MITIGATION OF FAILURES IN HIGH PERFORMANCE COMPUTING VIA RUNTIME TECHNIQUES

BY

XIANG NI

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 at Urbana-Champaign, 2016

Urbana, Illinois

Doctoral Committee:

Professor Laxmikant V. Kalé, Chair
Professor Nitin Vaidya
Professor William Kramer
Doctor Franck Cappello, Argonne National Laboratory

Abstract

As machines increase in scale, it is predicted that failure rates of supercomputers will correspondingly increase. Even though the mean time to failure (MTTF) of individual component is high, the large number of components significantly decreases the system MTTF. Meanwhile, the decreasing size of transistors has been critical to the increase in capacity of supercomputers. The smaller the transistors are, silent data corruptions (SDC) are likely to occur more frequently. SDCs do not inhibit execution, but may silently lead to incorrect results.

In this thesis, we leverage runtime system and compiler techniques to mitigate a significant fraction of failures automatically with low overhead. The main goals of various system-level fault tolerance strategies designed in this thesis are: reducing the extra cost added to application execution while improving system reliability; automatically adjusting fault tolerance decisions without user intervention based on environmental changes; protecting applications not only from fail-stop failures but also from silent data corruptions.

The main contributions of this thesis are development of a semi-blocking checkpoint protocol that overlaps application execution with fault tolerance operation to reduce the overhead of checkpointing, a runtime system technique for automatic checkpoint and restart without user intervention, a holistic framework (ACR) for automatically detecting and recovering from silent data corruptions and a framework called FlipBack that provides targeted protection against silent data corruption with low cost.

Table of Contents

List of Figures

List of Tables

CHAPTER 1

Overview

Reliability is one of the most important characteristics expected from computer systems. Effectively utilizing hardware resources is difficult if faults are frequently encountered. In high performance computing (HPC), where the system comprises hundreds of thousands of components, even making sure that all components are functional at all the times is a daunting task. This is because although the mean time to failure (MTTF) of individual components is high, the aggregate MTTF of the full system is low due to the large number of system components. As a result, hardware vendors and software designers have made significant effort to enable a smooth user experience even in the presence of fail-stop failures [1].

Besides fail-stop failures, the major deterrent to achieving reliability is presence of soft errors. A soft error typically results from transient faults caused by electronic noise or high-energy particle strikes. For example, silent data corruption (SDC) may occur due to transient bit-flips [2]. SDCs are becoming more prevalent in HPC as lower power chips with smaller feature sizes are being developed. Research has shown that there exists a strong correlation between an increase in soft error rate and a decrease in device sizes and operating voltages [3]. Even recent and current systems face a modest number of soft errors. For example, ASC Q system at Los Alamos National Laboratory experienced on average 26.1 CPU failures induced by cosmic rays per week [4]. On Jaguar, double-bit errors were observed once every 24 hours in Jaguar's 360 TB memory [5].

In Figure 1.1, we model the system utilization and vulnerability to fail-stop failures and silent data corruptions with different sizes of machines as well as different SDC rates using a heatmap. The underlying assumption used in Figure 1.1 is that the application runs on the whole machine and takes 120 hours to complete. The mean time between fail-stop failures for each socket is assumed to be 50 years. Since the application is running without any fault tolerance support, it needs to restart from the beginning of the execution whenever a fail-

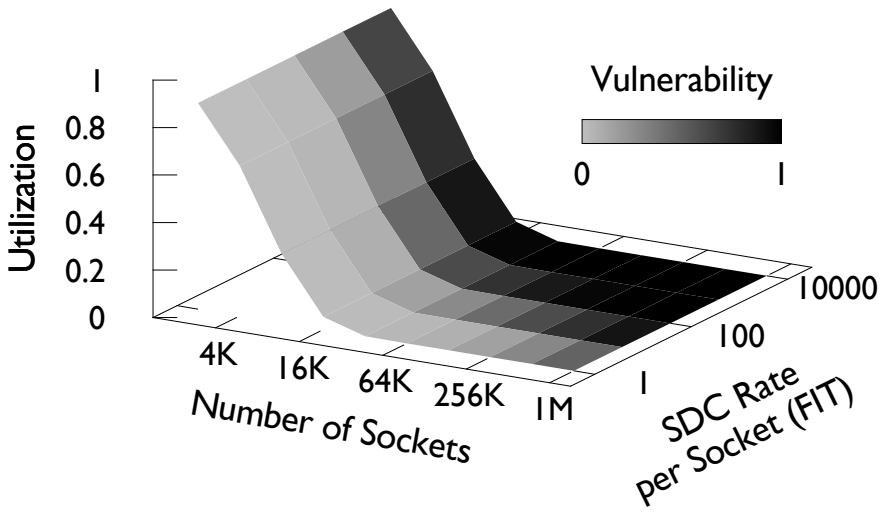


Figure 1.1: Overall system utilization and vulnerability to SDC without fault-tolerance protection.

