint probability distribution functions*. For example, if e_1 and e_2 share such a relationship, then it is represented by their joint probability distribution $\Pr(e_1, e_2; \Theta)$. Where, Θ refers to all tunable or learnable parameters of the distribution.

We assert that if nothing is known about the statistical relationships between the events, then $\Pr(e_i, \dots)$ can be approximated by a neural network and trained using data from HPCs. However, for most real systems, knowledge about the underlying microarchitectural resources being counted in a HPC can be correlated together to describe $\Pr(e_i, \dots)$. To do so, we use algebraic models of the composition of HPC measurements by using information about the CPU microarchitecture found in processor performance manuals [86, 87, 97]. For example, in an Intel x86 Sandy Bridge microarchitecture [97, 98], the fraction of cycles a CPU is stalled because of DRAM access is given by $(1 - \texttt{Mem_L3_Hit_Frac}) \times {}^{\texttt{STALLS_L2_PENDING}}/_{\texttt{CLKS}}$. Those stalls can be caused by either DRAM bandwidth issues or DRAM latency issues, which in turn can be measured as ${}^{\texttt{ORO_DRD_BW_Cycles}}/_{\texttt{CLKS}}$, and ${}^{\texttt{ORO_DRD_Any_Cycles}}/_{\texttt{CLKS}} - {}^{\texttt{ORO_DRD_BW_Cycles}}/_{\texttt{CLKS}}$ respectively. Here, `ORO_DRD_Any_Cycles`, `ORO_DRD_BW_Cycles`, `Mem_L3_Hit_Frac`, `STALLS_L2_PENDING`, and `CLKS` correspond to a set of fixed and programmable events, which are related to each other

[4] In general, this triggering event occurs based on the number of clock cycles or number of instructions executed.

via the algebraic relations described above. Given the equivalence of those three computed quantities, we can compute one, given values of the other. When some of these events are reported with measurement errors, the equivalence relationship becomes statistical (i.e., capture randomness because of errors). We then define a distribution function for individual events, where only valid combinations of the event values have a non zero probability of occurrence.

3.4.1 Scheduling

Problem. Given statistical dependencies between events, we need to ensure that the configurations created for two consecutive time slices (i.e., scheduler quanta) have at least one overlapping event in order to establish either a first-order or a transitive statistical relationship between consecutive time slices. For example, if we have four events e_1 to e_4 that are related by $f(e_1, e_2)$ and $g(e_2, e_3, e_4)$, we must ensure that samples of e_2 occur repeatedly across multiple time slices. Given (from a profiling application) an original schedule of configurations $C_1 \to C_2 \to \cdots \to C_n$, where C_i executes in time slice i, into another schedule of C_i's such that transitive statistical relationships hold, such that the validity criteria holds on each C_i'. In the case when it is not possible ensure the validity criteria on every C_i', we break the chain of repeated events, and start over again from a valid configuration.

Solution. The first step of the scheduling process is to aggregate all the statistical dependencies available for the processor in question into a graphical structure. The graph is produced by expanding the scheduled chain $C_1 \to \cdots \to C_n$ using the statistical relationships between the events in the chain. In the ML/statistics community, such a graph commonly referred to as *probabilistic graphical model*, and more specifically identified as a *factor graph* (FG) [99]. Remember from above that the statistical dependencies between the events are specified as joint probability functions $\Pr(S_i)$,[5] where $S_i \subseteq E$. Using those functions, we generate a bipartite FG $G = (E \cup \{\Pr_1, \ldots \Pr_n\}, \{(e, \Pr_i) | e \in S_i \forall i\})$. The FG represents the joint distribution of *all* the events in the schedule, composed together from every individual joint distribution.

Now, given the FG and two consecutive configurations from a schedule C_t and C_{t+1} (with events $E_t, E_{t+1} \subseteq E$ respectively), our scheduling problem reduces to (i) finding whether E_t and E_{t+1} share an event such that the transitive statistical dependency is met; and (ii) if they do not share such a dependency, producing the shortest sequence of C_*' such that $C_t \to C_{(1)}' \to \cdots \to C_{t+1}$. Solution of the first of the two problems is straightforward. We do it by computing the *Markov blanket* [99] of the sets E_t and E_{t+1} under the factor graph.

[5] We use the shorthand $\Pr_i = \Pr(S_i)$.

The Markov blanket B_{x_i} of a variable x_i in the factor graph defines a subset of $x_{\neg i}$ such that x_i is conditionally independent of $x_{\neg i}$ given B_{x_i}. If the Markov blankets of E_t and E_{t+1} overlap (i.e., $B_{E_t} \cap B_{E_{t+1}} \neq \varnothing$), then we are guaranteed that there exists at least one event that shares transitive dependencies between the time slices. The second problem is a little more involved. It can be solved by finding the shortest path (assuming unit cost for each edge traversed) from each $e \in E_t$ to each $e' \in E_{t+1}$ in the FG. That can be accomplished using Djikstra's algorithm, checking validity of the path at every step. In addition to the graph traversal, one must also apply the following optimizations to prune unnecessary C'_*s.

1. *Removing Common Steps:* If an intermediate step C'_i exists such that the Markov blankets $B_{e_1}, B_{e_2}, \ldots B_{e_n}$ of events $e_1, e_2, \ldots e_n$ overlap, the next transition state of the schedule can be condensed. That is, if there exists an $e_* \in B_{e_1} \cap \ldots B_{e_n}$, then composition of statistical relationships can happen through e_*, instead of through the larger set of events, i.e., $C'_{i+1} \mapsto (C'_{i+1} \setminus \{e_1, \ldots e_n\}) \cup e_*$

2. *Removing Redundant Steps:* If there exists two steps C'_i and C'_{i+1} such that there is no change in the Markov blanket (i.e., $B_{E_i} = B_{E_{i+1}}$), then we can skip the transition C'_{i+1} and instead transition to C'_{i+2}. That situation can occur because the Markov blankets in individual traversals $e \rightsquigarrow e'$ will change at every step; however, the union of all such blankets might not change. If it does not change, we have enough statistical information to skip the $i + 1^{\text{th}}$ step and go directly to $i + 2$.

Checking Validity of the Configuration. A key challenge in determining a valid transformation of a schedule is that of identifying the configurations that do not satisfy the microarchitectural constraints placed on HPCs. We check the validity of a new schedule using Linux's perf_event subsystem. It allows us to iterate over all HPCs in a configuration until it reaches an event that it fails to schedule, thereafter notifying the user of validity failure. To maximize the use of available counters, the perf iteration strategy starts with the most constrained events and goes to the least constrained events in a configuration. Linux's native scheduling for a group of events happens independently per PMU and per logical core. As some PMUs are shared between threads of the same core or package, their availability may change depending on what events are being measured on the other cores.

3.4.2 Modeling Errors in Event Samples

The first step to computing the full posterior distribution is to model errors in the capture of samples from HPCs. Recall that we listed sources of such errors in §3.2. For a single

event e programmed in an HPC c, if the error in measurement e_c can be modeled, then the measured/sampled values m_c can be modeled in terms of the true value v_c plus measurement noise e_c, i.e., $m_c = v_c + e_c$. Here, we focus only on random errors, by assuming zero systematic error. That is a valid assumption because the only reason for systematic errors will be hardware or software bugs. We assume that the error can be modeled as $e_c \sim \mathcal{N}(0, \sigma)$ for some unknown variance σ, hence $\Pr(m_c \mid v_c) = \mathcal{N}(m_c, \sigma)$ [71]. Now, given N samples of HPC, we compute their sample mean μ and sample variance S. A scaled and shifted Student's t-distribution describes the marginal distribution of the unknown mean of a Gaussian, when the dependence on variance has been marginalized out [100], i.e., $v_c \sim \mu + {}^{S}/_{\sqrt{N}}\ Student(\nu = N - 1)$. In all our experiments, the confidence level of the t-distribution was set to 95%. Now, since the measurement error model for an HPC is stochastic, when samples from these models are used in the algebraic relationships described above, they too become stochastic in nature. *The FG becomes one unified graphical representation of all of these statistical relationships, i.e., between the errored samples and true values of events, as well as among different events that measure related aspects of the CPU's microarchitecture.*

3.4.3 Inference Strategy

Once we have computed a schedule that ensures that events with statistical dependencies between them are measured in consecutive time slices, the next goal is to utilize the measurements to produce a posterior distribution for an event. Recall Fig. 3.2. In each scheduling time slice, we have measurements/samples from the current slice and the preceding slice. However, because of the transitive statistical dependencies, we would like to jointly compute inference for the FG (i.e., compute the posterior probability of some event in the FG given the sampled data) for some k time slices into the past.

Our approach to performing this computation with low-latency guarantees utilizes the idea that one can break the larger problem into k smaller parts, performing inference on each of the k parts, and then put the results together to get an approximate posterior inference, i.e., similar to map-reduce. There are two difficulties with such algorithms, as they are usually constructed. First, each of the k pieces has only partial information; as a result, for any of the pieces, a lot of computation is wasted in places that are contradicted by the other $k-1$ pieces. Second, the partial results from the k pieces must be carefully combined together to ensure that the prior (which is embedded into the FG model) is not counted multiple times. We use the Expectation Propagation (EP) algorithm [101, 102, 103] to overcome those difficulties to perform the inference. The EP algorithm naturally lends itself to *distributed inference* on partitioned datasets [103]. Hence we can perform inference on partitions of data, i.e., each

Algorithm 3.1 General EP algorithm.

Input: Target distribution $f(\theta) = \prod f_k(\theta)$
Output: Global approximation $g(\theta) = \prod g_k(\theta)$
1: Choose initial $g_k(\theta)$
2: **for** $k \in \{0, \dots K-1\} \wedge$ until g_k converges **do**
3: $g_{-k}(\theta) \propto {}^{g(\theta)}/_{g_k(\theta)}$ ▷ Cavity distribution
4: $g_{\backslash k}(\theta) \propto \Pr(y_k|\theta) g_{-k}(\theta)$ ▷ MCMC
5: $g^{new}(\theta) \propto g_{\backslash k}(\theta)$ ▷ Local update
6: $\Delta g_k(\theta) \approx {}^{g^{new}(\theta)}/_{g(\theta)}$
7: $g(\theta) \leftarrow g(\theta) \Delta g_k(\theta)$ ▷ Global update
8: **end for**
9: **return** $\{g_k(\theta) | k \in [0, K)\}$

scheduled configuration of the HPCs. In contrast, other techniques for Bayesian inference would require us to explicitly change the inference algorithm depending on the schedule of HPCs and the structure of the FG. Such changes might not be feasible for all possible schedules or all CPU architectures. The EP algorithm works by computing an effective region of overlap over our k pieces, i.e., for each piece, we use an approximate prior computed over the other $k-1$ pieces. The outline of the EP algorithm is illustrated in Alg. 3.1. The algorithm iteratively approximates a target density $f(\cdot)$ (in our case the FG) with a density $g(\cdot)$ that admits the same factorization, and uses a Gaussian *mean field approximation* [99].

Training. Training is not explicitly required for the proposed BayesPerf model. The advantage of using Bayesian models like FGs is that training on such models can be reduced to inference on the models' parameters. At runtime, for each time slice, we compute (infer) a full posterior distribution over the variables (i.e., E) and parameters (i.e., Θ) of the FG, and then use maximum likelihood estimation to pick the set of parameters (i.e. $\hat{\Theta}^{(MLE)}$) that can explain a data trace generated by the system.

3.5 THE BAYESPERF IMPLEMENTATION

In this section we describe the software and hardware components in which BayesPerf is deployed. Further, we describe the architecture and implementation of the BayesPerf accelerator that targets the execution of Alg. 3.1. Fig. 3.4 shows the architecture of the BayesPerf system, which works as follows.

Setup. BayesPerf is used by one or more "monitoring processes/threads" (labeled "Monitoring Application" in Fig. 3.4) to monitor hardware threads of a "Target Process." The BayesPerf user API is identical to the the Linux perf subsystem, and hence any user space

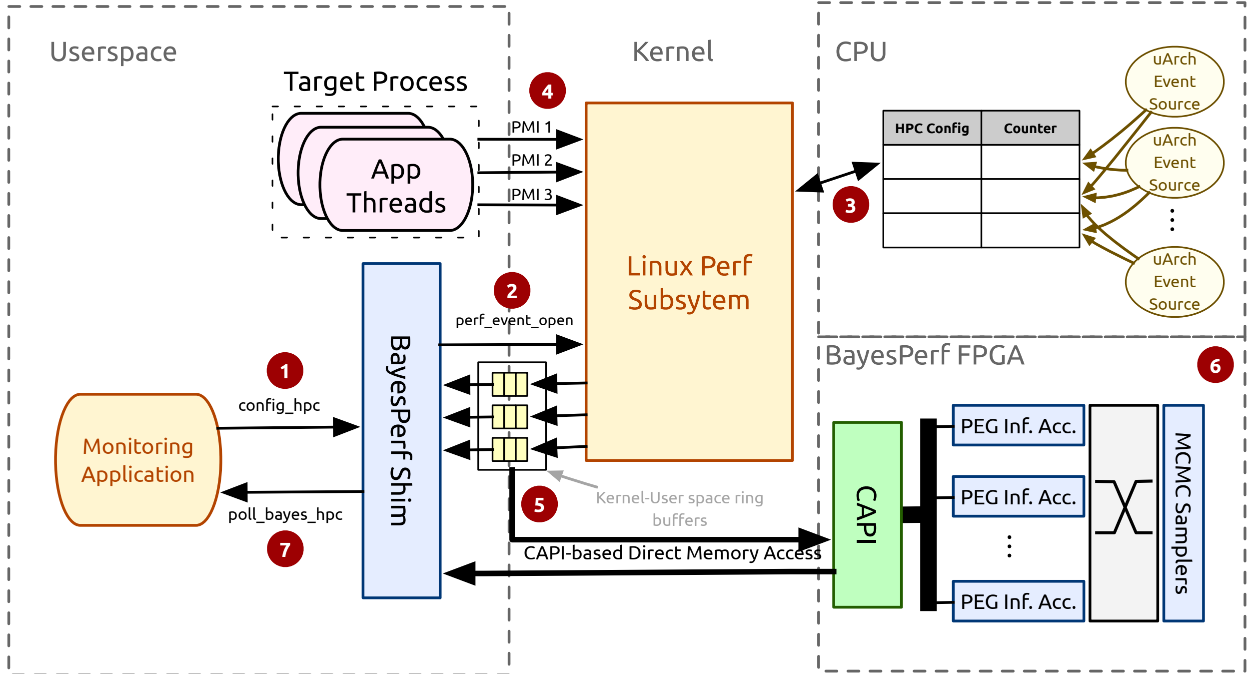


Figure 3.4: High-level architecture of the BayesPerf system.

program that uses the standard Linux interface can transparently use BayesPerf. Using this API, the monitoring process registers events of interest (labelled as 1 in Fig. 3.4[6]) with the userspace component of the BayesPerf system, labelled "BayesPerf Shim." The shim represents a userspace driver [104] that replicates the API of the Linux perf subsystem.

Linux perf. The shim registers HPCs on behalf of the user process with the Linux kernel. (labelled as 2). The kernel then manages the scheduling of performance counters onto the CPU (using the scheduling algorithm described in §3.4.1). This step is labelled as 3. When the target process raises *performance monitoring interrupts* (PMIs; labelled as 4), the Linux perf subsystem is responsible for reading the corresponding HPC and writing out the sampled value into a "ring buffer" (labelled as 5) that represents a segment of memory that is mapped into the address space of both the shim and the perf subsystem. The ring buffer represents a FIFO in which new samples are enqueued by the kernel and read from the userspace process. The ring buffer automatically provides a mechanism for managing backpressure between the shim and kernel as new samples are dropped if the ring buffer is full.

Interfacing with the Accelerator. As we will discuss in §3.6.1, we have prototyped the BayesPerf system on two different architectures: an Intel x86_64 and an IBM Power9 processor. The protocol for communication between the software and the BayesPerf accelerator (labelled as 6; described later) differ for the two architectures. On the Power9 system, we

[6]* refers to annotations in Fig. 3.4 if not otherwise specified.

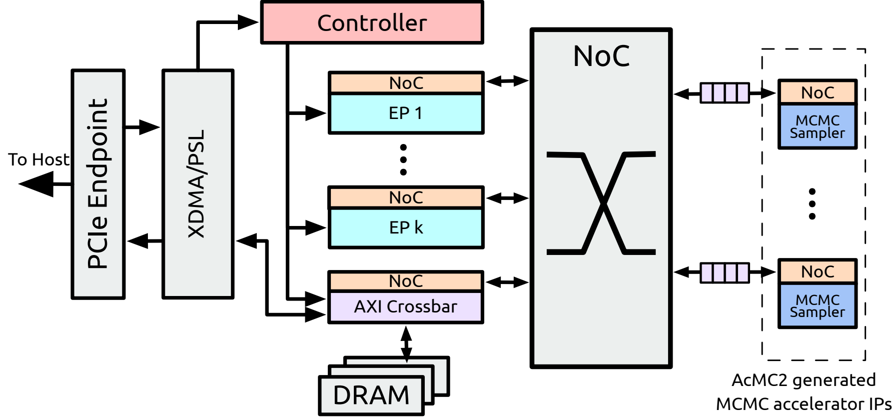


Figure 3.5: Architecture of the BayesPerf accelerator.

leverage CAPI 2.0 [96], a protocol that extends the processor's cache coherence protocol to PCIe-attached devices. In that case, as the accelerator can directly access the host memory, it can consume samples enqueued onto the ring buffer by the kernel (labelled as 5). It does so by snooping on cache invalidation messages for the cache lines corresponding to the ring buffer. Similarly, outputs of the accelerator are directly written back to the shim's virtual memory space. For Intel systems, the accelerator uses the base PCIe protocol and IOMMU-mediated PCIe DMA to read HPC samples and write the computed posterior distributions. Here, the shim must actively poll writes from the kernel to the ring buffer, and once the write has been made, initiate transfer of the samples to the FPGA. Similarly, the shim polls for interrupts from the accelerator that signify completion of computation, and initiates DMA transfers for the results. This added software interaction adds some latency overhead to the entire computation.

Polling Results. Finally, the monitoring application reads (polls) the results of the posterior computation in BayesPerf (labelled as 7) from ring buffers in the BayesPerf shim. These reads are always reads to the host memory of the CPU and do not need to initiate DMA requests with the accelerator. This design is able to mask almost all the latency that is added because of the added computation in BayesPerf (see Fig. 3.3).

Multi-Threaded Applications. OS-level monitoring contexts, like processes or threads are dealt with at the level of BayesPerf shim. Hence, when an OS context switch occurs, the memory references of the `perf` ring buffers are changed by setting configuration registers on the accelerator using MMIO. When the new references are written, the accelerator begins pulling data from a different buffer in memory. As a result, the accelerator can be shared across threads that are concurrently executing on the host CPU.

The Accelerator. Fig. 3.5 illustrates the architecture of the BayesPerf accelerator. The

accelerator exploits parallelism in the structure of Alg. 3.1 in two ways. First, we execute posterior inference on each of the k time-slices in parallel (recall §3.4.3). These parallel execution engines are labeled as "EP 1" through "EP k" and "Controller" in Fig. 3.5. The EPs execute lines 3–6 of Alg. 3.1 in parallel, and communicate their results to the global controller, which synchronously updates $g(\theta)$ and dispatches the new value to the idle EP. The values of the measurements from the HPCs (i.e., inputs) as well as the latest values of $g(\theta)$ are stored in the on board DRAM. Our target FPGA board (which we will describe in §3.6.1) supports 4 channels of 16GB LPDDR4 memory each. The input data and the current values of $g(\theta)$ (which together comprise $\sim$ 100 MB of data) are replicated across those modules to allow concurrent reads from different EPs to progress simultaneously.

The second level of parallelism exploited by the model is in the computation of MCMC inference in each of the EPs. Those are represented by the "MCMC Sampler" blocks in Fig. 3.5. They execute line 4 of Alg. 3.1 in parallel, by using MCMC to estimate $\Pr(y_k|\theta)$ (i.e., the likelihood that the data y_k is drawn from the local approximator g_k). Here we leverage our prior work, AcMC^2[95](see Chapter 5), a high-level synthesis compiler for MCMC applications, to generate IPs that can generate samples from the target distributions of the HPC measurements. The HPC statistical relationships (i.e., the FG) are fed into the compiler as a probabilistic program, i.e., a program in a domain specific language that can represent statistical dependencies between program variables. AcMC^2 then automatically generates efficient uniform random number generators, and automatically synthesizes other statistical constraints in FG. Instead of using the AcMC^2-generated controllers for the MCMC samplers, we use the EPs to directly control the pipelines of MCMC samplers. That is, (i) to set and update configuration parameters like seed values; and (ii) to update the state of the sampler with one which passes the rejection sampling test criteria for each random-walk. Allotment of the samplers to EPs, and all subsequent communication between the EPs and samplers, happen over a network-on-chip (NoC) generated with CONNECT [105]. This approach enables us to use samples from previous iterations as starting points for Markov-chain random walks. This optimization is possible only because we are using MCMC inside an EP algorithm, instead of by itself [106]. The NoC uses a butterfly topology to allow communication between EPs and samplers, as well as between the samplers themselves (as is required by AcMC^2). All our experiments use a 16 port NoC, with 4 of those ports being connected to the EPs, and the remaining 12 to the MCMC samplers. This is the maximal configuration for which we were able to meet timing requirements on the FPGA for a 250 MHz clock.

Table 3.1: Area & Power for components of the BayesPerf FPGA for the x86_64 and ppc64 configurations.

Component	Utilization (%)					Power (W)	
	BRAM	DSP	FF	LUT	URAM	Vivado	Measured
x86-PCIe	62	78	52	81	58	11.2	17.2
ppc64-CAPI	71	66	49	79	58	10.5	16.1

3.6 EVALUATION & DISCUSSION

This section discusses our experimental evaluation of the BayesPerf system and is organized as follows. First, in §3.6.1, we describe the experimental setup and explore the performance, power, and area requirements of BayesPerf accelerators when programmed onto an FPGA. Then, in §3.6.2 we evaluate the capabilities of the BayesPerf system in correcting measurement errors in HPCs. Finally, we demonstrate the integration of BayesPerf with ML-based resource management systems to improve their outcomes.

3.6.1 Experiment Setup

We evaluate BayesPerf on two system configurations: (i) an IBM AC922 dual-socket Power9 system (which we will refer to as the "ppc64" configuration), and (ii) a dual-socket Intel Xeon E5-2695 system (which we will refer to as the "x86" configuration). Both the systems are populated with two NVIDIA K80 GPUs, a single FDR Infiniband NIC, and a directly attached FPGA board (which we describe below). Both systems ran Ubuntu 18.04 with kernel version `v4.15.0`.

Accelerator: FPGA. The FPGA accelerator was based on the architecture in §3.5. All experiments were performed on an Alpha-Data ADM-PCIE-9V3 FPGA board (with Xilinx Virtex UltraScale+ VU3P-2 FPGA) clocked at 250 MHz. For the Power9 systems, the FPGA board was configured to use the CAPI 2.0 interface [96]. For the x86 configuration, the FPGA board was configured to use PCIe3 x16 along with the Xilinx XRT drivers. The power and FPGA utilization metrics for the two configurations of the BayesPerf accelerator are listed in Table 3.1. In comparison to a 100W TDP of the Intel processor and a 190W TDP Power9 processor, the FPGA performs 5.8× and 11.8× better, respectively, in terms of power consumption. The BayesPerf-ppc64 FPGA read latency is shown in Fig. 3.3. We observe that a single HPC read using the CPU implementation of BayesPerf has approximately 9× longer latency than native polling of the HPC. However, when the accelerator is being used, BayesPerf adds less than 2% overhead in read latency compared to the native solution.

Table 3.2: Area & Power for components of the BayesPerf ASIC for the x86_64 and ppc64 configurations.

Component	Type	Area (mm^2)	Power (W)
x86-PE	Logic	19.2	1.14
x86-PE	SRAM	66.0	3.39
ppc64-PE	Logic	18.6	1.12
ppc64-PE	SRAM	66.0	3.39
DRAM	LPDDR4 (4×32GB)	-	1.64

Compared to the BayesPerf-ppc64 implementation that uses CAPI, the BayesPerf-x86 has on average 15.8% larger latency. We can attribute that slowdown to the requirement that a userspace driver actively initiates DMA transfers to the FPGA accelerator, whereas the CAPI configuration snoops for cache invalidation messages.

Accelerator - ASIC. In addition to the experiments we perform on an FPGA to demonstrate the utility of BayesPerf, we provide a whole-chip performance and energy model for the design to demonstrate feasibility of integrating BayesPerf in future CPUs designs (in package or as an SoC component). The FPGA and ASIC designs differ in that the ASIC design does not consider the power and area requirements of the CAPI/PCIe IP. The design is evaluated in a 32 nm process, assuming a 1 GHz clock. For the energy model, energy numbers for arithmetic units are taken from [107] and scaled to 32 nm. SRAM energies are taken from CACTI [108]. We assume all SRAMs are itrs-lop as this decreases energy per access, but still yields SRAMs that meet timing at 1 GHz. DRAM energy is counted at 20 pJ/bit [107]. NoC energy is extrapolated based on the number and estimated length of wires in the design (using our PE area and L2 SRAM area estimates from CACTI). We assume the NoC uses low-swing wires [109], which are low power, however consume energy each cycle (regardless of whether data is transferred) via differential signaling. Ramulator [110] and DRAMPower [111] (using a LPDDR3 model, which should be a conservative estimate for LPDDR4 memory in our FPGA implementation) to estimate DRAM timing and power respectively. Table 3.2 shows the result of the analysis. The ASIC implementations perform 2.7 and 2.6× better than the FPGA accelerators respectively (see Table 3.1), in terms of power consumption. Using the cycle counts from the FPGA implementation, the ASIC throughput was estimated by scaling to ASIC frequency, resulting in near linear scaling. As a result, there is a significant drop (nearly 4×) in latency for the accelerator, and the performance bottleneck in terms of the architecture presented in Fig. 3.4 is in the round trip time over the PCIe interconnect.

3.6.2 Error Reduction Due to BayesPerf

To demonstrate the efficacy of BayesPerf in correcting HPC measurement errors, we employed the 29 workloads from the HiBench suite [112], which span microbenchmarks, machine learning, SQL, web search, graph analytics, and streaming applications. They represent real-world application workloads used in a cloud environment. We used the two machines in our experiment to simulate a cluster. Each of the machines hosted 32 workers, and the Spark master was deployed on the x86 node. We measured 10 derived events for each of the microarchitectures, where each derived event corresponded to a group of HPCs to be measured and aggregated using a mathematical relationship. We do not detail the events here for lack of space. The metadata corresponding to the events for the x86 configuration can be found in the Linux kernel source tree [92] for both the x86 and ppc64 configurations. In both cases, we measured all HPCs corresponding to the first 10 metrics.

Baselines. We use three baselines for comparison. First, we use Linux's inbuilt correction mechanism that uses enabled time and total time (recall from §3.4) to correct for measurement errors. This is the most realistic baseline for users who would use the default configuration available in Linux. Second, we use a variance reduction technique called CounterMiner [74] (CM), a state of the art HPC correction technique used in profiling analysis. Note that CM was originally meant to be used for offline analysis. As we will show in the remainder of this section, this requirement manifests as low average correction accuracy, with large variance, when used for online corrections. Third, we use the online technique by Weaver et. al. [71] (referred to as "WM+Pin") for correcting instruction counts in x86 processors. WM+Pin only corrects the number of instructions executed and was originally meant to correct core performance metrics like IPC or CPI. Further, it requires intercepting instructions through Pin [113] to collect opcodes for every dynamic instruction. This causes performance degradation of up to 198.2× across our benchmarks.

Error Correction. Fig. 3.7 shows the significant improvement in measurement values compared to the baseline. The average error across all benchmarks dropped from 39.25% and 40.1% for the "Linux (x86)" and "Linux (ppc64)," respectively, to 8.06% (i.e., $4.87\times = {}^{39.25\%}/_{8.06\%}$) and 7.6% (i.e., $5.28\times = {}^{40.1\%}/_{7.6\%}$). Similarly, when "BayesPerf (x86)" and "BayesPerf (ppc64)" are compared to "CM (x86)" and "CM (ppc64)," the average error dropped by 3.63× ($= {}^{29.28\%}/_{8.06\%}$) and 3.73× ($= {}^{28.31\%}/_{7.6\%}$), respectively. Similar improvements were observed in the CM configuration. That corresponds to a nearly 40% improvement in the quality of the result of the ppc64 configuration. The normalized improvement in average error for each of the benchmark applications when using BayesPerf, compared to the two baselines is shown Fig. 3.8. Recall from §3.3, that error in measurement is computed as the

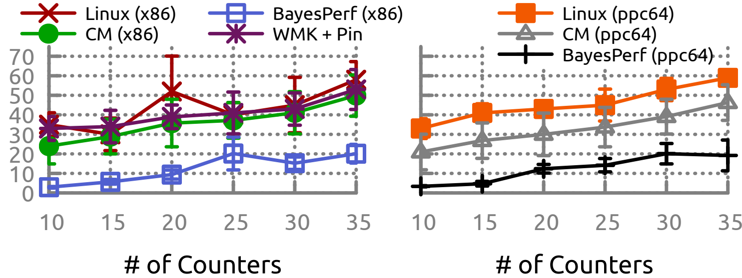


Figure 3.6: Scaling errors with the number of events sampled.

similarity between two time series sequences of performance counter samples [91]. In the case of the BayesPerf counters, we used a maximum likelihood estimator to provide the most likely value of the performance counter at a point in time. We normalize the similarity scores using an average similarity score between two runs of the application, with the application running in polling mode. That way, we could correct for any OS-based nondeterminism in the result. Just like in §3.2, where the magnitude of the error is a comparison between "polling" mode and "sampling" under Linux and CM (see Fig. 3.7).

Scaling. Fig. 3.6 shows the scaling behavior of the BayesPerf method with increasing numbers of counters for the "KMeans" workload in the HiBench suite. We observe that BayesPerf consistently reduced error by as much as 34% as the number of counters scaled up from 10 to 35 (for Linux). Further, WM+Pin performs worse than CM as it only corrects instruction counts. This justifies our choice of using CM as the main baseline for the evaluation. Interestingly we find that floating point initialization, which is a major source of errors in [71], doesn't result in overcounts, indicating that the issue is resolved in modern CPUs.

Latency Overhead. Since BayesPerf, performs significantly more compute than either Linux or the CM configurations, it is expected to be a significantly higher latency. Recall from Fig. 3.3 that the difference in latency between BayesPerf (when implemented in software) and the Linux correction is nearly 9×. The BayesPerf accelerator is designed to mitigate the effects of this increased latency. Again, from Fig. 3.3, we see that it successfully does so, reducing the 9× difference to 2%. This is on par with native HPC reads using `rdpmc` as well as kernel-assisted HPC reads.

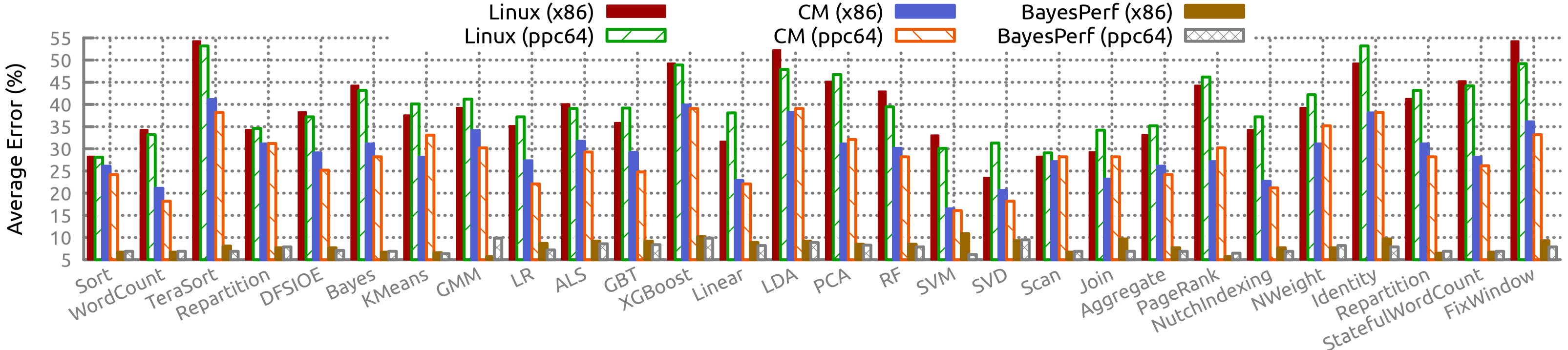


Figure 3.7: Error in performance counter measurements across the HiBench benchmarks.

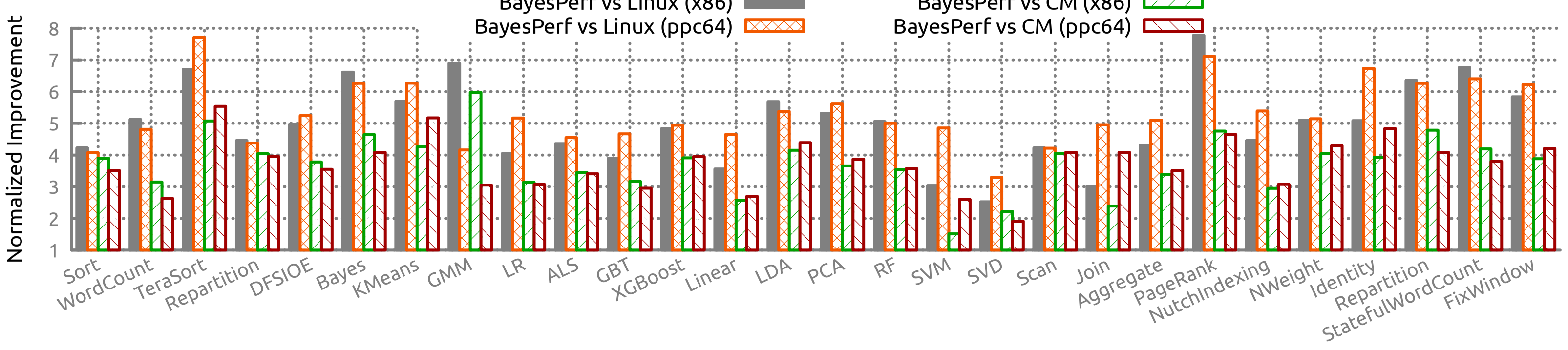


Figure 3.8: Normalized improvement in performance counter error measurements across the HiBench benchmarks.

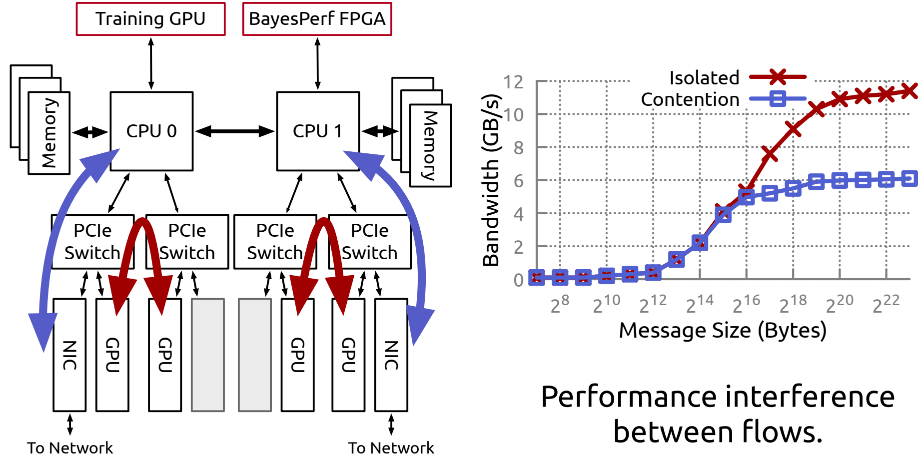


Figure 3.9: Topology of test system in §3.6.3 as well as the effect of the resource contention.

3.6.3 Case Study: BayesPerf in Feedback Loop

The core value of the BayesPerf approach in terms of it's error correction capability has been demonstrated in the previous section. Here we demonstrate the downstream value of BayesPerf to applications that use HPCs as inputs to control system resources. Examples of such applications include online performance hotspot identification (e.g., [21]), userspace or runtime-level scheduling (e.g., [29, 39, 70, 76, 77]), and power and energy management (e.g., [78, 79, 80, 81]), as well as attack detectors and system integrity monitors [82]. Further they often use as many as 45 HPCs in the case of [29, 76, 114].

The Problem. We now look at a situation in which BayesPerf measurements can be integrated into higher-level decision-making frameworks to perform resource management decisions. In this part of the experiment, we used HPC measurements to augment an Apache Spark Executor [115] that needed to run a distributed shuffle operation (which is part of the HiBench TeraSort benchmark [112]). Fig. 3.9 illustrates the rich dynamic information that can be extracted from HPC measurements, and how they can be used in higher-level controllers. Consider the case of a PCIe interconnect which is populated with NIC and GPU devices. Here, the Spark executor uses two GPUs to perform a halo exchange (for training a deep neural network). Fig. 3.9 shows the performance (in this case, bandwidth) of the exchange as "isolated" performance. If, at the same time, the application were to perform a distributed shuffle (across nodes in a cluster) using the NIC, we would observe that the original GPU-to-GPU communication is affected because of PCIe bandwidth contention at shared links. That phenomenon is shown as "contention" performance in Fig. 3.9, and it can cause as much as a 0–1.8× slowdown, depending on the size of the PCIe transactions. Online bandwidth and transaction size monitoring (which is enabled by HPCs) can be used by a

higher-level software framework to optimally schedule such transfers, so that the performance impact of shared resource contention is minimized. While the example is simple, it illustrates how errors in measurements can affect the ML algorithm, and hence the overall system performance.

We use two ML-based scheduling algorithms broadly based on those presented in [39] and in our prior work [76]. The first used collaborative filtering as the core ML algorithm, and the second used deep reinforcement learning. The goal of our ML-based scheduler was to decide which of the two NICs it would use to perform the shuffle operation, given that the GPUs were communicating with each other and contending for PCIe bandwidth. We simulated the GPU communication by using Tensorflow to train YoloNet on the ImageNet dataset.

The Models. The goal of this case study was to show the sensitivity of ML models to errors in their inputs (especially coming from HPCs). The inputs to the models included: (i) sampled HPC measurements corresponding to the numbers of allocating, full, partial, and non-snoop writes, (ii) sampled HPC measurements corresponding to demand code reads and partial/MMIO reads, (iii) DRAM Channel bandwidth utilization, (iv) memory-bus bandwidth utilization, and (v) the size of data to be shuffled (in or out), and the NUMA node on which the data would be resident. Note that all of the above are derived events, computation of which required us to capture 32 unique HPC events. Out of which, 12 were collected for each physical core (i.e., used 432 HPCs = 12 events $\times$ 18 cores $\times$ 2 sockets), and 20 were off-core events being collected per-socket (i.e., used 40 HPCs = 20 events $\times$ 2 sockets).

The first model, used collaborative filtering to impute values of application performance (in this case throughput) with data coming in from the inputs above, as well as data from training workloads of the SparkBench suite in HiBench. It is based on the technique presented in [39]. The second model used a straightforward neural network: a 4-layer, fully connected ReLU-activated neural network with 36 neurons in layer 1, 16 neurons in each of layers 2 & 3, and 2 neurons in the last layer. The two neurons in the last layer chose between the two NICs that were decided between as part of this task. The model was trained with actor-critic reinforcement learning based on the approach described in [76]. The loss function used for training the model minimized the total time taken to complete the shuffle. The model was trained on the HiBench benchmark suite without the TeraSort benchmark, and then evaluated using the TeraSort benchmark. When BayesPerf was used, the MLE estimate from the posterior distribution of the HPC was passed into the network. The GPU marked “Training GPU” was used to perform the collaborative filtering and reinforcement learning as well as runtime inference on the system. It did not contend for the same PCIe resources as the workloads that was being scheduled GPUs.

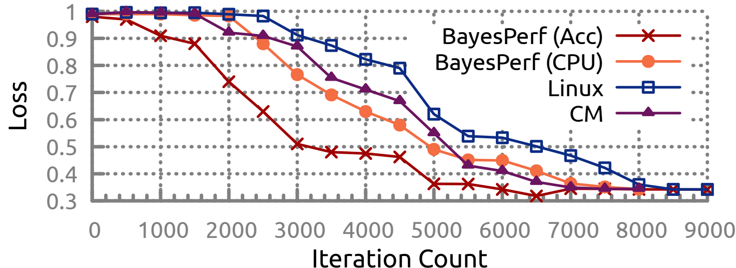


Figure 3.10: Decrease in training time due to BayesPerf.

Implementation Details: Training. Recall from §3.4 that the BayesPerf model in itself does not require training. However, the two models described above require training. The model from [76] learns by reinforcement. Hence, it does not have specific training and testing phases. The net epochs of data used to train the model are shown in Fig. 3.10. For the model in [39], which has specific training and test datasets, we calibrate against bias by using threefold cross-validation (i.e., across applications in Fig. 3.7).

Implementation Details: Hyperparameters. The hyperparameters used in the model are taken directly from [76] and [39]. These parameters include learning rate, LSTM-unroll-length, and epoch lengths, among others. In addition, we follow the procedure set out in [39] to determine the optimal value of sparsity. We sweep over the range between 30% and 80%. All results in this chapter uses the optimal (found from our sweep) value of 75% sparsity.

Results. We compare the results of using the above model with BayesPerf and without, using two metrics.

Results: Training Time. The collaborative filtering model does not have an explicit training phase. For the deep learning model, Fig. 3.10 illustrates the difference in training time when error-corrected measurements are used. In the figure, the loss is normalized using the time taken to run the same shuffle operation in a completely isolated system. We observed a nearly 37% reduction in the number of iterations before convergence. Each training iteration in Fig. 3.10 takes 63s; therefore, the overall saving of 37% corresponds to $\sim$52 hr. The reason for the reduction is apparent: a 40% error in the inputs of the neural network is slowing down the optimization process. Moreover, we observe that the time to convergence is effected by (i) the magnitude of error reduction, as seen by the difference between the Linux–CM (12.5% decrease) and –BayesPerf (37% decrease) configurations; and (ii) the timeliness of the error reduction, as seen by the difference between the CPU and accelerated versions of BayesPerf (28.5% decrease).

Results: Decision Quality. We observe that use of the ML-based scheduler (i.e., that makes Spark PCIe aware) leads to a $15.1 \pm 2.2\%$ and $22.3 \pm 7.9\%$ improvement in average

shuffle completion time for the two models respectively. Addition of BayesPerf to the model results in a further $8.7 \pm 0.9\%$ and $19 \pm 3.4\%$ reduction in average shuffle latency, respectively.

3.7 RELATED WORK

Error Correction in HPCs Measurement errors due to sampling in HPCs have been observed and reported on for the past decade [69, 70, 71, 72, 73, 74, 75]. Methods for correction of sampled HPC values can be broadly grouped into two separate approaches. The first group of methods artificially imputes data in the collected samples by interpolating between two sampled events using linear or piece-wise linear interpolation (e.g., [92]). The advantage of such interpolation methods is that they can be run in real time: however, they might not provide good imputations [70]. The second group of methods correct measurements by dropping outlier values, instead of by adding new interpolated values. Such methods are at the other extreme: they cannot be run in real time, as they need the entire trace of an application before providing corrections. For example, Lv et al. [74] use the Gumbel test for outlier detection, and Neill et. al. [116] use fork-join aware agglomorative clustering to remove outlier points. These methods are not suitable for dynamic control situations that need online HPC correction. Further, the core statistical technique used by these variance reduction approaches assume that the underlying distribution of the data remains unchanged, however, most workload exhibit distinct stages where workload behavior and thus the underlying distribution of the HPCs will change.

In contrast to those techniques, BayesPerf corrects measurements by using statistical relationships between events. For well-documented processors, such relationships can be known ahead of time, and the entire correction algorithm can be executed without any need to pre-collect data. The BayesPerf system (with its accelerator) allows nearly native latency access to the corrected HPCs, thereby enabling their use in dynamic control processes.

Using HPCs in Control. Several recent papers have explored the use of HPCs to perform higher-level resource management problems. Examples include online performance hotspot identification (e.g., [21]), userspace or runtime-level scheduling (e.g., [20, 29, 39, 70, 76, 77, 117]), power and energy management (e.g., [78, 79, 80, 81]), and attack detectors and system integrity monitors [82]. Most of the methods mentioned above do not explicitly use any techniques to correct for errors in HPC measurements. Further, while it is not impossible that some of the ML techniques can inherently correct for HPC errors, there are no guarantees that it does so.

3.8 SUMMARY

It is crucial to have reliable instrumentation/measurement in commercial CPUs, as exemplified by the inclusion of the `PEBS` (precision event-based sampling) and `LBR` (last branch record) technologies in modern Intel processors. However, as we showed in this chapter, such technology alone falls short of correcting errors in the values of HPCs accrued because of nondeterminism and sampling artifacts. This chapter presented the design and evaluation of BayesPerf, an ML model and associated accelerator that allows for correction of noisy HPC measurements, reducing the average error in HPC measurements from 42.11% to 7.8% when events are being multiplexed. BayesPerf is the first step in realizing a general-purpose HPC-error-correction system for real x86 and ppc64 systems today and potentially for future processors. We believe it will form the basis for performing large-scale measurement/characterization studies that use HPC data (i.e., offline analysis), but also enable a slew of applications that can use the HPC data to make control-decisions in a computer system (i.e., online analysis).

CHAPTER 4: LEARNING SCHEDULING POLICIES FOR HETEROGENEOUS CLUSTERS

4.1 INTRODUCTION

The problem of scheduling of workloads on heterogeneous processing fabrics (i.e., accelerated datacenters including GPUs, FPGAs, and ASICs, e.g., [10, 118]), is at its core an intractable NP-hard problem [119, 120]. System schedulers generally rely on application- and system-specific heuristics with extensive domain-expert-driven tuning of scheduling policies (e.g., [28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38]). Such heuristics are difficult to generate, as variations across applications and system configurations mean that significant amounts of time and money must be spent in painstaking heuristic searches. Recent work has demonstrated machine learning (ML) techniques [39, 40, 41, 42] for automating heuristic searches by using black-box approaches which require significant training data and time, making them challenging to use in practice.

This chapter presents Symphony, a scheduling framework that addresses the challenge in two ways: (i) we use a domain-guided Bayesian-model-based partially observable Markov decision process (POMDP) [121, 122] to decrease the amount of training data (i.e., sampled trajectories); and (ii) a sampling-based technique that allows one to compute the gradients of a Bayesian model without performing full probabilistic inference. We thus, significantly reduce the costs of (i) running a large heterogeneous computing system that uses an efficient scheduling policy; and (ii) training the policy itself.

Reducing Training Data. State-of-the-art methods for choosing an optimal action in POMDPs rely on training of neural networks (NNs) [123, 124]. As these approaches are model-free, training of the NN requires large quantities of data and time to compute meaningful policies. In contrast, we provide an inductive bias for the reinforcement learning (RL) agent by encoding domain knowledge as a Bayesian model that can infer the latent state from observations, while at the same time leveraging the scalability of deep learning methods through end-to-end gradient descent. In the case of scheduling, our inductive bias is a set of statistical relationships between measurements from microarchitectural monitors [125]. To the best of our knowledge, this is the first work to exploit those relationships and measurements to infer resource utilization in the system (i.e., latent state) to build RL-based scheduling polices.

Reducing Training Time. The addition of the inductive bias, while making the training process less data-hungry (i.e., requiring fewer workload executions to train the model), comes at the cost of additional training time: the cost of performing full-Bayesian inference at every

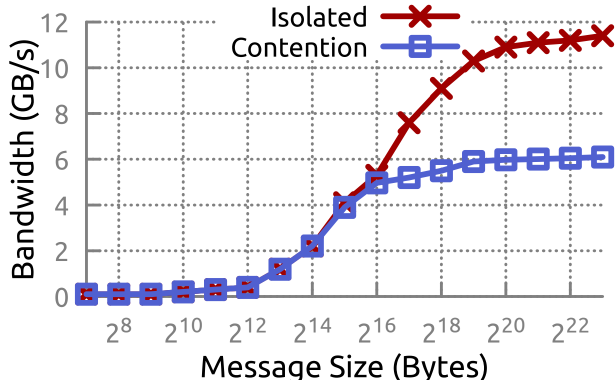


Figure 4.1: Performance degradation due to PCIe contention between GPU and NIC (averaged over 10 runs).

training step [126, 127, 128]. It is this cost that makes the use of deep RL techniques in dynamic real-world deployments (which require periodic retraining) prohibitively expensive. To address that issue, we have developed a procedure for computing the gradient of variables in the above Bayesian model without requiring full inference computation, unlike prior work [127, 128]. The key is to calculate the gradient by generating samples from the model, which is computationally simpler than inferring the posterior distribution.

Need for New Scheduler. Current schedulers prioritize the use of simple generalized heuristics and coarse-grained resource bucketing (e.g., core counts, free memory) to make scheduling decisions. Hence, even though they are perceived to perform well in practice, they do not model complex emergent heterogeneous compute platforms and hence leave a lot to be desired. Consider the case of a distributed data processing framework that uses two GPUs to perform a *halo exchange*.[1] Fig. 4.1 shows the performance (here, bandwidth) of the exchange as "isolated" performance. If the application were to concurrently perform distributed network communication, we would observe that the original GPU-to-GPU communication is affected because of PCIe bandwidth contention at shared links (i.e., a "hidden" resource that is not often exposed to the user). Such behavior is shown as "contention" in Fig. 4.1, and can cause as much as a $0 - 1.8\times$ slowdown, depending on the size of the transmitted messages. Traditional approaches would either have such a heuristic manually searched and incorporated into a scheduling policy, or would expect it to be found automatically as part of the training of a black-box ML model, and both approaches can require significant effort in profiling/training. In contrast, our approach allows the utilization of architectural resources (in this case, of the PCIe network) as an inductive bias for the RL-agent, thereby allowing the training process to

[1] A *halo exchange* occurs due to communication arsing between parallel processors computing an overlapping pieces of data, called *halo regions*, that need to be periodically updated.

automatically hone in on such resources of interest, without having to identify the resource's importance manually.

Results. The Symphony framework reduces the average job completion time over hand-tuned scheduling heuristics by as much as 32%, and to within 6% of the time taken by an oracle scheduler. It also achieves a training time improvement of 4× compared to full Bayesian inference based on belief propagation. Further, the technique outperforms black-box ML techniques by 2.2× in terms of training time. We believe that Symphony is also representative of RL applied to several other control-related problems (e.g., industrial scheduling, data center network scheduling) where data-driven approaches can be augmented with domain knowledge to build sample-efficient RL-agents.

4.2 BACKGROUND

4.2.1 Partially Observable Markov Decision Processes.

A POMDP is a stochastic model that describe relationships between an agent and its environment. It is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \Omega, O, R, \gamma)$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, and Ω is the observation space. We use $s_t \in \mathcal{S}$ to denote the hidden state at time t. When an action $a_t \in \mathcal{A}$ is executed, the state changes according to the transition distribution, $s_{t+1} \sim \mathcal{T}(s_{t+1}|s_t, a_t)$. Subsequently, the agent receives a noisy or partially occluded observation $o_{t+1} \in \Omega$ according to the distribution $o_{t+1} \sim O(o_{t+1}|s_{t+1}, a_t)$, and a reward $r_{t+1} \in \mathbb{R}$ according to the distribution $r_{t+1} \sim R(r_{t+1}|s_{t+1}, a_t)$.

An agent acts according to its policy $\pi(a_t|s_t)$, which returns the probability of taking action a_t at time t. The agent's goal is to learn a policy π that maximizes the expected future reward $J = \mathbb{E}_{\tau \sim p(\tau)}[\sum_{t=1}^{T} \gamma^{t-1} r_t]$ over trajectories $\tau = (s_0, a_0, \ldots, a_{T-1}, s_T)$ induced by its policy, where $\gamma \in [0, 1)$ is the discount factor. In general, a POMDP agent must infer the belief state $b_t = \Pr(s_t|o_1, \ldots, o_t, a_0, \ldots, a_{t-1})$, which is used to calculate $\pi(a_t|\hat{s}_t)$ where $\hat{s}_t \sim b_t$. In the remainder of the chapter, we will use $\pi(a_t|\hat{s}_t)$ and $\pi(a_t|b_t)$ interchangeably.

4.2.2 Related Work.

Finding solutions for many POMDPs involves (i) estimating the transition model T and observation model O, (ii) performing inference under this model, and (iii) choosing an action based on the inferred belief state. Prior work in this area has extensively explored the use of NNs, particularly recurrent NNs (RNNs), as universal function approximators for (i) and (iii) above because they can be easily trained and have efficient inference procedures

(e.g., [129, 130, 131, 132, 133, 134, 135]). Neural networks have proven to be extremely effective at learning, but usually require a lot of data (for RL-agents, sampled trajectories, which may be prohibitively expensive to acquire for certain classes of applications, such as scheduling). The ability to incorporate explicit domain knowledge (which in the case of scheduling, is based on system design invariants) could significantly reduce the amount of data required. To that end, other work [1, 134, 136] has advocated the integration of probabilistic models (including Bayesian filter models) for (i) above. The significant computational cost of learning and inference in such deep probabilistic models has spurred the use of approximation techniques for training and inference, including NN-based approximations of Bayesian inference [134, 135] and variational inference methods [136].

In this chapter, we too advocate the use of a domain-driven probabilistic model for b_t that can be trained through end-to-end back-propagation to compute a policy. Specifically, the technique handles the gradient descent procedure on a Bayesian network (BN) with known structure and incomplete observations without performing inference on the BN, only requiring generation of samples from the model. That approach is different from to prior work on learning BNs using gradient descent [127, 128] or expectation maximization, both of which require full posterior inference at every training step.

4.2.3 Actor-Critic Methods.

Actor-Critic methods [137] have previously been proposed for learning the parameters ρ of an agent's policy $\pi_\rho(a_t|s_t)$. Here (i) the "Critic" estimates the value function $V(s)$, and (ii) the "Actor" updates the policy $\pi(a|s)$ in the direction suggested by the Critic. In this chapter, we use n-step learning with the asynchronous advantage actor-critic (A3C) method [123]. For n-step learning, starting at time t, the current policy performs n_s consecutive steps in n_e parallel environments. The gradient updates of π and V are based on that mini-batch of size $n_e n_s$. The target for the value function $V_\eta(s_{t+i})$, $i \in [0, n_s)$, parameterized by η, is the discounted sum of on-policy rewards up until $t + n_s$ and the off-policy bootstrapped value $V^*_\eta(s_{t+n_s})$. If we use an advantage function $A^{t,i}_\eta = (\sum_{j=0}^{n_s-i-1} \gamma^j r_{t+i+j}) + \gamma^{n_s-i} V^*_\eta(s_{t+n_s}) - V_\eta(s_{t+1})$, the value function is

$$\mathcal{L}^A_t(\rho) = -\frac{1}{n_e n_s} \sum_{e=0}^{n_e-1} \sum_{i=0}^{n_s-1} \mathbb{E}_{s_{t+i} \sim b_{t+i}} [\log \pi_\rho(a_{t+i}|s_{t+i}) A^{t,i}_\eta(s_{t+i}, a_{t+i})] \tag{4.1a}$$

$$\mathcal{L}^V_t(\eta) = \frac{1}{n_e n_s} \sum_{e=0}^{n_e-1} \sum_{i=0}^{n_s-1} \mathbb{E}_{s_{t+i} \sim b_{t+i}} \left[A^{t,i}_\eta(s_{t+i}, a_{t+i})^2 \right]. \tag{4.1b}$$

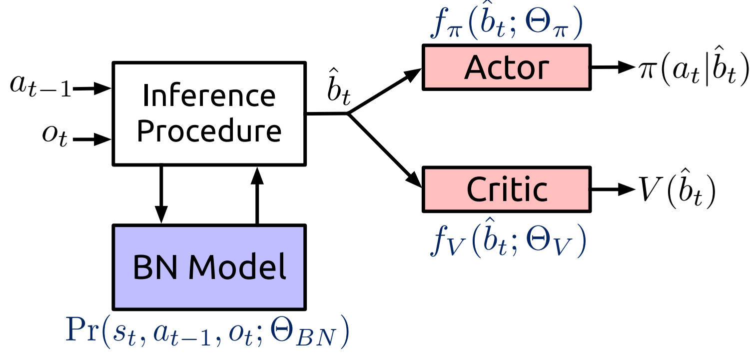


Figure 4.2: The proposed RL architecture for integrating Bayesian and deep learning models.

4.3 TRAINING THE POMDP RL-AGENT WITH BACK-PROPAGATION

We consider a special case of the POMDP formulation presented above (illustrated in Fig. 4.2). We assume that the domain knowledge about the environment of the RL-agent is presented as a joint probability distribution $\Pr(s_t, a_{t-1}, o_t; \Theta_{BN})$ that can be factorized as a BN (with parameters Θ_{BN}). A BN is a probabilistic graphical model that represents a set of variables and their conditional dependencies via a directed acyclic graph (DAG). We use probabilistic inference on the BN to calculate an estimate of the belief state $\hat{b}_t$. $\hat{b}_t$ is then used in an NN $f_\pi(\hat{b}_t; \Theta_\pi)$ (with parameters Θ_π) to approximate the RL-agent's policy, and an NN $f_V(\hat{b}_t; \Theta_V)$ (with parameters Θ_V) to approximate the state-based value function. We refer to all the parameters of the model as $\mathbf{\Theta} = (\Theta_{BN}, \Theta_\pi, \Theta_V) = (\rho, \eta)$. The model is then trained by propagating the gradient of the total loss $\nabla_{\mathbf{\Theta}} \mathcal{L}_t^{RL} = \nabla_{\mathbf{\Theta}} \mathcal{L}_t^A(\rho) + \nabla_{\mathbf{\Theta}} \mathcal{L}_t^V(\eta)$. Estimating this gradient requires us to compute $\nabla_{\Theta_{BN}} \hat{b}_t$. Traditional methods for computing the gradient require inference computation [127, 128]. However, even approximate inference in such models is known to be NP-Hard [126]. Below we describe an algorithm for approximating the gradient without requiring computation of full Bayesian inference. All that is required is the ability to generate samples from the BN. Only the subset of the BN necessary for generation of the samples is expanded. The samples are then used as a representation of the distribution of the BN. As a result, the proposed method decouples the training of the BN from the inference procedure used on it to calculate $\hat{b}_t$.

4.3.1 The Bayesian Network & Its Gradient

Let the BN described above be a DAG (V, E), and let $\mathbf{X} = \{X_v | v \in V\}$ be a set of random variables indexed by V. Associated with each node X is a conditional probability

density function $\Pr(X|\wp(X))$, where $\wp(X)$ are the parents of X in the graph. We assume that we are given (i) an efficient algorithm for sampling values of X given $\wp(X)$, and (ii) a function $f_X(x, y; \theta_X) = \Pr_{\theta_X}(X = x|\wp(X) = y)$ whose partial derivative with respect to θ_X is known and efficiently computable. The BN can also have deterministic relationships between two random variables, under the assumption that the relationship is a differentiable diffeomorphism. That is, for random variables X, Y, and diffeomorphism F, $\Pr(Y = y) = \Pr(X = F^{-1}(y))|DF^{-1}(y)|$ where DF^{-1} is the inverse of the Jacobian of F.

Computing Gradient. For a random variable X in the BN, we define its parents as $\wp(X)$, its ancestor set as $\Xi(X) = \{Y|Y \rightsquigarrow X \ \wedge \ Y \notin \wp(X)\}$ (where $\rightsquigarrow$ represents a directed path in the BN). We now define a procedure to approximately compute the gradient of X with respect to Θ_{BN}. We do so in two parts: (i) ${\partial \Pr(X=x|\xi=\mathbf{a})}/{\partial \theta_X}$ and (ii) $\nabla_{\Theta_{BN} \setminus \theta_X} \Pr(X = x|\xi = \mathbf{a})$ for $\xi \subseteq \Xi(X)$. First,

$$\begin{aligned}
\frac{\partial \Pr(X = x|\xi = \mathbf{a})}{\partial \theta_X} &= \frac{\partial}{\partial \theta_X} \int \Pr(\wp(X) = \mathbf{y}|\xi = \mathbf{a}) \times \Pr(X = x|\wp(X) = \mathbf{y}, \xi = \mathbf{a}) d\mathbf{y} \\
&= \frac{\partial}{\partial \theta_X} \int \Pr(\wp(X) = \mathbf{y}|\xi = \mathbf{a}) f_X(x, \mathbf{y}; \theta_X) d\mathbf{y} \\
&= \int \Pr(\wp(X) = \mathbf{y}|\xi = \mathbf{a}) \frac{\partial f_X(x, \mathbf{y}; \theta_X)}{\partial \theta_X} d\mathbf{y} \\
&\approx \sum_{i=1}^{S} \frac{\mathfrak{n}_S(\mathbf{a}, \mathbf{y}_i)}{\mathfrak{n}_S(\mathbf{a})} \frac{\partial f_X(x, \mathbf{y}_i; \theta_X)}{\partial \theta_X}. \qquad (4.2)
\end{aligned}$$

Here, S samples are drawn from a variable(s) Z such that $\mathfrak{n}_S(j)$ is the number of times the value j appears in the set of samples $\{z_i\}$, i.e., $\mathfrak{n}_S(j) = \sum_{i=1}^{S} \mathbb{1}\{z_i = j\}$. Next,

$$\begin{aligned}
\nabla_{\Theta_{BN} \setminus \theta_X} \Pr(X = x|\xi = \mathbf{a}) &= \nabla_{\Theta_{BN} \setminus \theta_X} \int \Pr(\wp(X) = \mathbf{y}|\xi = \mathbf{a}) \times \Pr(X = x|\wp(X) = \mathbf{y}, \xi = \mathbf{a}) d\mathbf{y} \\
&= \int f_X(x, \mathbf{y}; \theta_X) \nabla_{\Theta_{BN} \setminus \theta_X} \Pr(\wp(X) = \mathbf{y}|\xi = \mathbf{a}) d\mathbf{y} \\
&\approx \sum_{i=1}^{S} \frac{\mathfrak{n}_S(\mathbf{y}_i)}{S} f_X(x, \mathbf{y}_i; \theta_X) \nabla_{\Theta_{BN} \setminus \theta_X} \Pr(\wp(X) = \mathbf{y}_i|\xi = \mathbf{a}) \qquad (4.3)
\end{aligned}$$

When $|\wp(X)| > 1$, variables in $\wp(X)$ might not be conditionally independent given $\Xi(X)$.

Hence we find a set of nodes N such that $I \perp J | \Xi(X) \cup N \ \forall I, J \in \wp(X)$. Then,

$$\begin{aligned}
\Pr(\wp(X) = \mathbf{y}_i | \xi = \mathbf{a}) &= \int \Pr(N = \mathbf{n} | \xi = \mathbf{a}) Pr(\wp(X) = \mathbf{y} | N = \mathbf{n}, \xi = \mathbf{a}) d\mathbf{n} \\
&= \int \Pr(N = \mathbf{n} | \xi = \mathbf{a}) \prod_{j=1}^{m} \Pr(P_j = y_j | N = \mathbf{n}, \xi = \mathbf{a}) d\mathbf{n} \\
&\approx \sum_{k=1}^{S} \frac{\mathfrak{n}_S(\mathbf{a}, \mathbf{n}_i)}{\mathfrak{n}_S(\mathbf{a})} \prod_{j=1}^{m} \Pr(P_j = y_j | N = \mathbf{n}_k, \xi = \mathbf{a}),
\end{aligned} \tag{4.4}$$

where $\wp(X) = (P_1, \ldots, P_m)$ and $\mathbf{y}_i = (y_{i,1}, \ldots, y_{i,m})$. Thus, we obtain,

$$\begin{aligned}
\nabla_{\Theta_{BN} \setminus \theta_X} \Pr(\wp(X) = \mathbf{y}_i | \xi = \mathbf{a}) &\approx \sum_{k=1}^{S} \frac{\mathfrak{n}_S(\mathbf{a}, \mathbf{n}_k)}{\mathfrak{n}_S(\mathbf{a})} \times \nabla_{\Theta_{BN} \setminus \theta_X} \prod_{j=1}^{m} \Pr(P_j = y_{i,j} | N = \mathbf{n}_k, \xi = \mathbf{a}) \\
&= \sum_{k=1}^{S} \frac{\mathfrak{n}_S(\mathbf{a}, \mathbf{n}_k)}{\mathfrak{n}_S(\mathbf{a})} \times \\
&\qquad \sum_{l=1}^{m} \left(\prod_{h=1, h \neq l}^{m} \Pr(P_h = y_{i,h} | N = \mathbf{n_k}, \xi = \mathbf{a}) \right) \times \\
&\qquad\qquad \nabla_{\Theta_{BN} \setminus \theta_X} \Pr(P_l = y_{i,l} | N = \mathbf{n}_k, \xi = \mathbf{a}) \\
&\approx \sum_{k=1}^{S} \frac{\mathfrak{n}_S(\mathbf{a}, \mathbf{n}_k)}{\mathfrak{n}_S(\mathbf{a})} \sum_{l=1}^{m} \left(\prod_{h=1, h \neq l}^{m} \frac{\mathfrak{n}_S(y_{i,h}, a, \mathbf{n}_k)}{\mathfrak{n}_S(a, \mathbf{n}_k)} \right) \times \\
&\qquad \overbrace{\nabla_{\Theta_{BN} \setminus \theta_X} \Pr(P_l = y_{i,l} | N = \mathbf{n}_k, \xi = \mathbf{a})}^{\text{Expand by recursion using (4.2), (4.3) and (4.5)}}.
\end{aligned} \tag{4.5}$$

The term $\nabla_{\Theta_{BN} \setminus \theta_X} \Pr(P_l = y_{i,l} | N = \mathbf{n}_k, \xi = \mathbf{a})$ represents the gradient operator on a subset of the original BN, containing only the ancestors (from the BN's graphical structure) of X. Hence that gradient term can be recursively expanded using (4.2), (4.3) and (4.5). Repeating that process for all variables in $\hat{b}_t$ allows us to calculate the $\nabla_{\Theta_{BN}} \hat{b}_t$.

Computational Complexity. The cost of computing (4.2) and (4.3) is $O(S)$. The cost of computing (4.5) is $O(mS)$. The cost of finding N is $O(|\wp(s_t)|^2(|V| + |E|))$ (i.e., the cost of running the Bayes ball algorithm [138] for every pair of nodes in $\wp(X)$). The total computational complexity of the entire procedure hinges on finding the number of times (4.2), (4.4) and (4.5) are executed, which we refer to as Q. Q depends on the size of N and on the graphical structure of the BN. Hence, the total cost of computing $\nabla_{\Theta_{BN}} \hat{b}_t$ is $O(Q(|\wp(s_t)|^2(|V| + |E|) + mS))$ (where $|\wp(s_t)| \leq |V| - 1$), which is computed $n_s n_e |b_t|$ times during training. Note that for a polytree BN (the graphical structure of the BN we will use

in §4.4), $N = \varnothing$, and $Q \leq |V|$. This is still better than belief propagation on the polytree with the gradient computation technique from [127, 128], which is $O(|V| \max_{v \in V}(dom(X_v)))$, where $dom(X)$ is the size of the domain of X, which could be exponentially large.

4.4 SCHEDULING DATA CENTER WORKLOADS BY USING REINFORCEMENT LEARNING

We now demonstrate an application of the POMDP model and training methodology presented in §4.3 to the problem of scheduling tasks on a heterogeneous processing fabric that includes CPUs, GPUs, and FPGAs. The model integrates real-time performance measurements, prior knowledge about workloads, and system architecture to (i) dynamically infer system state (i.e., resource utilization), and (ii) automatically schedule tasks on a heterogeneous processing fabric.

4.4.1 Workload & Programming Model.

The system workload consists of multiple user programs, and each program is expressed as a *data flow graph* (DFG). A DFG is a DAG where the nodes represent computations (which we refer to as *kernels*, e.g., matrix multiplication), and edges represent input-output relationships between the nodes. Prior work has shown that a large number of applications can be expressed as compositions of such kernels [139, 140]. Prominent examples of such compositions include modern data analytics and ML frameworks that describe workloads as DFGs [141, 142, 143, 144]. We assume that the kernels are known ahead of time and have multiple implementations available for different processors and accelerators. That assumption is correct for many ML workloads; for other workloads, it is an area of active research wherein accelerator designers and architects are trying to decompose larger applications into smaller pieces. Once trained, our approach can schedule any composition (DFG) of the kernels, but requires retraining when the set of available kernels change.

4.4.2 POMDP Architecture.

The overall architecture of the Symphony POMDP model is illustrated in Fig. 4.3. The framework functions as follows.

1. The scheduler first makes measurements by using the available processor performance counters (e.g., instructions retired, cache misses).

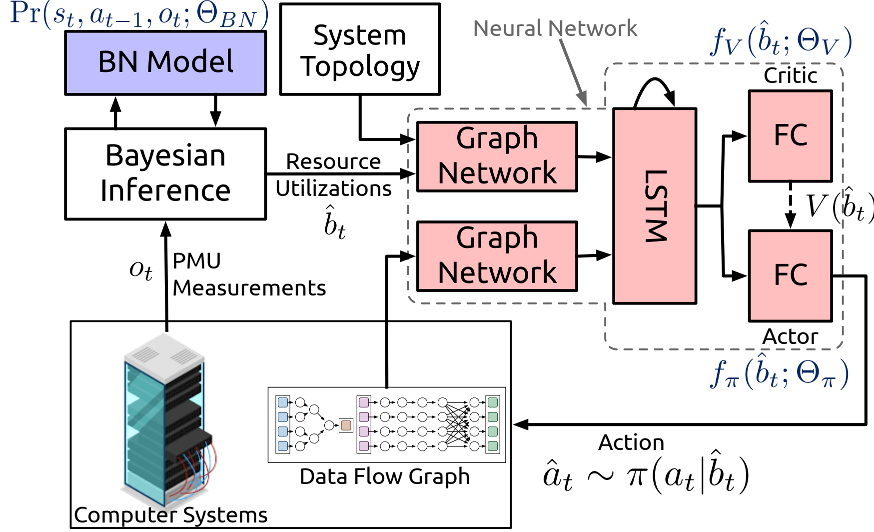


Figure 4.3: Architecture of the Symphony ML model.

2. When a processor becomes idle (finishes running the current kernel), it invokes the scheduler.

3. The measurements are fed into the scheduler's POMDP model as input. Using those measurements, the BN model computes the utilization of different levels of architectural resources in the system (e.g., memory bandwidth utilization, PCIe link utilization). We refer to those utilizations as the *state* of the system.

4. The computed utilization numbers, user programs represented as a DFG, and a system topology graph are fed into an NN. The NN produces a scheduling decision that is actuated in the system. The action space consists of a kernel-processor pair.

5. Finally, the scheduler gets feedback from the system (i.e., the reward) in terms of the time it took for the job to run as a result of its scheduling decision.

6. While in *training mode*, if an incorrect decision is made, Symphony enqueues an update of the policy parameters using back-propagation on the A2C/A3C loss function. An incorrect decision is one where kernel input-output dependencies are not respected, or a kernel-accelerator pair is picked where the accelerator does not provide an implementation of the kernel.

The first part of the POMDP models the latent state $\hat{b}_t$ of the computer system. For the scheduling problem, $\hat{b}_t$ corresponds to resource utilization of various components of the computer system. Utilization of some of the resources can be measured directly in software (e.g., the amount of free memory); however, the different layers of abstraction of the computer

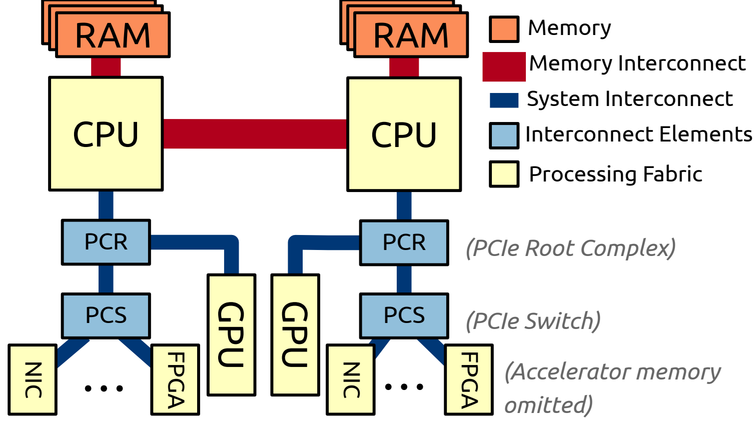


Figure 4.4: Organization of a multi-CPU computer.

stack hide some others from direct measurement. For example, consider the example in Fig. 4.1 in §4.1; here, PCIe link (see Fig. 4.4 for PCIe network organization) bandwidth cannot be directly measured. However, it can be measured indirectly by using the number of outstanding requests to memory from each PCIe device and by using the topology of the PCIe network. In essence, we statistically relate the back pressure of one resource on another, until we can find a resource that can be directly measured via real-time performance counter (PC) measurements (o_t) [125]. We refer to such resources whose utilization cannot be directly measured as *hidden* resources. PCs are special-purpose registers present in the CPU and other accelerators for characterization of an application's behavior and identification of microarchitectural performance bottlenecks. Specifically, we use a BN to (i) model aleatoric uncertainty in measurements, and (ii) encode our knowledge about system architecture in terms of invariants or statistical relationships between the measurements. Inference on that BN then gives us an accurate estimate of the latent state of the system. Second, we use an RNN (i.e., $f_\pi(\cdot)$ and $f_V(\cdot)$) to learn scheduling policies for user programs that minimize resource contention and maximize performance. Those two ML models effectively decouple system-architecture-specific and measurement-specific aspects of scheduling (the BN) from its optimization aspects (the NN). The compelling value of the above architecture (and its two constituent models) is that it can automatically generate scheduling policies for the deployment of DFGs in truly heterogeneous environments (that have CPUs, GPUs, and FPGAs) without requiring configuration specifics, or painstakingly tuned heuristics. The model improves overall performance and resource utilization, and enables fine-grained resource sharing across workloads.

Performance Counters. PCs are generally relied upon to conduct low-level performance analysis or tuning of performance bottlenecks in applications. As the source of such bottlenecks

is generally the unavailability of system resources, the performance counter can naturally be used to estimate resource utilization of a system. Another benefit of using PCs is that it is not necessary to modify an application's source code in order to make measurements. PCs can be grouped into three categories: (i) those pertaining to the processing fabric (CPU core or accelerators); (ii) those pertaining to the memory subsystem; and (iii) those pertaining to the system interconnect (in our case, PCIe). Figs. 4.4 and 4.5 illustrates the organization of a computer system as well as the categories above. Fig. 4.6 shows a mapping between the system organization and the PCs that are used in the BN model (described below).[2]

Performance counters' configuration and access instructions require kernel mode privileges, and hence those operations are supported by Linux: system calls to configure and read the performance counter data. Symphony uses a combination of user-space tools, e.g., libPAPI [145], PMUTools [146], and perf that wrap around the system call interface to make both system-specific and system-independent measurements. We configure the performance counters to make system-wide measurements (i.e., for all processes). All kernel (i.e., computation) executions are non-preemptive in the context of the proposed runtime, however the OS scheduler can preempt CPU threads. Further we prevent the OS scheduler from re-balance tasks/threads once assigned to a particular CPU, so as to ensure re-balancing actions happen only through the runtime. This is achieved by explicitly setting affinities of threads to cores (i.e., pinning them).

Monitoring of performance counters without having to perform interrupts is almost free. In our implementation, we capture on-core performance counters directly before and after a single kernel invocation. Un-core performance counters are measured periodically (every million dynamic instructions on a core) by using a *performance monitoring interrupt*. On an IBM PowerPC processor, the interrupt handler initiates a DMA transfer of the performance counters to memory [147], thereby incurring no performance penalty (other than the time to service the interrupt). On Intel processors, the interrupt handler has to explicitly read the performance counter registers and write them to memory. In our tests (on Intel processors), we observed a $\sim$3% performance penalty for applications with interrupts enabled. That corresponds to an execution of a usermode interrupt with an average 900-ns latency.

Topology Information. Consider the example of standard NUMA based computing system with PCIe based accelerators shown in Figs. 4.4 and 4.5. The system contains (i) multiple CPUs which have non-uniform access to memory, (ii) several accelerators (including GPUs and FPGAs) each with their own memory, and (iii) a system interconnect which connects all of the components of the system together. Symphony encodes the system topology

[2]A complete list of the PCs used in this chapter can be found in the supplementary material.

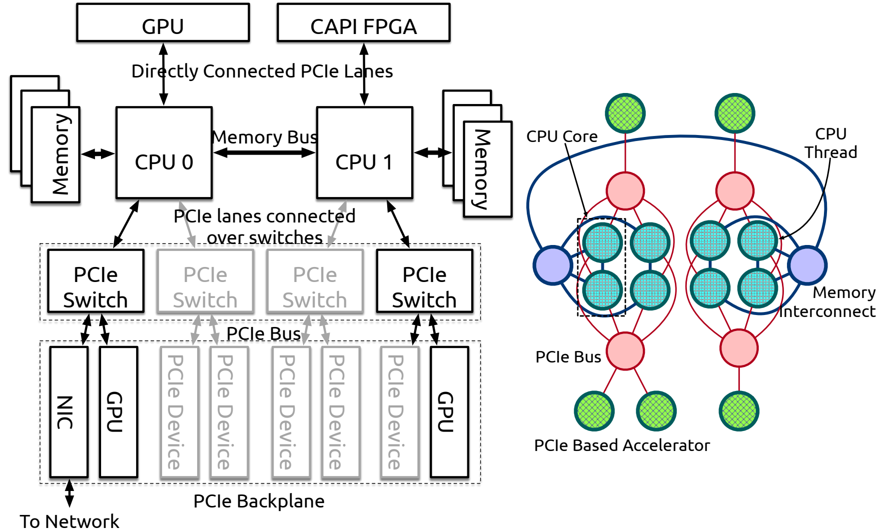


Figure 4.5: Example of a dual-socket NUMA-based system topology with a PCIe-interconnect and -devices. Figure on the right shows an graph-encoding of the topology.

as a graph $T = (P, N)$ (also shown in Fig. 4.5). The nodes of the graph P correspond to the processing elements (and attached memory) and memory/system interconnects. Each of the these nodes $p \in P$ have an attached resource utilization vector. For example, in an Intel processor, the utilization vector would include utilization like that of micro-op issue ports, floating point unit utilization etc. [89, 148].

The scheduler queries the system topology and builds the topology graph T (which is used as an input to the RL agent) using hwloc [149]. hwloc provides information about CPU cores, caches, NUMA memory nodes, and the PCIe interconnect layout (i.e., connections between the PCIe root complex and PCIe switches), as well as connection information on peripheral accelerators, storage, and network devices in the system. The scheduler does not explicitly model the rack-scale or data center network (unlike some previous approaches, e.g., [28, 150]), but the BN and RL model can be extended to do so. Our measurements considers injection bandwidth at the network interface card (NIC) to be a proxy for network performance, i.e., the NIC is modeled as an accelerator that accepts data at min(PCIe Bandwidth, Injection Bandwidth).

BN Model (see Fig. 4.6)**.** Measurements made from PCs have some inherent noise [71]. The measurements can only be stored in a fixed number of registers. Hence, only a fixed number of measurements can be made at any one point in time. As a result, one must make successive measurements that capture marginally different system states. Particular performance counters might become unavailable (or return incorrect values). Finally, if a single scheduling agent is controlling a cluster of machines (which is common in data centers),

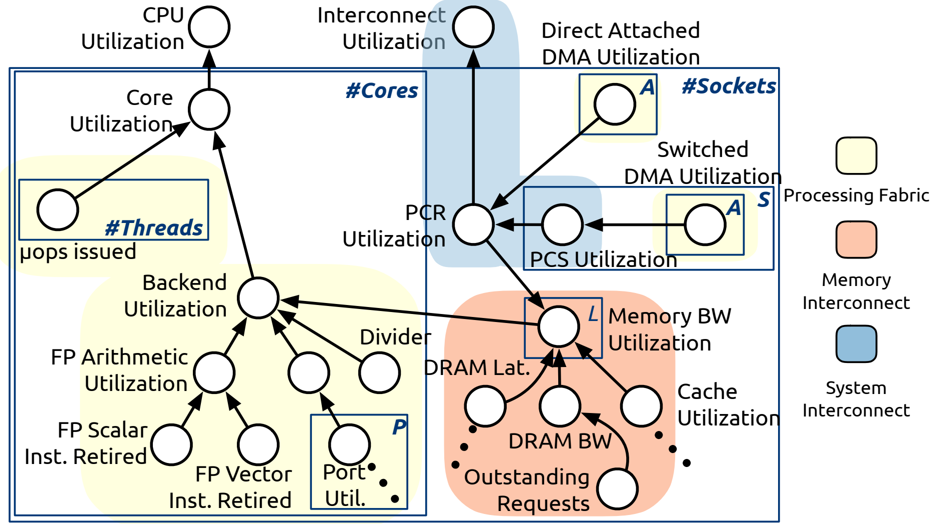


Figure 4.6: Bayesian network (uses the plate notation) used to estimate resource utilization.

measurements made on different machines will not be in sync and will often be delayed by network latency. As a result, PCs are often sampled N times between successive scheduler invocations to get around some of the sources of error. To maximize the performance estimation fidelity, we apply statistical methods to systematically model the variance of the measurements. For a single performance counter $o_t[c]$, if the error in measurement e_c can be modeled, then the measured value m_c can be modeled in terms of the true value v_c plus measurement noise e_c, i.e., $m_c = v_c + e_c$. Here, we focus only on random errors, and assume zero systematic error. That is a valid assumption because the only reason for systematic errors is hardware or software bugs. We assume that the error can be modeled as $e_c \sim \mathcal{N}(0, \sigma)$ for some unknown variance σ; hence, $\Pr(m_c \mid v_c) = \mathcal{N}(m_c, \sigma)$. That follows from prior work based on extensive measurement studies [71]. Now, given N measurements of the value of the performance counter, we compute their sample mean μ and sample variance S. A scaled and shifted t-distribution describes the marginal distribution of the unknown mean of a Gaussian, when the dependence on variance has been marginalized out [100]; i.e., $v_c \sim \mu + {}^{S}/_{\sqrt{N}}\ \mathit{Student}(\nu = N - 1)$. In our experiments, the confidence level of the t-distribution was 95%.

Now, given a distribution of v_c for every element of o_t, we describe the construction of the BN model. Our goal is to model resource utilization (a number in $[0, 1]$) for a relevant set of architectural resources R. To do so, we use algebraic models for composing PC measurements (v_c) by using algebraic (deterministic) relationships derived from information about the CPU architecture. Processor performance manuals [86, 87, 97] and or vendor contributions in OS codebases (e.g., in the `perf` module in Linux) provide such information. When available in

the later format (which is indeed the case for all modern Intel, AMD, ARM, and IBM CPUs), these relationships can be automatically parsed and be used to construct the BN.

As our error-corrected measurements are defined in terms of distributions, the algebraic models that encode static information about relationships (based on the microarchitecture of the processor or topology of the system) now define statistical relationships v_cs (based on the Jacobian relationships described in §4.3). Fig. 4.6 shows an example of the BN model. However, the types and meanings of hardware counters vary from one kind of architecture to another because of the variation in hardware organizations. As a result, the model defined by the BN is parametric, changing with different processors and system topologies (i.e., across all the different types of systems in a data center).

Consider the example of identifying memory bandwidth utilization for a CPU core. According to the processor documentation, the utilization can be computed by measuring the number of outstanding memory requests (which is available as a PC), i.e.,

$$\frac{\text{Outstanding Requests}[\geq \theta_{MB}]}{\text{Outstanding Requests}[\geq 1]}.^{3} \tag{4.6}$$

That is, identify the fraction of cycles in some time window that CPU-core stalls because of insufficient bandwidth. Naturally, in order to sustain maximum performance, it is necessary to ensure that no stalls occur. The value θ_{MB} is processor-specific and might not always be known. In such cases, we use the training approach described in §4.3 to learn θ_{MB}. The procedure is repeated for all relevant system utilization counters (marked as "Util." in Fig. 4.6), which together represent $\hat{b}_t$. Such a BN model for a 16-core Intel Xeon processor (with all PCIe lanes populated) has 68 nodes, of which 32 are directly measured and the remainder are computed through inference.

BN Retraining. The architectural information required to build the BN can be found in processor manuals [86, 87, 147] as well as in machine-parsable databases in the Linux kernel source code as part of the `perf` package. The only human intervention required in the process of building the BN is for filtering out those resources that cannot be controlled with software (because they change too quickly). The BN model should only be rebuilt when the underlying hardware configuration changes, which [32] observe happens every 5–6 years in a data center.

Implementation Details. We collect system-wide (for all processes) performance counter measurements for a variety of hardware events (described in Table 4.1). The system wide collection leads to occasional spurious measurements (e.g., from interrupt handlers), however,

[3] Here X[$\geq t$] counts cycles in which X exceeds threshold t.

Table 4.1: Performance counters used in test evaluation. We have disambiguated the names to ensure platform independence.

Performance Counters/Events
On-core Events
`Core Clock Cycles, Reference Clock Cycles, Temperature, Instructions (µops for Intel) issued, Instructions (µops for Intel) retired, Un-utilized slots due to miss-speculation`
Un-core & Memory Controller Events (per socket)
`#Read/Write requests to DRAM (from all channels), #Local DRAM accesses, #Remote DRAM Accesses, #Read/Write requests to DRAM (from all channels) from IO sources, #PCIe Read, #PCIe Write, QPI(for Intel)/Nest(for IBM) Transactions`
OS/Driver Events
`Free memory (CPU, GPU, FPGA), Total memory (CPU, GPU, FPGA)`

this allows us to make holistic measurements (e.g., capture system calls or drivers that perform memory and DMA operations). We make the minimum measurements to infer if a kernel scheduled to a CPU-hardware thread is core-bound (floating point- and integer-intensive). This allows us to make scheduling decisions on co-located kernels, i.e., those that get scheduled to SMT/hyperthreads bound to a core. The majority of measurements are made at the level of un-core events that captures performance of the memory interconnect and the system bus: to identify kernels that are bandwidth bottle necked. We do not explicitly model GPU performance counters as low-level scheduling decisions (e.g., warp-level scheduling) in GPUs are obfuscated by the NVIDIA runtime/driver.

NN Model. The second part of the POMDP-based scheduling model uses an NN (see Fig. 4.3) to learn the optimal policy with which to schedule user tasks given a belief state. The NN takes two graphs as inputs. The first input is the belief state $\hat{b}_t$, encoded as vertex labels on a graph that describes the topology of a computer system (i.e., the organization shown in Fig. 4.4), and input labels that correspond to the locations of inputs in the topology. The color coding in Figs. 4.4 and 4.6 shows a mapping (i.e., vertex labels) between nodes in the topology graph and $\hat{b}_t$. The second input is the user's program expressed as a DFG. We use *graph network* (GN) layers [68] to "embed" the graphs into a set of *embedding vectors*. GNs have been shown to capture node, edge, and locality information. We chose small, fully connected NNs for modeling the functional transformations in the GN layers. Prior work in scheduling (e.g., [151, 152]) has shown the benefit of considering temporal information to capture the dependencies of system resources over time as well as the time evolution of the user DFG. We capture those relationships (between the embeddings of the input graphs) by using an RNN, specifically an LSTM layer [153].

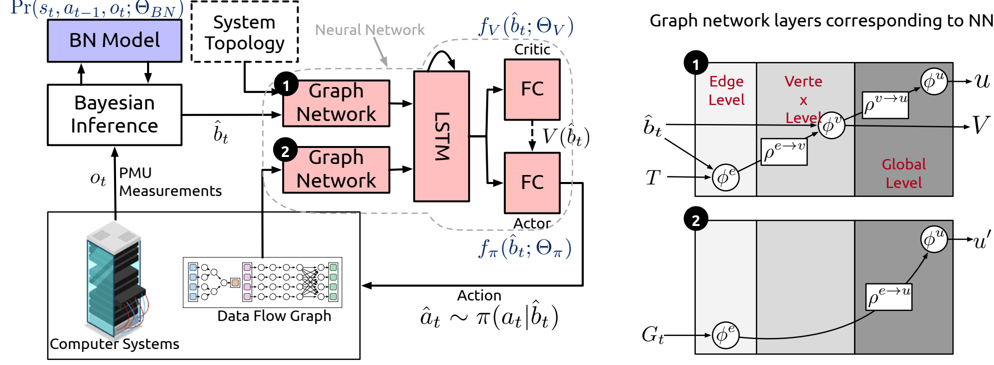


Figure 4.7: Structure of the graph networks used in Fig. 4.3.

Table 4.2: Mapping of the graph network layer functions in Fig. 4.7. We use the notation $FCNN(a, b)$ to denote a 2-hidden fully-connected layers with a and b hidden units, respectively.

Function in GN	Function in 1	Function in 2
ϕ^e	$FCNN(64, 32)$	$FCNN(64, 32)$
ϕ^v	$FCNN(32, 16)$	–
ϕ^u	$FCNN(16, 16)$	$FCNN(32, 16)$
$\rho^{e\to v}$	$\sum e$	–
$\rho^{v\to u}$	$\sum v$	–
$\rho^{e\to u}$	–	$ReLU(e)$

The action space $\mathcal{A}$ of the model is fixed as the number of kernels/processors available in the system and is known ahead of time. The action space consists of the following types of actions. (i) *Execution actions* correspond to execution of a kernel on a processor/accelerator. (ii) *Reconfiguration actions* correspond to reconfiguration of a single FPGA context to a kernel. (iii) *No-Op actions* correspond to not scheduling any task in a particular scheduler invocation. No-Ops are useful when the system resources are maximally subscribed, and execution of more tasks will hinder performance. The scheduler is invoked every time there is an idle processor/accelerator in the system (i.e., every time a processor finishes the work assigned to it), causing the system to take one of the above actions.

The structure of the graph network used in the proposed model is illustrated in Fig. 4.7. The numbers of parameters used in the different layers of the graph network are listed in Table 4.2.

Reward Function. The reward r_t is based on the objective of minimizing the runtime of a user DFG. At time t, $r_t = -\sum_{i=0}^{t} {}^{1}\!/_{T_i}$, where T_i is the wall clock time taken to execute the i actions executing in the system at time t. We picked r_t as it represents the "makespan"

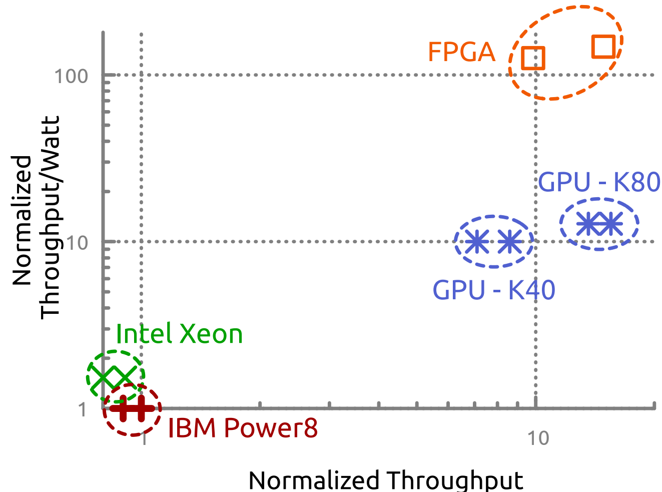


Figure 4.8: Architectural diversity leading to varied performance for the PairHMM kernel.

of the schedule, a metric that is popularly used in the traditional scheduling literature and accurately represents the user-facing performance of the system. Note that parallel actions are not double-counted in this formulation. The BN and NN models are trained end-to-end using minimization of (4.1) through back-propagation, as described in §4.3.