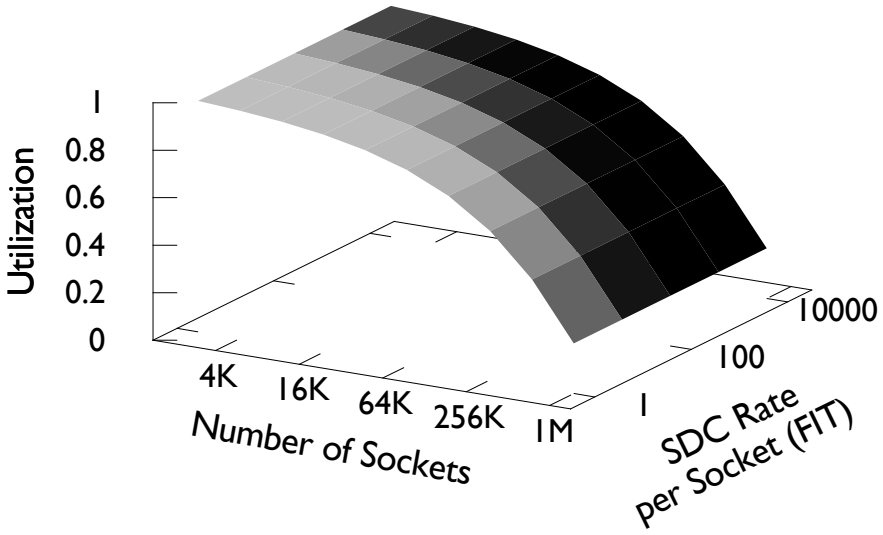


Figure 1.2: Overall system utilization and vulnerability to SDC with slow checkpointing to file system.

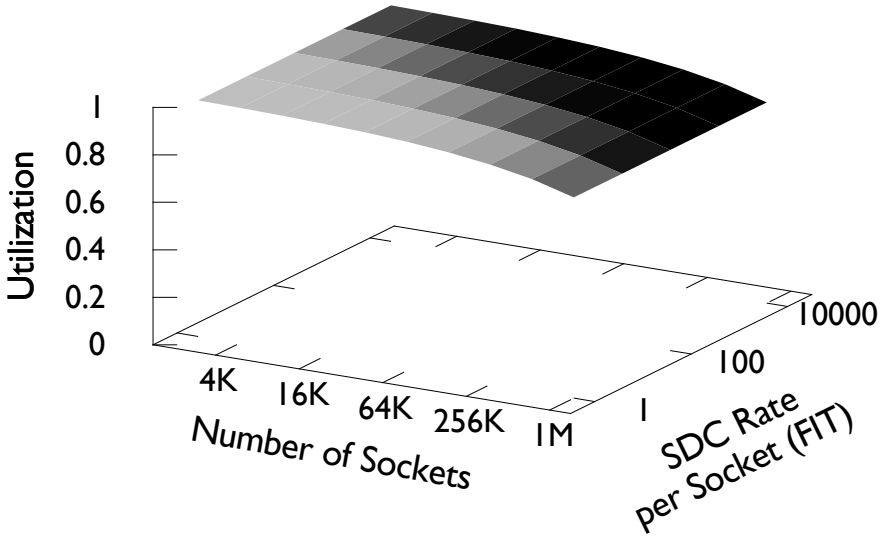


Figure 1.3: Overall system utilization and vulnerability to SDC with faster checkpointing.

stop failure occurs. Hence, as can be seen in Figure 1.1, the system utilization decreases very fast as the socket count increases. This is because the increase in socket count leads to high probability of fail-stop failures. Another trend that can be observed from Figure 1.1 is that as the socket count and the SDC rate increase, the vulnerability shown by the application becomes close to one (represented by darker color). This means that the probability of getting incorrect result is higher for these scenarios.

With all the aforementioned uncertainties, it becomes harder to ensure smooth execution of high performance computing applications. The common practice to protect applications from failures in HPC community is to periodically store the application state as checkpoints in a relatively reliable file system. However, writing the entire application state to the file system is time-consuming. As the failure rate increases, the application may easily end up spending more time on checkpoint and restart rather than performing useful computation. Also this approach fails to protect applications from silent data corruptions. In Figure 1.2, we model the system utilization and vulnerability when application performs periodical checkpoint to the file system. As can be seen from Figure 1.2, this fault tolerance strategy helps improve the system utilization compared to the case when no fault tolerance strategy is used. However, as the number of socket count increases, the utilization decreases significantly. The color pattern in Figure 1.2 is the same with Figure 1.1, which indicates that checkpointing to file system cannot help protect applications from silent data corruptions.