4.5 APPLICATION SPECIFIC FEATURES FOR SCHEDULING POLICIES

Current schedulers prioritize the use of simple online heuristics [151] and coarse-grained resource bucketing (e.g., core counts, free memory) and require user labeling of commonly used system resources [154, 155] to make scheduling decisions. Those approaches are untenable in truly heterogeneous settings as (i) defining such heuristics is difficult over the combinatorial space of application-processor/accelerator configurations; and (ii) user-based resource usage labeling requires in-depth understanding of the underlying system. This section demonstrates the use of ML to automatically infer such heuristics and their evolution over time as new user workloads and/or new accelerators are added.

4.5.1 Dealing with Architectural Heterogeneity

We reiterate that state-of-the-art schedulers do not model the emergent heterogeneous compute platforms that are being widely adopted in data centers and hence leave a lot to be desired (as can also be seen in the performance of our baselines). Consider, for example, the execution of the *forward algorithm on PairHMM models* [12], a computation that is commonly performed in computational genomics workloads. Fig. 4.8 shows the significant diversity

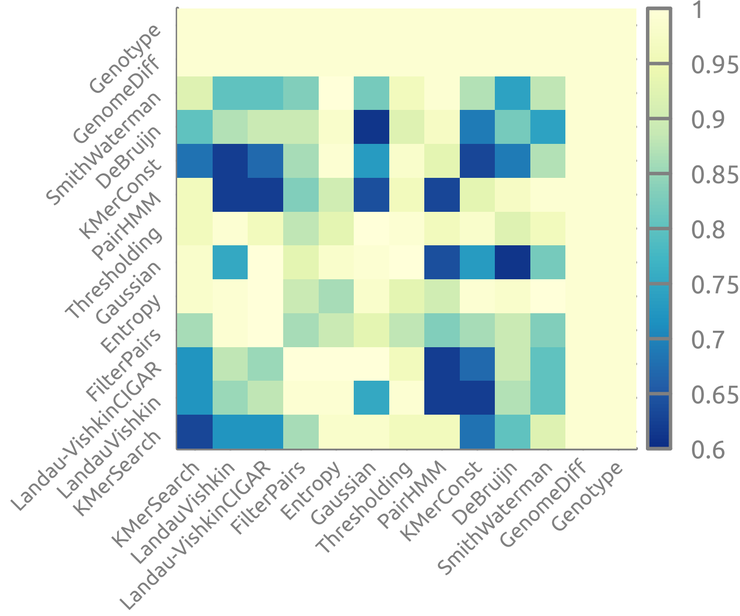


Figure 4.9: Degradation in runtime of co-located kernels due to shared resource contention.

(nearly 100×) in performance of this single workload across CPUs (from Intel and IBM), GPUs (two models of GPUs from NVIDIA) and FPGA implementations. The increasing heterogeneity necessitates rethinking of the design and implementation of future schedulers, as the current approach will require an extraordinary amount of manual tuning and expertise to adapt to the emergent systems. In contrast, the proposed technique eliminates that work and automates the whole process of learning the right granularity of resources and scheduling workloads in cloud-based, dynamic, multi-tenant environments, thereby improving application performance and system utilization, all with minimal human supervision. Prior work uses microarchitectural throughput metrics such as clock cycles per instruction [29, 31, 32, 39, 40] as proxies for processor affinities. In our case, such metrics are not usable because of the wide diversity in processors, i.e., CPU-centric units cannot describe the performance of GPUs/FPGAs.

4.5.2 Dealing with Low-level Resource Contention

Traditional schedulers use coarse-grained resource bucketing, i.e., they schedule macro-resources like CPU core counts and GBs of memory. That simplifies the design of the scheduling algorithms (both the optimization algorithms and attached heuristics), resulting in an inability to measure low-level sources of resource contention in the system. The contention of such low-level resources is often the cause for performance degradation and variability.

Table 4.3: Hardware specifications of test cluster.

Type	#	Specifications
`M1`	2	CPU IBM Power8 (SMT 8); 870 GB RAM; GPU NVIDIA K80; FPGA Alpha Data 7V3
`M2`	4	CPU IBM Power8 (SMT 4); 512 GB RAM; GPU NVIDIA K40; FPGA Nallatech 385
`N`	1	Mellanox FDR Infiniband

Consider, for example, the concurrent execution of several compute kernels on co-located hyper-threads (i.e., threads that share resources on a single core) on an Intel CPU. If we abstract the problem at the level of CPU threads and memory allocated, then those kernels should execute in isolation. The normalized runtime variation is illustrated in Fig. 4.9. We observe a slowdown of as much as 40% (i.e., the co-located runtime is 60% of the isolated runtime) for some combinations of kernels, and almost no slowdown for others. That problem is further exacerbated by the architectural diversity in processors that we described earlier. The proposed technique accounts for such contention by explicitly collecting information on low-level system state by using performance counter measurements, and by estimating resource usage in the system by explicitly encoding the measurements in its POMDP model.

4.6 EVALUATION & DISCUSSION

We evaluated the Symphony along the following dimensions. (i) *How well does Symphony perform compared to the state of the art?* (ii) *How does the Symphony's runtime affect scheduling decisions?* (iii) *What are the savings in training time compared to traditional methods?* The evaluation testbed consisted of a rack-scale cluster of twelve IBM Power8 CPUs, two NVIDIA K40, six K80 GPUs, and two FPGAs (listed in Table 4.3). We illustrated the generality of techniques on a variety of real-world workloads that used CPUs, GPUs, and FPGAs: (i) *variant calling and genotyping analysis* [156] on human genome datasets using tools presented in [12, 13, 140, 157, 158, 159, 160, 161, 162, 163, 164]; (ii) *epilepsy detection and localization* [165] on intra-cranial electroencephalography data; and (iii) in online *security analytics* [166] for intrusion detection systems. Table 4.4 lists the exact implementations of these workloads.

State of the Art. Traditional dynamic scheduling techniques (e.g., [28, 29, 30, 33, 37, 38]) use manually tuned heuristics (e.g., fairness, shortest-job-first) that prioritize simplicity and generality over achieving the best-case workload performance, often allocating coarse-grained

Table 4.4: Enumeration of workloads used to evaluate Symphony.

Application	Processors			Implementations
	CPU	GPU	FPGA	
Alignment *(Align)*	✓	✓	✓	[13, 140, 157, 158, 159, 164],
Indel Realignment *(IR)*	✓	✗	✗	[160, 161]
Variant Calling *(HC)*	✓	✓	✓	[12, 160, 162, 163, 167]
EEG-Graph *(EEG)*	✓	✓	✓	[165, 168]
AttackTagger *(AT)*	✓	✓	✓	[166, 168]

resources (e.g., GBs of memory, CPU threads) and making simplifying assumptions about the underlying workload. Several ML-based scheduling strategies have also been proposed, wherein the above heuristics are learned from data. They use a variety of black-box ML models, e.g., model-free deep RL in [41, 42], collaborative filtering [39, 40], and other traditional ML techniques like SVMs (e.g., [31, 32, 35, 36]). A common theme in these techniques is that of treating the system as a black-box and performing scheduling to optimize application throughput metrics. The above approaches are not well-suited to heterogeneous, accelerator-rich systems in which architectural diversity necessitates the use of low-level resources, which cannot be measured directly and are not semantically comparable across processors. As points of comparison to Symphony, we used *Graphene* [151], a heuristic-accelerated job shop optimization solver[4]; *Sparrow* [33], a randomized scheduler; and *Paragon* [39], a collaborative filtering-based scheduler.

Baseline for Comparison. We defined the *oracle schedule* to correspond to the best performance possible for running an application on the evaluation system. It corresponds to a completely isolated execution of an application. Here, different concurrently executing kernels of the same application contend for resources and might cause performance degradation. For the benchmark applications, we accounted for that by exhaustively executing schedules of the application DFGs to find the one with the lowest runtime (i.e., the *oracle run*). We measured the runtime of kernel i in workload (in the oracle run) j as $t_{i,j}^{\text{oracle}}$ across all kernels and workloads. $t_{i,j}^{\text{oracle}}$ serves as the baseline for assessing the performance of Symphony.

Effectiveness of Scheduling Model. First, we quantified how well Symphony can handle scheduling of kernels in a DFG taking into account of resource contention and interference at (i) intra-DFG level; and (ii) when executing with an unknown co-located workload utilizing compute and I/O resources. To do so, we measured the runtimes of each of the kernels i in the workload j (as above) to compute $t_{i,j}^{s}$ for each scheduler s under test.

[4]Graphene was not originally designed to execute on heterogeneous systems. In the supplementary material, we explain modifications we made to the algorithm.

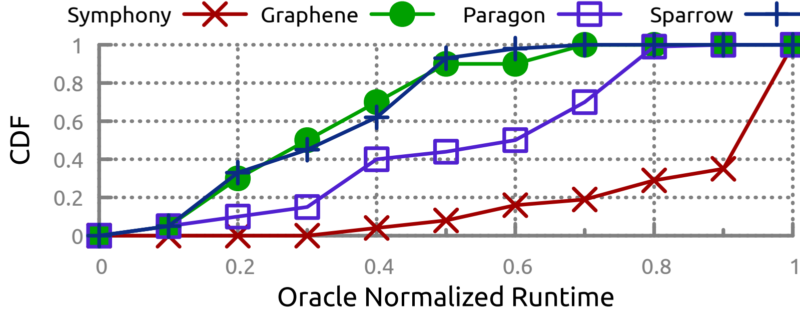


Figure 4.10: Comparing performance of Symphony to that of other popular schedulers for kernel executions in DFGs.

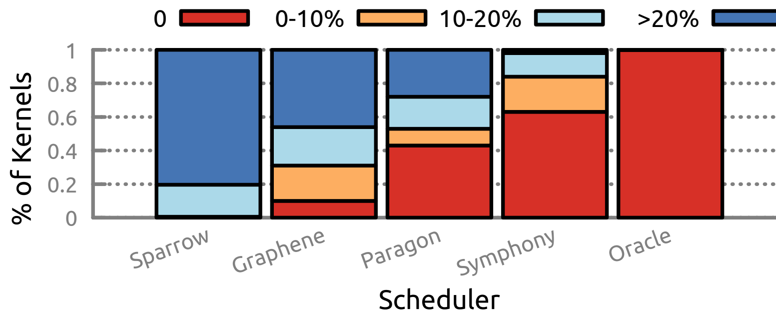


Figure 4.11: Percentage of application executions that show a degradation in performance.

In Fig. 4.10, we illustrate the distribution of oracle-normalized runtimes for each of the kernels in the workloads we tested, i.e., a distribution of $t^s_{i,j}/t^{\text{oracle}}_{i,j}$ across 500 executions of the three above workloads. In the figure, a distribution whose probability mass is closest to 1 is preferred, as it implies the least slowdown compared to the oracle. We observe that the proposed technique significantly outperformed the state-of-the-art. In our experiments, the median and tail (i.e., 99$^{\text{th}}$ percentile) runtime of Symphony outperformed the second best (in this case, Paragon) by close to 32%. At the 99$^{\text{th}}$ percentile, the generated schedules performed at a 6% loss relative to the oracle. Next, we quantified the performance of end-to-end user workloads, shown in Fig. 4.11. Here, we calculated $1 - (\sum_i t^s_{i,j})/(\sum_i t^{\text{oracle}}_{i,j})$ for all 500 runs of the DFGs and grouped them into buckets of different kinds of normalized performance. Symphony significantly outperformed the other scheduling techniques, running up to 60% of the applications with no performance loss relative to the oracle execution, and the rest with a performance loss of less than 20%.

Latency. There are two latencies to consider in comparing schedulers: the latency of the entire user workload ("LW", shown in Fig. 4.10), and the latency of the scheduler execution ("LS", shown in Fig. 4.12). In Fig. 4.12, we show two configurations of the Symphony

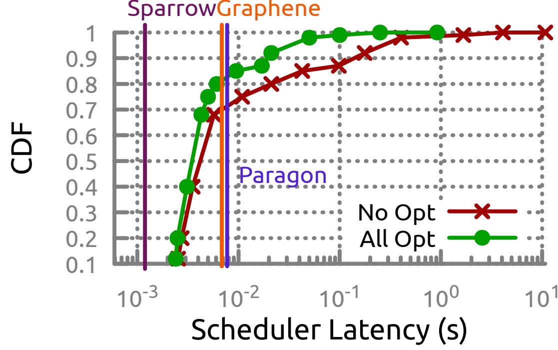


Figure 4.12: Symphony's latency ("All Opt" & "No Opt") compared to prior work.

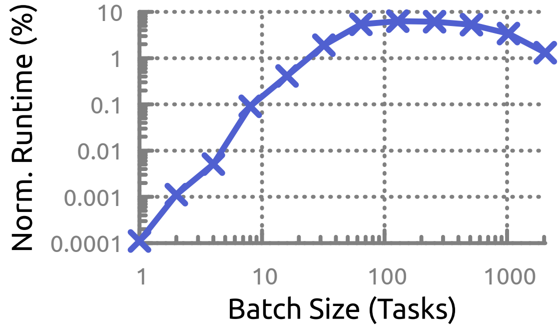


Figure 4.13: Symphony's performance (oracle normalized, in %) with varying batch size.

scheduler: (i) "No-Opt" which uses a belief propagation-based update for the BN (and MCMC-based inference); and (ii) "All-Opt" which uses the sampling technique described in §4.3, accelerators[5] to perform inference, and *task batching* (described below). LW ($\geq$ LS) is the user-facing metric of interest. Symphony outperforms all baselines in terms of LW. In terms of median LS, the Symphony is 1.8× and 1.6× faster than Paragon and Graphene, respectively. In contrast, Sparrow, which randomly assigns tasks to processors, has 3.6× lower median latency than Symphony. However, the reduced LS comes at the cost of increased LW (see Fig. 4.10).

Batching Task Execution. A key concern with Symphony is its large tail latency (100× larger than its median; see Fig. 4.12) compared to the other schedulers (which have deterministic runtime). This increased latency is brought about by Symphony having to perform significantly more compute if the RL-policy-update is triggered. The scheduler latency adversely affects LW as the time spent executing scheduler calls, is time not utilized to make progress on the user workload. In order to deal with this issue, our evaluation

[5]The accelerators include an NVIDIA K80 GPU for NN inference and an FPGA for BN inference using [168].

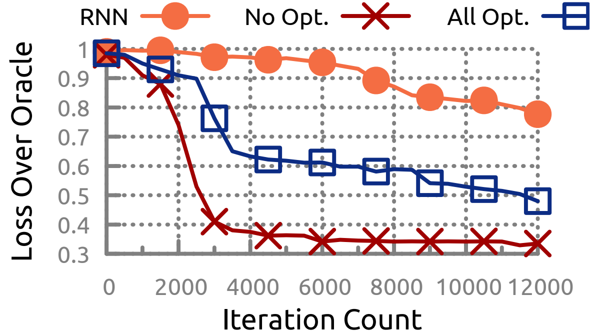


Figure 4.14: Training time for Symphony. An iteration is 2 RL episodes of 20 steps.

executed Symphony on batches of tasks instead of single tasks, thereby amortizing the cost of executing Symphony across the batch. Task batching works synergistically with the sampling based gradient propagation technique to reduce the tail latency by as much 12× (see Fig. 4.3). Fig. 4.13 demonstrates the average improvement in LW normalized to the oracle over a range of batch sizes. We observe that the optimal value for batch size is about 128 tasks per batch. This corresponds to the "All Opt" configuration in Figs. 4.12 and 4.14 as well as Figs. 4.10 and 4.11. The "No Opt" configuration in Fig. 4.12 is computed at a batch size of one.

Training Time. Finally, we quantified the improvement in training time offered by Symphony using the sampling-based gradient computation methodology presented in §4.3. We used the following baselines for evaluation: (i) model-free RNN (labeled "RNN" in Fig. 4.14); and (ii) the "All Opt." and "No Opt." configurations from above. The RNN model here replaces the BN (and inference) and system-topology-embedding GN (in Fig. 4.3) with a 3-layer, fully connected NN to compute an embedding for o_t. Fig. 4.14 illustrates the differences in performance of the these configurations with respect to degradation in performance of the user DFGs relative to the oracle schedule (i.e., $1 - {(\sum_i t^s_{i,j})}/{(\sum_i t^{\text{oracle}}_{i,j})}$). We observe that the RNN is significantly less sample-efficient than the proposed POMDP is; specifically, it is ∼2.2× worse than Symphony. Further linearly extrapolating time to convergence from iteration 12×10^3, the RNN would need $> 48 \times 10^3$ iterations to achieve the same accuracy as Symphony.

The difference in training time for the "No Opt." and "All Opt" in Fig. 4.14 can be attributed to (i) time taken to perform back-propagation for policy updates; and (ii) effective scheduler latency. Linearly extrapolating the training-loss, we observe that "All Opt" is at least 4.3× more sample efficient than "No Opt" to reach a 30% mean loss relative to the oracle. That reduction is significant because the continuous churn of user workloads and machine configurations in a cloud, as pointed out in [31], would require that the scheduling model be periodically retrained. In absolute terms, the "All Opt" configuration is able to

achieve ∼30% mean loss relative to the oracle scheduler in 700 hours of training and ∼4400 iterations of workload execution. That corresponds to approximately 500 hours of system execution; hence, the total process takes 1200 hours. Though this might appear to be over 7 weeks of time, in wall clock time this is approximately 2 week because we use parallel A3C-based training. In fact, the limiting factor here is the availability of FPGAs, of which we have only 2 in the evaluation cluster, hence limiting the number of RL episodes that can be run in parallel.

4.7 SUMMARY

This chapter presents (i) a domain-driven Bayesian RL model for scheduling that captures the statistical dependencies between architectural resources; and (ii) a sampling-based technique that allows the computation of gradients of a Bayesian model without performing full probabilistic inference. As data center architectures become more complex [10, 118], techniques like the one proposed here will be critical in the deployment of future accelerated applications.

CHAPTER 5: ACCELERATED AND REAL-TIME INFERENCE

5.1 INTRODUCTION

Many statistical- and machine-learning (ML) applications automatically detect patterns in data, and then use the uncovered patterns to predict future data, or to perform other kinds of decision-making under uncertainty. Probabilistic models (PMs; e.g., Markov models or Bayesian networks) and inference techniques have been shown to successfully integrate prior and structural relationships to quantify this uncertainty [99]. This allows PMs to naturally complement many ML methods (like deep learning [169]; DL) that (i) do not quantify uncertainty in their outputs [170], (ii) seldom produce interpretable results, and (iii) do not generalize well from small datasets or in cases with *class imbalance*. In fact, there are ongoing efforts in the ML community to combine PMs and DL to produce a *Bayesian DL* paradigm that can take advantage of both the flexibility of PMs in encoding model-related information (e.g., uncertainty, interpretability) with the immense scalability of DL [169].

Creation of optimized accelerators for DL models is well-developed [3, 141, 171]. The creation of accelerators that can execute infernence on PMs in real-time is substantially nonexistent, or is done only on a very problem-specific, hand-optimized basis [172, 173, 174, 175, 176, 177, 178, 179]. Development of such accelerators is the focus of this chapter. They will be fundamental not just to the addressing of PMs, but also to the integration PMs and DL.

Development of accelerators for execution of inference on PMs requires (i) a high-level language representation of PMs, and (ii) a method to map this representation into an architecture and correspondingly synthesized hardware that meets the real-time constraints. To address (1) above, we leverage prior work that proposes *probabilistic programming languages* (PPLs) [180] as a way to represent complex PMs as programs (e.g., [181, 182, 183, 184, 185, 186, 187, 188]).

This chapter addresses (2) above by proposing AcMC^2, a compiler that transforms general PMs expressed in a PPL into optimized hardware accelerators to infer query distributions (i.e., quantities of interest) over the posterior samples of a PM. Inference over PMs is analytically intractable in general [189]; therefore, we focus on methods that compute approximate answers, in particular the sampling-based Markov-Chain Monte Carlo (MCMC) methods. The crux of our approach is three fold. (i) We identify and accelerate common *computational kernels* used across multiple models. In the case of MCMC-based inference, that corresponds to the use of multiple types of random number generators. (ii) We use marginal and conditional

independences to maximally exploit the parallelism and data locality available in the structure of a PM for its inference. (iii) We integrate the above pieces with compositional MCMC techniques, i.e., where different variants of basic MCMC algorithms can be integrated together to solve an inference problem.[1] AcMC^2 then automatically generates HDL that corresponds to system-on-chip (SoC) components that can be integrated into CPU-based SoCs, FPGAs, or ASICs, which can then be used in both large servers and embedded devices.

Contributions. Our primary contributions are

1. We present a compiler workflow for generating hardware accelerators (both their architecture and implementation) from PMs described in PPLs. The compiler uses:
    (a) Conditional statistical independences (captured using *Markov blankets*) in a PM to generate maximally parallel, deeply pipelined, problem-specific random number generators.
    (b) Speculative execution to execute several independent MCMC chains in parallel on the generators above.
    (c) Bounded approximation techniques that reduce off-chip bandwidth for storage of intermediate results.
2. We describe an FPGA-based prototype (on a Xilinx Virtex 7 FPGA) for AcMC^2-generated accelerators that communicate to host CPUs (IBM POWER8) by using the CAPI interface [96].
3. We demonstrate AcMC^2's performance using a set of PPL micro-benchmarks. The AcMC^2-generated accelerators provided an average 46.8× improvement in runtime and a 753.5× improvement in terms of performance-per-watt over CPU-based software implementations.
4. We illustrate the generality of AcMC^2 by solving two real-world problems that require real-time analytics.
    (a) *Precision Medicine:* Identification of seizure-generating brain regions through analysis of electroencephalograms (EEGs) [165].
    (b) *Datacenter Security:* Detecting data-center-scale security incidents by analysis of alerts generated from network- and host-level security monitoring tools [166].

[1]For example, we use *Gibbs sampling* [190] for discrete variables, and *Hamiltonian Monte Carlo* (HMC) [191] for continuous variables when gradient information is available.

We demonstrate a $48.4 - 102.1\times$ improvement in performance over a CPU baseline; and a $8.6 - 18.2\times$ improvement in performance over a NVIDIA K80 GPU.

Placing AcMC² in Perspective. Traditional methods have been unsuccessful at addressing the challenge of accelerating the execution of inference in PMs. (i) Optimizing compilers and high-level synthesis engines [192] (HLS; C/C++ to HDL compilers) have used control and data dependences in programs to drive parallelism and SIMD optimizations [193, 194]. That approach is inherently limited because it analyzes only the inference procedure (i.e., the program that is executed), and not the dependence (and hence the parallelism) available in the PM. (ii) General-purpose accelerators like GPUs have limited success with MCMC algorithms that are inherently sequential (i.e., compute as a chain of steps) and present significant *branch divergence* across multiple chains. (iii) The use of domain-specific languages (DSLs) to describe parallel patterns [44, 45] that generate efficient code/accelerators have not shown much promise, as they do not offer the abstractions required to easily represent PMs. As a result, accelerated applications for PMs have generally required manual optimization on a problem-by-problem basis, e.g., [172, 173, 174, 175, 176, 177, 178, 179]. In contrast, AcMC² effectively analyzes statistical properties of the PM at compile time and is able to achieve significant parallelism that the traditional methods described above are not designed to accomplish.

5.2 BACKGROUND

5.2.1 Bayesian Modeling

AcMC² considers PMs with joint distribution factorization

$$p(\theta, x_D) = p(\theta)p(x_D|\theta), \tag{5.1}$$

where $p(\theta)$ is a distribution over parameters $\theta = \{\theta_1, \ldots, \theta_m\}$ called the *prior*, and $p(x_D|\theta)$ is the conditional distribution of the dataset $x_D = \{[x_{0,0}, \ldots, x_{n,0}], \ldots, [x_{0,D}, \ldots, x_{n,D}]\}$ given the parameters θ. Given observed data point $x = [x_0, \ldots, x_n]$, the goal of an AcMC² accelerator is to compute the posterior distribution,

$$p(\theta|x) = \frac{p(\theta, x)}{p(x)} = \frac{p(\theta)p(x|\theta)}{\int p(\theta)p(x|\theta)d\theta}. \tag{5.2}$$

A user can then query this distribution (called an *inference*) to obtain required information about the model/data.

For example, consider the use of a commonly used PM to solve the problem of clustering a set of points into K clusters. Here, the PM models how we believe the observations are generated. For instance, one explanation might be (i) that there are K cluster centers chosen randomly according to some distribution, and (ii) that each data point (independent of all other data points) is normally distributed around a randomly chosen cluster center. That explanation describes what is known as a *Gaussian mixture model (GMM)*. We can then query this model to ask, "What is the number of clusters for a given dataset under this model?" or "What is the most likely cluster assignment for a data point under the model?"[2]

Application characteristics often bring additional latent structure to the density factorization shown in (5.1). In the example above, the explanation of the generative process defines this latent structure. Prior work at the conjunction of graph theory and probability theory has developed a powerful formalism called *factor graphs (FGs)* [99]. An FG factorizes (5.1) into C sets of dependent variables:

$$p(x_1, \ldots, x_n, \theta_1, \ldots, \theta_m) = \frac{\prod_{c \in C} f_c(x_c)}{\int \cdots \int \prod_{c \in C} f_c(x_c) d\theta_1 \ldots d\theta_m}, \tag{5.3}$$

where we use the shorthand $x_c = \{x_i | i \in C\}$, and f_c represents *factor functions* describing the statistical relationships between different x_cs. Overall, FGs provide an intuitive and compact representation to parse out the independences.

Inference. Probabilistic inference is the task of deriving the probability that one or more random variables will take a specific value or set of values. Inference tasks are generally structured as in (5.2), where a set of variables θ are being queried over a PM described by $p(\theta, x)$. Inference is analytically intractable for general PMs [189]. Therefore, approximate Bayesian inference methods have become popular. These approximations can be divided into two categories: variational inference and Monte Carlo methods. In this paper, we focus on the second method.

5.2.2 MCMC Methods & Hastings Samplers

MCMC presents a direct method for simulating samples from the posterior distribution in (5.1) or estimating other properties of the distribution. The idea behind MCMC is to have a Markov chain whose stationary distribution is the target distribution; then, samples can be generated by simulating the chain until convergence. In practice, it is common to discard samples from the chain before it converges. This stage is referred to as the *burn-in* stage.

[2] Fig. 5.2 shows the AcMC^2 workflow of the model described above.

Algorithm 5.1 Generic Hastings sampler.

Input: (i) Initial distribution D, (ii) Proposal distribution q, (iii) Number of samples N, (iv) Number of burn-in samples b

Output: Samples from target distribution p

1: Initialize $X_0 = \{X_0^1, \dots, X_0^n\}$ from some distribution D

2: **for** $i \leftarrow [1, N]$ **do**

3: Generate $X \sim q(X|X_{i-1})$

4: $\alpha \leftarrow \min\left\{1, {p(X)q(X_{i-1}|X)}/{p(X_{i-1})q(X|X_{i-1})}\right\}$

5: $X_i \leftarrow \begin{cases} X & \text{with probability } \alpha \\ X_{i-1} & \text{with probability } 1-\alpha \end{cases}$

6: **end for**

7: **return** $(X_{b+1}, \dots, X_N)$

In particular, we consider the Hastings sampler [195] (described in Algorithm 5.1) that generates sample candidates from a proposal distribution q that is generally different from the target distribution p (above). The algorithm then decides whether to accept or reject candidates based on an acceptance test (α).

$$\alpha = \min\left\{1, \frac{p(x') \times q(x|x')}{p(x) \times q(x'|x)}\right\} \tag{5.4}$$

Here x and x' represent the current and proposed values, respectively. The choice q and the acceptance test produce a variety of MCMC methods, e.g., Gibbs sampling and HMC.

We consider three variants of the Hastings sampler. The simplest variant is the Metropolis-Hastings algorithm [195, 196], which combines a Gaussian random walk proposal with an accept-reject test as described above. In general this method scales poorly with increasing dimension and complexity of the target distribution. The Gibbs sampling variant [190] utilizes the structure of the target distribution by taking its element-wise conditional distribution as the transition proposal, forcing the conditionals (in Line 3 of Algorithm 5.1) to be analytically computable. The third variant, called *HMC* [191] uses Hamiltonian dynamics [197] (H) to define a continuous-time transition (i.e., $p(\theta, \rho|x)$) and the stationary distribution of the corresponding Markov chain. To sample from $p(\theta|x)$, HMC introduces an auxiliary momentum variable ρ with the same dimensionality as θ, and effectively samples from the joint distribution $p(\theta, \rho|x) = p(\theta|x)\exp\{-\frac{1}{2}\rho^T M^{-1}\rho\}$, where M is called the *mass matrix*. Samples are generated by simulating

$$\frac{\partial\theta}{\partial t} = \nabla_\rho H \text{ and } \frac{\partial\rho}{\partial t} = \nabla_\theta H. \tag{5.5}$$

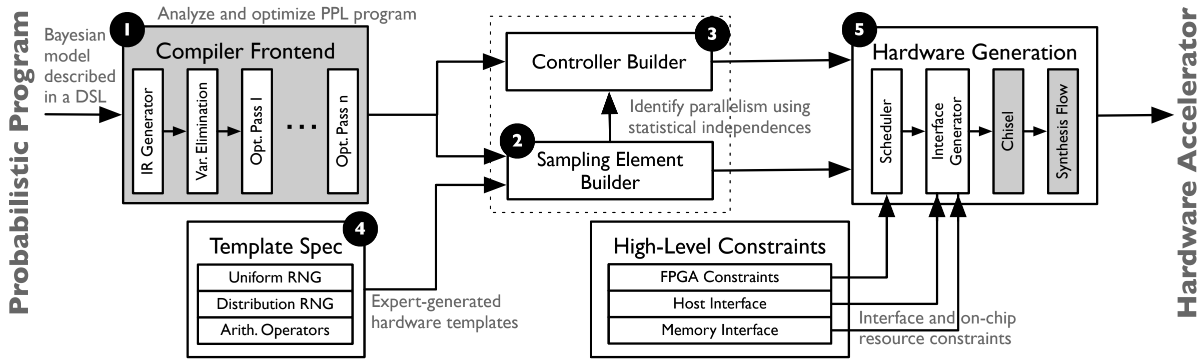


Figure 5.1: Overview of the AcMC^2 approach. Boxes shown in gray represent third-party components integrated into AcMC^2.

5.3 APPROACH OVERVIEW

The §5.3–§5.7 describe the AcMC^2 system. The key optimizations that drive the system are:

1. Identification of dependences between variables in a PM by constructing and identifying non-intersecting Markov blankets, and use of this information to build parallel multidimensional random number generators.

2. Use of the dependences from (1) and MCMC update strategies to enable concurrent speculative execution of multiple proposal distribution samples and acceptance tests, thereby creating a maximally parallel execution plan.

3. Finally, use of bounded approximation provided by counting bloom filters to optimally utilize the on-chip memory for staging intermediate results.

Fig. 5.1 illustrates the workflow that integrates the above optimizations. We briefly describe each of its components below.

Compiler Frontend (1 in Fig. 5.1). The compiler frontend converts a high-level PPL program (in our case, PMs expressed in the BLOG programming language [181]) into an FG that is used in the following steps of the workflow. FGs [198] allow the description of general probability distributions and subsume all other probability-modeling formalisms [199]. AcMC^2 is decoupled from the choice of frontend PPL language through the use of this IR. We describe this stage further in §5.4.

Sampling Element Builder (2 in Fig. 5.1). Using the IR generated from the PPL program, AcMC^2 first computes a partition of the input FG such that different MCMC update strategies (e.g., Gibbs sampling, HMC) are applied to different portions of the PM. The partition strategy is based on a heuristic approach, unless otherwise specified by the

user. For each partition, we separately optimize an accelerated *sampling element* (SE) by identifying the parallelism that is available in the model through the identification of statistical independences. We describe this process in detail in §5.5.

Hardware Templates (4 in Fig. 5.1). Several components of the hardware accelerators are reused across PMs. They include random number generators (RNGs; e.g., following uniform, Gaussian and exponential distributions); arithmetic operators (e.g., floating point adders and multipliers); interfaces to off-chip memory; and host memory. We call those components *templates*, and pre-design them to make optimal trade-offs between on-chip resource utilization and performance. In this chapter, we consider optimizations of these templates for Xilinx FPGA devices. We describe these templates in §5.5.2.

Controller Builder (3 in Fig. 5.1). We construct *Controllers* to synchronize the actions of SEs. For example, to ensure maximal parallelism in the sampling process, we execute several chains of MCMC samplers in parallel, as well as speculatively sample propsal distributions in a single MCMC chain. Aggregation (mixing) of these results identifies the probability distribution (i.e., a histogram) corresponding to a user's query. We describe these optimizations further in §5.6.

Hardware Generation (5 in Fig. 5.1). In the final step of the AcMC^2 workflow, the above statistical relationships and hardware templates are combined together in an execution schedule for an accelerator. A statically generated schedule significantly simplifies the generated hardware. We then use the Chisel [43] to automatically generate HDL corresponding to the computed schedule. The Chisel-generated Verilog can then be fed into a traditional hardware synthesis workflow. We describe the hardware synthesis process in detail in §5.7.

5.4 COMPILER FRONT-END

5.4.1 The BLOG Language

BLOG [181] represents a strongly typed first-order programming language that can be used to define probability distributions over worlds with unknown numbers of variables. Fig. 5.2 illustrates the mapping from a statistical model describing a Gaussian mixture model (GMM; used for clustering data) in BLOG to the underlying FG representation, and its correspondence to the final accelerator generated by AcMC^2.

Why BLOG? Research into PPLs has resulted in the development of several languages (e.g., [181, 182, 183, 184, 185, 186]) that allow users to describe PMs as programs. These PPLs can be categorized into two groups. The first is DSLs that are embedded in higher-level languages like Lisp (for Church [182]) or Python (for PyMC3 [187]). The second group (which

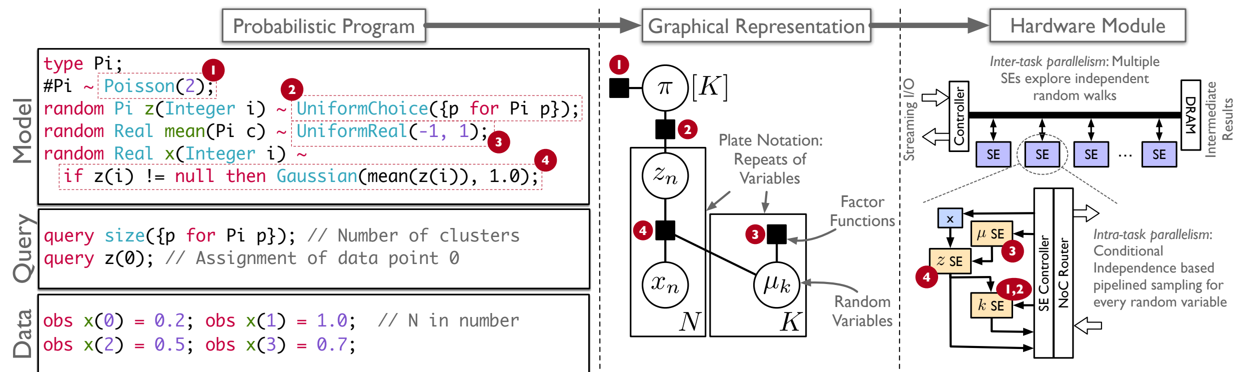


Figure 5.2: Example of the end-to-end workflow: A Bayesian Gaussian mixture model described as a as a BLOG PPL program, converted into an HDL-based hardware module that van be integrated as SoC components.

includes BLOG) consists of essentially standalone languages (with their own interpreters and compilers). The first group of languages is unsuitable for hardware synthesis, as solutions to the PPLs' synthesis must necessarily include solutions to the "high-level synthesis" problems of the higher-level language in which they are embedded (e.g., dealing with unbounded recursion, unbounded loops, and library calls). That was the primary motivation for selection of BLOG as the front-end language for AcMC^2. Further, among the second group of languages, BLOG is one of the few PPLs that can represent PMs that contain both discrete and continuous random variables.

Extensions to BLOG. The BLOG language (and compiler [200]), however, has one drawback. It has no method for describing abstract inputs without binding them to particular values. For example, in Fig. 5.2, the `obs` keyword is used to describe both the input and its value. We have extended the language by adding an `input` keyword to define formally named inputs that will be made available at runtime. These inputs correspond to input ports on the AcMC^2-generated SEs. Outputs are defined using the `query` keyword.

5.4.2 FG Generation

AcMC^2 uses the lexer and parser of the BLOG compiler presented in [200] to generate an abstract syntax tree (AST) of the input BLOG program.[3] We then proceed as follows:

1. Identify `query` statements in the AST, replacing them with new variables. We will use these new variables to define named outputs in later steps of the workflow.

[3]It does so after adding the extension keyword `input` mentioned above.

2. Search the AST for subtrees for arithmetic expressions that can be statically evaluated (i.e., deterministic code containing constant values) and evaluate them.

3. The AST is traversed to find a list of deterministic variables/functions (i.e., those which are not randomly generated) in the model. This identification is done based on the *type* of the variable or the *return type* of the function. AcMC^2 statically composes these deterministic variables/functions with other random functions, so as to build an FG with only random components.

4. Convert the AST into an FG by associating each BLOG function with a factor function, and its inputs and outputs with associated random variables.

5. Apply *variable elimination* [99] to the FG to reduce the size and complexity of the FG. This method is roughly equivalent to *static function execution* and *dead code elimination* in traditional compilers. Note that variable eliminations that lead to *marginal probability distributions* that cannot be directly sampled are dropped.

Note that the optimizations listed above resemble traditional compiler optimizations that most PPL compilers should perform. The FG conversion is specific to the problem at hand, as the downstream steps of AcMC^2 expect an FG as input. The inputs and outputs of the overall process are illustrated in Fig. 5.2. Note that we keep track of repetitions of variables and factor functions in the model that correspond to repeated or indexed variables in the original BLOG program. Dynamic PMs (i.e., which express time varying behavior of variables) are expressed in two parts: (i) the FG corresponding to one instant of time, and (ii) factor functions corresponding to statistical relationships across timesteps.

The process described above (specifically, Step 4) converts a directed graphical model into an undirected one (i.e., an FG). Both of them provide a formalism for representing independences; however, each of them can represent independence constraints that the other cannot. The conversion process occurs by the construction of a *moral graph* [99] from the original directed acyclic graph. If the original PM is *moral*, then the converted PM is a perfect map. The *moralization* process can cause loss of conditional independence if it introduces new edges. In AcMC^2 such a loss does not change the accuracy of the MCMC, merely the effective amount of parallelism (described in §5.5).

5.5 SAMPLING ELEMENT DESIGN

This section describes an algorithm ("SE Builder" from Fig. 5.1) for generating the design of a single SE (see Fig. 5.3) based on the input FG. The process has the following steps.

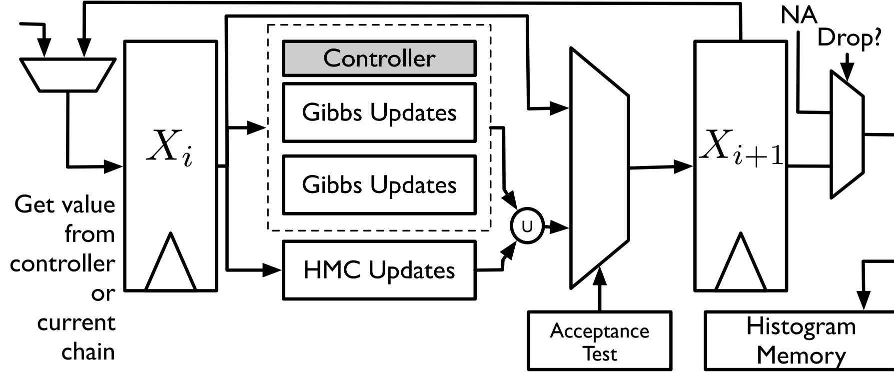


Figure 5.3: Architecture of sampling element: The SE design is based on a compositional Hastings sampler (see Algorithm 5.1) that uses Gibbs sampling and HMC updates. X_i is the state of the sampler in its ith iteration.

1. Depth-first traversal of the FG identifies (i) variables that will be provided as runtime inputs, (ii) variables that will have to be generated by random sampling, and (iii) output variables corresponding to a user's query.

2. Variables that need to be generated through sampling are identified and partitioned into sets corresponding to their MCMC proposal and update strategy.

3. AcMC^2 then constructs samplers corresponding to the proposal distributions q. This step identifies conditional independences in FG that can be used to extract the maximal parallelism in each of the partitions.

4. Using the set of template components available to it, AcMC^2 generates samplers for each of the FG partitions.

After those samplers are executed for their burn-in phases, the values corresponding to the query variables are extracted, tabulated, and stored as histograms (as described in §5.5.3). Each SE generates a fixed number of samples, which represents one execution of an MCMC chain. The output of the "SE Builder" stage is a data-flow graph corresponding to the high-level schematic in Fig. 5.3. This data-flow graph does not incorporate timing information among the different blocks shown in the figure. Timing is described further in §5.7.

5.5.1 Compositional MCMC

Often the high dimensionality of the vectors x and density $q(x'|x)$ being estimated in a Hastings sampler make the sampling process difficult (and, in some cases, intractable).

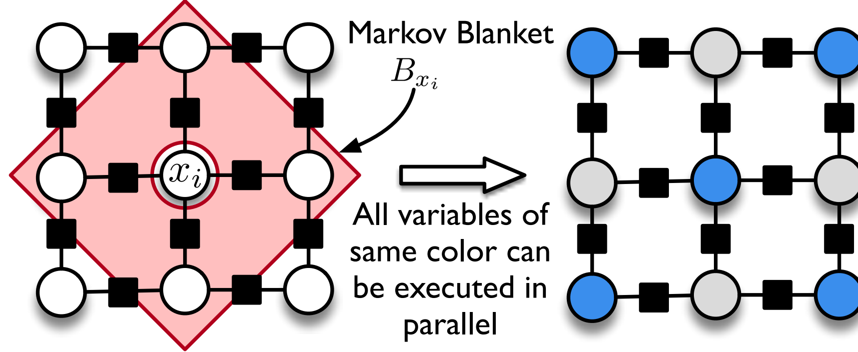


Figure 5.4: Conditional independences of an FG node can be utilized to identify strategies for parallel execution when a Gibbs-sampling-based Hastings sampler is used.

However, it has been shown that it is possible to find MCMC updates for x' that consist of several sub-steps, each of which updates one component or a group of components in x [99, 201]. Finding the optimal division of an FG into partitions for an arbitrary PM is still an open problem.

AcMC^2 relies on a straightforward heuristic to find those partitions. First, it identifies the variables on which it can perform Gibbs sampling. This set consists of discrete random variables in the model, along with continuous variables that exhibit conjugacy relationships. A conjugacy relation implies that a conditional distribution $p(\theta|x)$ takes the same (or an equivalent) functional form as $p(\theta)$. AcMC^2's list of conjugacy relationships are built based on [202]. The remaining continuous variables are sampled with HMC. We do not explicitly use the Metropolis-Hastings sampler, because of its bad scaling behavior in high-dimensional spaces. A user can override this heuristic and manually specify partitions.

Gibbs Sampling. Recall that a Gibbs sampler utilizes the target distribution as its proposal distribution q and also takes compositional steps, where each step targets the sampling of element-wise conditional distributions. Hence in each sub-step, where x_i is updated with x_i', the sampler uses $q(x_i'|x_i) = p(x_i|x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) = p(x_i|x_{-i})$. That means that in an arbitrary PM, every sub-step must proceed sequentially. However, in the case of AcMC^2, since the FG already encodes latent structure in the distribution $p(x_i|x_{-1})$, we can extract conditional independences encoded in the model to relax the dependency of x_i on the set x_{-i}. We do so through the computation of a Markov blanket B_{x_i} on the FG. B_{x_i} defines a subset of x_{-i} such that x_i is conditionally independent of x_{-i} given B_{x_i} (see Fig. 5.4). Hence $q(x_i'|x_i) = p(x_i|B_{x_i})$, which implies that the sub-steps corresponding to x_i can be executed in parallel with $x_{-i} \setminus B_{x_i}$.

One can generalize that observation to all nodes in the FG by studying the graphical structure of the FG. Variables that are in each other's Markov blankets (i.e., that share a

common factor function) cannot be sampled in parallel. Hence, computation of a k-coloring of the FG (i.e., solving a graph coloring problem on the FG), where variables that share a factor function are not given the same color, will give us the maximal parallelism available during the Gibbs sampling process. Here, k will represent the number of synchronization points in the sampling process. [203] provides a proof of correctness of the technique. Fig. 5.4 demonstrates the property on a factor graph. The coloring is synthesized into a state machine that drives the "Controller" in Fig. 5.3.

Hamiltonian Monte Carlo. AcMC^2 uses reverse-mode automatic differentiation [204] to automatically compute the gradients required in (5.5) from the joint distribution of the FG. The current implementation of AcMC^2 performs a source-to-source translation of the symbolic gradients to OpenCL for high-level synthesis through Xilinx SDAccel [205]. That allows us to generate optimized data-paths corresponding to the HMC proposal distribution.

5.5.2 Template-Based Elements

SE components that are reused across a range of PE models are provided to AcMC^2 as a library of manually designed template patterns that provide low latency, high throughput, and low on-chip resource utilization. In our implementation, all of the template-based components are specialized for FPGAs (which we use as a prototyping platform). However, AcMC^2 can also be used to generate ASICs by replacing the template components. We describe these components below.

Random Number Generators. AcMC^2's RNG library provides three types of generators: (i) uniform RNGs, (ii) discrete RNGs corresponding to particular probability mass function definitions, and (iii) RNGs for general probability distributions (e.g., Gaussian, Exponential). Our minimum requirement for these RNGs is that they be high-quality generators that pass common statistical tests; they are not required to be cryptographically secure. The composition of RNGs across complex PMs ensures that even though we might exhaust the period of a single RNG, we will never exceed the period of all the RNGs used in an SE. Further, large PMs require many RNGs, so they have to use on-chip resources optimally. Finally, we are interested in RNGs that have deterministic performance. For example, rejection-sampling-based RNGs [206], which might retry an indefinite number of times before generating a random number, are not suitable in our use case, because the hardware generation step in the AcMC^2 requires definite latency characteristics to produce a static schedule of SEs.

The Uniform RNG (see Fig. 5.5) is the simplest type of random number generator available in AcMC^2. Our implementation draws heavily from [207], and represents a modified XOR-shift [208] generator that is optimized for low resource usage on FPGAs. The 4-bit RNG

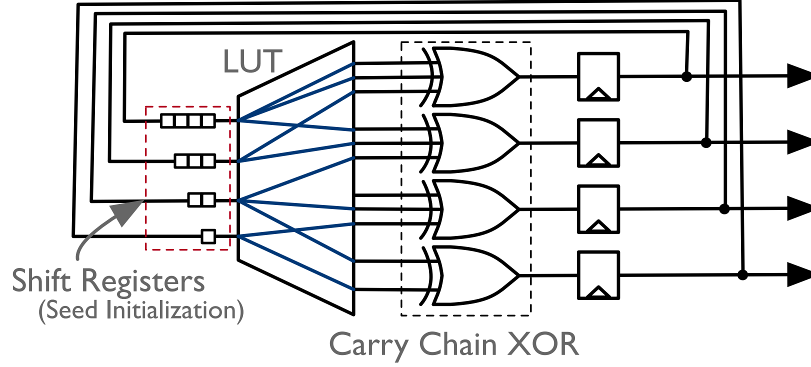


Figure 5.5: XOR-shift-based FPGA-optimized uniform random number generator.

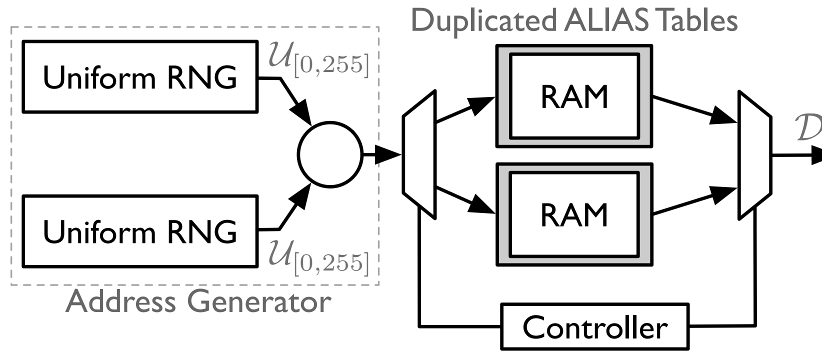


Figure 5.6: Alias-table-based RNG to generate arbitrary discrete random variables with static distributions.

completely utilizes a single logic cell available on an FPGA: it utilizes a 4-input look-up table (LUT), the XOR-gates from a carry chain adder, and the output registers to buffer output. Overall it produces a single 4-bit random number per clock cycle, utilizing only a single LUT and a single shift register. These RNGs are used as a source of randomness for the other types of RNGs.

The second type of generator in AcMC^2 uses an alias-table-based strategy [209] to generate discrete random numbers whose probability distribution is provided at compile time. Fig. 5.6 shows the schematic layout for this RNG. It reuses two uniform RNGs to generate addresses and store them into a locally stored alias table, which is accessed to retrieve the value of the random number. The address lookup happens in two clock cycles; hence, the memory element is duplicated to ensure a throughput of 1 operation per cycle. The dynamic range of the probability values in the alias table is set to a 32-bit floating-point number.

The final type of RNG used in AcMC^2 generates values from well-known distributions by using the inverse transform method [210]. Fig. 5.7 shows a schematic of the generator. This method transforms integer uniform random numbers into fractional numbers in $[0, 1]$ and

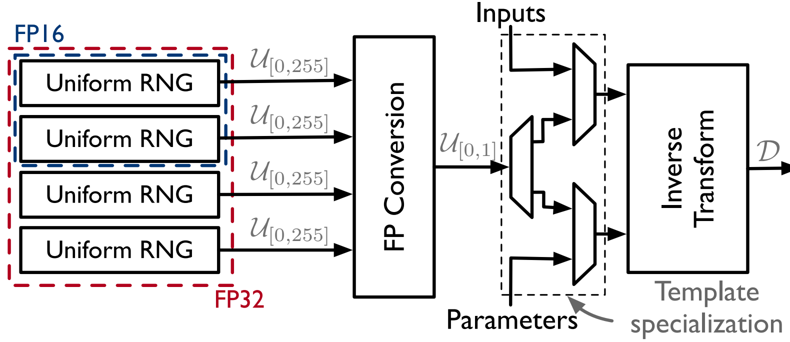


Figure 5.7: Inverse-transform-based RNGs.

then uses the inverse of the target distribution's cumulative density function to generate the required random numbers. The method is computationally intensive, as computing the inverse transform often requires several floating-point operations, leading to higher latencies and lower throughputs than the other RNGs mentioned above. AcMC^2 provides support for exponential, Poisson, Gaussian, and binomial distributions.[4] One can add more RNGs to AcMC^2 using template implementations.

5.5.3 Storing Sampled Results

The final step of the SE pipeline (see Fig. 5.3) corresponds to saving the values generated by the SE into *Histogram Memory*. It is implemented using on-chip memory as follows.

- When output variables take a finite number of values (i.e., types corresponding to the output variable are defined over finite sets), AcMC^2 generates counters corresponding to each of the values. The counters are incremented when a corresponding sample is produced.

- When the output variables' domain corresponds to large sets, the user is required to annotate a binning criterion corresponding to each query.

- Binning provides only a partial solution, as there is a limited amount of on-chip memory for storing histogram information. Off-chip DRAM provides an attractive alternative for storing the histograms; however, write latencies to DRAM, as well as write contention across multiple SEs, make the use of DRAM intractable from the point of view of performance. In order to remove this bottleneck, we allow for an approximate solution by using counting Bloom filters [211] (see Fig. 5.8). The core idea is to allow for storage

[4]The binomial distribution is generated through a look up-table-based approximation of the distribution's inverse density function.

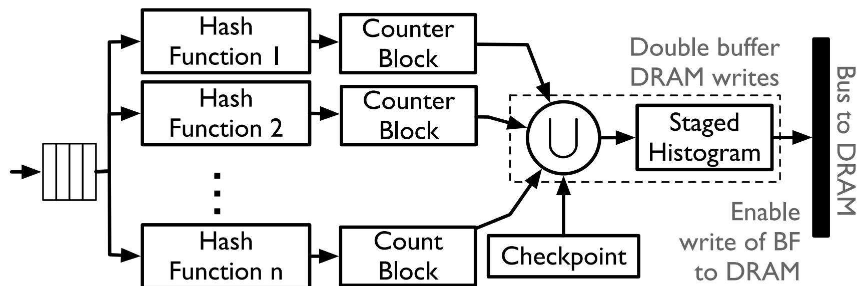


Figure 5.8: Counting Bloom-filter-based approximate memory for storing histogram outputs of an SE.

of the histograms in an approximate fashion whereby counts corresponding to some bins can be larger than their true values, in order to trade off the amount of memory required to store the values. Counting Bloom filters provide bounded approximations for storage that allow us to tune the parameters of the Bloom filters to stay within the noise margins of the MCMC simulations. A problem with the Bloom filter approach is that they eventually fill up over time when they must deal with a large stream of data. As a result, at some point the Bloom filter becomes unusable due to its high error rates. Hence we need to periodically checkpoint the state of the Bloom filter (by storing it to DRAM), and reset it to avoid such problems.

AcMC^2 requires a user to actively *opt in* to any of the above storage types, annotating an FG model with information about histogram binning and Bloom filter size. Our implementation of the counting Bloom filter uses MurmerHash [212].

5.5.4 Handling Infinite Models

Plate-based models, like the GMM example in Fig. 5.2, in which portions of the model get repeated multiple times are handled by synthesizing SEs that correspond to the plates in the PM. These SEs are then repeatedly executed based on the plate specification in the PM. This repetition is encoded into state machines (in the "Controller" block in Fig. 5.3), which control the execution of the proposal distributions across partitions. Broadly speaking, there can be two types of plates in a model: plates that repeat based on user input at runtime (e.g., a plate corresponding to N in Fig. 5.2), and plates that correspond to repetitions due to model variables (e.g., a plate corresponding to K in Fig. 5.2). AcMC^2 automatically handles the second type, and requires explicit human annotation of the number of repeats in the first type.

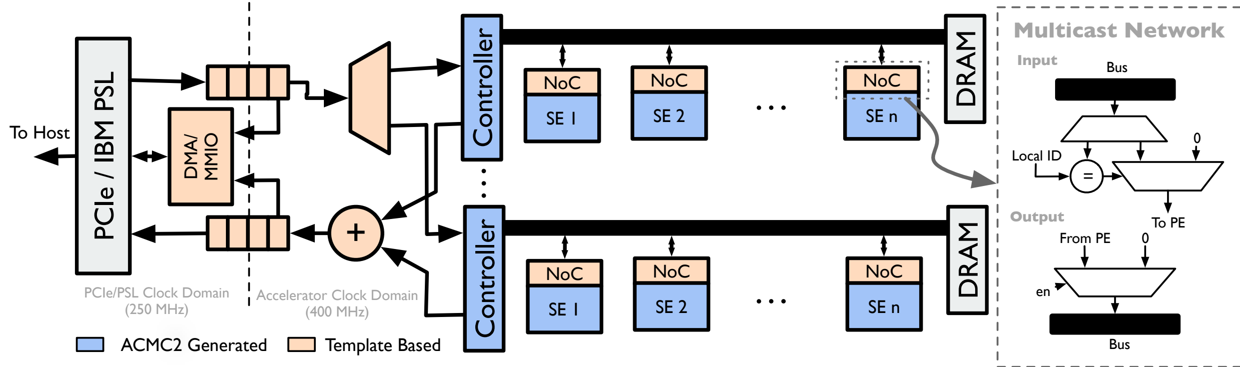


Figure 5.9: Architecture of the generated accelerator, including SEs, controllers, and host-accelerator communication.

5.5.5 Importance of RNG Efficiency

RNG efficiency does indeed play a significant role in overall performance. However, the latency throughput characteristics of these RNGs have to be tuned with other components of the system to ensure the best performance. We describe these trade-offs below.

Throughput. Accelerators generated by AcMC^2 leverage data-flow between the RNGs and computational elements (e.g., adders, multipliers) and matches throughput between these elements. In most cases, Xilinx-provided IPs for computational elements execute at 1 op/cycle, which is matched by the template-based RNGs described in §5.5.2. Thus we can generate MCMC samples at throughputs close to 1 sample/cycle for a single SE. Using worse RNGs could impact the overall throughput of the accelerator and result in SE stalls.

Latency. We prefer high-throughput high-latency RNGs. Note that there is a trade-off point after which the latency negatively affects performance: the assumption in this argument is that a single RNG latency is significantly lower than DRAM write latency. Guaranteeing high throughput (1 sample/cycle) often results in increased RNG latency. The trade-off between sample latency and overall performance has to be tuned to the performance of the memory system. For example, the time taken to generate the output histogram of an SE has to be greater than the time taken to write that histogram to onboard DRAM (i.e., the accelerator is not memory-bound). In fact, we use double buffering of histogram writes (see Fig. 5.8) to effectively hide the increased RNG latency. An improved memory system on the FPGA board could alleviate these problems.

5.6 CONTROLLER DESIGN

AcMC^2 uses multiple independent instances of the Hastings sampler executed in parallel to generate several target distributions. The final results can then be aggregated from individual samplers. Further, multiple SEs can be used to speculatively execute future steps of Hastings chains in parallel. The "Controller Builder" block from Fig. 5.1 identifies the scope of that parallelism and constructs a controller that can coordinate the execution of SEs to enable the above optimizations. Fig. 5.9 illustrates the integration of the controller and SEs into a single accelerator.

Ensemble Samplers. Individual MCMC chains have internal serial dependence; however, multiple chains can be computed in parallel to generate several independent estimates of the target PM posterior distribution. The final result is calculated by pooling the results of these different chains (i.e., by aggregating the output sample histograms from all the chains). This corresponds to exploiting the *embarrassingly parallel* nature of the MCMC process. Overall, this optimization improves throughput and accuracy of the inference process. The only drawback is that each sampler has an independent burn-in phase, so that the number of redundant burn-in samples grows linearly with the number of ensemble samplers employed. We achieve this optimization in the generated accelerator by generating multiple instances of the single-Controller and multiple-SE block (see Fig. 5.9).

Speculative and Predictive Evaluation. Hastings samplers show branching behavior, i.e., possible random walks explored by the samplers form a *branch tree*. An MCMC chain will traverse several paths (corresponding to the proposal distributions) in this branch tree, but will eventually take only a single path (corresponding to a successful acceptance test). That provides a scope for exploiting parallelism through speculation. For example, in depth-first traversal, a generated sample is assumed to be accepted, and its subtree is generated speculatively. Similarly, in breadth-first traversal, a sample is assumed to be rejected, and other samples from the same level are generated speculatively. In contrast to prior work in statistics (e.g., [213, 214]) that adopted the depth-first approach presented above, AcMC^2 uses the breadth-first approach, as the generated hardware is much simpler. All SEs are set to start with one initial state; thereafter, each SE explores individual samples from the proposal distribution. When an SE generates a value that passes the acceptance test, it broadcasts the new state value to all other SEs, and proceeds with the next step in its own pipeline. The controller arbitrates the bus and ensures race-free executions. Some aspects of the depth-first approach are captured in each individual SE's pipeline, where, as soon as a proposal value is generated, the next level of the branch tree can start execution in the pipeline.

Other auxilliary functions of the controller include:

1. *Initialization*: The controller initializes all SEs with the seeds required to start random number generation, and computes the initial starting state of the Hastings sampler (which is generated from a Gaussian distribution).

2. *Bus Scheduling*: The controller acts as an arbiter to give individual SEs the ability to write data to the multicast bus (described further in §5.7.2).

3. *Batching*: If a dataset is being used that is larger than the available memory on the accelerator, the data have to be divided into batches, and the inference has to be run one batch at a time. The controller is responsible for copying input batches from the host memory to the accelerator.

4. *Moving Results to Host Memory*: The controller is responsible for moving the final outputs of the MCMC chains (i.e., histograms generated over the user query) from the on board DRAM on which it is stored to the host memory space. Doing so involves aggregating the counting Bloom filters from each of the ensemble samplers.

Communication between the controller and SEs is encapsulated in an AXI-Stream protocol. That protocol allows us to decouple the controller from the SEs and synthesize them separately, relying on the communication protocol between them to ensure synchronization properties.

5.7 ACCELERATOR SYNTHESIS

The final step of the AcMC^2 workflow converts the data flow graph corresponding to an SE into a synthesizable Chisel model, which can then be fed into traditional FPGA/ASIC synthesis flows.

5.7.1 Scheduling

The scheduler is responsible for converting the SE data-flow graph into a cycle-by-cycle schedule for a given set of resource constraints. AcMC^2 uses this schedule to generate synchronization between inputs and outputs of stages of the compositional MCMC SE (see Fig. 5.3). AcMC^2 uses the *As-Soon-As-Possible scheduling algorithm* (based on [215]) to compute such schedules. The scheduling algorithm works by scheduling an operation as soon as all of its predecessor operations (in the data-flow graph) have completed. The maximum number of parallel operations permitted are defined by resource constraint parameters fed in

by the user. AcMC^2 generates the design for a single SE that accommodates this schedule. Note that the scheduling procedure in AcMC^2 does not consider resource limitations or routing issues between SE components – these decisions are left to the subsequent synthesis flow (recall from §5.3).

5.7.2 Overall Accelerator Design

The SE data-flow graph, along with its static schedule, is converted into a Chisel HDL-based description of an SE pipeline. Through the use of the cycle counts for operations in the SE design, a Chisel HDL description of the *Controller* (described in §5.6) is also generated. These units then interface with off-chip DRAM and host-memory interfaces to produce the final probabilistic inference accelerator. The overall architecture of the accelerator is shown in Fig. 5.9. We describe the remaining components of the accelerator next.

Broadcast-Based NoC for SEs. AcMC^2 uses a template-based, bus-based network-on-chip (NoC) design to enable point-to-point and broadcast messaging between the Controller, SEs, and DRAM controllers. The bus uses an AXI-Stream-based communication protocol for data transfer. Fig. 5.9 shows the design of the routers used in the network. To read from the bus, a router matches its local identifier to that of the stream being sent on the bus and connects the SE to the bus if there is a match. Writing to the bus is mediated by the controller in a single-writer, multiple-reader protocol.

Host-Accelerator Communication. AcMC^2 uses the IBM Coherent Accelerator Processor Interface (CAPI) [96] for Power8 processors to facilitate host-accelerator communication. CAPI is layered over PCIe and provides low-latency, high-speed device-to-host memory transfers. In particular, CAPI simplifies the generated host CPU code, reduces dependency on DMA drivers, and eliminates the need for page-pinning and bounce buffering to extract high performance from the underlying PCIe bus. A DMA and MMIO controller for the IBM Power Service Layer (PSL; the IBM IP component that interfaces with the accelerator) is provided as a template for the accelerator. This interface is used to send inputs to the accelerator, receive inputs from the accelerator, and initialize the accelerator with *seed* values for RNGs. The PSL and DMA/MMIO interfaces are clocked at 250 MHz, and the remainder of the accelerator is clocked at 400 MHz.

Host-accelerator communication can be of the following types. (i) In the *batch data transfer* mode, the host loads the input dataset into host memory, from which the accelerator reads data in batches. Similarly, outputs are transferred to a host buffer. (ii) In the *streaming data transfer* mode, the host and accelerator share circular buffers corresponding to inputs and outputs. Synchronization between host and accelerator is ensured using CAPI's atomic

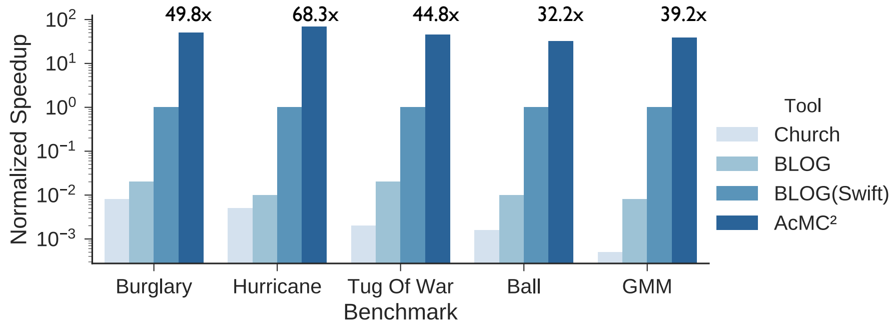


Figure 5.10: Comparison of runtime performance for AcMC^2-generated accelerators with PM inference on other PPLs (normalized to `Blog(Swift)` [200] compiler).

operations. In both cases, the accelerator is initialized with the addresses of the input/ouptut buffers. The accelerator actively prefetches new data (in cacheline-sized 128-byte chunks) and adds it into the input FIFOs (see Fig. 5.9) for further processing.

5.8 EVALUATION AND DISCUSSION

Experimental Setup. AcMC^2 has been implemented in $\sim$ 2k lines of Scala. The templates used in the compiler were developed in System Verilog and use IPs from Xilinx to implement single-precision floating-point math operators, BRAM blocks, an off-chip DRAM interface, and shift registers. The accelerator communicates with the host through the IBM CAPI interface [96, 216]. All generated accelerators (and CPU baselines) were evaluated on an IBM Power8 S824L system with an Alpha-Data ADM-PCIE-7V3 FPGA board (with Xilinx Virtex 7 XC7VX690T FPGA) and an NVIDIA K80 GPU.

5.8.1 Comparison With CPU-Based PPLs

As a first-level comparison, we compared the runtime of accelerators generated through AcMC^2 with common benchmarks that are used to evaluate PPLs. We used the set of benchmarks from [181, 200], which represent tests over a wide set of PPL features available in BLOG. They are indicative of performance, as they compare the performance of AcMC^2-generated accelerators to that of accelerators generated by CPU-based PPL compilers. However, these benchmarks do not represent complex the PMs that a user would encounter in the real world. We consider such real-world problems (and their implementations on GPUs and HLS compilers) in §5.8.2.

A performance comparison of the techniques is shown in Fig. 5.10, and a comparison of peak power usage is presented in Table 5.1. Power usage estimates of the generated accelerators are collected from the Xilinx Vivado tools and via the S824L's in-built "system-level" power measurement infrastructure.[5] Power estimates for CPU-based code were made based on publicly available TDP values. In all cases, the programs were instrumented to run 2 million samples before stopping. Overall, we observed an average speedup of 46.8× and an average reduction in power of 16.1× (which corresponds to an overall improvement of 753× in terms of performance-per-watt). We observe that FPGA BRAM utilization was the dominant resource used, as seen in Table 5.1. Note that all performance measurements presented above were collected over 1000 runs of each of the programs, to amortize OS costs in process creation and setting up of communication through CAPI. The GMM benchmark uses the counting Bloom filter approximation described in §5.5.3. It infers the distribution of the number of mixture components, as well as the distribution of means and variances for each mixture component. The histograms corresponding to the means and variances are stored in the Bloom filter; in each case, a 1000-bin histogram was stored in 100 counters and checkpointed/refreshed every 10000 samples. All benchmarks used the batch data-transfer model for host-accelerator communications. The number of ensemble samplers was limited to 4 to ensure that each sampler was mapped to a single onboard memory DIMM.

Overall, Fig. 5.10 and Table 5.1 suggest that when there is no unbounded repetition in the PM, AcMC^2-generated accelerators fare better (with respect to both runtime performance and power usage) with discrete variable PMs. That is expected, because (i) the use of continuous distributions (even if they have conjugacy relationships) requires expensive floating-point computations to compute inverse transforms, which significantly increases the latency of the RNGs and results in higher resource cost; and (ii) unbounded worlds require re-execution of the MCMC chain (i.e., the SE) with different (sampled) initial worlds. It is important to note that the performance comparison across Church, BLOG, and Swift also compares the overhead of the language runtimes: Church uses Lisp [182], BLOG uses Java [181], Swift uses C++ [200]. AcMC^2 does not have any of these overheads.

5.8.2 Real-World Case Studies

Case Study 1: Epilepsy SoZ Localization. We applied AcMC^2 to a PM [165] (see Fig. 5.11) that is used to infer characteristics of human-brain activity by using electroencephalogram (EEG) sensors, with the goal of identifying brain regions responsible for the

[5]The system reports power measurements averaged over 30s intervals. We measure a distribution of power consumption when the system was idling, and when the accelerator was being used.

Table 5.1: Power and resource utilization for benchmark BLOG programs. Numbers of SEs are described as $x \times y$: x is the number of controllers and y is the SEs per controller.

Benchmark	Power (Watts) CPU	Vivado	Measured	# of SEs	BRAM %
Burglary	190	10.1	16.3 ± 2.1	4×4	35%
Hurricane	190	10.3	16.8 ± 1.9	4×4	31%
Tug of War	190	12.5	19.4 ± 3.1	4×4	38%
Ball	190	12.9	19.3 ± 2.6	4×4	41%
GMM	190	14.1	15.3 ± 3.4	4×4	46%
EEG-Graph	190	11.8	15.8 ± 3.8	4×8	64%

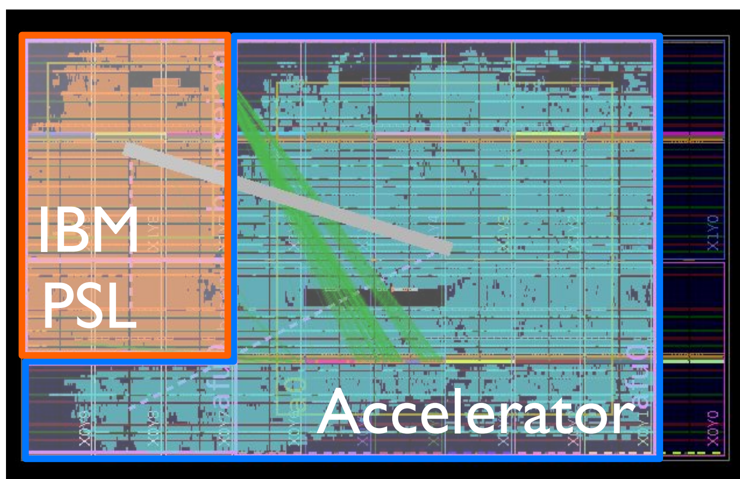


Figure 5.11: AcMC^2 generated accelerator targeting a Bayesian neurology model for epilepsy [165].

onset of epilepsy. [165] presents a generative FG model that estimates brain activity and localizes it to a particular EEG sensor, allowing clinicians to identify regions of the brain (called *seizure onset zones* or *SoZs*) that show pathological behavior like epileptic seizures. This application represents a typical use of PMs in the field of precision medicine, where data are obtained from medical sensors in a streaming fashion and used to make decisions in real time.

Case Study 2: Network Security. The second real-world problem for which we show the application of AcMC^2 is in the domain of *network security*. Here, a PM [166] is used to describe the relationships between user intent (i.e., whether a user is benign, suspicious, or malicious and represents a threat to the integrity of a networked computer system) and events observed by security monitors (e.g., network monitors like Bro [217]). Using these statistical relationships, [166] aims to infer the user state given real-time data from the

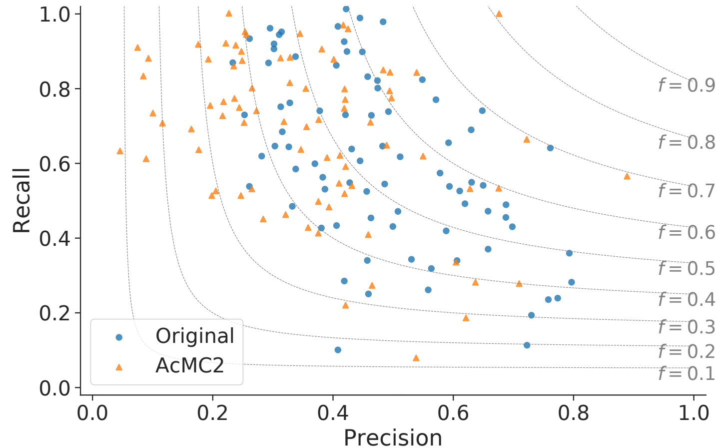


Figure 5.12: Accuracy (in terms of precision and recall) of the EEG-GRAPH energy minimization method (labeled `Original`) vs. the MCMC accelerator generated by AcMC^2.

security monitors. This application represents the typical use of PMs to build "intelligent" compute-embedded network devices like network interface cards (NICs) or switches that can automatically detect and preempt malicious intrusions.

For each of the case studies described above, we constructed (i) a BLOG program and (ii) an OpenCL program corresponding to the model. The OpenCL program was hand-tuned to use the model-specific optimizations presented in §5.5 and §5.6. Two separate versions of this program were created, using compiler-specific attributes targeting NVIDIA's OpenCL and Xilinx's SDAccel compilers. In both cases, the CPU baseline corresponds to software obtained from the original authors. Note that the CPU baselines represent research software, which, as such, might not be perfectly tuned to the system architecture. However, that is a common situation for research software for which an underlying compiler is expected to perform meaningful performance optimizations.

Why these models? (i) These models represent real-world applications of PMs in performance-critical applications across varied application domains. (ii) These models use a large subset of the PPL language features: discrete and continuous random variables, distribution conjugacy relationships, and unbounded dynamic models that obey the Markov property. (iii) Further, they represent challenging situations for AcMC^2 because they use multivariate factor functions in relatively small FGs. The PM is relatively densely connected so that the generation of random samples in the Gibbs sampling-based SE is almost completely serialized.

Comparisons to CPU: Case Study 1. The generated accelerator can infer the query posterior distribution for a $3s$ chunk of input EEG data in an average of ~ 41.8 ms compared

to the CPU baseline, which uses $4.3s$ (i.e., it does not meet the $3s$ real time requirement of the applications). Following Little's Law, a single FPGA accelerator would be able process $71 \approx {}^{3s}/_{41.8ms}$ patients in parallel in real-time. That is the largest accelerator configuration we have successfully fit onto the FPGA. Fig. 5.12 shows the difference in accuracy between the original EEG-GRAPH technique and the AcMC^2 accelerator. Overall there is a drop in accuracy from an f-score[6] of 0.47 to one of 0.465. That can be attributed to the approximate nature of the MCMC procedure. The method proposed in [165] uses exact inference procedures based on energy minimization, which might produce marginally more accurate answers.

Comparisons to CPUs: Case Study 2. The AcMC^2-generated accelerator performed the inference at the rate of 0.4 ms/event. That is $\sim 48.4\times$ faster than the CPU-based implementation in [166]; here, an event corresponds to the output security monitor. The CPU implementation uses threading as well as high-performance math libraries that are optimized for the SIMD and memory locality. In this case, the approximate Bloom filter optimization did not affect the statistical correctness of the output (i.e., 74.2% true-positive rate, 98.5% true-negative rate, 1.5% false-positive rate, and 25.8% false negative rate).

All our experiments were setup to use the same number of accepted samples. The number of samples was chosen in each case to ensure that the MCMC procedure would be close to convergence. However, the approximate nature of MCMC implies that independent runs do not give the same answers. To verify the correctness of our generated acclerators, we performed a Kolmogorov-Smirnov test across the CPU, GPU, and AcMC^2 implementations to ensure that the sampled distributions were identical with high probability. Tables 5.2 and 5.3 show a comparison of power and FPGA resource utilization for the case studies presented above.

OpenCL Comparison: Code Complexity. Table 5.4 shows a comparison of the complexity of the AcMC^2 and OpenCL accelerators in terms of lines of code (LoC). For example, in Case Study 1, compared to the AcMC^2 accelerator, which is described in 183 LoC. The GPU and FPGA OpenCL require 622 and 961 LoC, respectively. Their added

[6]The f-score is two times the harmonic mean of precision and recall. An F-Score can take values between $[0, 1]$, with 1 being the best possible score.

Table 5.2: Performance & Power consumption improvements when using AcMC^2.

		Power (Watts)			
Benchmark	**Perf. CPU**	**CPU**	**Vivado**	**Measured**	**# of SEs**
Case Study 1	102.1×	190	11.8	19.3 ± 3.7	4×8
Case Study 2	48.4×	190	10.4	11.2 ± 0.8	4×8

Table 5.3: Summary of FPGA resource utilization when using AcMC^2.

	BRAM		DSP		FF		LUT	
	Avail.	**%**	**Avail.**	**%**	**Avail.**	**%**	**Avail.**	**%**
Case Study 1	1470	64%	3600	49%	866400	22%	43200	50%
Case Study 2	1470	83%	3600	29%	866400	34%	43200	61%

Table 5.4: Comparing complexity and performance of AcMC^2-generated accelerators with that of OpenCL accelerators.

	NVIDIA K80 GPU			FPGA		
Implementation	**LoC**	**Perf.**	**Power (W)**	**LoC**	**Perf.**	**Power (W)**
AcMC^2- Case Study 1	–	–	–	183	18.2×	11.8
OpenCL - Case Study 1	622	1×	300 (TDP)	961	0.2×	14.2
AcMC^2- Case Study 2	–	–	–	146	8.6×	10.4
OpenCL - Case Study 2	586	1×	300 (TDP)	8984	0.8×	15.6

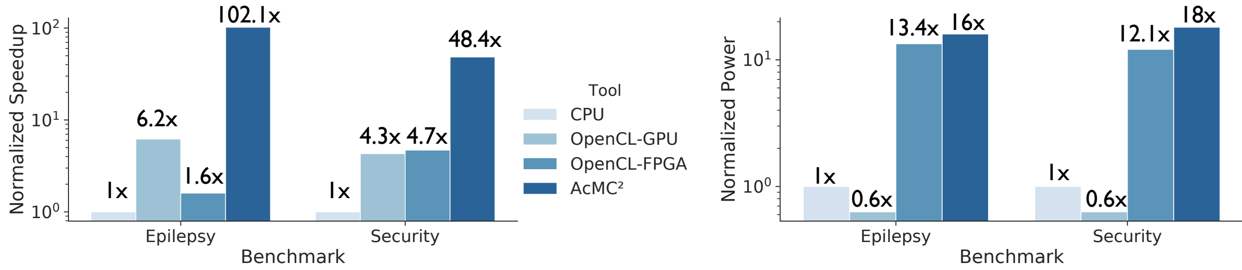


Figure 5.13: Comparison of runtime performance and power consumption for AcMC^2-generated accelerators for two real-world applications [165, 166].

complexity is derived from writing memory and synchronization code on the GPU. In the case of the FPGA, explicit code annotation (e.g., `__attribute__` directives), statically bound loops and other code snippets is used to force effective pipelining. In addition to LoC, the expertise (of the underlying hardware system) required to construct the OpenCL version clearly emphasizes the superiority of AcMC^2.

OpenCL Comparison: Performance (GPUs & FPGAs). Table 5.4 further shows a comparison between performance and power requirements for the four configurations. For each case study, performance has been normalized to that of the GPU. We observe that the AcMC^2-generated accelerators performed $8-18\times$ better than the K80 GPU and $248-462\times$ better in performance-per-Watt terms. We speculate that the reduced performance resulted from (i) *control divergence* between threads that were exploring separate parts of the MCMC

search space, and (ii) *throughput-optimized* RNG libraries that perform better when they have to generate a batch of values rather than the single values used in MCMC's inherently sequential random walks. We drew these conclusions based on the rejection-sampling-based algorithms used in the GPU implementations; i.e., the kernels generate several RNs and accept/reject a fraction of them, which means that the control flow is different on different threads. Further, the GPU libraries for RNGs work by generating batches of RNs; they are later consumed to generate samples for the MCMC. Thereby maximizing the throughput of RNG but not the throughput of the MCMC computation. The exact measurement of divergence in this case is difficult to make as the mapping between PTX and SASS is unknown on NVIDIA devices. Measurement on a microarchitectural simulator might not lead to exact results.

The HLS-generated FPGA accelerator performs an order of magnitude worse than the AcMC^2 and GPU implementations, in terms of absolute performance terms. There are multiple reasons: (i) the accelerator can attain a maximum clock speed 163 MHz, while the AcMC^2-generated accelerator can achieve 400 MHz; and (ii) the automatically generated XOR-shift RNGs are higher-latency and lower-throughput than the 1-RNG-per-cycle generators described in §5.5.

Choice of CPU Baseline. In our measurements Intel Xeon E5 CPUs performed less than $6-8\%$ better than the IBM Power8. Our choice of that baseline did not change our conclusions. However, the use of the CAPI-based interface for host-accelerator communication significantly simplified the implementation (recall §5.7.2). The Power8 and Xilinx FPGA use different process technologies, i.e., 22nm and 28nm respectively, and hence it might not be completely fair to compare their results. However, the decision to use CAPI limits us to using FPGA boards that are supported by IBM (with their PSL IP components). We believe that changing the FPGA technology will not change our performance (as the 250 Mhz clock should be replicable on even newer FPGA parts); however, performance-per-Watt measurements will change with the design of the FPGA routing network.

Performance Implications of CAPI. All the microbenchmarks correspond to transmission of $100-960$ KB of input data to the accelerator for computation. The total time for these transfers (in streaming mode) is included in the results presented in the paper and contribute $< 10\%$ of the runtime. We observe in Case Study 2 that the runtime in is bound by the PCIe messaging latency (which is on the order of 100 ns for the 128-byte cache-line transferred over PCIe in CAPI). A real deployment of this accelerator would involve at least 3 such messages over PCIe, i.e., network interface card (NIC) to CPU, CPU to the accelerator on the FPGA, and, finally, returning the result to the CPU. We will address this communication latency issue in future work.

5.9 RELATED WORK

Parallelization of Probabilistic Inference. Low et al. [218] present a distributed computing framework for machine learning algorithms. They demonstrate the distributed parallelization of probabilistic inference using belief propagation [99]. Gonzales et al. [203] provide a proof of correctness for parallelization of Gibbs sampling through the use of conditional independences; this motivated our use of Markov blankets in §5.5. Recht et al. [219] suggest *asynchronous Gibbs sampling*, whereby conditional independences are not honored by the sampler; [220] provides bounds on the asymptotic correctness of such a sampler. [213, 214] provide methods for speculative execution (called *dynamic prefetch*) using biased proposal distributions. We use the approximation from [197] to generate samples in HMC. Homan and Gelman [221] present a further-optimized algorithm (which converges more quickly than traditional HMC with [197]) to generate samples from the proposal distribution of an HMC. [186] provides an implementation of [221] in a PPL.

Accelerated Probabilistic Inference. Several prior FPGA/ASIC efforts solve some form of probabilistic inference; however, they cannot be generalized to apply to all PMs. For example, [172] proposes an architecture for LDPC code encoding/decoding; [173, 178, 179] propose architectures for several bioinformatics applications; [174] proposes an architecture for image segmentation; and [176] proposes an architecture for inference on a class of state space models. In comparison, AcMC^2 has the ability to generate accelerators for general PMs expressed in the BLOG language. The use of GPUs in MCMC has been explored, but only for particular applications, e.g., in [222, 223, 224]. [177] comes closest to AcMC^2. It describes the design of an FPGA-based accelerator and a compiler to target the acceleration of belief propagation in Bayesian networks. In contrast, AcMC^2 can be applied to a much larger set of PMs. [225] explores the use of analog circuits to perform statistical inference.

Hardware Generators. AcMC^2 uses the Chisel HDL [43] to generate RTL corresponding to the accelerator. Other HDL generators, e.g., [44, 45], provide higher-level constructs that can be used to declare and annotate the parallelism available in a program; they cannot be used directly with PMs as PPLs.

Programming Language Design. [226] proposes Edward, a PPL for Bayesian neural networks (BNN) probabilistic inference that uses variational inference techniques with MCMC methods. It utilizes Tensorflow's [141] GPU-based tensor operations, and potentially Google's TPUs [3]. [171] provides an optimized architecture for inference (which includes both variational inference and MCMC) on BNNs with Gaussian priors. In comparison, AcMC^2 only targets MCMC applications and cannot handle variational inference.

5.10 SUMMARY

This chapter presented the design and evaluation of AcMC^2, a compiler that can transform PMs expressed in the BLOG PPL into optimized accelerators that compute inference queries by using an ensemble of MCMC methods. We believe that AcMC^2 significantly simplifies the process of constructing workload-optimized accelerators for executing inference on PMs, making the benefits of such optimization available to a larger group of researchers. As a result, AcMC^2 forms the basis for creating future high-performance SoCs for AI applications that can be deployed as edge devices or in clouds.

Future Work. The design of AcMC^2-generated accelerators in this paper assumes that the PM models being compiled fit on a single FPGA. It is not impossible to conceive of a PM for which that assumption would fail. We believe a solution will involve the distributed execution of the ensemble samplers over multiple FPGAs and unreliable network links.

CHAPTER 6: CONCLUSIONS

This thesis takes a first step towards building computer systems that can learn to efficiently optimize performance on their own through reinforcement learning. We demonstrate that learning-based systems are able to achieve superior performance than human-engineered heuristics in a wide range of environments, from to cluster scheduling for complex data processing jobs, to online error correction in performance counter measurements. The key advantage of these data-driven approaches is their ability to tailor for the specific deployment settings (e.g., network types, workload patterns, etc.) and to automatically adapt to challenging environments, especially those for which the fixed heuristics were not custom-designed in advance. Moreover, we have identified several key problem structures and scalability issues that are unique to learned-computer-systems and have proposed solutions to address these problems. First, that measurements, which drive the learned policies in computer systems are intrusive, and can affect the overall performance of the system. Majority of techniques that address measurement overheads, suggest limiting the number of simultaneous measurements in some way. Hence, these techniques do not allow us to consistently capture the state of the system, often leading to the collection of noisy or imprecise state-data. Moreover, the noise increases as more parallel measurements are made. A key contribution of this thesis is to demonstrate a Bayesian denoising (filtering algorithm that can take support of coincident measurements to quantify the uncertainty (error) in a measurement, and using this information, predict the correct value of the measurement. We demonstrate how such a Bayesian filtering algorithm can be composed with downstream deep reinforcement learning agent (for scheduling tasks to accelerators) that are trained jointly through backpropagation. This composition technique is general, thereby allowing it to be used across measurement modalities and controller/agent types. Second, we identify the issue that as the number of learned-policies in a system increases, these policies can interact with each other in complex, non-linear ways, leading to less than optimal (and less than isolated) performance. The traditional approach for dealing with this issue requires the ML-developer to jointly design and optimize all the policies that are deployed on a system. A key contribution of this thesis is to demonstrate that each of these individual policies can be automatically combined in a multi-agent reinforcement learning setting to alleviate this challenge. We show that the functionality of modern OSs can be extended to provide "virtualization" to ML-developers, where they can focus on building one model at a time, and the OS can automatically integrate all of these individual models into a single large system-level policy.

To conclude this thesis, we outline some important research directions for extending the ideas presented in this thesis.

6.1 LOOKING FORWARD

6.1.1 ML Models

Safe exploration and deployment. RL fundamentally requires the agent to explore different actions in order to compare the empirical returns and learn. Context-free random explorations, such as entropy-based policy randomization, can bring the system into an unsafe region and lead to catastrophic outcomes. For example, when load balancing among several heterogeneous servers, random workload assignment can systematically overwhelm some servers while leaving others idle, which effectively reduces the service capacity. Moreover, future data-driven systems should ideally train in a live system — in order to avoid inaccuracies of a offline simulator [227]. Random exploration thus becomes a critical danger to a running system and may even inhibit further learning (e.g., by creating an insurmountable backlog of work that halts the generation of new learning experience). Moreover, learning methods that can provide provable guarantees on system behavior and runtime performance, can be central to future learned systems. The fundamental challenge in achieving this goal in deep RL is that the agent's superior performance often comes at the cost of using complex and not (yet) interpretable neural network. Following this thesis' line inquiry, of using assertions as a way to capture poor RL decisions and falling back to the original policy, many research questions are still open. For example, fallback policy can often dominate the system and prevent RL exploration (e.g., when initial RL policy is weak and easily triggers safety violation), how can we leverage the fallback policy experience for RL training as well? Can we robustly learn the system dynamics and find a safeguarding policy when the dynamics of the system is not unknown a priori?

Model-based learning. In this thesis, we mostly focus on model-free RL where we assume the complex dynamics (i.e., state transition map) is unknown or hard to model. However, system operators often do understand most parts of the system dynamics very well; it is usually only a small component that creates all the uncertainty and complexity. Intuitively, only learning the dynamics or response of the unknown component, rather than learning the control policy for the entire system end-to-end, should have lower sample complexity (i.e., need a smaller amount of data to train). Therefore, efficient model-based RL is viable to train in slow-to-interact environments (e.g., waiting to download a real video chunk without simulation, executing the actual job binary in a cluster). In general, however, it is yet unclear

how to optimally integrate a learned prediction model — which may be inherently inaccurate — with the end-to-end system control.

Adapting to changing workloads. The high level promise of RL is to automatically adapt and optimize for different kinds of system dynamics. In practice, however, the RL-trained agent can only generalize to a narrow set of environments that are share similar characteristics as the training environment. Hierarchical RL seeks to co-train a high-level planning agent with a low-level action-taking agent [**? ?**]. Traditionally researchers apply this approach to tasks that require sequential planning in a long time horizon (e.g., complex games with multiple sub-tasks to solve). Based on different stage of a task, the planner adaptively decides how to activate or tune the action-maker at the low level. This separation of controller may be applicable to dealing with the change in workloads. Further, this can help separate aspects of the decision making process that need to proceed at different rates of incidence. For example, controlling low-level hardware processes as well as high-level cluster level processes and policies. In principle, one can imagine a high-level agent that builds a model for different families of workload patterns and adaptively tunes the low level agent to adjust its policy for different actions.

APPENDIX A: ACCELERATED LEVENSHTEIN DISTANCE COMPUTATION

A.1 INTRODUCTION

The advent of high-throughput next-generation sequencing technology (NGS) has created a deluge of genomic data for computational analysis [11]. Efficiently processing this data requires the development of a new generation of high-performance computing systems that can efficiently handle such data. This new generation of application-specific and accelerator-rich computing systems are expected to gain performance, power, and energy improvements over traditional systems [10].

A crucial step in a significant number of NGS data analytics applications (e.g., variant discovery, genome-wide association studies, and phylogeny creation) is the mapping of short fragments of sequenced genetic material (called *reads*) to their most likely points of origin in the genome, popularly called the *short-read alignment* problem. This chapter presents the design and implementation of ASAP, an accelerator for computing Levenshtein distance [228, 229] (LD; used interchangeably with edit-distance) in the context of the short-read alignment problem. LD is a measure of the similarity between strings, which is computed by counting the number of single-character edits required to change one string into the other. LD computation is a prominent underlying mathematical kernel that is common to a large number of short-read alignment algorithms and tools (e.g., BLAST [230], Bowtie [159, 231], BWA [232], and SNAP [164]), and is responsible for 50% – 70% of their runtime [140].

ASAP represents a novel approach to accelerate the LD computation, in that it uses algorithmic approximations, and maps these approximations into hardware to significantly improve overall performance ($\sim 200\times$ compared to the CPU baseline). The core algorithm in ASAP leverages two key observations about the computation and datasets involved in the short-read alignment problem:

1. Although all the tools mentioned above calculate the exact value of LD between pairs of nucleotide strings, they use them only *to build a total ordering* (i.e., an ordered list) of the most likely points of origin in the genome. The best alignment is the pair of strings corresponding to the minimum LD in the ordered list. Hence, it is sufficient to only calculate the total ordering (in this instance, returning the pair that corresponds to the minimum LD), and not essential to compute the exact value of the LD. This distinction enables approximation in the computation of LD to gain performance, while preserving the overall accuracy of the alignment algorithm (which comes from the total

ordering).

2. Modern sequencing platforms (like the Illumina HiSeq 2500) represent a very low sequencing error regime ($\leq 1\%$) [233, 234], and modern alignment tools (mentioned above) have accurate candidate region-matching algorithms (described in Appendix A.2). Hence, LD computations process significantly more "matches" than "mismatches," in the majority of sequencing experiments.[1] The ASAP architecture uses this heuristic to accelerate LD computation (described in Appendices A.3.1 and A.3.2).

To take advantage of these observations, ASAP augments RaceLogic [235][2] using application heuristics, as well as hardware architectural optimizations to realize the design on FPGAs. In particular, this chapter proposes (i) a mechanism to encode LD computation parameters (e.g., *gap-penalties*; described further in Appendix A.2) into the ASAP architecture, making it possible to map the time taken to process a "match" exactly as a circuit delay. This mapping gives us the ability to tune the performance of ASAP to match data characteristics; and (ii) the use of "zero delay" circuit elements to explore large portions of the search space (LDs of substrings of the strings being compared) in parallel within one clock cycle, and to ignore parts of the search space that do not contribute to an answer, thereby saving energy. Overall, ASAP can compute alignments quickly ($\sim 200\times$ faster than the CPU baseline and $\sim 50\times$ faster than an equivalent RaceLogic design), and with the same accuracy as traditional software- or hardware-based alignment tools. We leverage reconfigurable FPGA devices to prototype ASAP, thereby allowing us to reconfigure the accelerator based on user decisions on input parameters (described in Appendix A.2), as well as to adapt the accelerator to input NGS datasets of varying read lengths.

Contributions. To summarize, the primary contributions of this chapter are as follows:

1. Presents a measurement-driven study that demonstrates that computation of LD represents a significant portion of the runtime of several short-read alignment programs.

2. Builds on top of the delay-based computation paradigm presented in [235] to encode gap-penalties as "zero delay" circuit elements. This allows us to calculate approximate the LD between strings by using combinational circuit elements. We prove the correctness of this encoding and demonstrate that the result of the approximation can be used as a proxy for computing LD in short-read aligners. That is, a tool using the approximation

[1] This is a facet of the accurate sequencing process and the thoroughly validated reference genome for human subjects. This observation will also apply to most model organisms whose genome has been extensively studied.

[2] RaceLogic uses propagation delay of circuit elements to perform computations.

and the accelerator produces alignments identical to those of tools based on traditional methods (e.g., BWA-MEM [232]).

3. Presents an FPGA-based implementation of the accelerated LD computation in the ASAP accelerator that leverages the coherent accelerator-processor interface (CAPI) [96, 236] for communication between the host and accelerator.

4. Demonstrates that ASAP on an FPGA is able to accelerate the runtime of the LD computation by $200\times$ compared to a Smith-Waterman-based and $40-60\times$ compared to a Landau-Vishkin based IBM Power8 CPU execution. As well as $5\times$ better that competing FPGA implementations.

5. Demonstrates that integration of the ASAP accelerator into a short-read alignment frameworks SNAP and BWA-MEM. In both cases this results in a $2\times$, $1.9\times$ performance improvement respectively, which is close to the Amdahl's law limits for these applications.

Other Applications. Our approach can be adapted to a variety of other problems in which a total ordering of LDs is computed. For example, in signal processing, where similarity between signals is computed [228]; in text retrieval, where misspelled words have to be accounted for in a dictionary [237]; and in computer-security where virus- and intrusion-detection requires comparison of signatures [238].

Organization. The remainder of this chapter is organized as follows. Appendix A.2 describes LD computation and its use in popular short-read alignment tools. Appendix A.3 briefly describes (i) a mathematical formalism for encoding computation in circuit delays; (ii) the approximation at the core of ASAP and prove its correctness; and (iii) presents the hardware architecture of ASAP leverages this approximation algorithm. Appendix A.4 presents the evaluation of the accelerator. Appendix A.5 compares the ASAP approach to other hardware accelerated approaches for computing LD, and, finally, we conclude in Appendix A.6.

A.2 LEVENSHTEIN DISTANCE COMPUTATION AND SHORT-READ ALIGNMENT

Traditional methods for aligning reads to a reference genome find the position (*locus*) of a single read in the reference by minimizing the maximum edit distance between the short read being aligned (called the *query*, and denoted by Q) and the reference genome sequence. The Smith-Waterman algorithm (SW) [239] and Needleman-Wunsch algorithm (NW) [240] utilize a dynamic programming-based algorithm to calculate the alignment score

(Levenshtein distance) between the read and a particular section R of the reference genome, accounting for base pair substitutions, insertions, and deletions. Both of these algorithms work by constructing a matrix S (which is used interchangeably with lattice) of size $l_Q \times l_R$, where l_Q and l_R are the lengths of the two strings, between which the edit distance must be calculated. Consider a matrix S in which the $(i,j)^{th}$ entry, $S(i,j)$, is the minimum edit distance between the sub strings $Q[1:j]$ and $R[1:i]$. $S(i,j)$ is recursively defined as

$$S(i,j) = min \left\{ \begin{array}{l} S(i-1,j) + \Delta(-,R_j), \\ S(i,j-1) + \Delta(Q_i,-), \\ S(i-1,j-1) + \Delta(Q_i,R_j) \end{array} \right\} \tag{A.1}$$

where Δ corresponds to input parameters called *gap penalties*. These Δ-parameters assign scores for insertion, deletion, match,[3] or mismatch between the sequences such that a more desirable outcome has a smaller score associated with it. The parameters $\Delta(Q_i, R_j)$, $\Delta(-, R_j)$ and $\Delta(Q_i, -)$ correspond to the match/mismatch, deletion, and insertion penalties respectively. These parameters are chosen to optimize the accuracy of alignments based on prior information about the sequences being compared (e.g., evolutionary information about mutations in a population [241, 242, 243]). This chapter describes the use of constant gap penalties (i.e., a fixed score is assigned to every gap between nucleotides). That is,

$$\left. \begin{array}{rcl} \Delta(Q_i,R_j) & = & \Delta(\text{Match}) \text{ if } Q_i = R_j \\ \Delta(Q_i,R_j) & = & \Delta(\text{Mismatch}) \text{ if } Q_i \neq R_j \\ \Delta(-,R_j) & = & \Delta(\text{Delete}) \\ \Delta(Q_i,-) & = & \Delta(\text{Insert}) \end{array} \right\} \forall R_i, Q_j. \tag{A.2}$$

Such gap penalties are are commonly used in DNA alignment (e.g., in NCBI-BLASTN, or WU-BLASTN [241]).

The NW algorithm computes a global alignment in which the entirety of the query is matched to the reference, as shown in Fig. A.1. It does so by computing the value of $S(m,n)$. The SW algorithm computes a local alignment and matches the largest (substring) of the query to the reference, and, hence, needs to calculate the minimum value in the row $S(m,-)$. For example, when the strings `AGCACACA` and `ACACAACT` are compared with constant penalties $\Delta(\text{Match}) = 0$, $\Delta(\text{Mismatch}) = 2$, and $\Delta(\text{Insert}) = \Delta(\text{Delete}) = 1$, we get the matrix described in Fig. A.1. The optimal alignment is then calculated from this matrix by finding the minimum weighted path (in S) from (m,n) to $(0,0)$ in the NW algorithm and $(m,\mathcal{N})$ to

[3]Gap penalties traditionally do not have match scores. We group them together for simplicity in our notation.

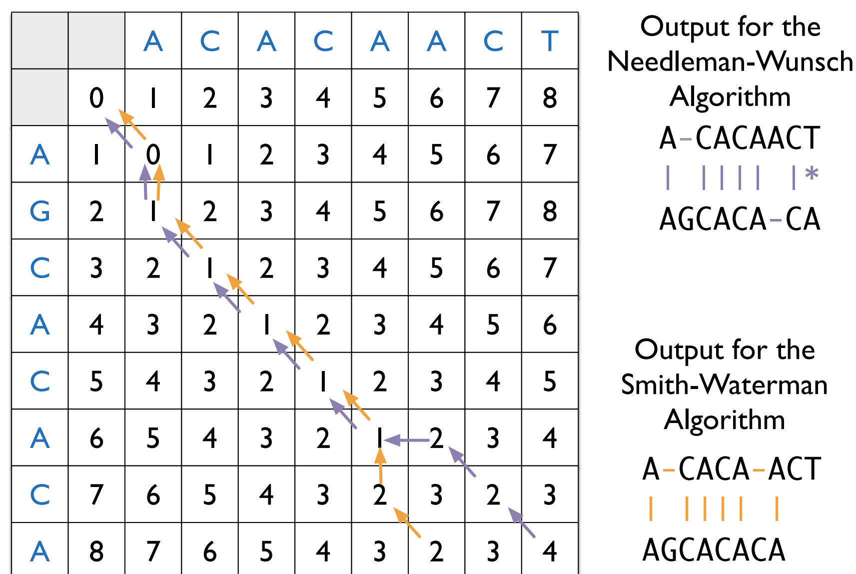


Figure A.1: The matrix S for the strings `AGCACACA` and `ACACAACT`, assuming $\Delta(\text{Match}) = 0$, $\Delta(\text{Mismatch}) = 2$, and $\Delta(\text{Insert}) = \Delta(\text{Delete}) = 1$. The colored paths from $S(8, 8)$ and $S(8, 6)$ to $S(0, 0)$ show the optimal alignments produced by the NW and SW algorithms, respectively.

$(0, 0)$ in the SW algorithm. $\mathcal{N}$ corresponds to the largest substring of the reference to which the query string maps with the lowest LD.

Although these methods are guaranteed to produce the optimal alignment, they are prohibitively expensive for whole-genome alignments because of $O(l_Q \times l_R)$ space and time complexity. Therefore, a large number of alignment tools are designed to heuristically reduce the search space required to find the optimal match of a query in the reference. An extensive amount of research, e.g., [159, 164, 230, 231, 232], has been conducted, focusing on indexing strategies for the reference genome to rapidly reduce the number of candidate locations that have to be searched. Most of these tools use some variant of a backwards search algorithm utilizing an FM-index [244] or a hash-table-like data structure. As a result of this reduction in the search space, linear-time heuristic algorithms like the Landau-Vishkin algorithm (LV) [245] (in addition to traditional algorithms like SW and NW) can be applied to the sequence alignment problem in SNAP [164], to compute edit distance accurately up to a particular number of mismatches (assuming that correct alignments have lower numbers of mismatches). Alg. A.1 describes the skeleton of these heuristic accelerated algorithms for single-ended read alignment [246]. The definitions of the `Build_Index`, `Candidate_Locations`, `Edit_Distance`, and `Find_Config` functions define different variants of these algorithms. For example, Table A.1 defines the BWA-MEM and SNAP alignment

Table A.1: Mathematical formulation of different aligners to fit them into the structure of Alg. A.1.

Function	**BWA-MEM** [232]	**SNAP** [164]
`Build_Index`	Burrows-Wheeler transform [247] of prefix trie	Ukkonen's algorithm [248]
`Candidate_Locations`	Prefix trie traversal	Hash table lookup
`Edit_Distance`	Smith-Waterman algorithm [239]	Landau-Vishkin algorithm [245]
`Find_Config`	Smith-Waterman algorithm [239]	Landau-Vishkin algorithm [245]

Table A.2: Distribution of runtime across the steps of Alg. A.1 for the SNAP tool aligning an in-sillico human genome with 50× coverage.

Lines in Alg. A.1	**% of runtime**	**# of calls**
Line 5	6.79	1.5×10^{10}
Line 6	18.59	6×10^{10}
Line 7	59.22	8.3×10^{11}
Line 8	9.25	1×10^{10}
Misc	6.15	–

tools by substituting these placeholder functions with specific algorithms.

We performed a profiling study of the SNAP aligner on an in-sillico (from an Illumina HiSeq 2500) whole human genome [249] with 50× coverage (i.e., each nucleotide of the reference is backed by an average of 50 reads that align to that base) on the Blue Waters[250] supercomputer. We chose the SNAP aligner in particular because it is significantly faster than other alignment tools like BWA and Bowtie. Also, as the LV algorithm used in SNAP has a linear time complexity, its comparison to ASAP as the CPU baseline is much more

Algorithm A.1 Algorithmic skeleton for single-ended short-read-alignment algorithms.

Input: NGS Read Dataset, Reference Genome
Output: Aligned positions and mapping of reads in Reference Genome

1: $ngsdata \leftarrow$ Set of reads
2: $reference \leftarrow$ String(s) corresponding to a reference
3: $index \leftarrow$ `Build_Index`$(reference)$
4: $alignment \leftarrow \emptyset$
5: **for** $read \in ngsdata$ **do**
6: $\quad locs \leftarrow$ `Candidate_Locations`$(read, index)$
7: $\quad opt \leftarrow \arg\min_{loc \in locs}($`Edit_Distance`$(read, loc))$
8: $\quad config \leftarrow$ `Find_Config`$(read, opt)$
9: $\quad alignment \leftarrow alignment \cup config$
10: **end for**
11: **return** $alignment$

challenging. Table A.2 describes the distribution of runtime across for the SNAP aligner for corresponding steps of Alg. A.1.[4] These measurements, along with static analysis of Alg. A.1, show the following:

1. The LD computation corresponds to nearly 60% of the running time of the SNAP aligner.
2. The LD computation is one of the most frequently called algorithmic kernels in the alignment process (on average called 54.1 times per read).
3. The LD kernel is used to build a total ordering of all candidate locations for a read in the reference; refer to Line 7 of Alg. A.1.
4. The backtrack-based alignment [239, 240] is computed only for the best-matched location in the reference.
5. The remaining portion of SNAP's runtime (after the LD computation) is spent in either memory or IO bound computation (e.g., hash table look-ups and reading/writing files). This part is unsuitable for acceleration on PCIe-based devices because of the time-cost associated with performing data transfer over the bus.

A.3 DESIGN OF THE ASAP ACCELERATOR

This section describes the approximation algorithm that drives the design of ASAP, provides a proof for its correctness, and describes its implementation in programmable hardware. Appendix A.3.1 briefly summarizes the RaceLogic chapter [235], describing an formalizing the encoding the computation of LD scores into circuit propagation delay. Appendix A.3.2 describes the approximation at the heart of ASAP: using the ability to directly tune the performance of the algorithm to input-data characteristics (i.e., using circuit propagation delays encode both the algorithm and its computation time), we show a method to chose appropriate propagation delays to compute approximate answers for LD while maintaining their total ordering (i.e., satisfy the application invariant for correctness). Finally, Appendices A.3.3 and A.3.4 describes the ASAP FPGA implementation.

[4]Note that some steps of the SNAP aligner implementation includes a variety of other miscellaneous tasks, e.g., memory allocation, IO. These are collectively described in the "Misc" category. Also note, the SNAP aligner is optimized to perform asynchronous pre-fetch based disk IO. Hence wait time for IO is minimized.

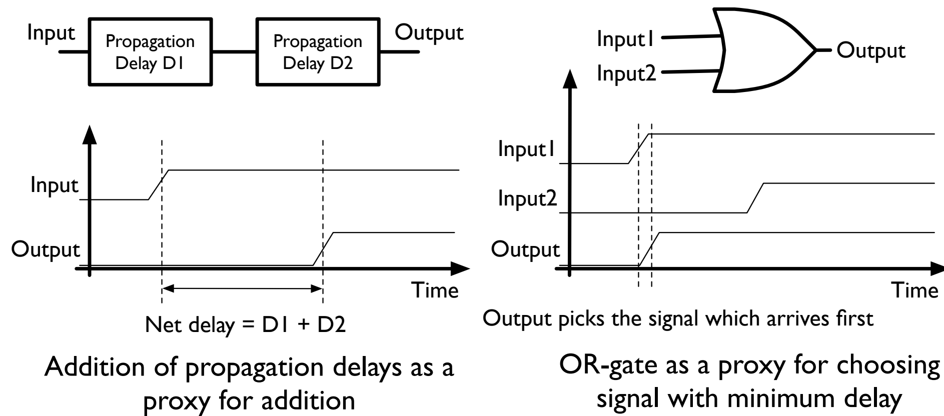


Figure A.2: Computing with propagation delays: Delay-based proxy for the addition operator is a series connection, and the proxy for the min operator is the `OR` gate.

A.3.1 Encoding LD Computation as Propagation Delays

The core idea is to map addition and minimization, the two mathematical operators necessary for the recursive computation defined in (A.1), to particular topologies of circuit elements. Fig. A.2 illustrates the mapping explained here:

1. If circuit elements are combined in series, the net propagation delay of a signal is the sum of the propagation delays for all of the individual elements. This construction is a proxy for addition.

2. If two circuit elements are connected to an OR gate, the signal that emerges out of the OR gate corresponds to the signal that arrived first at the gate. This construction is a

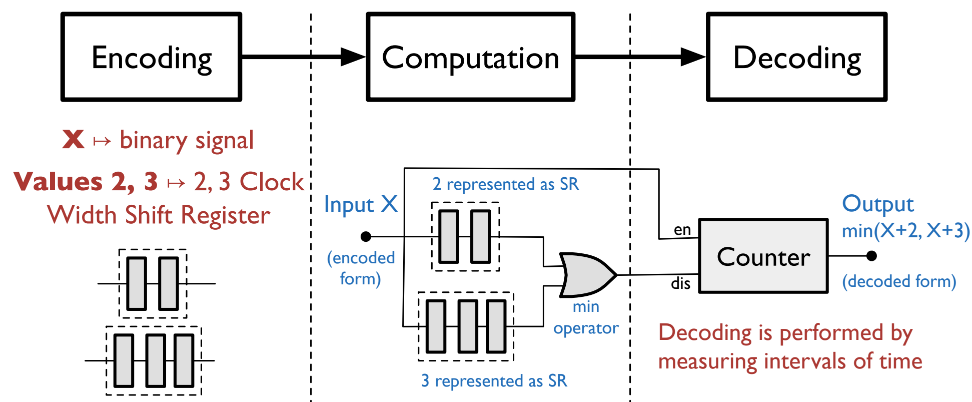


Figure A.3: Example of the encoding, computation, and decoding phase for computing "min$(X+2, X+3)$" using the circuit-delay proposed in RaceLogic [235]. Note that we present this example using shift-registers for delay elements as opposed to comparators proposed in [235].

proxy for the minimization operator (in particular, the rising edge of the OR gate's output computes a minimization in time).

For example, Fig. A.3 demonstrates the computation of "$\min(X+2, X+3)$" using the aforementioned delay based computing. In the example, X corresponds to an arbitrary input signal that is represented in the delay encoding, the 2- and 3-length shift register serving as the delay element implementing the $\cdot + 2$ and $\cdot + 3$ operator respectively, the OR gate serves as the minimization operator and the counter serving as the decoder.

We formalize this delay based computation succinctly in the following lemma.

Lemma A.1. Propagation-delay-based computation can occur on a tropical semiring structure $\mathcal{T}$ over $\{0\} \cup \mathbb{Z}^+$ (i.e., time measured in clock ticks) that defines a binary addition operation, a minimization operator (using an OR gate), and a maximization operator (using an AND gate).

The delay-based proxies for the addition and minimization operators can be used by replacing the LD values $S(i,j)$ in (A.1) with the equivalent propagation delays. The resulting circuit represents the application of the addition and minimization operators in the computation of $S(i,j)$. Fig. A.4 shows the structure of the circuit that produces this computation. It is composed of a lattice of $l_Q \times l_R$ delay elements (DEs). The connections in the lattice build on the recursive definition of S: each DE $\mathcal{D}(i,j)$'s inputs are connected to the outputs of the preceding elements $\mathcal{D}(i-1,j-1)$, $\mathcal{D}(i-1,j)$, and $\mathcal{D}(i,j-1)$, and its outputs are connected to the input of $\mathcal{D}(i+1,j+1)$, $\mathcal{D}(i+1,j)$, and $\mathcal{D}(i,j+1)$. At a high level, each DE is composed of three delay blocks: (i) D_M (delay due to match or mismatch at (i,j)), (ii) D_I (delay due to insertion at (i,j)), and (iii) D_D (delay due to deletion at (i,j)). This design is specialized for FPGAs in Appendix A.3.3)

The computation can be started by injecting a high signal (logic value 1) at the inputs of index $\mathcal{D}(0,0)$ in the array. The time-encoded value of the LD is then found by measuring the propagation delay of the signal exiting the array of delay elements. Note that the delay-based computation can be applied to all variants (SW, NW, and LV) of the LD computation as follows.

1. The delay-based version of the SW variant can be computed by measuring the delay between the introduction of the input signal in the lattice, and its emergence at any of the delay elements on the last row, *i.e.,* $(l_R, -)^{th}$ DE. Fig. A.4 illustrates this configuration.

2. The delay-based version of the NW variant can be computed by measuring the delay

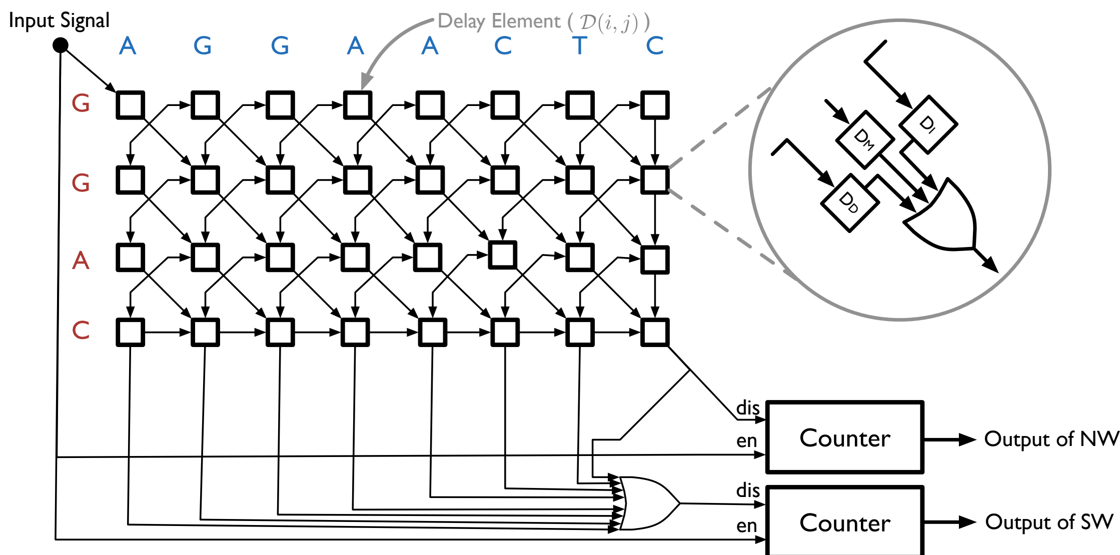


Figure A.4: High-level design of the ASAP accelerator to compute the minimum edit distance between two strings. The accelerator lattice is of size $l_Q \times l_R$, where l_Q and l_R are the sizes of the query and reference, respectively.

between the introduction of the input signal in the lattice, and its emergence at the $(l_R, l_Q)^{th}$ DE. This configuration is also shown in Fig. A.4.

3. The delay-based version of the *LV* variant can be computed by assigning the maximum permissible LD as the result of the computation. This represents the "timeout" with which the signal wavefront will emerge from the DE lattice. If the timeout is triggered, the maximum value of LD, as set by the user, is used as the result of the computation. One delay element and one AND gate (not shown in the Fig. A.4) suffice to implement the timeout.

A.3.2 Approximating LD Computations in ASAP

A key aspect of the aforementioned method is the mapping of gap-penalty parameters (Δ-parameters) to their corresponding circuit delays. The ASAP accelerator uses this mapping both to encode the approximation (mentioned in Appendix A.1), and to reduce the time taken to do the "match"-based computation. Both actions are formally stated below.

Definition A.1. A *Delay Encoding Function* $\mathcal{E} : \mathbb{R} \rightarrow \mathcal{T}$ is a mapping between the set of real numbers and its propagation-delay-based representation. $\mathcal{E}$ is constrained to obey the Cauchy functional equation ($\mathcal{E}(x+y) = \mathcal{E}(x) + \mathcal{E}(y)$).

More general delay encoding functions can be considered, for example in analog circuits, where circuit elements do not exhibit linear behavior for all inputs. We constrain ourselves to those that satisfy the Cauchy functional equation (CFE) because of simplicity in proving of correctness of the transformation. Although the domain of $\mathcal{E}$ can be the set of real numbers $\mathbb{R}$, the ASAP implementation presented in this chapter uses integer or rational gap penalties which can be easily mapped to integer delay values (which can further be represented as a multiples of the clock width).

Definition A.2. A δ-parameter is the time-encoded representation of a user-inputted Δ-parameter. That is

$$\begin{aligned}\delta(\text{Insert}) &= \mathcal{E}(\Delta(\text{Insert})) \\ \delta(\text{Delete}) &= \mathcal{E}(\Delta(\text{Delete})) \\ \delta(\text{Match}) &= \mathcal{E}(\Delta(\text{Match})) \\ \delta(\text{Mismatch}) &= \mathcal{E}(\Delta(\text{Mismatch}))\end{aligned} \tag{A.3}$$

These parameters are used to define the delays in the D_M, D_I, and D_D blocks. Note that we have assumed that $\Delta(\text{Match}) = 0$, and thus $\delta(Match) = \mathcal{E}(0)$ is also 0 based on Definition A.1.

Based on definitions A.1 and A.2, we now show that any encoding of δ-parameters based on $\mathcal{E}$ produces the same ordering of LDs as the original algorithm.

Lemma A.2. When a query string Q and a reference string R are compared under the traditional (see (A.1)) and delay-based algorithm for computing LD at loci $l_1, \ldots, l_n$ of the reference, to produce LDs $e_1, \ldots, e_n$ and propagation delays $d_1, \ldots, d_n$, respectively, then $d_i = \mathcal{E}(e_i)$, and consequently

$$e_i \leq e_j \iff \mathcal{E}(e_i) \leq \mathcal{E}(e_j) \iff d_i \leq d_j \; \forall i, j. \tag{A.4}$$

Lemma A.2 is sufficient to show that using the ASAP accelerator to compute LD in the context of Alg. A.1 (in line 7; i.e., using an "arg min" operator over the results of multiple executions of the ASAP accelerator) produces the same result as the traditional algorithm (without requiring the computation of the inverse for $\mathcal{E}$). A key observation in the formalism of $\mathcal{E}$ is that the choice of the numerical values of δ can be tuned to directly change the performance of the accelerator, as they corresponds to circuit propagation delays. That is, the parameters and inputs to the accelerator jointly define the net propagation delay of

the circuit. Below we demonstrate one such transformation, which forms the core of the approximation used in ASAP.

Lemma A.3. When a query string Q and a reference string R are compared at loci $l_1, \ldots, l_n$ of the reference, they produce LDs $e_1, \ldots, e_n$ for gap penalties Δ, and LDs $e'_1, \ldots, e'_n$ for gap penalties $\Delta + k$, for some number k. The e'_i obey the relationship: $e'_i = e_i + n_i k$, for some $n_i \in \mathbb{Z}$ such that $(n_i \geq 0) \wedge (e_i \leq e_j \iff n_i \leq n_j)$, and consequently

$$e_i \leq e_j \iff e'_i \leq e'_j \; \forall i, j. \tag{A.5}$$

Our algorithm for the approximation at the core of ASAP uses Lemmas A.2 and A.3 to select values of the delay-encoded parameters that correspond to minimizing the time taken to process a dataset. For example, to optimize performance for our observed case of most nucleotides corresponding to "matches," we modify the gap-penalties to set the match penalty (i.e., $\delta(\text{Match})$) to 0 cycles[5]. This transformation uses a two-step process to convert (encode) user-inputted Δ-parameters into δ-parameters:

1. $\Delta \mapsto \Delta + k$, choosing k so that $\Delta(\text{Match}) = 0$ after the transformation;

2. $\Delta + k \mapsto \mathcal{E}(\Delta + k)$, with $\mathcal{E}(x) = mx$ to produce the required delay value.[6]

As a result, the parameters in the LD algorithm are tweaked to better suit the delay-based computation hardware. The answer (i.e., the exact values of LD) produced by this approximate version of the algorithm is not identical to that produced by the original algorithm. However, based on the aforementioned lemmas, we can see that the total ordering created by the approximated LDs is identical to that of the original algorithm. Furthermore, assuming that most nucleotide comparisons are matches (which is true for the indexed reference-based techniques described in Appendix A.2), this encoding ensures that (almost) zero time is taken to explore large portions of the search space that correspond to matches. We explore the relation of this optimization to timing closure on the FPGA design in Appendix A.3.3. In other re-sequencing experiments, where "matches" do not represent the common computation, a user can set $\delta(k) = 0$ for $k \in \{\text{Insert, Delete, Mismatch}\}$. Note that in our formulation of the problem (as described in Appendix A.2), $\Delta(\text{Match})$ is required to be the minimum positive value amongst all the Δ-parameters.

[5]True "0 cycle" propagation delay is not possible because of finite combinational and wire delays in the circuit. Here we imply that the computation is done in combinational logic, whose propagation delay is much much lower than the clock width of the circuit (i.e., 0 time). This is explained further in Appendix A.3.3.

[6]The choice of k and m has to ensure that none of the encoded gap penalties are negative. As the encoded values represent circuit propagation delays, negative numbers are meaningless.

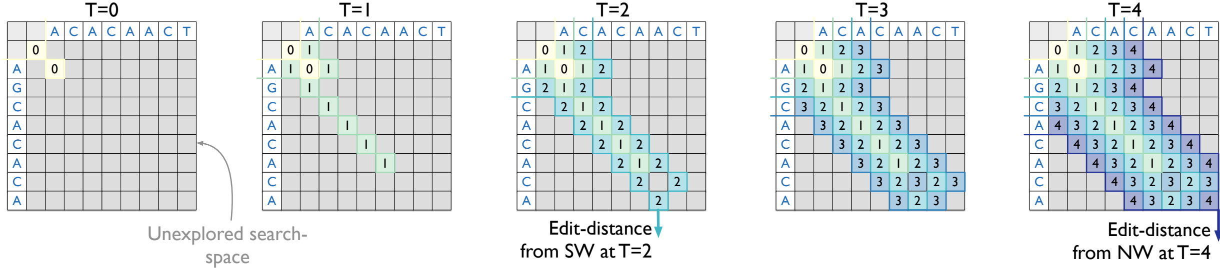


Figure A.5: An example of the ASAP accelerator processing the same inputs used in Fig. A.1. The signal wavefront is shown progressing through the ASAP lattice until the outputs of the SW and NW algorithms are produced in 2 and 4 clock cycles, respectively. The values in the matrix represent the clock cycles in which the corresponding DEs were enabled.

Consider the example of computing the LD between the strings `AGCACACA` and `ACAACAACT`, presented in Appendix A.2. Based on our encoding mechanism ($k = 0, m = 1$), we compute the δ-parameters of the ASAP accelerator as $\delta(\text{Match}) = 0$, $\delta(\text{Mismatch}) = 2$, and $\delta(\text{Insert}) = \delta(\text{Delete}) = 1$. Fig. A.5 illustrates the propagation of the signal wavefront through the ASAP accelerator for that example. The accelerator produces an output for the SW notion of LD (local alignment) in two clock cycles and the NW notion of LD (global alignment) in four clock cycles. The figure shows the portion of the array explored and the value of the propagation delay at each element $\mathcal{D}(i, j)$ of the lattice. Note that some portions of the array are not explored at all (e.g., for SW and NW, only 25 and 53 DEs out of a total of 81 are triggered, respectively). This design thus provides a large savings in both time (using "zero delay" circuit components for the most commonly used computation) and power (clock-gating unused DEs with their input signals ensures minimal power usage) compared to traditional methods.

To summarize, using the encoding of δ-parameters described in this section, the ASAP accelerator has two clear advantages over traditional techniques:

1. *Faster Processing*: One can explore large portions of the search space in a small amount of time by setting delay parameters appropriately.

2. *Energy Savings*: DEs in the ASAP lattice are used only when their output can contribute to the answer; otherwise, they are switched off to save energy. This can be accomplished by clock-gating the DEs with their input signal.

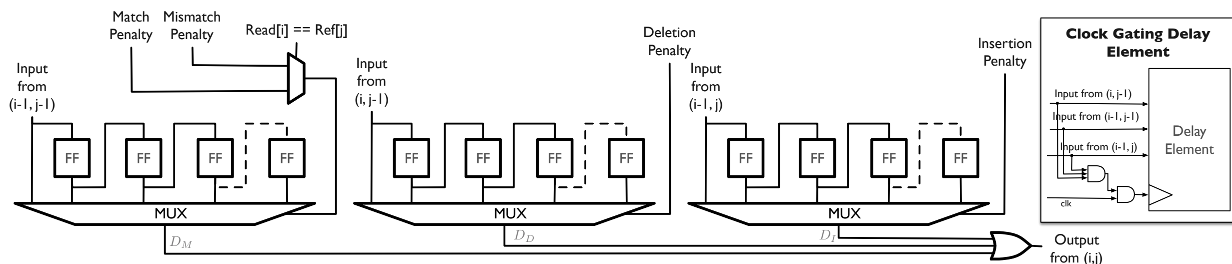


Figure A.6: Design of a single delay element $\mathcal{D}$ in ASAP. The DE is composed of three separate delay units corresponding to D_M, D_I, and D_D in Fig. A.4.

A.3.3 ASAP: The FPGA Implementation

Why FPGA?

The techniques discussed so far in the chapter represent an approximation technique and architecture, one which can be implemented ASICs, FPGAs, or any other platform. The original RaceLogic design was demonstrated in simulation as an ASIC [235]. However, some key characteristics of the short-read alignment problem and the ASAP architecture make ASAP particularly suitable for FPGAs, as they offer programmability and reconfiguration. The ASAP accelerator is runtime-programmable only for changing the values of gap penalties. The input data size, which defines the size of the accelerator lattice, is fixed at compile time. To allow users to sweep experiment such "meta-parameters" (i.e., input data size, gap-penalty bit-width, and input encoding), ASAP is designed to be re-synthesized and re-programmed on an FPGA. Potentially, the use of partial reconfiguration can allow users to change these parameters on the fly. We leave this possibility for future work. We discuss the advantages of the ASAP design compared to the commonly used systolic array based design (e.g., [251, 252, 253, 254, 255]) in Appendix A.5.

Design of a Delay Element

The overall architecture of the ASAP accelerator is shown in Fig. A.4. Fig. A.6 shows the design of a single DE. A DE utilizes sequential logic in the form of a shift-register to add a user-specified amount of delay. Each DE has (i) three input signals (representing input wavefront) that connect it to its preceding DEs in the grid, (ii) two input signals representing the nucleotides being compared by the element, and (iii) three input signals representing the δ-parameters. Each DE has one output signal representing the propagated wavefront after the delay has been added. The match, mismatch, insertion, and deletion delay penalties

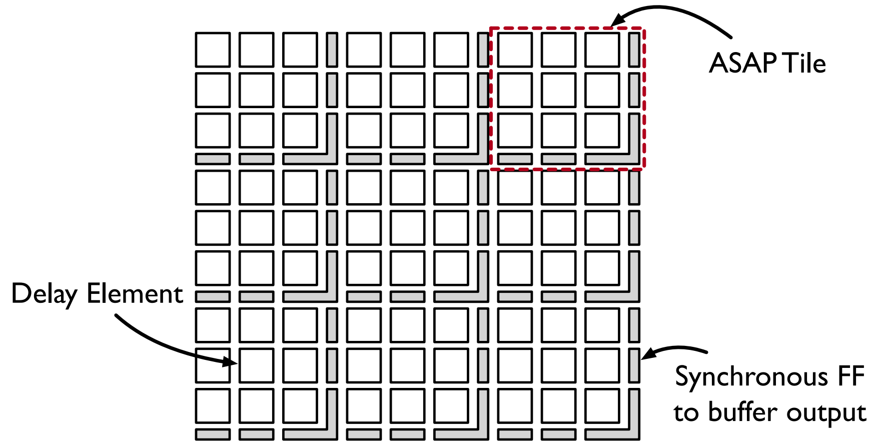


Figure A.7: The architecture of the ASAP accelerator in terms of tiles whose output is buffered by clock synchronous flip-flops (FFs).

are defined in terms of multiples of the clock period. When the input signal wavefront first reaches an element, it is propagated through a shift register to create delay. Based on the gap penalty specified for match/mismatch, insertion and deletion, the DE propagates the input signals to the output. The output of each flip-flop in the shift register is muxed to allow for the selection of the bit corresponding to the gap-penalty of the block (illustrated in Fig. A.6). The ASAP array allows the user to program (i.e., dynamically set at runtime) the values of the select lines of these `MUX`s. This provides the ASAP array with a degree of programmability, allowing it to be reused across computations that merely require re-parameterization of the gap-penalties. Changes in input-sizes, or the dynamic range of the gap penalties (i.e., number of bits required to represent the gap-penalties) requires a re-synthesis and reconfiguration of the accelerator on the FPGA.

As described in the motivating example for the ASAP accelerator given in Fig. A.5, the power of the ASAP accelerator is that it can explore a large portion of the search space of possible mappings between the query string and the reference within a clock cycle by setting $\delta(\text{Match}) = 0$. This improvement in computational speed can be coupled with a decrease in energy consumed by the accelerator by clock-gating the DE (illustrated in Fig. A.6) with the input signal.

The approach mentioned above has problems with long chains of combinational logic and may lead to timing violations on large lattices of DEs. To get around this problem, larger lattices of delay elements are composed by using the smaller tiles of ASAP accelerators (for which the timing violations do not occur) and by adding a sets of clock-triggered flip-flops between the tiles to break the chains of combinational logic (see Fig. A.7). Further, the diagonal tile crossing (i.e., the flip-flops at the lower right corner of the tile) corresponds

to a 2 cycle delay (i.e., two flip-flops in serial). Although the additions of the tile flip-flops changes the results of ASAP from what was described in the last section, the overall total-ordering is preserved, as this constitutes a constant addition of delay to all outputs of the ASAP accelerator. Each tile is synthesized, optimized, and placed-and-routed separately by defining separate design partitions. This approach prevents the compiler from performing optimizations across partition boundaries [256]. This approach also ensures that unintended wiring delays do not creep into the netlist of the ASAP accelerator.

The counter that decodes the delayed signal output from the ASAP lattice (shown in Fig. A.4) is designed based on a computation of the number of clock cycles for the signal wavefront to emerge from the lattice. The bit-width of this counter, N_o, is calculated from the sizes of the input strings and the user-input gap-penalty parameters, and is given by

$$N_o = \left\lceil \log_2 \min \left\{ \begin{array}{l} \delta_I l_Q + \delta_D l_R, \\ \delta_M l_Q + \delta_D (l_R - l_Q) \end{array} \right\} \right\rceil . \tag{A.6}$$

This expression is an upper bound (albeit a loose one) on the maximum delay caused by a DE.

Scalability Issues in the ASAP Accelerator

There are challenges involved in scaling the ASAP accelerator to large input sizes and large gap penalties. Those challenges can be addressed as follows:

1. *Large Input Sizes.* The size of the reference and read strings being compared in the ASAP accelerator plays a role in the size of the lattice defined by the ASAP accelerator. The size of the accelerator grows as $O(l_Q \times s)$ with the input size[7]. The tile size parameter defines a tunable knob to control the critical combinational path in the circuit. It can be used to trade off performance against meeting timing closure as the size of the accelerator grows to a significant portion of the resources available on the FPGA. Appendix A.4 demonstrates our scaling experiments with the accelerator.

2. *Large Gap Penalties.* A large dynamic range of the gap-penalty values negatively affects the ASAP accelerator, as it increases the size of the shift-registers and multiplexers in the DE (see Fig. A.6). We work around this problem by using BRAM-based shift registers, which can be $\sim 10^3$ bits long (without intermediate routing). In general, we

[7]This corresponds to quadratic growth in size of the ASAP lattice (i.e., $O(n^2)$) when $l_Q = s = n$.

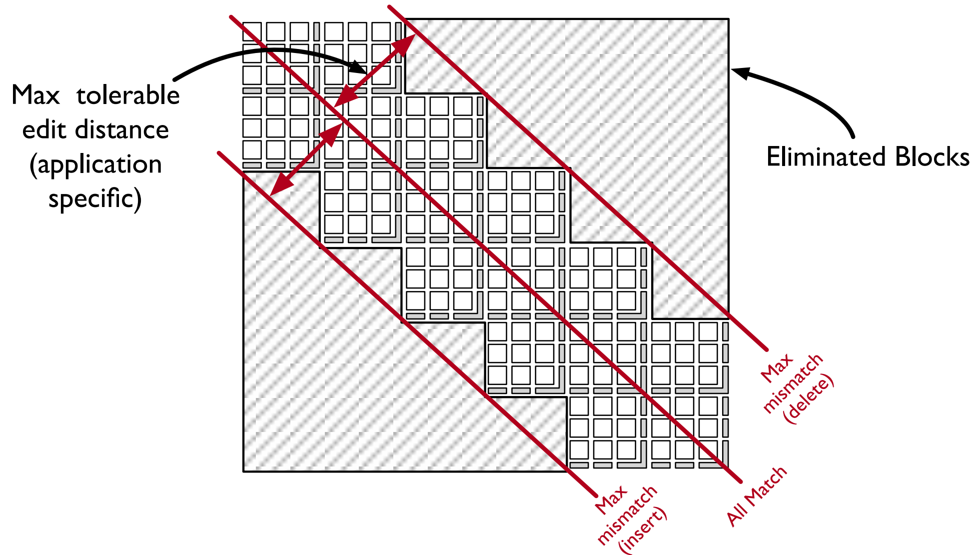


Figure A.8: Elimination of unused tiles from the ASAP lattice in the case of LV variant of the LD algorithm

do not expect large gap penalties to be a problem for genomic sequences (as opposed to protein sequences), for which the dynamic range in gap-penalties is low.

3. *Potentially Unused Tiles.* Fig. A.5 shows that a large part of the ASAP array is not involved in computation when the input strings have low LD (which is indeed the case in the short read alignment problem). There are several ways to tackle the problem of unused tiles across the three variants of the LD computation (i.e., SW, NW, and LD). As mentioned earlier, in the case of SW or NW, clock-gating individual delay elements ensures minimal power consumption. Further, in the LV case, as a the worst case LD is specified, we can use this information at compile (in this case synthesis) time to eliminate part of the ASAP lattice that will not contribute to an answer. Fig. A.8 illustrates such an elimination on an 18×18 lattice with a maximum of 6 insertions or deletions permitted, resulting in a $56\%(= {}^{20}/_{36} \times 100)$ reduction in area.

Issues with Timing Closure

Computing with propagation delays is disadvantaged by the fact that thermal dissipation and temperature variations at different parts of the FPGA chip to change the physical time associated with unit delay. However, the ASAP accelerator is resilient to these thermal changes up to the maximum operating temperature of the FPGA (i.e., timing violations

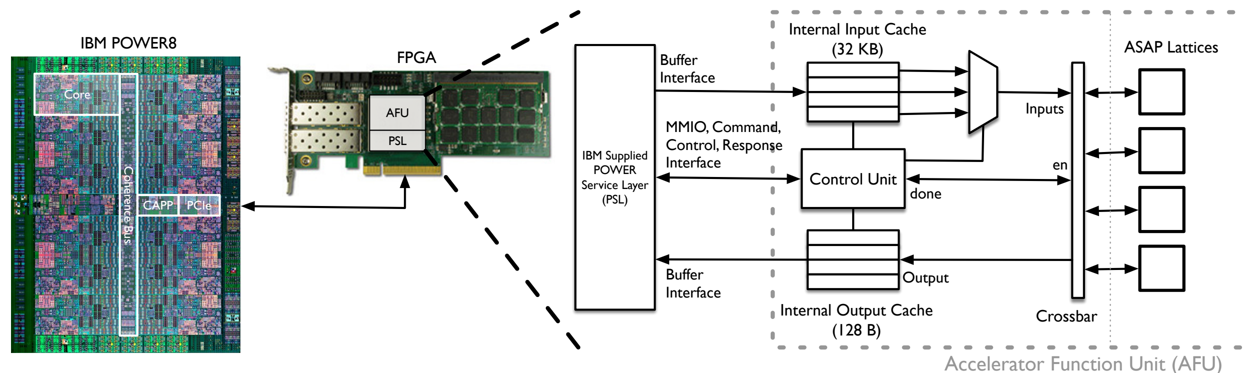


Figure A.9: The design of the interface between the host Power8 processor and the FPGA running the ASAP accelerator using the CAPI interface. The diagram assumes an ASAP accelerator that computes on input strings that are 64 nucleotides long and encoded as 2 bits per nucleotide.

do not occur). Further, only delays that are multiples of the clock period can affect the computed LD. The tile length serves as a tunable knob between runtime performance and worst case negative slack for the circuit. This slack is enforced by the compiler (e.g., Xilinx Vivado, Altera Quartus) as only values of tile length for which timing closure can be met can be used in the FPGA. Furthermore, the counters in Fig. A.4 that measure edit distance are synchronously triggered by the clock, thereby ensuring that all delay-based LDs are computed as multiples of the clock cycle.

Encoding Input Sequences

The implementation of the ASAP accelerator assumed use for genomic data, implying that the entire alphabet can be represented in two bits (i.e., `A`, `C`, `G` and `T`). The bases `N`, `-`, `R`, `Y`, `K`, `M`, `S`, and `W` (which represent an unknown or ambiguous nucleotide) are removed from the alphabet. Our design could potentially be extended to larger alphabets, e.g., for protein sequence alignment.

A.3.4 Host-to-Accelerator Communication via CAPI

Communication between the host and accelerator is implemented using the CAPI interface [96, 236] provided on an IBM Power8 CPU. The CAPI interface gives an accelerator (a PCIe-attached FPGA) coherent access to the virtual address space of a process running on the host CPU, with all address translations from virtual to physical memory done in the CPU. Fig. A.9 shows the interface and mechanism by which the host CPU communicates

with the ASAP accelerator. The Power8 is a superscalar symmetric multiprocessor, that has 12 cores per chip, with up to 8 hardware threads per core. All cores have access to shared memory through a PowerBus (shared memory bus). The Coherent Attached Processor Proxy (CAPP) enables the interface (CAPI) by maintaining a directory of cache lines held by the processor and providing coherency by snooping the PowerBus on behalf of the accelerator (or any other PCIe device). The PCIe host bridge provides connectivity between the CAPP and the Power Service Layer (PSL) on the FPGA over the PCIe bus. The PSL on the accelerator acts as a proxy for the CAPI protocol on the FPGA, communicating between the CAPP and the Accelerator Functional Unit (AFU). The AFU contains the custom acceleration logic and reads/writes coherent data across the PCIe. The PSL unit runs at the same speed as the PCIe bus (250 MHz). It contains a memory management unit (MMU) to handle address translation on the accelerator side on its copy of the processor's cache directory.

The AFU interacts with the PSL to provide word-level read and write commands. If these requests are made to cache lines (which are 1024 bits long) in a shared or exclusive state on the device, they are served locally. Otherwise the PSL interacts with the CAPP over the PCIe bus to attempt virtual to physical address translation, loading of the cache line from main memory (if it is already not present in the processor's cache), moving (or copying of) the cache line to the PSL, and changing the coherence of the cache line in the processor's directory [236, 257]. We use the AFU in dedicated mode, meaning only one MMU context is supported by the accelerator. That is, only one user-space process can use the accelerator at one time.

Fig. A.9 shows the configuration of the interface to the PSL for an ASAP accelerator that computes on two 64-bp strings, with each nucleotide encoded by two bits. Hence the accelerator takes 256-bit inputs (64 bp $\times$ 2 bits/bp $\times$ 2) and produces a propagation delay measurement encoded in 32 bits (to keep with the signed integer implementation in short-read aligner), which is the number of clock cycles for the signal to emerge from the ASAP accelerator (depending on whether the SW or NW algorithm is used). There is an internal 32 kB cache, which has a 1024-bit input port connected to the PSL, and a 1024-bit output port that is connected to the input of the ASAP accelerator. This cache is configured in a modified FIFO configuration; each entry in the FIFO contains multiple input cases (in this case, four). A 4×1 MUX controlled by the AFU control unit is responsible for producing 256 bits at a time from the 1024-bit input. The AFU packs the 32 bit outputs from the ASAP array into 1024 bit cache-lines before writing them back to the address space of the host over DMA. The AFU uses the work element descriptor (WED; [236]) to communicate the pointer to the input and output, as well as the progress of the accelerator.

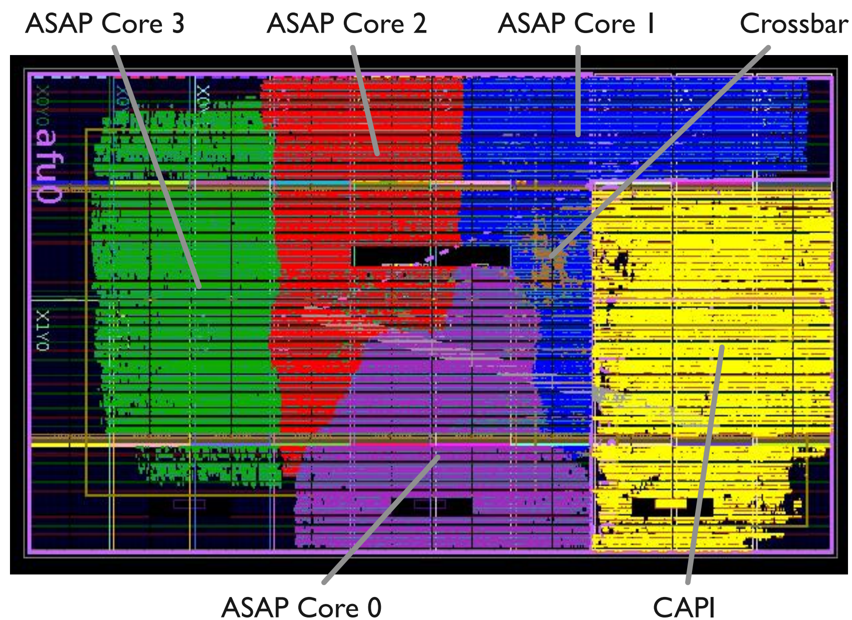


Figure A.10: Layout of the accelerator on the Xilinx Virtex 7 XC7VX690T FPGA. The design implemented above has 4 instances of the ASAP accelerator and the IBM CAPI interface for host-accelerator communications.

A.4 EVALUATION AND DISCUSSION

Experimental Setup. The ASAP accelerator is implemented in Chisel [43] and can potentially be compiled across FPGAs by Xilinx and Altera. The host-accelerator interface (which utilizes IBM CAPI) is implemented in VHDL and is specific to an IBM Power8 S824L system with an Alpha-Data ADM-PCIE-7V3 board (that uses a Xilinx Virtex 7 XC7VX690T FPGA) clocked at 250 MHz. All measurements (baseline CPU as well as FPGA-based) were done on this machine. Fig. A.10 illustrates the layout of four ASAP lattices and the CAPI based interface on the Virtex 7 FPGA mentioned above.

Input Data & Validation. All inputs for the experiments presented in this section are derived from the human reference genome `hg38` by simulating [249] 100 million reads of appropriate length. The read simulation introduced random mutations and simulated sequencing-error models from an Illumina HiSeq 2500 with a 0.1% sequencing error rate. We verified the correctness of our implementation through comparison with (i) answers generated from the software tools (i.e., BWA [232] or SNAP [164]); (ii) the ground truth values generated by the simulator.

The remainder of this section is organized as follows. In Appendix A.4.1 we discuss microbenchmark performance (in terms of runtime, communication bottlenecks, FPGA resource

Table A.3: Comparison of median run-time for LD computation on CPU and ASAP (SW & LV configurations). Rows marked with "*" are simulated results. The LV configuration uses $k = 1/4 \times$Read Size.

	SW Configuration			**LV Configuration**		
Read Size	**CPU**	**ASAP**	**Speedup**	**CPU**	**ASAP**	**Speedup**
64	1890 μs	10.3 μs	183×	238 μs	6.8 μs	34.8×
128	2083 μs	10.7 μs	194×	497 μs	10 μs	49.7×
192*	3326 μs	16.4 μs	203×	729 μs	16.3 μs	44.7×
256*	3906 μs	17.2 μs	219×	944 μs	17.2 μs	54.9×
320*	4484 μs	18.9 μs	237×	1190 μs	18.8 μs	63.3×

utilization, and energy consumption) of various configurations of the ASAP accelerator. Then in Appendix A.4.2 we discuss the end-to-end performance of integrating the ASAP accelerator into the SNAP [164] and BWA-MEM [232] aligners.

A.4.1 Microbenchmark Performance

Performance of the Accelerator

In this section we compare the performance of ASAP-accelerated LD computation against their respective CPU baselines. Here we do not account for time taken to perform disk IO, serialization/de-serialization (i.e., parsing inputs, writing in-memory data structures to disk), and reference lookups (see Appendix A.2) that are required as a part of the end-to-end computation. These other factors are described in Appendix A.4.2.

SW Configuration. The ASAP accelerator is approximately 200× faster than the baseline C implementation of the SW algorithm for computing LD that is optimized to use single instruction multiple data (SIMD; e.g., Intel AVX instructions) and simultaneous multi-threading (SMT; e.g., Intel Hyperthreads) based multi-threading [258]. The baseline implementation exploits inter-task parallelism (i.e., data parallelism) by processing multiple reads across threads. Table A.3 describes the comparison of the performance of a single lattice ASAP accelerator. Having multiple cores on the CPU or multiple ASAP lattices on the FPGA does not change this comparison, as each core/lattice is expected to be computing a separate unrelated instance of the LD computation. The performance of ASAP depends not only on the size of the inputs, but on the inputs themselves (i.e., more mismatched inputs mean a higher computation time). Hence we present all ASAP measurements as the median across all the randomly generated reads. We observe that a single ASAP lattice

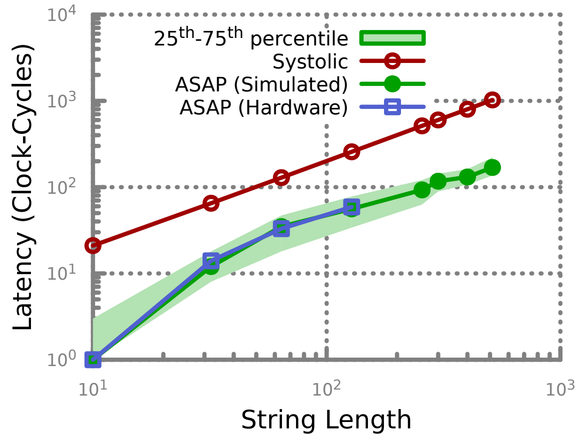
shows $\sim 200\times$ speedup relative to a single CPU core (containing 8 SMT threads and SIMD units), with potential improvements in performance with growing input size (see Table A.3). Overall, a Power8 CPU chip contains six such cores, whereas our implementation of ASAP can scale to four lattices (see Fig. A.10). Hence a chip-to-chip comparison yields a $133\times$ improvement in performance.

Fig. A.11(a) illustrates the latency of the accelerator (without the overhead of communication between the host and device) in computing LD (in the SW sense) for a single read-reference pair. In contrast to traditional systolic-array-based accelerators, ASAP needs to update only the cells (DEs) that can contribute to the LD computation (i.e., corresponding to the colored cells in Fig. A.5). Hence, throughput of the ASAP accelerator can be computed in two ways: we can compute it either by considering the total number of cells in the LD lattice, or by considering only the cells updated by ASAP. The first method which we refer to as *effective-GCUP/s* is directly comparable to traditional techniques as they too consider updating all elements in the LD lattice. In terms of the first method, ASAP achieves an average of 609.6 GCUP/s (10^9 cell updates per second) for 128-bp reads; the second method, it achieves an average of 204.8 GCUP/s. This implies that in the median case, ASAP is approximately $5\times$ better than an equivalent systolic-array-based FPGA implementations (e.g., 122 GCUP/s were physically achieved on an FPGA in [259][8]). Fig. A.11(b) shows the effect of changing tile-length on the latency of the accelerator. It is evident that there are diminishing returns for increasing the tile length, with almost no improvement beyond tile size 16.

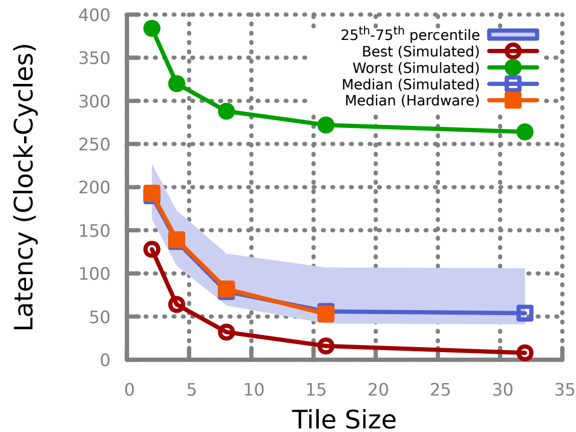
LV Configuration. Table A.3 shows a comparison of the ASAP accelerator running in the LV configuration to the C++-based LV implementation in the SNAP alignment software [164] (which is multi-threaded and uses SIMD instructions). Overall, LV has a lower computational complexity than SW (i.e., $O(nk)$ compared to $O(n^2)$ for SW). This difference in performance is apparent in the baseline CPU implementations shown in Table A.3. Further, we observe that ASAP-accelerated LV is $40 - 60\times$ faster than the baseline for representative input sizes. This corresponds to a $\sim 4\times$ better performance of the LV configuration compared to the SW configuration of the ASAP accelerator. Note that this difference is input-dependent, with the LV variant performing significantly better (as the maximum delay in the ASAP LV lattice is upper-bounded by k) for string pairs that have larger degrees of mismatches.

Input-Dependence. Another point to note about Fig. A.11 is that ASAP represents a method to trade-off worst-case performance and average-case performance. The approximations that we present may be slower than the baseline performance for the worst-case

[8]The comparison to [259] is made based on numbers presented in their chapter, and has not been re-implemented by us.



(a) Input string length (Tile length = 16).



(b) Tile length (Input length = 128).

Figure A.11: Latency of the ASAP-SW accelerator as a function of the input string length. The shaded area in both the graphs show 25th and 75th percentile measurement from simulation.

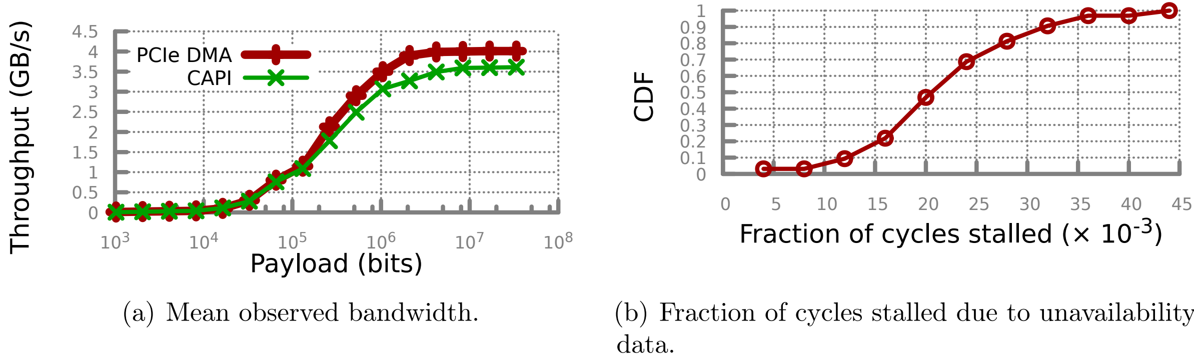
Table A.4: Measured CAPI-based memory access performance. Latency measurements includes round-trip latency to shared memory as seen from the accelerator.

Interface	Payload (B)	Type	Measurement
PCIe	128	Mean read/write latency	0.87 μs
CAPI	128	Mean read/write latency	126 ns
CAPI	128	Mean read/write bandwidth	3.88 GB/s

(i.e., when read mismatches reference completely). However, we see that for representative data sets, the median performance as well as the 75th percentile performance are significantly better than the baseline. For the short read alignment problem, we observe that matches occur more frequently than insertions, deletions or mismatches. The ASAP accelerator can also be applied to other cases where insertions or deletions are more frequent by dealing with those cases in combinational logic.

Performance of the CAPI Interface

The ASAP accelerator benefits from the use of the CAPI interface, because CAPI (i) significantly simplifies, and (ii) significantly streamlines the process of initializing and communicating with the accelerator. We benefit from using a unified virtual memory space across the PCIe bus with hardware-supported address translation, compared with the traditional model, which requires significant hand-holding by an OS. For example, a typical device driver



(a) Mean observed bandwidth.

(b) Fraction of cycles stalled due to unavailability of data.

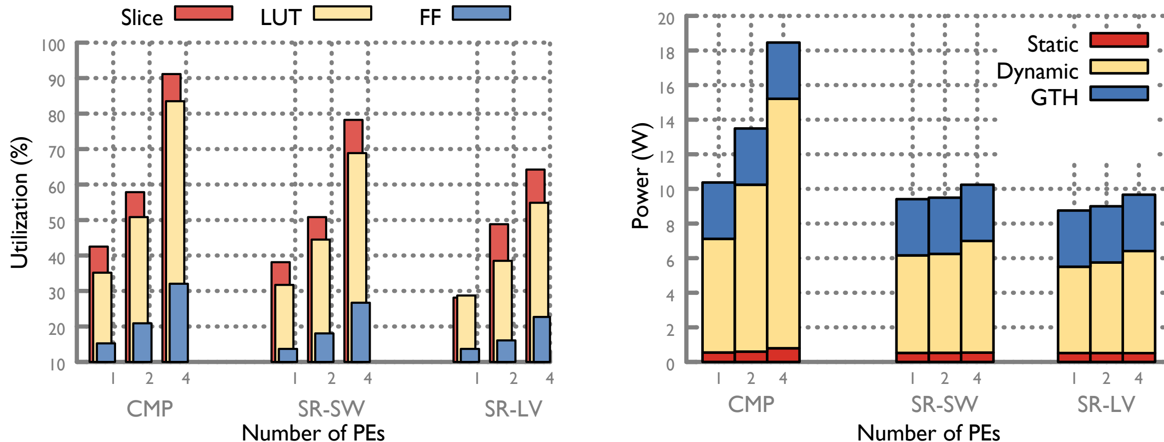
Figure A.12: Mean host-accelerator bandwidth over the CAPI interface and its effect on the performance of the ASAP accelerator.

would execute approximately $20k$ instructions, PCIe bounce-buffering, and page-pinning to perform communication between host and accelerator. We performed measurements on the CAPI interface using a loopback accelerator [257] (i.e., an accelerator reads a cache-line and writes it back to a different location). We observed that (see Table A.4 and Fig. A.12(a)) the CAPI interface can perform random reads and writes with (i) sub-μs latency, and (ii) 4 GB/s bandwidth which are both close to the measured native PCIe latency/bandwidth for the FPGA board used in the evaluation. The one disadvantage that we observe with the CAPI interface is that it allows an AFU to use only 50% of the available peak-theoretical PCIe bandwidth. Our measurements of PCIe goodput (i.e., bandwidth for user data to and from the accelerator) are similar to those from CAPI (see Fig. A.12(a)).[9] Bandwidth is currently not a limitation for the accelerator. Fig. A.12(b) shows the fraction of the runtime of the accelerator spent in stall over the execution of a large number of reads. However, moving to a larger FPGA that supports larger ASAP lattices or multiple smaller ASAP lattices (executing in parallel), or clocking the ASAP accelerator higher than 250 MHz will require larger bandwidth for the host-accelerator interface.

FPGA Resource Utilization

This section describes the overall on-chip resource utilization to implement the CAPI interface and multiple ASAP lattices on the FPGA. Fig. A.13 illustrates this utilization with the increasing number of lattices for two implementation styles for the ASAP delay

[9]We speculate that this limitation occurs because of non-optimal interactions between the OS-modules (e.g., CAPI cache misses trigger TLB/ERAT or page misses) and the PCIe-endpoint ASIC (e.g., dealing with out-of-order packet delivery) on the FPGA board. We leave the optimization of such direct memory access (DMA) issues to future work.



(a) Scaling of FPGA resource utilization (accelerator size) with increase in number of ASAP lattices.

(b) Power dissipation from the ASAP accelerator with increase in number of ASAP lattices per chip.

Figure A.13: Comparison of on-chip resource utilization of the `CMP` and `SR` implementations of the ASAP design. Each ASAP lattice is of size 128×128.

element. First, the comparator based design that was presented in the original RaceLogic chapter [235] (referred to as `CMP` in the figure), and second, the shift-register based design (presented in Appendix A.3) that has been optimized for FPGAs (referred to as `SR-SW` and `SR-LV` in the figure). Fig. A.13(a) demonstrates the significant reduction (nearly 15%) in number of logic elements (i.e., slice resources) required to implement `SR` compared to `CMP`. This further translates to a $\sim 1.9\times$ reduction in power consumed by the `SR` design (shown in Fig. A.13(b)). In the SW configuration, the proposed design is nearly $18.8\times$ more power efficient than the IBM Power8 CPU (~ 10.1 W compared to 190 W). This implies an overall $3,760\times$ ($= 200 \times 18.8$; based on Appendix A.4.1) improvement over the CPU in performance/Watt terms. The LV configuration of the accelerator utilizes 19% less FPGA resources and consumes 10% less power than the SW configuration. The diminished returns from the LV optimizations are a result of the IBM PSL module's occupying $\sim 30\%$ of the FPGA area, thereby dominating the relative decrease in resource utilization and power.

Note that the power consumption for the chip is calculated from the synthesis tool (i.e., Xilinx Vivado) and represents worst-case power consumed by the accelerator. However, the real power consumption is input-dependent and lower than that mentioned above, as clock-gating on off-diagonal delay elements will be enabled differently based on inputs (recall Fig. A.5). We computed this difference in power consumption using the S824L's on-board power meters on the Flexible Service Processor (FSP).[10] The FSP measurements report power

[10]The FSP is an auxiliary processor on the S824L that is an always-on management processor enabling out-of-band management of the server.

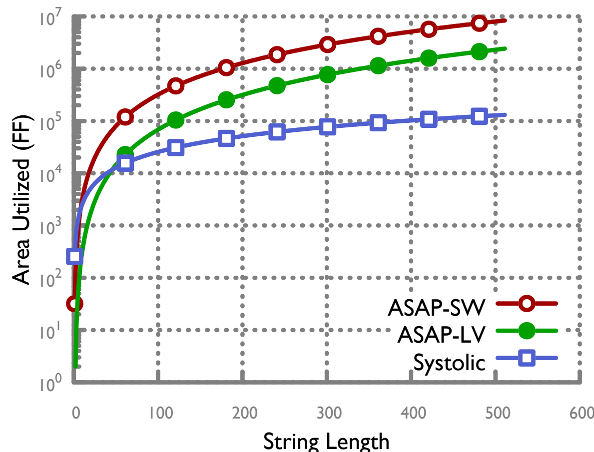


Figure A.14: Scaling of FPGA resource utilization (accelerator size) with increase in input string size for the ASAP lattice in SW configuration.

consumption of the entire computer system averaged over 30 s intervals. To calculate the power consumption of the ASAP accelerator, we measured the difference in power consumed by the system when executing the 4-lattice instance of the ASAP accelerator shown in Fig. A.10, and when idling. We observed an average difference (i.e., the ASAP accelerator's average power consumption) over 100 executions (of the entire benchmark dataset) of 6.9 W $\pm$ 2.8 W (error is expressed as standard deviation) for the SW configuration and 6.8 W $\pm$ 1.6 W for the LV configuration. These measurements support our claim that the actual power consumption of ASAP is lower than that reported by the synthesis tool.

Area-based Scaling. The resource utilization of the ASAP accelerator scales quadratically with the lengths of the sequences being compared. For example, Fig. A.14 shows the number of flip-flops (including those used in shift registers) used by the ASAP accelerator with increasing string length, based on a 16×16 square tile size[11]. In comparison, an FPGA-based systolic array implementation of the LD computation [251] (described in Appendix A.5) scales linearly (i.e., $2N + 1$, where N is the length of the strings being compared). It is apparent that for larger sequences, ASAP quickly exhausts the FPGA resources.

However, ASAP is able to compute LD for short-read sequences (e.g., the 100-150 bp sequences that are typically obtained from an Illumina HiSeq 2500) which are popularly used in resequencing experiments. In addition, we leave approximately 20% of the area of the FPGA free, to allow the CAD tools to place-and-route the circuit without timing violations due to wiring delays.[12] As a result, we are able to fit a maximum 128 bp read

[11]This example does not include flip-flops required for the CAPI interface.

[12]There is no simple analytical method to derive the optimal tile size, sequence size and free area on the FPGA, as the synthesis tools are a black box.

accelerator on our FPGA. Fitting larger blocks leads to timing violations because of delays introduced by the on-chip interconnect. Given the industry trend towards FPGAs with larger programmable area, in the future it should be possible to extend ASAP to read sizes that are potentially thousands of nucleotides long.

Handling Inputs Larger Than Lattice Size. Currently, the ASAP accelerator can be used to compute LD for larger strings by adding a special software-based control algorithm in software to compute LD between sub-strings of the original queries, and combine them to compute the result. The algorithm works by measuring (and storing) the time at which the signal wavefront leaves the extremal DEs of the ASAP lattice, and reintroducing this signal wavefront in the same lattice after updating the nucleotides to be another disjoint substring of the queries. We leave the hardware implementation of this approach for future work.

NW Configuration. Note that the NW and SW configurations of the ASAP accelerator are identical in terms of performance and FPGA-resource utilization. Appendix A.3.1 describes how the NW and SW configurations differ in how delay between the input and output are measured.

A.4.2 Integration Into End-to-End Alignment Software

In this section, we compare ASAP-accelerated versions of end-to-end alignment software tools with their baseline CPU versions. This comparison includes time taken for LD computation as well as other auxiliary functions like, disk IO, and marshaling and un-marshaling of data (from disk and from accelerator). As a result, improvements in LD computation provide diminishing returns (i.e., asymptotic behavior similar to Amdahl's law); we show that the current ASAP represents a speedup that is very close to the asymptotic limits for this computation. We use two alignment tools, SNAP [164] (which uses ASAP in the LV configuration) and BWA-MEM [232] (which uses ASAP in the SW configuration).[13] These results are described below.

Host-Accelerator Communication. The baseline SNAP & BWA aligners exploit parallelism in the alignment problem by dividing the work of aligning a set of reads among all of the 192 threads available on the system. We use the same communication algorithm to dispatch LD computations to the accelerator in both cases. Since our current implementation of ASAP allows for only one calling context on the host-side, accelerator executions are dispatched by maintaining a pool of memory shared among all threads to communicate with the accelerator. The procedure for each thread communicating with the accelerator is as follows: (i) picks a read from the set it was assigned; (ii) queries the reference index for

[13]We used version 1.0 of the SNAP tool and version 0.7.17 for BWA.

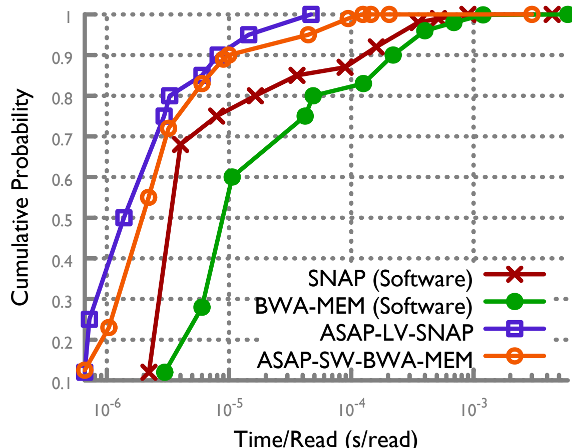


Figure A.15: Performance comparison of the ASAP-accelerated SNAP and BWA-MEM alignment tools (called `ASAP-LV-SNAP` and `ASAP-SW-BWA-MEM`, respectively) with their baseline CPU versions (called `SNAP (Software)` and `BWA-MEM (Software)`, respectively).

candidate locations for the read; (iii) contends for a lock, then writes nucleotides for the read and the candidate locations into shared memory; (iv) at this point, the accelerator reads from the shared memory and writes out the results to another shared segment of memory; and (v) polls for results from the accelerator using a test and test-and-set based locking protocol [260], then consumes the output. This algorithm exemplifies CAPI's benefit, as we can make use of cache coherence between the CPUs and FPGA to easily implement mutual exclusion.

The SNAP & BWA-MEM Aligners. Fig. A.15 shows the distribution of time taken per read by the baseline and the ASAP accelerator for all LD computations. We see that there is a large spread for total time spent in computing LD because some reads map to more regions of the reference than others. This variation is an artifact of both the nature of the human genome and the read simulator's practice of picking reads at random from the genome. We observe that the SNAP aligner is accelerated by 2× (i.e., $^{1.85\text{ hr}}/_{0.92\text{ hr}}$) and that the BWA-MEM aligner is accelerated by 1.86× (i.e., $^{2.64\text{ hr}}/_{1.42\text{ hr}}$), respectively. These results are representative of the mean time spent in processing a single read. Fig. A.15 shows the long-tailed behavior of some of the input pairs in the datasets that are significantly mismatched (for the accelerated implementations). The long-tailed behavior of the CPU baselines stem from non-determinism in the IO and thread scheduling subsystems of the host. The LV upper-bound for maximum time spent in computing LD ensures that the ASAP-accelerated SNAP version (called `ASAP-LV-SNAP` in Fig. A.15) has a significantly shorter tail than the SW configuration in BWA (called `ASAP-SW-BWA-MEM` in Fig. A.15).

The performance measurements presented above are close to the Amdahl's law limit of

the SNAP algorithm based on our measurements presented in Table A.2. In the case of the BWA-MEM aligner, we observe that we achieve a lower absolute improvement, that is expected, as the asymptotic limit for improvement is lower. The BWA-MEM algorithm performs larger amounts of non-LD computation compared to SNAP, i.e., Burrows-Wheeler transform based index lookup while SNAP computes hashes for a hash table lookup. Both baselines are measured with huge-page support turned on in the host's Linux-kernel to negate effects of ERAT-misses (TLB-misses in Intel parlance).

Gap Penalty Models. A subtlety to be noted in the comparison presented above is that BWA-MEM's default behavior uses *affine gap-penalties* in addition to the SW local-alignment algorithm (instead of the *constant gap-penalties* used by ASAP). Hence we have to use the tool's command line arguments to set the gap-penalty parameters such that they replicate a constant gap-penalty model (i.e., set the requisite parameters to 0). We discuss handling of affine gap-penalties in ASAP as part of our future work in Appendix A.6.

A.5 RELATED WORK

The sequence alignment problem has been addressed by an extensive body of work that looks at algorithms and their high-performant implementations on CPUs and on accelerators like GPUs and FPGAs. This section focuses on comparing ASAP to other implementations of the LD computations. Refer to Appendix A.2 for a discussion of algorithms.

On CPUs and GPUs. The LD computation and sequence-alignment problem has been studied on SIMD and MIMD processors that exploit parallelism in the problem at two levels. Inter-task parallelism [261] (using multiple cores to independently compute alignments of different short reads), and intra-task parallelism [262, 263] (using SIMD instructions and efficient use of the memory hierarchy to effectively compute (A.1)). Most of the popular SW or NW implementations exploit the use of both of these techniques. These techniques have also been applied to GPUs [264, 265, 266]. One such example is NVIDIA's NVBIO [267] library and the accompanying set of tools nvBWT, nvFM-server. These look at accelerating the construction and look-up of data structures that index the reference genome. The major disadvantages of this approach is the large power consumption of these processors, and their restrictive lock-step parallelism based programming models.

On FPGAs and ASICs. Custom hardware acceleration of the problem on FPGAs and ASICs has also been widely studied. Most of the popular hardware architectures are based on systolic arrays [251, 252, 253, 254, 255]. These architectures like the SIMD and MIMD approaches, are limited by the amount of parallelism they can exploit. It has been shown in [268], that exploiting deeper pipelines with much larger inter-task parallelism can potentially

enable more efficient use of FPGAs. We may be able to use this optimization to futher increase the throughput of the accelerator, particularly on larger FPGAs that can sustain larger off-chip bandwidth. Kaplan et al. [269] present an ASIC design for a Processing-in-Memory accelerator for the SW algorithm that leverages resistive content-addressable memory to compute matches/mismatches of nucleotides. ASAP represents a significant improvement over [269] in throughput/Watt terms, i.e., ASAP achieves 61 GCUP/s/W (= $^{609.6}/_{10.1}$) compared to their 53 GCUP/s/W. Turakhia et al. [14] present an accelerator to perform long-read assembly, one step of which includes a SW-based alignment (through a seed-and-extend approach). Alser et al. [270] present an FPGA based accelerator to efficiently filter candidate locations to calculate LD. This accelerator is targeted at Line 6 of Alg. A.1, as opposed to ASAP which targets Line 7, hence the accelerator can be used in addition to ASAP to accelerate the end-to-end alignment process. More recent work [271] has also shown the benefit of distributing the compute intensive LD computation across multiple accelerators (including CPUs, GPUs, FPGAs, Xeon Phis). We observe that ASAP significantly outperforms such multi-accelerator systems both in terms of performance and performance per-Watt. The Host + 2× FPGA design presented in [271] only achieves a 441.6 GCUP/s performance at 1.51 GCUP/s/W. In comparison ASAP achieves 609.6 effective GCUP/s at 61 GCUP/s/W on a single FPGA.[14] Other work, e.g., [178, 179, 272], has demonstrated the use of systolic-array-based designs to accelerate computations on Pair-HMM models, where gap-penalties are replaced by probability distributions. That may be a future direction for the extension of the ASAP design.

ASAP's design philosophy is most closely related to Madhavan et. al.'s RaceLogic [235] ASIC design, which also encodes LD computations as circuit delay. However, ASAP builds on this basic model to further optimize the design by using (i) approximation algorithms for the LD computation which maintains the total ordering of LDs, and (ii) accelerating the most common computation (in this case the processing of "matches") in combinational circuitry thereby spending minimal runtime in its computation. This is demonstrated by the fact that ASAP is $\sim 50\times$ faster than a RaceLogic implementation. Further, the nature of the alignment problem and the rapidly evolving sequencing technology (i.e., read lengths), implies that fixed function ASICs are not favorable because of the large monetary investment required and the inability of the accelerator to adapt to new input sizes. ASAP circumvents these problems by using reconfigurable FPGAs. Of course, an ASIC will almost always outperform an FPGA in energy efficiency because of its customized layout. Hence going forward, a design with a fixed function (i.e., ASIC-based) IO interface (i.e., CAPI) with a

[14] The comparison is made across an equivalent generation of Altera and Xilinx FPGAs, using *effective-GCUP/s* (described in Appendix A.4.1).

configurable substrate for ASAP accelerators might present an ideal trade-off.

Comparison to Systolic Arrays. Relative to the related work described above, ASAP has some decided advantages:

1. The systolic array based approaches require each element of the array to compute on as many bits as the maximum LD computed. Our approach requires only as many bits per delay element as the maximum delay between inputs at that point in the lattice.

2. The earlier accelerators have to explore the entirety of the lattice before computing the LD. We show that the ASAP accelerator explores only the portions of the lattice that is reachable before the final result is produced. This represents a significant savings in run time and energy expended for computation.

3. The ASAP accelerator can explore multiple elements in the lattice in under one clock cycle by setting $\delta(\text{Match}) = 0$. Systolic array based architectures cannot perform this optimization, as this creates large combinational chains which make timing closure difficult to obtain.

On Neuromorphic Computers. Neuromorphic computing is modeled on biological neurons that communicate and compute using temporal-encoding of information as voltage pulses, or spikes. This is similar in principle to the delay based computation outlined in this chapter, however it is still an open research question [273, 274].

On sequencing technologies. Recall that the alignment computation (shown in Appendix A.2) is composed of the LD computation as well heuristics to identify candidate reference regions. The use of novel sequencing technologies (e.g., PacBio, Nanopore which are based on single-molecule sequencing), introduces new sequencing error regimes which will change the heuristic components of the alignment computation, but not the LD. As ASAP targets the LD computation, we believe it can be applied to data generated from these long-read sequencing machines. Turakhia et al. [14] present one such accelerator for long reads. Their accelerator targets the acceleration of the entirety of Alg. A.1 and uses LD computation as a submodule. ASAP can replace that module and provide significant performance and energy benefits as shown in this chapter.

A.6 SUMMARY

This chapter proposed ASAP, an accelerator for rapid computation of LD, in the context of the short-read alignment problem. ASAP builds upon the idea that the LD between strings can be approximated for the short-read alignment problem by encoding gap penalties

in propagation delays of circuit elements. We show that by effectively setting these delays, it is possible to accelerate performance significantly, and at the same time ensure that the accuracy of alignment is maintained. ASAP significantly outperforms (both in performance and performance-per-Watt terms) purely CPU/GPU-based as well as sytolic array-based accelerator implementations of LD computation in the all the SW, LV and NW configurations.

The ASAP accelerator, and the approach (based on heuristic approximations) presented in this chapter, can also be adapted to a variety of other problems in which a total ordering of LDs is computed. For example, in signal processing, where different instances of a signal have to be aligned to compute similarity [228]; in text retrieval, where misspelled words have to be accounted for in a dictionary [237]; and in virus- and intrusion-detection, where signatures have to be aligned to a baseline [238].

Future Work. Our future work will primarily look to extend ASAP to handle more complex gap-penalty models. This chapter describes the use of constant gap penalties (i.e., a fixed score is assigned to every gap), which are commonly used in DNA alignment (e.g., in NCBI-BLASTN, or WU-BLASTN [241]). We can extend ASAP to handle linear, affine, and convex gap penalties by letting each DE track the propagation of the signal wavefront in the portion of the lattice before it. We can do so by dynamically resizing the length of the shift registers on off-diagonal DEs depending on their positions (i.e., i, j coordinates). Further, re-using the lattice for input strings larger than the lattice dimensions would involve dynamic reconfiguration of the FPGA to allow for different *taps* in the shift registers. Further, ASAP can also be extended for use in the alignment of proteins by using substitution matrices, like BLOSUM [230], which assign unique scores to each pair of residues.

APPENDIX B: ACCELERATED PAIRHMM FORWARD ALGORITHM

B.1 INTRODUCTION

An important computation (and a critical bottleneck) in medical sequence analysis pertaining to the analysis of variants (mutations) in sequenced genomic data is the *forward algorithm* [275] (FA) on *Pair-Hidden Markov Models* [276] (PHMMs) (see Appendix B.2). The FA algorithm, which is generally viewed as one of the best ways to compute the statistical similarity between two sequences, is widely used in genomic data analysis workflows for gene prediction, functional similarity analysis between protein sequences, multiple sequence alignment, phylogeny, and germline- and somatic-variant calling [277]. This chapter addresses the problem of accelerating the GATK HaplotypeCaller [160], a popular and trusted variant calling and genotyping tool[1] that incorporates a PHMM model (described in Appendix B.2.2) and is widely used in clinical settings (e.g, in the diagnosis of critical diseases like cancer). The FA constitutes the most computationally complex phase of this application, accounting for nearly 70-80% of the runtime while processing human clinical datasets.

The FA algorithm is inherently parallelizable at two levels: (i) at the level of the algorithm, i.e., *intra-task* parallelism through the anti-diagonal recurrence pattern in a single FA execution; and (ii) at the level of the data, i.e., *inter-task* parallelism by computing independent instances of FA in parallel. Prior efforts to address this problem [178, 272, 279, 280, 281, 282, 283, 284] used some form of a systolic array (SA) architecture. These architectures optimize only for intra-task parallelism, and thus underutilize on-chip resources and waste energy when input sizes are not multiples of the number of processing elements.[2] As a consequence, such designs cannot efficiently handle realistic data where input sizes (i.e., the lengths of the query DNA fragments) can vary significantly (see Appendix B.3.1). A common thread of research in this area has been to utilize control algorithms and data placement strategies to overcome these shortcomings, thereby leading to increased algorithmic complexity (for CPUs and GPUs) and larger on-chip areas for FPGAs [178, 272, 279, 284].

This chapter proposes (in Appendix B.3) the design of an accelerator for the FA algorithm that overcomes the aforementioned shortcomings. Unlike previous approaches, our approach uses it's entire resource budget optimizing for inter-task parallelism (thereby exploiting the embarrassingly parallel nature of the problem). Intra-task parallelism is addressed by deep pipelining to maximize temporal sharing (reuse) of computational resources. We demonstrate

[1]It is a part of the Broad Institute Best Practices Workflow [278] for variant-calling.

[2]For example, a 32 processing element systolic array performs 1568 (56×28) unnecessary computations when processing two sequences of lengths 100, and 200.

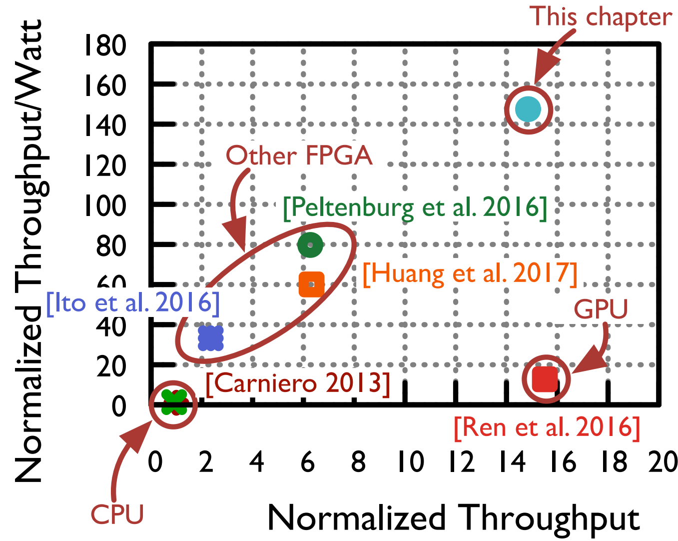


Figure B.1: Comparing this chapter with related work based on throughput (MCUP/s) and throughput-per-watt (MCUP/s/Watt). Values are normalized to our CPU baseline running on a Power8 CPU.

that this design maximizes overall throughput by optimally using parallelism, and minimizes control related hazards and stalls. Our accelerator produces a speedup of 14.85× over an 8-core Power8 processor executing the baseline software implementation, and is 147.49× better in speedup-per-unit-energy terms. Fig. B.1 demonstrates this speedup compared to the related work available on this problem (explained further in Appendix B.6). In our initial design implementation on an IBM Power8 system and using an FPGA attached over the CAPI [96] interface, we observe that the key performance-limiting factors are (i) latency overhead in accelerator invocation through the software stack, and (ii) redundant computation done across multiple unrelated instances of the FA. To alleviate these performance bottlenecks, we propose (in Appendix B.4) two additional algorithmic optimizations that span the hardware-software interface. These techniques prune the inputs of the FA algorithm and memoize its output (i) to reduce the number of invocations of the FA kernel, and (ii) to reduce the size of the sequences being compared. The result is an effective reduction in the amount of data that has to be transferred from the host to the accelerator to complete a batch of FA computations, which contributes a further 2.8× improvement in performance.
The main contributions of this chapter are as follows:

1. Identifies the performance-limiting issue with today's CPU/GPU and systolic-array-based FA accelerators.

2. Presents the architecture of an ensemble of processing elements that maximizes inter-

task parallelism and uses aggressive pipelining to address intra-task parallelism, thereby overcoming the inefficiencies of the SA architectures.

3. Evaluates the proposed design on an FPGA, and couples it with a coherent interface to the CPUs memory, allowing work-sharing between the CPU and accelerator.

4. Presents two pruning strategies to memoize results to reduce the input data-set size as well as the number of invocations of the FA accelerators.

5. Demonstrates that integration of the accelerator and pruning techniques into the GATK HaplotypeCaller can accelerate it by 3.287× (close to the Amdahl's law limit) over the baseline CPU implementation.

6. Evaluates the potential impact of several emerging high-bandwidth memory technologies to alleviate the host-accelerator bandwidth limitations in PCIe/CAPI.

B.2 BACKGROUND

B.2.1 Pair-HMM Model

PHMM models are instances of Bayesian multinets that allow for a probabilistic interpretation of the alignment problem [276]. An alignment models the relationship (homology) between two sequences via a series of mutations (M), insertions (I), and deletions (D) of nucleotides. The FA algorithm of the PHMM allows the computation of statistical similarity by considering all alignments between two sequences and computing the overall alignment probability by summing over them. These divergent sets of alignments are caused by evolutionary mutations or sequencing errors. Specifically, given two sequences S_1 and S_2 of lengths n and m respectively, the FA algorithm defines the computation of three probabilities, $f_M(i,j)$, $f_I(i,j)$, and $f_D(i,j)$ (see Fig. B.2). The value $f_k(i,j)$ corresponds to the combined probability of all alignments for substrings $S_1[0:i]$ and $S_2[0:j]$ that end in state $k \in \{M, I, D\}$. The FA algorithm can be recursively defined as follows:

$$\left.\begin{array}{rcl} f_M(i,j) & = & P(a_{mm} f_M(i-1,j-1)+ \\ & & a_{im} f_I(i-1,j-1)+ \\ & & a_{dm} f_D(i-1,j-1)) \\ f_I(i,j) & = & a_{mi} f_M(i-1,j) + a_{ii} f_I(i-1,j) \\ f_D(i,j) & = & a_{md} f_M(i,j-1) + a_{dd} f_D(i,j-1) \end{array}\right\} \quad \text{(B.1)}$$

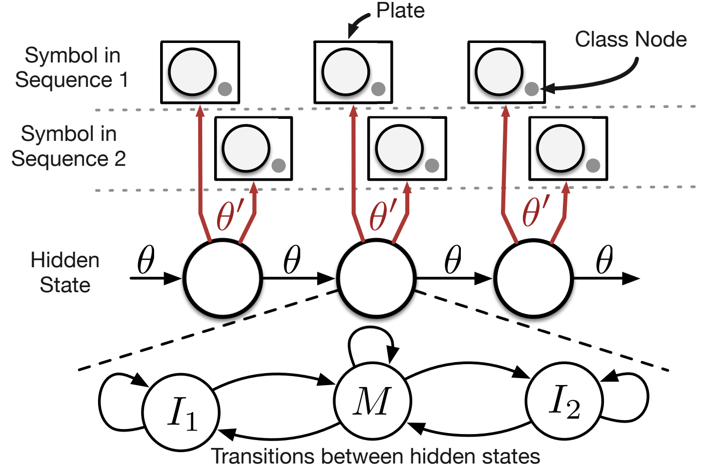


Figure B.2: The PHMM model describes the pairwise alignment of two sequences. The plates and class-nodes in the figure indicate the presence or absence of a symbol.

The parameters a_{mm}, a_{im}, a_{dm}, a_{mi}, a_{ii}, a_{md}, a_{dd} and P are derived from the values of base-quality scores, map-quality scores, and the values $S_1[i]$ and $S_2[j]$. They represent a statistical model that jointly describes (i) dependence between adjacent nucleotides, (ii) dependence between hidden and observed sequences that describes a multi-nucleotide mutation model, a point mutation model, and (iii) sequencing and alignment errors using an affine gap score model [276]. Patcher et al. [275] describes the rationale behind using these quality metrics in the PHMM model to set the a_* parameters. Finally, the overall similarity metric between the sequences is the sum of the probabilities across the states M, I, and D when comparing the entire strings S_1 and S_2, i.e., $f_M(n,m) + f_I(n,m) + f_D(n,m)$. Hence, each FA needs to compute the recursion stated in (B.1) $n \times m$ times. That corresponds to a total computational-time complexity of $O(nm)$ and a total space complexity of $O(\max(n,m))$.

B.2.2 Germline Variant Calling

In this work, we accelerate the germline variant-calling and genotyping tool called the GATK HaplotypeCaller (or GATK), which statistically infers differences between sequenced genomes and *reference genomes*, where reference genomes represent "average" genomes for a population of the candidate species. The base algorithm of variant calling and genotyping is straightforward: input sequence fragments are aligned to a reference sequence, and at every position the number of mismatches are counted. However, this process is complicated by the fact that data from sequencing machines are inherently noisy (from sequencing errors), alignments are often incorrect (from mapping errors) and polyploidy of an organism (i.e.,

Algorithm B.1 Algorithmic skeleton of the GATK HaplotypeCaller. The functions `EnumerateHaplotypes`, `Align`, and `Genotype` are described in [140].

```
 1: alignment ← Aligned set of reads in an active region
 2: reference ← Reference genome for an organism in an active region
 3: n ← Number of ploids in the organism
 4: haplotypes ← EnumerateHaplotypes(alignment.reads)
 5: for h ∈ haplotypes do
 6:     for r ∈ reads do
 7:         score[h, r] ← PairHMM(h, r)
 8:     end for
 9: end for
10: best_haplotypes ← Find n-best haplotypes
11: new_align ← ∅
12: for r ∈ alignment.reads do
13:     new_align ← argmax_{h∈bh} Align(r, h)
14: end for
15: variants ← ∅
16: for haplotype ∈ best_haplotypes do
17:     for loc ∈ haplotype do                ▷ Every position in the haplotype
18:         variants ← variants ∪ Genotype(new_align.atLocus(loc))
19:     end for
20: end for
21: return variants
```

there are many copies of a genome per individual). The PHMM model is applied in this context to model the aforementioned errors (e.g., sequencing errors, alignment errors, or mutations) statistically and to assign sequenced DNA fragments to their corresponding ploids. GATK computes the variants in a sequenced genome by filtering the genome into *active regions* that might contain possible mutations. Alg. B.1 is then applied in parallel to all active regions in order to reconstruct haplotypes (using DeBruijn graphs [285]) for the ploids for each active region. The FA algorithm is then used to compute the probability that a sampled sequence fragment originated from a certain haplotype. These probabilities are used to weight each haplotype to find a candidate set that might best represent a ploid. Finally, the reads are realigned to their best haplotype. A count of the number of mismatches to the haplotype (instead of the reference) is then used to determine the presence of a variant and its genotype [286]. GATK uses single-precision floating-point numbers to compute (B.1). In the case of an underflow, double precision floating-point numbers are used to recompute the result. The FA algorithm constitutes the bulk (nearly 70%; see Table B.1) of the runtime of GATK on the popular NA12878 sample from the GIAB consortium [287], and as such is a

Table B.1: Distribution of runtime between the phases of Alg. B.1 in the GATK Haplotype-Caller for the NA12878 sample executing on the baseline hardware configuration described in Appendix B.5.

Stage	Absolute Time (hr)	Percentage Time	Line Number
Assembly	2.87	13.8	1–4
PHMM FA	14.78	71.1	5–9
Realign + Misc	3.13	15.1	10–21

good candidate for acceleration. Banerjee et al. [140] shows that GATK and transitively the FA computation also forms a significant portion of the runtime of sequencing data-analytics workflows on modern compute infrastructures (e.g., clouds and supercomputers).

B.3 ACCELERATOR

B.3.1 Design Philosophy

Based on our analysis of the algorithms and input datasets, we offer a set of insights that drove our design philosophy:

Insight 1. *Diversity in input size.* Haplotypes generated by the HaplotypeCaller show great diversity in size (see Fig. B.3). Traditional SA-based architectures for accelerating the FA algorithm are often not able to handle this diversity effectively because of the cycles wasted when the size of the recursion lattice is not divisible by the number of PEs (processing elements) in the SA, resulting in holes in the processor's pipeline. Traditional SA-based architectures deal with these issues by using complicated control mechanisms [178, 272, 284] that improve pipeline utilization. CPU- and GPU-based architectures that exploit SIMD instructions [279, 280] also experience this issue, albeit to a lesser degree.

Insight 2. *Exploiting inter-task parallelism.* Systolic array architectures exploit anti-diagonal parallelism to minimize latency for a single task. However, given that the FA algorithm itself demonstrates several orders of magnitude greater parallelism across tasks than within tasks, we believe that it is more prudent to exploit inter-task parallelism. Such an approach also addresses the problem of input diversity, as we can exploit the data-parallel nature of the problem without data dependencies between PEs. As we show in our results (Appendix B.5), the increased data set size needed for inter-task parallelism does not limit our implementation.

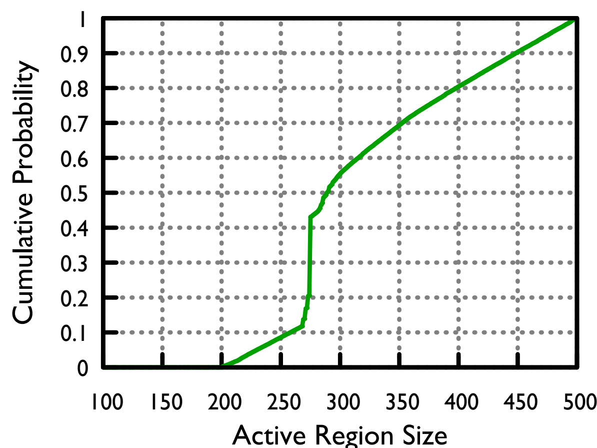


Figure B.3: Distribution of active region (haplotype) sizes in a sample of the NA12878 dataset.

Insight 3. *Why not GPUs, and why FPGAs?* Insights 1 & 2 discourage the use of GPUs because of the prevalence of lock-step parallelism in their programming models. Previous efforts of using GPUs in the context of this problem have successfully extracted intra-task parallelism at the warp-level. However, input diversity leads to control divergence when inter-task parallelism is exploited. In contrast, FPGAs provide the flexibility to build a processing pipeline that is tailored to the computation at hand and its input characteristics. We explore the difference in performance between GPUs and our method in Appendix B.6. We demonstrate that the our design is unmatched in the performance-energy trade-off space, but concede that the GPU represents a different (and in some cases a preferable) point on the performance–developer-productivity spectrum.

Fig. B.4 illustrates the overall design and implementation of the accelerator. The accelerator is organized as an array of independent processing elements (PEs; see Appendix B.3.2), which asynchronously pull inputs from the CPU's memory space over a cache-coherent CAPI bus (see Appendix B.3.4). Reads and writes to the host processor's memory space are made in a streaming fashion to overlap computation of haplotype-read pairs on the host with the accelerator.

B.3.2 Processing Element (PE)

Fig. B.4 also shows the design of a single PE. The nucleotide inputs to the PE are encoded as 4-bit unsigned numbers. These correspond to nucleotides `A`, `C`, `G`, `T`, and the ambiguous

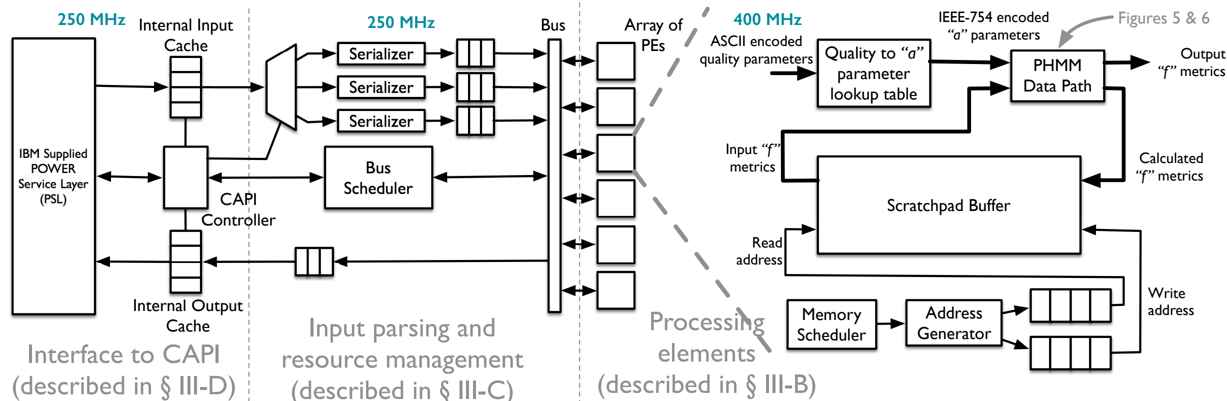


Figure B.4: Architecture of the data-path and the control-path of the proposed accelerator comprising the 1) host-accelerator interface over CAPI, and 2) input parsing and load balancing.

nucleotides `N`, `-`, `R`, `Y`, `K`, `M`, `S` and `W`. The remaining symbols can be used to accommodate FA on protein sequences. The quality-score inputs are in their standard ASCII encoding [276]. These are used to compute the FA algorithm parameters. Each PE has: (i) a table-lookup-based function to compute floating-point parameter (probability) values from input quality scores; (ii) a data-path consisting of single-precision floating-point adder and multipliers; (iii) a scratchpad buffer to store intermediate values of the f_k matrices (where $k \in \{M, I, D\}$); and (iv) scheduling circuitry that generates a valid sequence of read-write addresses for the scratchpad buffer. Using table-lookup allows us to use 8-bit encoding of quality scores as opposed to their 32-bit floating-point encoding when transferring inputs to the accelerator, thereby reducing I/O bandwidth requirements. Each PE is fed from a BRAM bank that stores the values of each of its inputs. We now briefly describe the design of the data-path and the scheduler.

Data-Path

The data-path of the PE has to implement the recurrence relations in (B.1). Each step of this recurrence has 8 multiplication operations and 4 addition operations. Our design finds the optimal trade-off point between latency/throughput and resource utilization on the chip to implement this computation. We use a Xilinx intellectual property (IP) that provides a 5-cycle latency and 1-op/cycle throughput for the float-multiplier and a 10-cycle latency and 1-op/cycle throughput for the float-adder. Both the adders and multipliers run at 400 MHz. Our design utilizes two multipliers and one adder for a 32-cycle latency and 0.25-recursion-step/cycle throughput schedule. Fig. B.5 demonstrates the utilization of the

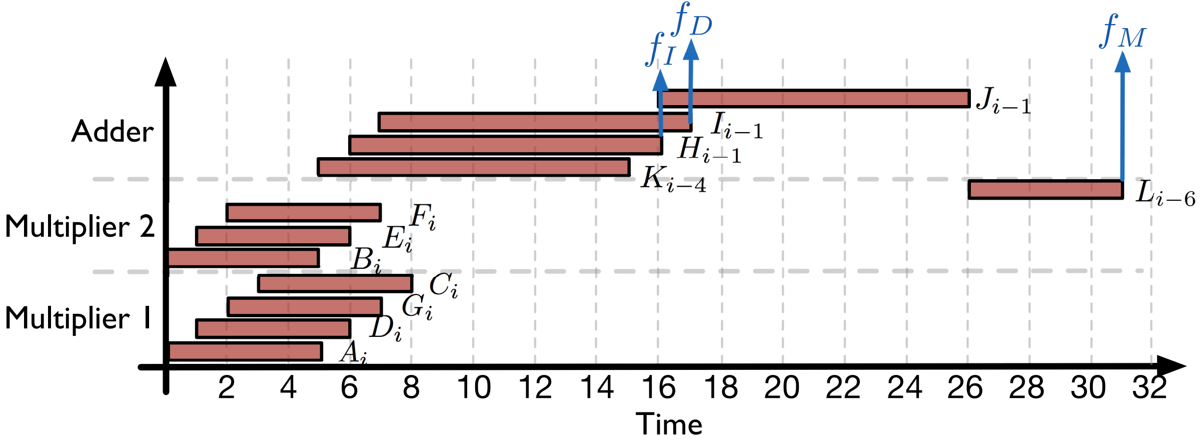


Figure B.5: Schedule of the adders and multipliers on the data-path illustrated in Fig. B.6.

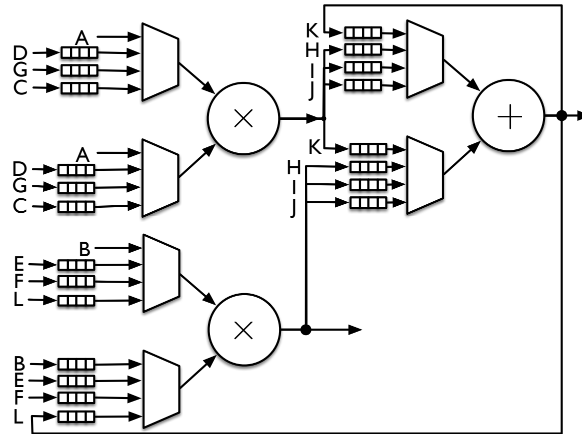


Figure B.6: Circuit diagram for the PE data-path shown in Fig. B.4.

two multipliers and the adder in the 32-cycle period to compute one step of the recurrence. The circuit representation corresponding to this schedule is shown in Fig. B.6. The inputs of the circuit elements, labeled A through L are shown in both Figs. B.5 and B.6, where the subscripts represent the step of the recurrence currently being computed. For example, A_i and L_{i-6} (in Fig. B.5) represents that the ith recurrence step for A and the $i-6^{th}$ recurrence step for L is computed in the same 32-cycle window. We achieve synchronization in this scheme by using shift registers attached to the muxed inputs of the adder and multipliers, as shown in Fig. B.6. The lengths of these registers can be derived from Fig. B.5. The outputs of this data-path is fed back into the inputs of the following stages via the scratchpad buffer (implemented as a BRAM block) shown in Fig. B.4.

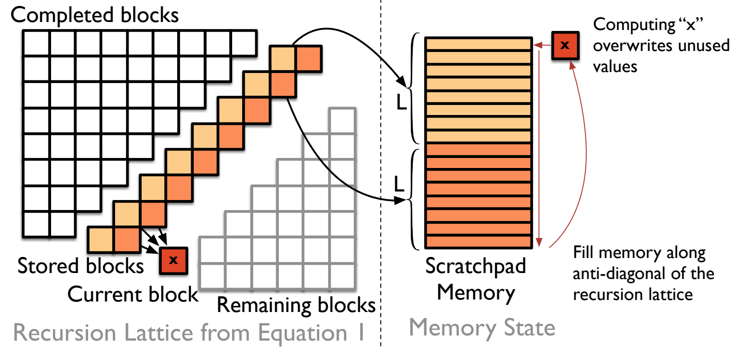


Figure B.7: Exploiting anti-diagonal *wavefront* pattern of (B.1) to optimize memory size to $L-2$, where $L = \min(n, m)$

Memory Scheduler

To minimize the size of the scratchpad buffer (in Fig. B.4) we compute steps of the recurrence (i.e., (B.1)) in an anti-diagonal fashion, as shown in Fig. B.7. In the figure, we illustrate the antidiagonal pattern by dividing the entire recurrence lattice into four parts as follows:

1. *Completed Blocks.* Blocks for which the value of the recurrence has been computed and is no longer required.

2. *Stored Blocks.* Blocks for which the value of the recurrence has been computed and is required in subsequent steps of the computation.

3. *Current Block.* Block whose inputs have been produced and can start computation.

4. *Remaining Blocks.* Blocks whose inputs have not yet been generated.

Here each block refers to the three tuple $(f_M(i,j), f_I(i,j), f_D(i,j))$. We observe that that limiting the maximum size of the buffer to $2L$, where L is the size of the largest anti-diagonal, is sufficient to compute the FA algorithm. Fig. B.7 demonstrates that once this buffer is full, simply starting over at the beginning only overwrites data that is no longer required for the computation. Our current implementation supports matrices up to $L = 512$. This limit is sufficient to accommodate the largest haplotype (500 bases) generated by GATK, as well as reads from the popular Illumina HiSeq sequencing platforms (150–250 bases long). The scheduler generates a pattern of read and write addresses into the memory for the aforementioned data-path. The scheduler deals with the upper and lower triangles in the recurrence lattice (e.g., $i + j \leq 8$ where $8 \times {}^{1}/_{0.25}$ is the latency of the pipeline), when a PE's

pipeline cannot be kept full because of the dependencies between the inputs and outputs of the recurrence.

B.3.3 On-Chip Bus-Scheduing and Load-Balancing

On the accelerator side, we multiplex inputs from the CPU (1024-bit cache lines) among the array of processors by parsing the input stream through a *"Serializer"* and storing the FA executions in FIFOs to be fed to idle processors over a bus (see Fig. B.4). We observed that using cache line aligned inputs significantly reduces the complexity of parsing the input stream on the accelerator side, though this alignment incurs memory overhead on the CPU side. Our experimental system (described in Appendix B.5) had 1TB of RAM attached to the CPUs and hence this trade-off is acceptable. This design decision can be revisited for other machine configurations. Note that this choice does not affect performance of the accelerator, merely the complexity of the "Serializer." We use a straightforward arbitration mechanism for the bus described in Fig. B.4. In the case of ties, the bus scheduler arbitrates inputs in a round-robin fashion. Outputs are handled in a similar fashion.

B.3.4 Host-Accelerator Interface

Communication between host and accelerator is implemented using the CAPI interface [96] with an IBM Power8 CPU. The CAPI interface allows an accelerator (a PCIe-attached FPGA) coherent access to the virtual address space of a process running on the host CPU. Our accelerators pull data directly from three circular buffers in the processor's memory space. These correspond to the reads, haplotypes, and quality scores. The CPU generates new tasks (new instances of the FA algorithm to be computed) by executing Alg. B.1 and enqueues inputs to the accelerator into the respective circular buffers. The accelerator and other threads on the CPU then consume these inputs from the buffers. CAPI is beneficial in this instance, as we can make use of the cache coherency between the CPUs and FPGA to easily implement mutual exclusion.

B.4 ALGORITHMIC OPTIMIZATIONS

In building the accelerator as described in Appendix B.3, we noticed that *batch size* (number of tasks streamed to the accelerator at a time) has a particularly dominant effect in the performance. Fig. B.8 illustrates this loss in performance as a function of batch size. We attribute this behavior to the software overheads (e.g., system calls, IRQ handlers)

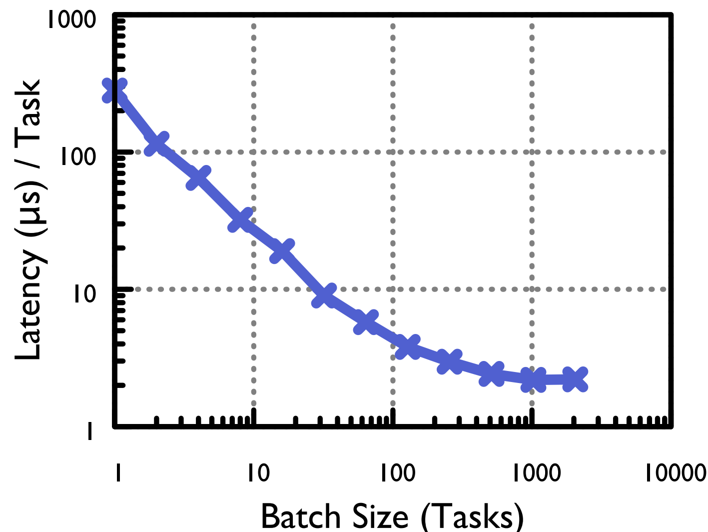


Figure B.8: Quantifying the accelerator call and data transfer overhead.

that initiate the accelerator. Our observations can be explained by the fact that simply batching tasks amortizes this cost over several individual accelerator invocations. To further improve (i) batching efficiency (latency amortization), (ii) host-accelerator data transfer bottlenecks (PCIe limitations), and (iii) reuse of precomputed results (across multiple tasks), we developed two algorithmic methods for pruning inputs and memoizing outputs of the FA algorithm when it is used in conjunction with the HaplotypeCaller. Both optimizations reduce the throughput of the FPGA accelerator, but improve end-to-end performance by reducing the total number of computations.

B.4.1 Common Prefixes in Reads

We observe that when Alg. B.1 is invoked on active regions of high-coverage, high-quality datasets, a large number of reads share a common prefix. This property is an artifact of the alignment process, of having similar reads start close to each other, and of repeats in the reference genome. According to the formulation of the FA algorithm, for the same haplotype, reads with common prefixes produce exactly same results under (B.1). We exploit this observation by reusing values of the recurrence relation for common prefixes. This reuse is done by constructing a compressed trie [288] of the reads in an active region, and computing the longest common prefix (LCP) in the trie. The accelerated FA algorithm is then computed on the LCP and the answer is memoized. All substrings of the LCP in the trie can then use the lattice values computed in the LCP as a starting point for further computation. In fact, this optimization can be reapplied once the LCP is removed from the trie. Though

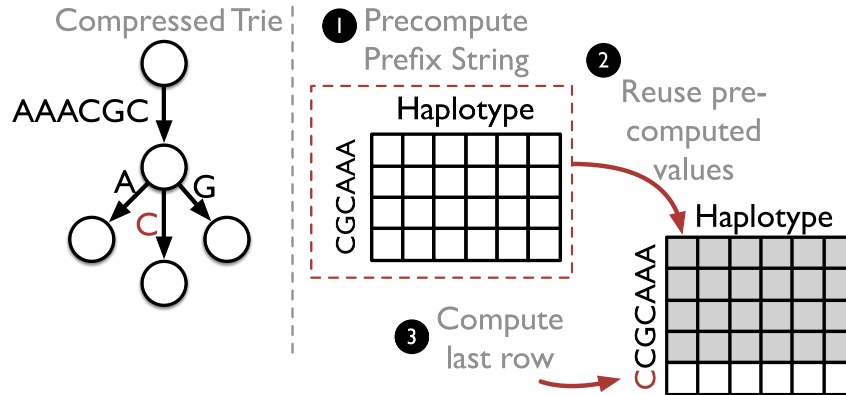


Figure B.9: Exploiting common subsequences between reads in an active region to memoize the computation of the FA

straightforward, this heuristic significantly reduces the amount of computation required in the HaplotypeCaller (by $20-30\%$ for human clinical datasets). For example, consider the read sequences `AAACGCA`, `AAACGCC` and `AAACGCG`; they share a common prefix `AAACGC`. Fig. B.9 illustrates a compressed trie consisting of these reads as well as the use of the LCP to memoize the result of the FA algorithm. We see that the LCP can be reused across the three reads and the computation can be reduced to (i) FA on the LCP, and (ii) Computing 3 rows corresponding to `A`, `C` and `G`. This optimization is carried out on the host-side (CPU side). The CPU schedules the LCP lattices on to the FPGA accelerator, and the remaining computations are carried out on the CPU using the SIMD (AVX or AltiVec instructions). This optimization significantly reduces: (i) the number of invocations of the accelerator, (ii) the amount of data that has to be moved to the accelerator, and (iii) the amount of redundant computation that is performed on the accelerator.

B.4.2 FA on DeBruijn Graphs

In addition to the reads that share common prefixes, the haplotypes generated by the assembly process in Alg. B.1 also share large common subsequences. The DeBruijn graph [285] produced as a result of the assembly on an active region encodes these common subsequences in a graphical format much like the compressed trie in the previous section. Computing the FA of a read and the DeBruijn graph potentially allows us to reuse these values as well as reduce the number of invocations of the FA algorithm from the number of reads $\times$ the number of haplotypes to just the number of reads (when executing independent of the optimization in Appendix B.4.1). The FA algorithm has to be modified to allow computing similarity between a DeBruijn graph and a read. Given that the graph represents a partially

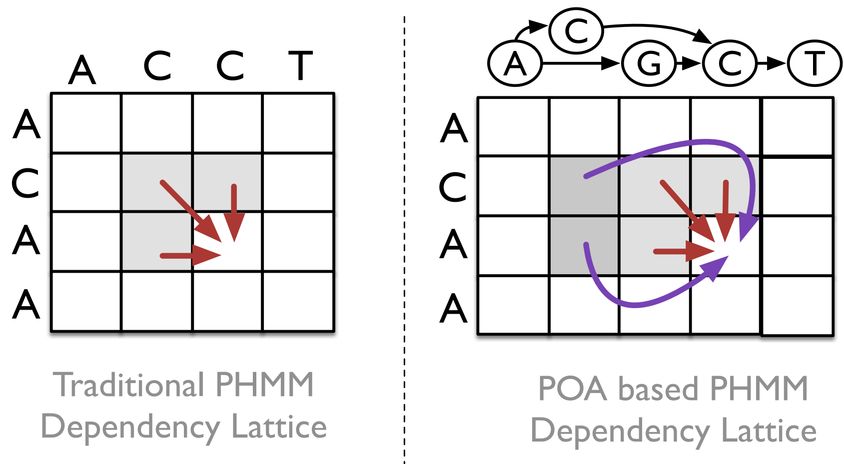


Figure B.10: Dependencies between cells when calculating FA on a DeBruijn graph

ordered set of strings (haplotypes), we first compute a topological sort of the graph to convert it into a total order. Then, we follow the same FA algorithm as described in (B.1), with one major difference: dependencies between lattice elements now incorporate the DeBruijn graph. Fig. B.10 gives an example of conflating graph dependencies with FA dependencies. As the quantities being added in (B.1) represent probabilities, and the branches on the DeBruijn graph represent mutually exclusive subsequences of haplotypes, this transformation produces the correct answer. This approach is a generalization of Lee et al.'s POA algorithm [289] to PHMM models. The software controller is augmented with the ability to dispatch subsequences of haplotypes to the FA accelerator, instead of traversing the DeBruijn graph and dispatching entire haplotypes. The final reduction (addition), of the various topologically sorted subsequences is computed on the CPU. This reduction corresponds to the additional dependencies shown in Fig. B.10.

B.5 EXPERIMENTAL RESULTS

The accelerator is implemented in mixed-language HDL. We used IPs from Xilinx to implement the single-precision floating point adder and multipler and BRAM blocks. We implemented the accelerator on an IBM Power8 S824L system with an Alpha-Data ADM-PCIE-7V3 board (that uses a Xilinx Virtex 7 XC7VX690T FPGA). All the IO interfaces were clocked at 250MHz, and the PEs were clocked at 400MHz. All measurements (baseline CPU as well as FPGA based) were done on this system. Our input data-set for this section was derived from sample G15512.HCCI954.1 (same as used in [272, 283]) and the hg38 reference human genome. We verified the correctness of our FPGA implementation by comparing it

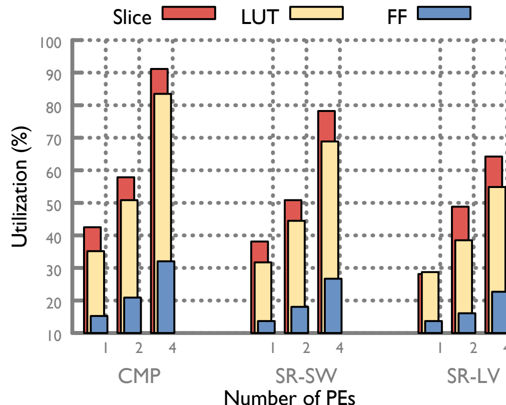


Figure B.11: Resource utilization on the FPGA as a function of the number of PEs.

Table B.2: Comparing the power and performance of the proposed design to that of a Power8 CPU

	Throughput (MCUP/s)	Power (W)		
		Static	Dynamic	Total
Power8 Core	100.8	–	–	–
Power8 Chip	806.6	–	–	190
Single PE on FPGA	296.1	0.5	4.686	8.437
44 PEs on FPGA	11983.6	2.448	13.485	19.139

to the CPU-only version of the GATK HaplotypeCaller (`v3.6` - Provides SIMD optimized and multi-threaded implementations of the algorithm). We use the C++ implementation of the FA on the Power8 CPU as a baseline [290]. This version has been optimized using AltiVec SIMD instructions and multi-threading. Appendix B.6 describes a comparison of this implementation to one [279] that uses AVX256 SIMD for x86 processors.

B.5.1 Resource Utilization

We observed near-linear scaling of the utilization of on-chip resources for the accelerator with the number of PEs (see Fig. B.11). Even though the figure shows a high utilization of logic slices on the FPGA, the actual numbers of LUTs and FFs are much lower. For example, in the 32 PE case, even though we used 60.47% of the slices, we used only 46.85% of LUTs and 26.64% of FFs on the FPGA. Fig. B.12 and Table B.2 report the power consumption of the accelerator. These reports were generated by the Vivado Design Suite.

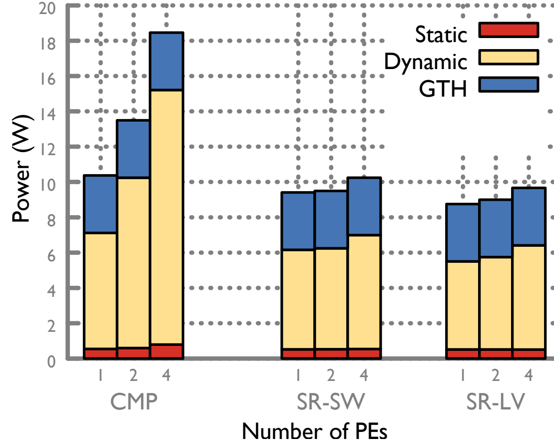


Figure B.12: Power consumption of the accelerator as the number of PEs is increased.

B.5.2 Performance of the Accelerator

Compared with a C++ implementation optimized by IBM for their 8-core Power8 architecture, the proposed accelerator increases aggregate throughput by 14.85×(i.e., 11983.6/806.6) in terms of throughput. We quantify throughput using the popular MCUP/s measure. A MCUP or mega cell update represents the computation of 10^6 steps of the recursion in (B.1) (traditionally each recursion step is called a cell). Further, adding the algorithmic optimizations from Appendix B.4, we observe a 41.59×(*i.e.,* 14.85 ×2.8) improvement in performance. Table B.2 demonstrates that the proposed accelerator significantly outperforms the Power8 CPU in terms of power and performance. A single PE outperforms a Power8 core and a 44-PE accelerator outperforms an 8-core Power8 processor by 147.49×(i.e.,

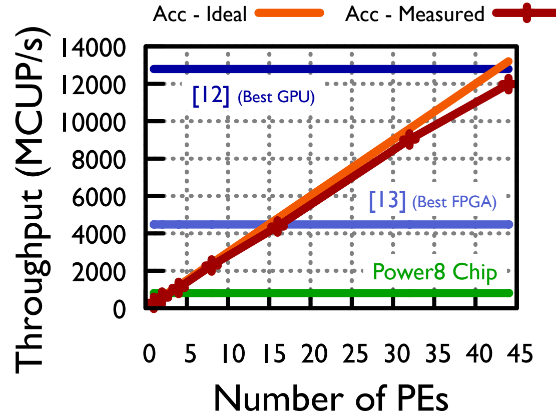


Figure B.13: Mean throughput of the accelerator as a function of the number of PEs.

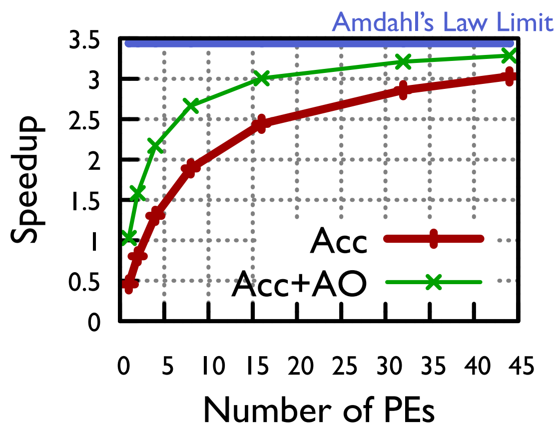


Figure B.14: Mean end-to-end speedup of the HaplotypeCaller when applying the accelerator (ACC) and algorithmic optimizations (AO).

$^{11983.6}/_{806.6} \times {}^{190}/_{19.139}$) in terms of performance per unit energy (i.e., MCUP/Joule). Fig. B.13 describes the performance scaling of the accelerator with the number of PEs. At 44-PEs we observe some non-linearities (the difference between ideal and measured performance) because of insufficient off-chip bandwidth and limitations with the round-robin bus scheduling strategy in Appendix B.3.3.

B.5.3 Integration into GATK

Use of the proposed accelerator and algorithmic optimizations inside the GATK HaplotypeCaller demonstrated a maximum acceleration of 3.287×in runtime when using 44 PEs (see Fig. B.14). The algorithmic optimizations presented in Appendix B.4 account for approximately 2.8×reduction in runtime of the FA algorithm. Fig. B.14 shows this improvement as it applies to end-to-end GATK application. It is to be noted that the optimizations from Appendix B.4 are input-dependent and can produce varying results for other datasets. Furthermore, Fig. B.14 demonstrates diminishing returns from adding more processors in GATK because of Amdahl's law (3.44×asymptotic limit from Table B.1). After using the accelerator and optimizations presented in this chapter, the `Align` function in Alg. B.1 dominates the runtime.

B.5.4 High-Bandwidth Memory Interfaces

With the industry trend of increasing the FPGA area in each successive generation, the number of PEs that fit into an FPGA will also grow. However, simple scaling of the number

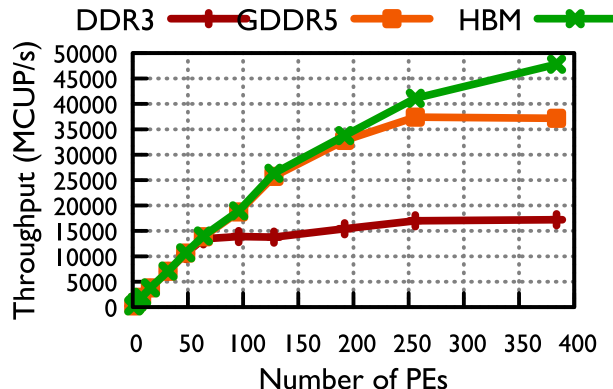


Figure B.15: Simulated throughput when replacing the CAPI-based interface with a DRAM interface.

of processors leads to sub-linear performance scaling, as performance is limited by off-chip bandwidth for the FPGA through the PCIe/CAPI interconnect. To test the scalability of our accelerator, we replaced the CAPI interface with that of a simulated memory controller through the trace-driven simulation framework called Ramulator [110]. We observe that changing effective bandwidth can lead to significant non-linearities in scaling behavior (see Fig. B.15). For example, using HBM (which is already commercially available on flagship Xilinx Ultrascale+ FPGAs) leads to near-linear scaling of performance up to 256 PEs, after which non-linear scaling is observed. This performance scaling, though significant in terms of the FA algorithm, has almost no impact in the GATK HaplotypeCaller because of the diminished returns from Amdahl's law (as seen in Fig. B.14).

B.6 RELATED WORK

In this section, we briefly describe related work (see Table B.3) that has accelerated the FA algorithm on a variety of processors (e.g., CPU, GPU, and FPGA devices). The accelerators in Table B.3 are generally based on SA architectures, which suffer from the shortcoming of inefficient use of hardware resources for handling varied input sizes of real sequence data. Fig. B.1 is a graphical representation of Table B.3. Some of the related work does not clearly mention throughput and power measurements. In all such cases the Intel AVX implementation of the FA algorithm [279, 280] is used to normalize performance stated in the respective papers to that of our baseline. In the situation that power measurements are not provided in cited papers, we assume the publicly stated TDP for the device used in the implementation. Our results demonstrate that the design proposed in this chapter outperforms all the previous designs in terms of both the PHMM micro-benchmark, and transitively to the GATK

Table B.3: List of related high-performance implementations of PHMM. We use the best result presented in the respective paper to compare performance.

Paper	HLS	SA	Platform	Speedup	Speedup / Power
C++ Baseline [290]	–	–	Power8 CPU	1×	1×
[279, 280]	–	–	Intel Xeon CPU	0.91×	1.33×
[281]	✓	✓	Convey HC2	NA[3]	NA[a]
[282]	✓	✓	Stratix V D8	NA[a]	NA[a]
[283]	✗	✓	Power8 & Xilinx KU060	2.35×	33.24×
[178]	✗/✓[4]	✓	Arria 10	6.32×	60.04× (TDP = 20W)
[284]	–	–	Power8 & NVIDIA K40	15.51×	12.80× (TDP = 235W)
[272]	✗	✓	Power8 & Xilinx XC7VX	6.27×	79.74×
This chapter	✗	✗	Power8 & Xilinx XC7VX	14.85×	147.49×

HaplotypeCaller on representative datasets. Our performance results in Table B.3 do not contain the algorithmic optimizations (from Appendix B.4) that further improves performance by 2.8× (i.e., we compare *only* the design of the accelerator). The GPU implementation in [284] demonstrates marginally higher absolute performance (i.e., 1.04× = $^{15.51}/_{14.85}$), but our design represents a significant improvement (i.e., 11.5× = $^{147.49}/_{12.80}$) in terms of performance-per-energy consumed. Furthermore, the K40 GPU has `x16` PCIe connections compared to our `x8`, and on-board GDDR5 memory, use of which will also benefit our design (see Fig. B.15).

Similarity to Smith-Waterman and other Levenshtein distance (LD) algorithms. The PHMM computation is an generalization of edit-distance formulation proposed by Levenshtein in [228] to probabilistic gap penalties [275]. Several LD variants have been accelerated in ASICs and FPGAs (e.g., [235, 251, 259]). However the key point of difference between the LD and the PHMM computations is in the use of floating point math which produces a significantly more complicated data-path. Furthermore, LD computations have been shown to be memory-bound for large lattices, whereas PHMM computations are significantly more compute intensive [288].

B.7 SUMMARY

This chapter presented an approach to accelerating the computation of the FA algorithm in hardware. Our key insight of using input data characteristics to inform architectural

design patterns allows our design to outperform traditional architectures in terms of both energy per operation and runtime performance. The use of this accelerator in genomic data analysis represents a significant acceleration in terms of time spent in computation. The proliferation of sequencing platforms and the resulting explosion in genomic data [11] will make this accelerator even more important in the future. Though most of the techniques presented in this chapter are applicable only to the FA algorithm and transitively only in bioinformatics applications, the general design philosophy of using input data characteristics, in addition to the algorithmic definition, for design specialization of accelerators can be broadly applied across a large number of domains.

REFERENCES

[1] D. Silver, H. van Hasselt, M. Hessel, T. Schaul, A. Guez, T. Harley, G. Dulac-Arnold, D. Reichert, N. Rabinowitz, A. Barreto et al., "The Predictron: End-to-end learning and planning," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70.* JMLR, 2017, pp. 3191–3199.

[2] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Dębiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse et al., "Dota 2 with large scale deep reinforcement learning," *arXiv preprint arXiv:1912.06680*, 2019.

[3] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 1–12.

[4] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 13–24.

[5] A. Maricq, D. Duplyakin, I. Jimenez, C. Maltzahn, R. Stutsman, and R. Ricci, "Taming performance variability," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18).* Carlsbad, CA: USENIX Association, Oct. 2018. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/maricq pp. 409–425.

[6] N. J. Yadwadkar, G. Ananthanarayanan, and R. Katz, "Wrangler: Predictable and faster jobs using fewer resources," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2670979.2671005 p. 1–14.

[7] R. Shea, F. Wang, H. Wang, and J. Liu, "A deep investigation into network performance in virtual machine based cloud environments," in *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, 2014, pp. 1285–1293.

[8] S. Jha, A. Patke, J. Brandt, A. Gentile, B. Lim, M. Showerman, G. Bauer, L. Kaplan, Z. Kalbarczyk, W. Kramer, and R. Iyer, "Measuring congestion in high-performance datacenter interconnects," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, February 2020. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/jha pp. 37–57.

[9] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, p. 74–80, Feb. 2013. [Online]. Available: https://doi.org/10.1145/2408776.2408794

[10] Y. S. Shao and D. Brooks, "Research infrastructures for hardware accelerators," *Synthesis Lectures on Computer Architecture*, vol. 10, no. 4, pp. 1–99, 2015.

[11] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson, "Big data: Astronomical or genomical?" *PLOS Biology*, vol. 13, no. 7, p. e1002195, jul 2015.

[12] S. S. Banerjee, M. El-Hadedy, C. Y. Tan, Z. T. Kalbarczyk, S. S. Lumetta, and R. K. Iyer, "On accelerating pair-HMM computations in programmable hardware," in *Proc. 27th International Conference on Field Programmable Logic and Applications, FPL 2017, Ghent, Belgium, September 4-8, 2017*, 2017, pp. 1–8.

[13] S. S. Banerjee, M. El-Hadedy, J. B. Lim, Z. T. Kalbarczyk, D. Chen, S. S. Lumetta, and R. K. Iyer, "ASAP: Accelerated Short-Read Alignment on Programmable Hardware," *IEEE Transactions on Computers*, vol. 68, no. 3, pp. 331–346, March 2019.

[14] Y. Turakhia, G. Bejerano, and W. J. Dally, "Darwin: A genomics co-processor provides up to 15,000x acceleration on long read assembly," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18, 2018, pp. 199–213.

[15] D. Fujiki, A. Subramaniyan, T. Zhang, Y. Zeng, R. Das, D. Blaauw, and S. Narayanasamy, "Genax: A genome sequencing accelerator," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 69–82.

[16] T. J. Ham, D. Bruns-Smith, B. Sweeney, Y. Lee, S. H. Seo, U. G. Song, Y. H. Oh, K. Asanovic, J. W. Lee, and L. W. Wills, *Genesis: A Hardware Acceleration Framework for Genomic Data Analysis*. IEEE Press, 2020, p. 254–267. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00031

[17] L. Jiang and F. Zokaee, "Exma: A genomics accelerator for exact-matching," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 399–411.

[18] M. Alser, Z. Bingöl, D. S. Cali, J. Kim, S. Ghose, C. Alkan, and O. Mutlu, "Accelerating genome analysis: A primer on an ongoing journey," *IEEE Micro*, vol. 40, no. 5, pp. 65–75, 2020.

[19] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, Y. He, and C. Delimitrou, "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems," in *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2019.

[20] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: An intelligent fine-grained resource management framework for slo-oriented microservices," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/qiu pp. 805–825.

[21] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3297858.3304004 p. 19–33.

[22] H.-C. Kuo, J. Chen, S. Mohan, and T. Xu, "Set the Configuration for the Heart of the OS: On the Practicality of Operating System Kernel Debloating," in *Proceedings of the 2020 ACM SIGMETRICS Conference (SIGMETRICS'20)*, June 2020.

[23] X. Sun, R. Cheng, J. Chen, E. Ang, O. Legunsen, and T. Xu, "Testing Configuration Changes in Context to Prevent Production Failures," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, Nov. 2020.

[24] Q. Chen, T. Wang, O. Legunsen, S. Li, and T. Xu, "Understanding and Discovering Software Configuration Dependencies in Cloud and Datacenter Systems," in *Proceedings of the 2020 ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'20)*, Nov. 2020.

[25] R. Cheng, L. Zhang, D. Marinov, and T. Xu, "Test-Case Prioritization for Configuration Testing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'21)*, July 2021.

[26] J. Dean, "The Deep Learning Revolution and Its Implications for Computer Architecture and Chip Design," *CoRR*, vol. abs/1911.05289, 2019, https://arxiv.org/abs/1911.05289. [Online]. Available: http://arxiv.org/abs/1911.05289

[27] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[28] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair scheduling for distributed computing clusters," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09, 2009, pp. 261–276.

[29] J. Giceva, G. Alonso, T. Roscoe, and T. Harris, "Deployment of query plans on multicores," *Proc. VLDB Endow.*, vol. 8, no. 3, pp. 233–244, Nov. 2014.

[30] R. Lyerly, A. Murray, A. Barbalace, and B. Ravindran, "Aira: A framework for flexible compute kernel execution in heterogeneous platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 2, pp. 269–282, Feb 2018.

[31] J. Mars, L. Tang, and R. Hundt, "Heterogeneity in "homogeneous" warehouse-scale computers: A performance opportunity," *IEEE Comput. Archit. Lett.*, vol. 10, no. 2, pp. 29–32, July 2011.

[32] J. Mars and L. Tang, "Whare-map: Heterogeneity in "homogeneous" warehouse-scale computers," *SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 619–630, June 2013.

[33] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, low latency scheduling," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 69–84.

[34] L. Xu, A. R. Butt, S. Lim, and R. Kannan, "A Heterogeneity-Aware Task Scheduler for Spark," in *Proc. 2018 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2018, pp. 245–256.

[35] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13, 2013, pp. 607–618.

[36] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang, "SMiTe: Precise QoS Prediction on Real-System SMT Processors to Improve Utilization in Warehouse Scale Computers," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2014, pp. 406–418.

[37] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing Shared Resource Contention in Multicore Processors via Scheduling," *SIGPLAN Not.*, vol. 45, no. 3, pp. 129–142, Mar. 2010.

[38] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling," in *Proceedings of the 5th European Conference on Computer Systems*, 2010, pp. 265–278.

[39] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 77–88.

[40] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and QoS-aware Cluster Management," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14, 2014, pp. 127–144.

[41] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*. ACM, 2016, pp. 50–56.

[42] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," *arXiv preprint arXiv:1810.01963*, 2018.

[43] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*, June 2012, pp. 1212–1221.

[44] D. Koeplinger, R. Prabhakar, Y. Zhang, C. Delimitrou, C. Kozyrakis, and K. Olukotun, "Automatic generation of efficient accelerators for reconfigurable hardware," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 115–127.

[45] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel paterns," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 389–402, June 2017.

[46] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis, and K. Olukotun, "Spatial: A language and compiler for application accelerators," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3192366.3192379 p. 296–311.

[47] R. Nigam, S. Atapattu, S. Thomas, Z. Li, T. Bauer, Y. Ye, A. Koti, A. Sampson, and Z. Zhang, "Predictable accelerator design with time-sensitive affine types," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3385412.3385974 p. 393–407.

[48] D. A. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*. IEEE, 2001, pp. 197–206.

[49] S. Han, H. Cai, L. Zhu, J. Lin, K. Wang, Z. Liu, and Y. Lin, "Design automation for efficient deep learning computing," *arXiv preprint arXiv:1904.10616*, 2019.

[50] A. Margaritov, D. Ustiugov, E. Bugnion, and B. Grot, "Prefetched address translation," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3352460.3358294 p. 1023–1036.

[51] Y. Lin, D. Hafdi, K. Wang, Z. Liu, and S. Han, "Neural-hardware architecture search," *Workshop on ML for Systems at NeurIPS*, 2019, http://mlforsystems.org/assets/papers/neurips2019/neural_hardware_lin_2019.pdf.

[52] A. Samajdar, P. Mannan, K. Garg, and T. Krishna, "Genesys: Enabling continuous learning through neural network evolution in hardware," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 855–866.

[53] C. Mendis, C. Yang, Y. Pu, D. Amarasinghe, and M. Carbin, "Compiler auto-vectorization with imitation learning," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 14 625–14 635. [Online]. Available: http://papers.nips.cc/paper/9604-compiler-auto-vectorization-with-imitation-learning.pdf

[54] C. Mendis, A. Renda, S. Amarasinghe, and M. Carbin, "Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks," in *Proceedings of the 36th International Conference on Machine Learning (ICML)*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. Long Beach, California, USA: PMLR, Jun 2019. [Online]. Available: http://groups.csail.mit.edu/commit/papers/19/ithemal-icml.pdf pp. 4505–4515.

[55] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to optimize tensor programs," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS'18. Red Hook, NY, USA: Curran Associates Inc., 2018, p. 3393–3404.

[56] H. Mao, R. Netravali, and M. Alizadeh, "Neural adaptive video streaming with pensieve," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3098822.3098843 pp. 197–210.

[57] N. Jay, N. Rotman, B. Godfrey, M. Schapira, and A. Tamar, "A deep reinforcement learning perspective on internet congestion control," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. Long Beach, California, USA: PMLR, 09–15 Jun 2019. [Online]. Available: http://proceedings.mlr.press/v97/jay19a.html pp. 3050–3059.

[58] A. Dziedzic, V. Sathya, M. I. Rochman, M. Ghosh, and S. Krishnan, “Machine learning enabled spectrum sharing in dense lte-u/wi-fi coexistence scenarios,” *CoRR*, vol. abs/2003.13652, 2020. [Online]. Available: https://arxiv.org/abs/2003.13652

[59] Y. Kong, H. Zang, and X. Ma, “Improving tcp congestion control with machine intelligence,” in *Proceedings of the 2018 Workshop on Network Meets AI & ML*, ser. NetAI’18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3229543.3229550 p. 60–66.

[60] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, “The case for learned index structures,” in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD ’18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3183713.3196909 p. 489–504.

[61] T. Kraska, M. Alizadeh, A. Beutel, E. H. Chi, J. Ding, A. Kristo, G. Leclerc, S. Madden, H. Mao, and V. Nathan, “Sagedb: A learned database system,” in *Conference on Innovative Data Systems Research*, 2019, http://cidrdb.org/cidr2019/papers/p117-kraska-cidr19.pdf.

[62] S. Krishnan, Z. Yang, K. Goldberg, J. Hellerstein, and I. Stoica, “Learning to Optimize Join Queries With Deep Reinforcement Learning,” *CoRR*, vol. abs/1808.03196, 2018, https://arxiv.org/abs/1808.03196. [Online]. Available: http://arxiv.org/abs/1808.03196

[63] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul, “Neo: A learned query optimizer,” *Proc. VLDB Endow.*, vol. 12, no. 11, p. 1705–1718, July 2019. [Online]. Available: https://doi.org/10.14778/3342263.3342644

[64] M. Pradel and K. Sen, “Deepbugs: A learning approach to name-based bug detection,” *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018. [Online]. Available: https://doi.org/10.1145/3276517

[65] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, “Neuzz: Efficient fuzzing with neural program smoothing,” in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 803–817.

[66] H. Mao, P. Negi, A. Narayan, H. Wang, J. Yang, H. Wang, R. Marcus, r. addanki, M. Khani Shirkoohi, S. He, V. Nathan, F. Cangialosi, S. Venkatakrishnan, W.-H. Weng, S. Han, T. Kraska, and D. Alizadeh, “Park: An open platform for learning-augmented computer systems,” in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/f69e505b08403ad2298b9f262659929a-Paper.pdf

[67] F. Y. Yan, H. Ayers, C. Zhu, S. Fouladi, J. Hong, K. Zhang, P. Levis, and K. Winstein, “Learning in situ: a randomized experiment in video streaming,” in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/yan pp. 495–511.

[68] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner et al., "Relational inductive biases, deep learning, and graph networks," *arXiv preprint arXiv:1806.01261*, 2018.

[69] G. Ammons, T. Ball, and J. R. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," *SIGPLAN Not.*, vol. 32, no. 5, pp. 85–96, May 1997.

[70] G. Zellweger, D. Lin, and T. Roscoe, "So many performance events, so little time," in *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, ser. APSys '16. New York, NY, USA: ACM, 2016, pp. 14:1–14:9.

[71] V. M. Weaver and S. A. McKee, "Can hardware performance counters be trusted?" in *2008 IEEE International Symposium on Workload Characterization.* IEEE, 2008, pp. 141–150.

[72] T. Mytkowicz, P. F. Sweeney, M. Hauswirth, and A. Diwan, "Time interpolation: So many metrics, so few registers," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, Dec 2007, pp. 286–300.

[73] M. Dimakopoulou, S. Eranian, N. Koziris, and N. Bambos, "Reliable and efficient performance monitoring in linux," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2016, pp. 396–408.

[74] Y. Lv, B. Sun, Q. Luo, J. Wang, Z. Yu, and X. Qian, "Counterminer: Mining big performance data from hardware counters," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2018, pp. 613–626.

[75] V. M. Weaver, D. Terpstra, and S. Moore, "Non-determinism and overcount on modern hardware performance counter implementations," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2013, pp. 215–224.

[76] S. Banerjee, S. Jha, Z. Kalbarczyk, and R. Iyer, "Inductive-bias-driven reinforcement learning for efficient schedules in heterogeneous clusters," in *Proceedings of the 37th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, H. D. III and A. Singh, Eds., vol. 119. Virtual: PMLR, 13–18 Jul 2020. [Online]. Available: http://proceedings.mlr.press/v119/banerjee20a.html pp. 629–641.

[77] 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 *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 29–44.

[78] R. P. Pothukuchi, S. Y. Pothukuchi, P. Voulgaris, and J. Torrellas, "Yukta: Multilayer resource controllers to maximize efficiency," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, June 2018, pp. 505–518.

[79] R. P. Pothukuchi, J. L. Greathouse, K. Rao, C. Erb, L. Piga, P. G. Voulgaris, and J. Torrellas, "Tangram: Integrated control of heterogeneous computers," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: ACM, 2019, pp. 384–398.

[80] S. J. Tarsa, R. B. R. Chowdhury, J. Sebot, G. Chinya, J. Gaur, K. Sankaranarayanan, C.-K. Lin, R. Chappell, R. Singhal, and H. Wang, "Post-silicon cpu adaptation made practical using machine learning," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: ACM, 2019, pp. 14–26.

[81] Y. Ding, N. Mishra, and H. Hoffmann, "Generative and multi-phase learning for computer systems optimization," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: ACM, 2019, pp. 39–52.

[82] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose, "Sok: The challenges, pitfalls, and perils of using hardware performance counters for security," in *2019 IEEE Symposium on Security and Privacy (SP)*, May 2019, pp. 20–38.

[83] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 2672–2680. [Online]. Available: http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf

[84] P. Dash, M. Karimibiuki, and K. Pattabiraman, "Out of control: Stealthy attacks against robotic vehicles protected by control-based techniques," in *Proceedings of the 35th Annual Computer Security Applications Conference*, ser. ACSAC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3359789.3359847 p. 660–672.

[85] S. Jha, S. Cui, S. S. Banerjee, T. Tsai, Z. T. Kalbarczyk, and R. K. Iyer, "ML-driven Malware for Targeting AV Safety," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2020.

[86] I. Corp., "Intel® 64 and IA-32 Architectures Software Developer Manuals," https://software.intel.com/en-us/articles/intel-sdm, 2016, accessed 2019-03-05.

[87] B. Hall, P. Bergner, A. S. Housfater, M. Kandasamy, T. Magno, A. Mericas, S. Munroe, M. Oliveira, B. Schmidt, W. Schmidt et al., *Performance optimization and tuning techniques for IBM Power Systems processors including IBM POWER8*. IBM Redbooks, 2017.

[88] Linux Community, "perf: Linux profiling with performance counters," https://perf.wiki.kernel.org/index.php/Main_Page, 2019, [Online; accessed 19-November-2019].

[89] Intel, "Intel 64 and ia-32 architectures optimization reference manual," *Intel Corporation, Sept*, 2014.

[90] J. M. May, "Mpx: Software for multiplexing hardware performance counters in multi-threaded programs," in *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*, April 2001, pp. 8 pp.–.

[91] D. J. Berndt and J. Clifford, "Using dynamic time warping to find patterns in time series," in *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, ser. AAAIWS'94. AAAI Press, 1994, p. 359–370.

[92] L. Torvald, "Linux Perf Subsystem Userspace Tools." https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/tools/perf/pmu-events/arch, 2020, accessed 2020-03-05.

[93] D. Tran, A. Kucukelbir, A. B. Dieng, M. Rudolph, D. Liang, and D. M. Blei, "Edward: A library for probabilistic modeling, inference, and criticism," *arXiv preprint arXiv:1610.09787*, 2016.

[94] J. V. Dillon, I. Langmore, D. Tran, E. Brevdo, S. Vasudevan, D. Moore, B. Patton, A. Alemi, M. Hoffman, and R. A. Saurous, "Tensorflow distributions," *arXiv preprint arXiv:1711.10604*, 2017.

[95] S. S. Banerjee, Z. T. Kalbarczyk, and R. K. Iyer, "Acmc2 : Accelerating markov chain monte carlo algorithms for probabilistic models," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: ACM, 2019, pp. 515–528.

[96] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel, "CAPI: A Coherent Accelerator Processor Interface," *IBM J. Research and Develop.*, vol. 59, no. 1, pp. 7:1–7:7, Jan 2015.

[97] A. Yasin, "A top-down method for performance analysis and counters architecture," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2014, pp. 35–44.

[98] Intel, "Top-down microarchitecture analysis method," https://software.intel.com/en-us/vtune-cookbook-top-down-microarchitecture-analysis-method, 2019, [Online; accessed 19-November-2019].

[99] D. Koller and N. Friedman, *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[100] A. Gelman, J. Carlin, H. Stern, and D. Rubin, *Bayesian Data Analysis*. Chapman & Hall, New York, 1995.

[101] M. Opper and O. Winther, "Gaussian processes for classification: Mean-field algorithms," *Neural Comput.*, vol. 12, no. 11, pp. 2655–2684, Nov. 2000.

[102] T. P. Minka, "Expectation propagation for approximate bayesian inference," in *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*, ser. UAI'01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 362–369.

[103] A. Gelman, A. Vehtari, P. Jylänki, T. Sivula, D. Tran, S. Sahai, P. Blomstedt, J. P. Cunningham, D. Schiminovich, and C. Robert, "Expectation propagation as a way of life: A framework for bayesian inference on partitioned data," *arXiv preprint arXiv:1412.4869*, 2017.

[104] H.-J. Koch, "The userspace i/o howto," https://www.kernel.org/doc/html/v4.12/driver-api/uio-howto.html, 2006, [Online; accessed 19-November-2019].

[105] M. K. Papamichael and J. C. Hoe, "Connect: Re-examining conventional wisdom for designing nocs in the context of fpgas," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2145694.2145703 pp. 37–46.

[106] M. D. Hoffman, D. M. Blei, C. Wang, and J. Paisley, "Stochastic variational inference," *The Journal of Machine Learning Research*, vol. 14, no. 1, pp. 1303–1347, 2013.

[107] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, Feb 2014, pp. 10–14.

[108] N. Muralimanohar and R. Balasubramonian, "Cacti 6.0: A tool to understand large caches."

[109] A. N. Udipi, N. Muralimanohar, and R. Balasubramonian, "Non-uniform power access in large caches with low-swing wires," in *2009 International Conference on High Performance Computing (HiPC)*, Dec 2009, pp. 59–68.

[110] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, Jan 2016.

[111] K. Chandrasekar, B. Akesson, and K. Goossens, "Improved power modeling of ddr sdrams," in *2011 14th Euromicro Conference on Digital System Design*, Aug 2011, pp. 99–108.

[112] Intel, "Sparkbench: The big data micro benchmark suite for spark 2.0," https://github.com/intel-hadoop/HiBench/, 2016, accessed 19-November-2019.

[113] 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," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[114] W. Yoo, K. Larson, L. Baugh, S. Kim, and R. H. Campbell, "Adp: Automated diagnosis of performance pathologies using hardware events," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '12. New York, NY, USA: ACM, 2012, pp. 283–294.

[115] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, Oct. 2016.

[116] R. Neill, A. Drebes, and A. Pop, "Fuse: Accurate multiplexing of hardware performance counters across executions," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 4, Dec. 2017. [Online]. Available: https://doi.org/10.1145/3148054

[117] J. Chen, S. S. Banerjee, Z. T. Kalbarczyk, and R. K. Iyer, "Machine learning for load balancing in the linux kernel," in *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, ser. APSys '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3409963.3410492 p. 67–74.

[118] K. Asanović, "FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers." Santa Clara, CA: USENIX Association, Feb. 2014.

[119] M. Mastrolilli and O. Svensson, "(acyclic) job shops are hard to approximate," in *2008 49th Annual IEEE Symposium on Foundations of Computer Science*, Oct 2008, pp. 583–592.

[120] M. Mastrolilli and O. Svensson, "Improved bounds for flow shop scheduling," in *International Colloquium on Automata, Languages, and Programming.* Springer, 2009, pp. 677–688.

[121] K. J. Astrom, "Optimal control of Markov processes with incomplete state information," *Journal of Mathematical Analysis and Applications*, vol. 10, no. 1, pp. 174–205, 1965.

[122] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra, "Planning and acting in partially observable stochastic domains," *Artificial Intelligence*, vol. 101, no. 1–2, pp. 99–134, 1998.

[123] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Harley, T. P. Lillicrap, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ser. ICML'16. JMLR.org, 2016, pp. 1928–1937.

[124] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov, "OpenAI Baselines," https://github.com/openai/baselines, 2017.

[125] R. S. Dreyer and D. B. Alpert, "Apparatus for monitoring the performance of a microprocessor," Aug. 1997, uS Patent 5,657,253.

[126] P. Dagum and M. Luby, "Approximating probabilistic inference in Bayesian belief networks is NP-hard," *Artificial Intelligence*, vol. 60, no. 1, pp. 141–153, 1993.

[127] S. Russell, J. Binder, D. Koller, and K. Kanazawa, "Local learning in probabilistic networks with hidden variables," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, ser. IJCAI'95. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 1146–1152.

[128] J. Binder, D. Koller, S. Russell, and K. Kanazawa, "Adaptive Probabilistic Networks with Hidden Variables," *Machine Learning*, vol. 29, no. 2/3, pp. 213–244, 1997.

[129] M. Hausknecht and P. Stone, "Deep recurrent Q-learning for partially observable MDPs," in *2015 AAAI Fall Symposium Series*, 2015.

[130] K. Narasimhan, T. Kulkarni, and R. Barzilay, "Language understanding for text-based games using deep reinforcement learning," *arXiv preprint arXiv:1506.08941*, 2015.

[131] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski et al., "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[132] M. Jaderberg, V. Mnih, W. M. Czarnecki, T. Schaul, J. Z. Leibo, D. Silver, and K. Kavukcuoglu, "Reinforcement learning with unsupervised auxiliary tasks," *arXiv preprint arXiv:1611.05397*, 2016.

[133] J. N. Foerster, Y. M. Assael, N. de Freitas, and S. Whiteson, "Learning to communicate to solve riddles with deep distributed recurrent Q-networks," *arXiv preprint arXiv:1602.02672*, 2016.

[134] P. Karkus, D. Hsu, and W. S. Lee, "QMDP-Net: Deep learning for planning under partial observability," in *Advances in Neural Information Processing Systems*, 2017, pp. 4694–4704.

[135] P. Zhu, X. Li, P. Poupart, and G. Miao, "On improving deep reinforcement learning for POMDPs," *arXiv preprint arXiv:1804.06309*, 2018.

[136] M. Igl, L. Zintgraf, T. A. Le, F. Wood, and S. Whiteson, "Deep variational reinforcement learning for POMDPs," *arXiv preprint arXiv:1806.02426*, 2018.

[137] V. R. Konda and J. N. Tsitsiklis, "Actor-critic algorithms," in *Advances in neural information processing systems*, 2000, pp. 1008–1014.

[138] R. D. Shachter, "Bayes-ball: The rational pastime (for determining irrelevance and requisite information in belief networks and influence diagrams)," *arXiv preprint arXiv:1301.7412*, 2013.

[139] K. Asanović, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, "A View of the Parallel Computing Landscape," *Commun. ACM*, vol. 52, no. 10, pp. 56–67, Oct. 2009.

[140] S. S. Banerjee, A. P. Athreya, L. S. Mainzer, C. V. Jongeneel, W.-M. Hwu, Z. T. Kalbarczyk, and R. K. Iyer, "Efficient and scalable workflows for genomic analyses," in *Proceedings of the ACM International Workshop on Data-Intensive Distributed Computing*, ser. DIDC '16, 2016, pp. 27–36.

[141] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 265–283.

[142] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. Henry, R. Bradshaw, and Nathan, "FlumeJava: Easy, Efficient Data-Parallel Pipelines," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2010, pp. 363–375.

[143] M. McCool, J. Reinders, and A. Robison, *Structured Parallel Programming: Patterns for Efficient Computation*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

[144] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12, 2012, pp. 15–28.

[145] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting Performance Data with PAPI-C," in *Tools for High Performance Computing 2009*, M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 157–173.

[146] A. Kleen, "PMU-Tools," https://github.com/andikleen/pmu-tools, 2010, accessed 2019-03-05.

[147] A. T. Sudhakar and M. Srinivasan, "IBM POWER in-memory collection counters," https://developer.ibm.com/articles/power9-in-memory-collection-counters/, 2019, accessed 2019-03-05.

[148] J. Doweck, "Inside 6th generation Intel Core code named Skylake:: New Microarchitecture and Power Management," https://www.hotchips.org/wp-content/uploads/hc_archives/hc28/HC28.23-Tuesday-Epub/HC28.23.90-High-Perform-Epub/HC28.23.911-Skylake-Doweck-Intel_SK3-r13b.pdf, 2016, accessed 2019-03-05.

[149] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications," in *Proc. 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, Feb 2010, pp. 180–186.

[150] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14, 2014, pp. 443–454.

[151] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni, "Graphene: Packing and Dependency-aware Scheduling for Data-parallel Clusters," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, 2016, pp. 81–97.

[152] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili, "Kernel Weaver: Automatically Fusing Database Primitives for Efficient GPU Computation," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2012, pp. 107–118.

[153] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[154] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. USENIX Association, 2011, pp. 295–308.

[155] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan, "Altruistic Scheduling in Multi-resource Clusters," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 2016, pp. 65–80.

[156] G. A. Van der Auwera, M. O. Carneiro, C. Hartl, R. Poplin, G. del Angel, A. Levy-Moonshine, T. Jordan, K. Shakir, D. Roazen, J. Thibault, E. Banks, K. V. Garimella, D. Altshuler, S. Gabriel, and M. A. DePristo, "From fastq data to high-confidence variant calls: The genome analysis toolkit best practices pipeline," *Current Protocols in Bioinformatics*, vol. 43, no. 1, pp. 11.10.1–11.10.33, 2013.

[157] H. Li and R. Durbin, "Fast and accurate short-read alignment with burrows-wheeler rransform," *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, may 2009. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/btp324

[158] H. Li and R. Durbin, "Fast and accurate long-read alignment with Burrows–Wheeler transform," *Bioinformatics*, vol. 26, no. 5, pp. 589–595, 2010.

[159] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg, "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome," *Genome Biol*, vol. 10, no. 3, p. R25, 2009.

[160] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly, and M. A. DePristo, "The genome analysis toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data," *Genome Research*, vol. 20, no. 9, pp. 1297–1303, jul 2010.

[161] F. Nothaft, M. Massie, T. Danford, Z. Zhang, U. Laserson, C. Yeksigian, J. Kottalam, A. Ahuja, J. Hammerbacher, M. Linderman, M. J. Franklin, A. D. Joseph, and D. A. Patterson, "Rethinking data-intensive science using scalable analytics systems," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 631–646.

[162] F. Nothaft, "Scalable genome resequencing with adam and avocado," M.S. thesis, EECS Department, University of California, Berkeley, May 2015.

[163] A. Rimmer, H. Phan, I. Mathieson, Z. Iqbal, S. R. F. Twigg, A. O. M. Wilkie, G. McVean, and G. Lunter, "Integrating mapping-, assembly- and haplotype-based approaches for calling variants in clinical sequencing applications," *Nature Genetics*, vol. 46, no. 8, pp. 912–918, jul 2014.

[164] M. Zaharia, W. J. Bolosky, K. Curtis, A. Fox, D. Patterson, S. Shenker, I. Stoica, R. M. Karp, and T. Sittler, "Faster and more accurate sequence alignment with SNAP," *arXiv preprint arXiv:1111.5572*, 2011.

[165] Y. Varatharajah, M. J. Chong, K. Saboo, B. Berry, B. Brinkmann, G. Worrell, and R. Iyer, "Eeg-graph: A factor-graph-based model for capturing spatial, temporal, and observational relationships in electroencephalograms," in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Curran Associates, Inc., 2017, pp. 5371–5380.

[166] P. Cao, E. Badger, Z. Kalbarczyk, R. Iyer, and A. Slagell, "Preemptive intrusion detection: Theoretical framework and real-world measurements," in *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, ser. HotSoS '15. New York, NY, USA: ACM, 2015, pp. 5:1–5:12.

[167] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin et al., "The sequence alignment/map format and SAMtools," *Bioinformatics*, vol. 25, no. 16, pp. 2078–2079, 2009.

[168] S. S. Banerjee, Z. T. Kalbarczyk, and R. K. Iyer, "AcMC^2 : Accelerating Markov Chain Monte Carlo Algorithms for Probabilistic Models," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 515–528.

[169] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning.* MIT press Cambridge, 2016, vol. 1.

[170] Y. Gal, "Uncertainty in deep learning," *University of Cambridge*, 2016.

[171] R. Cai, A. Ren, N. Liu, C. Ding, L. Wang, X. Qian, M. Pedram, and Y. Wang, "Vibnn: Hardware acceleration of bayesian neural networks," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 476–488.

[172] M. Karkooti and J. R. Cavallaro, “Semi-parallel reconfigurable architectures for real-time ldpc decoding,” in *International Conference on Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004.*, vol. 1, April 2004, pp. 579–585 Vol.1.

[173] N. Bani Asadi, C. W. Fletcher, G. Gibeling, E. N. Glass, K. Sachs, D. Burke, Z. Zhou, J. Wawrzynek, W. H. Wong, and G. P. Nolan, “Paralearn: A massively parallel, scalable system for learning interaction networks on fpgas,” in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS ’10. New York, NY, USA: ACM, 2010, pp. 83–94.

[174] J. Choi and R. A. Rutenbar, “Hardware implementation of mrf map inference on an fpga platform,” in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2012, pp. 209–216.

[175] M. Hosseini, R. Islam, A. Kulkarni, and T. Mohsenin, “A scalable fpga-based accelerator for high-throughput mcmc algorithms,” in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2017, pp. 201–201.

[176] G. Mingas, L. Bottolo, and C.-S. Bouganis, “Particle mcmc algorithms and architectures for accelerating inference in state-space models,” *International Journal of Approximate Reasoning*, vol. 83, pp. 413–433, 2017.

[177] M. Lin, I. Lebedev, and J. Wawrzynek, “High-throughput bayesian computing machine with reconfigurable hardware,” in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA ’10. New York, NY, USA: ACM, 2010, pp. 73–82.

[178] S. Huang, G. J. Manikandan, A. Ramachandran, K. Rupnow, W.-m. W. Hwu, and D. Chen, “Hardware acceleration of the pair-hmm algorithm for dna variant calling,” in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA ’17. New York, NY, USA: ACM, 2017, pp. 275–284.

[179] S. S. Banerjee, M. el Hadedy, C. Y. Tan, Z. T. Kalbarczyk, S. Lumetta, and R. K. Iyer, “On accelerating pair-hmm computations in programmable hardware,” in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2017, pp. 1–8.

[180] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani, “Probabilistic programming,” in *Proceedings of the on Future of Software Engineering*, ser. FOSE 2014. New York, NY, USA: ACM, 2014, pp. 167–181.

[181] B. Milch, B. Marthi, and S. Russell, “Blog: Relational modeling with unknown objects,” in *ICML 2004 Workshop on Statistical Relational Learning and Its Connections*, 2004.

[182] N. D. Goodman, V. K. Mansinghka, D. Roy, K. Bonawitz, and J. B. Tenenbaum, “Church: A language for generative models,” in *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence*, ser. UAI’08. Arlington, Virginia, United States: AUAI Press, 2008, pp. 220–229.

[183] S. Hershey, J. Bernstein, B. Bradley, A. Schweitzer, N. Stein, T. Weber, and B. Vigoda, "Accelerating inference: towards a full language, compiler and hardware stack," *arXiv preprint arXiv:1212.2991*, 2012.

[184] F. Wood, J. W. Meent, and V. Mansinghka, "A New Approach to Probabilistic Programming Inference," in *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, S. Kaski and J. Corander, Eds., vol. 33. Reykjavik, Iceland: PMLR, 22–25 Apr 2014, pp. 1024–1032.

[185] V. K. Mansinghka, D. Selsam, and Y. Perov, "Venture: A higher-order probabilistic programming platform with programmable inference," *arXiv preprint*, vol. arXiv:1404.0099, 2014.

[186] B. Carpenter, A. Gelman, M. D. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. Brubaker, J. Guo, P. Li, and A. Riddell, "Stan: A probabilistic programming language," *Journal of Statistical Software*, vol. 76, no. 1, 2017.

[187] J. Salvatier, T. V. Wiecki, and C. Fonnesbeck, "Probabilistic programming in python using pymc3," *PeerJ Computer Science*, vol. 2, p. e55, Apr. 2016.

[188] D. Huang, J.-B. Tristan, and G. Morrisett, "Compiling markov chain monte carlo algorithms for probabilistic modeling," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: ACM, 2017, pp. 111–125.

[189] G. F. Cooper, "The computational complexity of probabilistic inference using bayesian belief networks," *Artificial Intelligence*, vol. 42, no. 2, pp. 393–405, 1990.

[190] S. Geman and D. Geman, "Stochastic relaxation, gibbs distributions, and the bayesian restoration of images," in *Readings in Computer Vision*. Elsevier, 1987, pp. 564–584.

[191] R. M. Neal et al., "Mcmc using hamiltonian dynamics," *Handbook of Markov Chain Monte Carlo*, vol. 2, no. 11, 2011.

[192] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *IEEE Design Test of Computers*, vol. 26, no. 4, pp. 18–25, July 2009.

[193] J. M. Mooij, "libDAI: A free and open source C++ library for discrete approximate inference in graphical models," *Journal of Machine Learning Research*, vol. 11, pp. 2169–2173, Aug. 2010.

[194] B. Andres, T. Beier, and J. Kappes, "OpenGM: A C++ library for discrete graphical models," *CoRR*, vol. abs/1206.0111, 2012.

[195] W. K. Hastings, "Monte carlo sampling methods using markov chains and their applications," *Biometrika*, vol. 57, no. 1, pp. 97–109, 1970.

[196] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of state calculations by fast computing machines," *The Journal of Chemical Physics*, vol. 21, no. 6, pp. 1087–1092, 1953.

[197] B. Leimkuhler and S. Reich, *Simulating hamiltonian dynamics.* Cambridge university press, 2004, vol. 14.

[198] F. R. Kschischang, B. J. Frey, and H.-A. Loeliger, "Factor graphs and the sum-product algorithm," *IEEE Transactions on information theory*, vol. 47, no. 2, pp. 498–519, 2001.

[199] B. J. Frey, "Extending factor graphs so as to unify directed and undirected graphical models," in *Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence*, ser. UAI'03. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003, pp. 257–264.

[200] Y. Wu, L. Li, S. Russell, and R. Bodik, "Swift: Compiled inference for probabilistic programming languages," in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, ser. IJCAI'16. AAAI Press, 2016, pp. 3637–3645.

[201] S. Brooks, A. Gelman, G. Jones, and X.-L. Meng, *Handbook of Markov Chain Monte Carlo.* CRC press, 2011.

[202] M. H. DeGroot and M. J. Schervish, *Probability and statistics.* Pearson Education, 2012.

[203] J. Gonzalez, Y. Low, A. Gretton, and C. Guestrin, "Parallel gibbs sampling: From colored fields to thin junction trees," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, G. Gordon, D. Dunson, and M. Dudík, Eds., vol. 15. Fort Lauderdale, FL, USA: PMLR, 11–13 Apr 2011, pp. 324–332.

[204] M. Bartholomew-Biggs, S. Brown, B. Christianson, and L. Dixon, "Automatic differentiation of algorithms," *Journal of Computational and Applied Mathematics*, vol. 124, no. 1-2, pp. 171–190, 2000.

[205] J. Fifield, R. Keryell, H. Ratigner, H. Styles, and J. Wu, "Optimizing opencl applications on xilinx fpga," in *Proceedings of the 4th International Workshop on OpenCL*, ser. IWOCL '16. New York, NY, USA: ACM, 2016, pp. 5:1–5:2.

[206] G. Casella, C. P. Robert, and M. T. Wells, "Generalized accept-reject sampling schemes," in *Institute of Mathematical Statistics Lecture Notes - Monograph Series.* Institute of Mathematical Statistics, 2004, pp. 342–347.

[207] D. B. Thomas and W. Luk, "Fpga-optimised uniform random number generators using luts and shift registers," in *2010 International Conference on Field Programmable Logic and Applications*, Aug 2010, pp. 77–82.

[208] G. Marsaglia, "Xorshift RNGs," *Journal of Statistical Software*, vol. 8, no. 14, 2003.

[209] A. Walker, "New fast method for generating discrete random numbers with arbitrary frequency distributions," *Electronics Letters*, vol. 10, no. 8, p. 127, 1974.

[210] C. P. Robert, *Monte Carlo Methods*. Wiley Online Library, 2004.

[211] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM transactions on networking*, vol. 8, no. 3, pp. 281–293, 2000.

[212] A. Appleby, "Murmurhash," 2008. [Online]. Available: https://sites.google.com/site/murmurhash

[213] I. Strid, "Efficient parallelisation of metropolis–hastings algorithms using a prefetching approach," *Computational Statistics & Data Analysis*, vol. 54, no. 11, pp. 2814–2835, 2010.

[214] E. Angelino, E. Kohler, A. Waterland, M. Seltzer, and R. P. Adams, "Accelerating mcmc via parallel predictive prefetching," in *Proceedings of the Thirtieth Conference on Uncertainty in Artificial Intelligence*, ser. UAI'14. Arlington, Virginia, United States: AUAI Press, 2014, pp. 22–31.

[215] D. C. Ku and G. DeMicheli, *High level synthesis of ASICs under timing and synchronization constraints*. Springer Science & Business Media, 2013, vol. 177.

[216] M. Brobbel, "capi-streaming-framework," 2015. [Online]. Available: https://github.com/mbrobbel/capi-streaming-framework

[217] V. Paxson, "Bro: a System for Detecting Network Intruders in Real-Time," *Computer Networks*, vol. 31, no. 23-24, pp. 2435–2463, 1999.

[218] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, Apr. 2012.

[219] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Advances in Neural Information Processing Systems 24*, J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2011, pp. 693–701.

[220] C. De Sa, K. Olukotun, and C. Ré, "Ensuring rapid mixing and low bias for asynchronous gibbs sampling," in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ser. ICML'16. JMLR.org, 2016, pp. 1567–1576.

[221] M. D. Homan and A. Gelman, "The no-u-turn sampler: Adaptively setting path lengths in hamiltonian monte carlo," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1593–1623, Jan. 2014.

[222] M. A. Suchard, Q. Wang, C. Chan, J. Frelinger, A. Cron, and M. West, "Understanding gpu programming for statistical computation: Studies in massively parallel massive mixtures," *Journal of computational and graphical statistics*, vol. 19, no. 2, pp. 419–438, 2010.

[223] K. Esler, J. Kim, D. Ceperley, and L. Shulenburger, "Accelerating quantum monte carlo simulations of real materials on gpu clusters," *Computing in Science & Engineering*, vol. 14, no. 1, pp. 40–51, 2012.

[224] A. Terenin, S. Dong, and D. Draper, "Gpu-accelerated gibbs sampling: a case study of the horseshoe probit model," *Statistics and Computing*, Mar 2018.

[225] B. Vigoda, "Analog logic: Continuous-time analog circuits for statistical signal processing," Ph.D. dissertation, Massachusetts Institute of Technology, 2003.

[226] D. Tran, M. D. Hoffman, R. A. Saurous, E. Brevdo, K. Murphy, and D. M. Blei, "Deep probabilistic programming," in *International Conference on Learning Representations*, 2017.

[227] L. A. Barroso and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines," *Synthesis lectures on computer architecture*, vol. 4, no. 1, pp. 1–108, 2009.

[228] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," Tech. Rep. 8, 1966.

[229] G. Navarro, "A guided tour to approximate string matching," *ACM Comput. Surv.*, vol. 33, no. 1, pp. 31–88, Mar. 2001.

[230] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, no. 3, pp. 403–410, oct 1990.

[231] B. Langmead and S. Salzberg, "Fast gapped-read alignment with bowtie 2," *Nature Methods*, vol. 9, pp. 357–359, March 2012.

[232] H. Li and R. Durbin, "Fast and accurate long-read alignment with burrows-wheeler transform," *Bioinformatics*, vol. 26, no. 5, pp. 589–595, jan 2010.

[233] T. C. GLENN, "Field guide to next-generation DNA sequencers," *Molecular Ecology Resources*, vol. 11, no. 5, pp. 759–769, may 2011.

[234] M. G. Ross, C. Russ, M. Costello, A. Hollinger, N. J. Lennon, R. Hegarty, C. Nusbaum, and D. B. Jaffe, "Characterizing and measuring bias in sequence data," *Genome Biology*, vol. 14, no. 5, p. R51, 2013.

[235] A. Madhavan, T. Sherwood, and D. Strukov, "Race logic: A hardware acceleration for dynamic programming algorithms," *SIGARCH Comput. Archit. News*, vol. 42, no. 3, pp. 517–528, June 2014.

[236] I. C. Systems and T. Group, "Coherent accelerator processor interface: User's manual," 2015. [Online]. Available: http://www.nallatech.com/wp-content/uploads/IBM_CAPI_Users_Guide_1-2.pdf

[237] R. A. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[238] S. Kumar and E. H. Spafford, "A pattern matching model for misuse intrusion detection," in *In Proceedings of the 17th National Computer Security Conference*, 1994, pp. 11–21.

[239] T. Smith and M. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981.

[240] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, mar 1970.

[241] W.-K. Sung, *Algorithms in Bioinformatics: A Practical Introduction (Chapman & Hall/CRC Mathematical and Computational Biology).* Chapman and Hall/CRC, 2009.

[242] S. Henikoff and J. G. Henikoff, "Amino acid substitution matrices from protein blocks." *Proceedings of the National Academy of Sciences*, vol. 89, no. 22, pp. 10 915–10 919, nov 1992.

[243] C. Wang, R.-X. Yan, X.-F. Wang, J.-N. Si, and Z. Zhang, "Comparison of linear gap penalties and profile-based variable gap penalties in profile–profile alignments," *Computational Biology and Chemistry*, vol. 35, no. 5, pp. 308–318, oct 2011.

[244] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, ser. FOCS '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 390–.

[245] G. M. Landau and U. Vishkin, "Efficient string matching with k mismatches," *Theoretical Computer Science*, vol. 43, pp. 239–249, 1986.

[246] Illumina, "Pair-end sequencing," 2010. [Online]. Available: http://www.illumina.com/technology/next-generation-sequencing/paired-end-sequencing_assay.html

[247] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," Tech. Rep., 1994.

[248] E. Ukkonen, "Algorithms for approximate string matching," *Information and control*, vol. 64, no. 1, pp. 100–118, 1985.

[249] Z. D. Stephens, M. E. Hudson, L. S. Mainzer, M. Taschuk, M. R. Weber, and R. K. Iyer, "Simulating next-generation sequencing datasets from empirical mutation and sequencing models," *PLOS ONE*, vol. 11, no. 11, p. e0167047, nov 2016.

[250] N. C. for Supercomputing Applications (NCSA), "Blue waters supercomputer," 2012. [Online]. Available: https://bluewaters.ncsa.illinois.edu/

[251] R. J. Lipton and D. Lopresti, "A systolic array for rapid string comparison," in *Proceedings of the Chapel Hill Conference on VLSI*, 1985, pp. 363–376.

[252] D. T. Hoang and D. P. Lopresti, "FPGA implementation of systolic sequence alignment," in *Lecture Notes in Computer Science*. Springer Science + Business Media, 1993, pp. 183–191.

[253] S. A. Guccione and K. Eric, *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream: 12th International Conference, FPL 2002 Montpellier, France, September 2–4, 2002 Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, ch. Gene Matching Using JBits, pp. 1168–1171.

[254] P. Zhang, G. Tan, and G. R. Gao, "Implementation of the smith-waterman algorithm on a reconfigurable supercomputing platform," in *Proceedings of the 1st International Workshop on High-performance Reconfigurable Computing Technology and Applications: Held in Conjunction with SC07*, ser. HPRCTA '07. New York, NY, USA: ACM, 2007, pp. 39–48.

[255] N. Ahmed, V. M. Sima, E. Houtgast, K. Bertels, and Z. Al-Ars, "Heterogeneous hardware/software acceleration of the bwa-mem dna alignment algorithm," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2015, pp. 240–246.

[256] Xilinx, "Hierarchical design methodology guide," 2010. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx12_2/Hierarchical_Design_Methodology_Guide.pdf

[257] M. J. Jaspers, "Acceleration of read alignment with coherent attached FPGA coprocessors," M.S. thesis, Delft University of Technology, The Netherlands, 2015.

[258] J. Daily, "Parasail: SIMD c library for global, semi-global, and local pairwise sequence alignments," *BMC Bioinformatics*, vol. 17, no. 1, feb 2016.

[259] A. Sirasao, E. Delaye, R. Sunkavalli, and S. Neuendorffer, "Fpga based opencl acceleration of genome sequencing software," in *Poster presented at Supercomputing 2015*, Austin, TX, Nov 2015. [Online]. Available: http://sc15.supercomputing.org/sites/all/themes/SC15images/tech_poster/tech_poster_pages/post269.html

[260] G. Andrews, *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000.

[261] E. Georganas, A. Buluç, J. Chapman, L. Oliker, D. Rokhsar, and K. Yelick, "meraligner: A fully parallel sequence aligner," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, May 2015, pp. 561–570.

[262] M. Farrar, "Striped smith-waterman speeds database searches six times over other SIMD implementations," *Bioinformatics*, vol. 23, no. 2, pp. 156–161, nov 2006.

[263] R. Hughey, "Parallel hardware for sequence comparison and alignment," *Comput. Appl. Biosci.*, vol. 12, no. 6, pp. 473–479, Dec 1996.

[264] Y. Liu, W. Huang, J. Johnson, and S. Vaidya, *Computational Science – ICCS 2006: 6th International Conference, Reading, UK, May 28-31, 2006, Proceedings, Part IV*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, ch. GPU Accelerated Smith-Waterman, pp. 188–195.

[265] Y. Liu, B. Schmidt, and D. L. Maskell, "CUSHAW: a CUDA compatible short read aligner to large genomes based on the burrows-wheeler transform," *Bioinformatics*, vol. 28, no. 14, pp. 1830–1837, may 2012.

[266] K. Zhao and X. Chu, "G-BLASTN: accelerating nucleotide alignment by graphics processors," *Bioinformatics*, vol. 30, no. 10, pp. 1384–1391, jan 2014.

[267] N. Corporation, "Nvbio," 2015. [Online]. Available: https://developer.nvidia.com/nvbio

[268] Y. T. Chen, J. Cong, J. Lei, and P. Wei, "A novel high-throughput acceleration engine for read alignment," in *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, May 2015, pp. 199–202.

[269] R. Kaplan, L. Yavits, R. Ginosar, and U. Weiser, "A resistive cam processing-in-storage architecture for dna sequence alignment," *IEEE Micro*, vol. 37, no. 4, pp. 20–28, 2017.

[270] M. Alser, H. Hassan, H. Xin, O. Ergin, O. Mutlu, and C. Alkan, "GateKeeper: a new hardware architecture for accelerating pre-alignment in DNA short read mapping," *Bioinformatics*, vol. 33, no. 21, pp. 3355–3363, may 2017.

[271] E. Rucci, C. Garcia, G. Botella, A. E. D. Giusti, M. Naiouf, and M. Prieto-Matias, "Oswald: Opencl smith–waterman on altera's fpga for large protein databases," *The International Journal of High Performance Computing Applications*, vol. 32, no. 3, pp. 337–350, 2018.

[272] J. Peltenburg, Johan, S. Ren, and Z. Al-Ars, "Maximizing systolic array efficiency to accelerate the PairHMM Forward Algorithm," *2016 IEEE Int. Conf. Bioinformatics and Biomedicine (BIBM)*, vol. 00, pp. 758–762, 2016.

[273] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, aug 2014.

[274] B. V. Benjamin, P. Gao, E. McQuinn, S. Choudhary, A. R. Chandrasekaran, J.-M. Bussat, R. Alvarez-Icaza, J. V. Arthur, P. A. Merolla, and K. Boahen, "Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations," *Proceedings of the IEEE*, vol. 102, no. 5, pp. 699–716, may 2014.

[275] L. Pachter and B. Sturmfels, *Algebraic Statistics for Computational biology.* Cambridge University Press, 2005., vol. 13.

[276] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison, *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids.* Cambridge University Press, 1998.

[277] B.-J. Yoon, "Hidden Markov Models and their Applications in Biological Sequence Analysis," *CG*, vol. 10, no. 6, pp. 402–415, Sep 2009.

[278] M. A. DePristo, E. Banks, R. Poplin, K. V. Garimella, J. R. Maguire, C. Hartl, A. A. Philippakis, G. del Angel, M. A. Rivas, M. Hanna, A. McKenna, T. J. Fennell, A. M. Kernytsky, A. Y. Sivachenko, K. Cibulskis, S. B. Gabriel, D. Altshuler, and M. J. Daly, "A framework for variation discovery and genotyping using next-generation DNA sequencing data," *Nature Genetics*, vol. 43, no. 5, pp. 491–498, Apr 2011.

[279] M. Carneiro, "Accelerating variant calling," https://hpc.mssm.edu/files/Carneiro_workshop.pdf, 2013, accessed: 2017-04-01.

[280] Intel, "Replication Recipe for HaplotypeCaller Tool Runtimes," http://www.intel.com/content/dam/www/public/us/en/documents/pdf/10380-2%20Recipe-GATK_021615.pdf, 2016, accessed: 2017-04-01.

[281] S. Ren, V. M. Sima, and Z. Al-Ars, "FPGA acceleration of the pair-HMMs forward algorithm for DNA sequence analysis," in *Bioinformatics and Biomedicine (BIBM), 2015 IEEE Int. Conf.*, Nov 2015, pp. 1465–1470.

[282] C. Rauer and N. J. Finamore, "Accelerating Genomics Research with OpenCL and FPGAs," https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01262-accelerating-genomics-research-with-opencl-and-fpgas.pdf, 2016, accessed: 2017-04-01.

[283] M. Ito and M. Ohara, "A power-efficient FPGA accelerator: Systolic array with cache-coherent interface for pair-HMM algorithm," in *2016 IEEE Symp. Low-Power and High-Speed Chips (COOL CHIPS XIX)*, Apr 2016, pp. 1–3.

[284] S. Ren, K. Bertel, and Z. Al-Ars, "Exploration of alternative GPU implementations of the pair-HMMs forward algorithm," in *2016 IEEE Int. Conf. Bioinformatics and Biomedicine (BIBM)*, Dec 2016, pp. 902–909.

[285] N. G. de Bruijn, "A Combinatorial Problem," *Koninklijke Nederlandsche Akademie Van Wetenschappen*, vol. 49, no. 6, pp. 758–764, Jun 1946.

[286] H. Li, "A statistical framework for SNP calling, mutation discovery, association mapping and population genetical parameter estimation from sequencing data," *Bioinformatics*, vol. 27, no. 21, pp. 2987–2993, 2011.

[287] J. M. Zook, D. Catoe, J. McDaniel, L. Vang, N. Spies, A. Sidow, Z. Weng, Y. Liu, C. E. Mason, N. Alexander, E. Henaff, A. B. McIntyre, D. Chandramohan, F. Chen, E. Jaeger, A. Moshrefi, K. Pham, W. Stedman, T. Liang, M. Saghbini, Z. Dzakula, A. Hastie, H. Cao, G. Deikus, E. Schadt, R. Sebra, A. Bashir, R. M. Truty, C. C. Chang, N. Gulbahce, K. Zhao, S. Ghosh, F. Hyland, Y. Fu, M. Chaisson, C. Xiao, J. Trow, S. T. Sherry, A. W. Zaranek, M. Ball, J. Bobe, P. Estep, G. M. Church, P. Marks, S. Kyriazopoulou-Panagiotopoulou, G. X. Zheng, M. Schnall-Levin, H. S. Ordonez, P. A. Mudivarti, K. Giorda, Y. Sheng, K. B. Rypdal, and M. Salit, "Extensive sequencing of seven human genomes to characterize benchmark reference materials," *Scientific Data*, vol. 3, p. 160025, Jun 2016.

[288] A. Konopka and M. Crabbe, *Compact Handbook of Computational Biology.* CRC Press, 2004.

[289] C. Lee, C. Grasso, and M. F. Sharlow, "Multiple sequence alignment using partial order graphs," *Bioinformatics*, vol. 18, no. 3, pp. 452–464, Mar 2002.

[290] G. Van Der Aura, "Speed up HaplotypeCaller on IBM POWER8 systems," http://gatkforums.broadinstitute.org/gatk/discussion/4833/speed-up-haplotypecaller-on-ibm-power8-systems, accessed: 2017-03-31.