Thesis statement: We explore the hypothesis that runtime system and compiler techniques

can be leveraged to mitigate a significant fraction of failures automatically with low overhead. We believe that scientists, especially users of HPC applications, should mainly focus on the scientific experiments being conducted and leave it to the runtime system to mitigate failures. Specifically, we strive to solve the following research challenges in this thesis:

- **Cost:** How do we reduce the overhead of runtime system based failure mitigation strategies so that the performance and progress rate of applications are high with and without the occurrence of failures?
- **Automation:** It is equally important to design runtime system based failure mitigation strategies that can transparently handle failures without user intervention. For example, features such as automatic recovery and automatic checkpoint decision should relieve users from performing tasks required to ensure correct execution of applications.
- **Coverage:** Runtime system based failure mitigation strategies should not only protect applications from fail-stop failures but should also handle silent data corruptions. This thesis explores various ways to increase the coverage of such strategies.

1.1 Thesis Organization

Chapter 2 describes the background work related to this thesis. It includes explanation of relevant key terminologies, analysis of failure logs obtained from a top supercomputer, overview of the parallel runtime system that we have chosen to demonstrate various failure mitigation strategies with, analysis of existing fault tolerance approaches, and description of memory bottleneck that challenges the practical use of fault tolerance strategies as well as normal application execution.

Chapter 3 presents a semi-blocking checkpointing algorithm which can protect applications from fail-stop failures with minimal overhead. With the support of parallel runtime system, the presented technique can automatically adapt to the application communication in order to schedule checkpoint communication at the right time. The proposed fault tolerance strategy not only minimizes the overheads, but also targets to relieve the memory pressure. Figure 1.3 presents the scenario when the time to perform one checkpoint is reduced. As can be seen from Figure 1.3, with faster checkpoint the system utilization is above 80% even when there are more than 1 million sockets.

Chapter 4 is focused on a generic approach to protect applications from silent data corruptions so that the system vulnerability shown in Figure 1.1, Figure 1.2 and Figure 1.3 is

reduced. Use of a parallel runtime system eases the development of this failure mitigation strategy by allowing multiple instances of the application to run at the same time. These multiple instances serve as replica for each other and provide a baseline to detect silent data corruptions. In Chapter 4, we also discuss how runtime system can automatically adjust the control knobs provided by the fault tolerance strategy on demand in pursuit of higher application progress rate.

Chapter 5 proposes FlipBack which provides automatic targeted protection against silent data corruption based on the characteristics of high performance applications. FlipBack guarantees high coverage at low cost by making use of both compiler based techniques and parallel runtime system based techniques.

Chapter 6 is dedicated to relieving memory pressure, which is yet another challenge that prevents correct execution of parallel applications. We propose Charm-HMC, a parallel runtime system that efficiently utilizes non volatile memory as an extension of the existing memory hierarchy so that applications that require more memory than is available in DRAM can run efficiently. Although this is a tenuous connection to resilience, the main theme of the thesis, the research presented in this chapter leverages the techniques developed for resilience in earlier chapters. We show that Charm-HMC is able to transparently and efficiently enable out-of-core execution of parallel applications. Using information on data dependences and data privileges, Charm-HMC successfully overlaps I/O transactions with computation, and automatically balances the load among different cores.

Finally in **Chapter 7**, we summarize the failure mitigation techniques presented in this thesis and suggest potential future directions.

CHAPTER 2

Background

In this chapter, we first explain basic terms that are used throughout this thesis. Next, we present a study of system logs from Titan, the top supercomputer in the world in 2012, and show that failures are no longer rare events in today's HPC system. Given that, automated fault tolerance strategies with good coverage at low overhead are necessary. To enable such strategies, we introduce a parallel runtime system and describe how different failure mitigation strategies can be easily built in it. Finally, we describe the challenges posed by memory pressure that may affect the normal executions of high performance applications and feasibility of various fault tolerance strategies.

2.1 Terminology

Fault is a defect within a system, which is the root cause of errors. Examples of faults are: software bugs, hardware unavailability, memory bit flip/stuck, omission or commission fault in data transfer, etc. Faults can be either active or inactive. For example, if a function that has a software bug is never invoked, it is an inactive fault. In contrast, if a bit flip changes the value of critical data and causes the program to crash, it is an active fault.

Error is the consequence of fault. By definition, error is a deviation from the required operation of system or subsystem. Fault may manifest itself as an error long time after it occurs. For example, if a bit flip happens, error only occurs when CPU accesses the data that has been modified.

Failure is triggered by error. It occurs when the system fails to perform the required function. For example, a system outage may happen if there is hardware voltage fault,

or a program's output may be incorrect if a memory bit flips or gets stuck. Such system outage or incorrect program output is failure. Another example that describes the fault-error-failure relationship is presented in a study of BlueGene/P logs [6]. In that study, since a power module cannot maintain proper outputs (fault), BPC clock chip shuts itself down after receiving the signal (error). As a result, a job that is running on that chip is affected (failure).

Hard Error causes fail-stop failures. Such errors are not transient. To fix them, either the system needs to be rebooted or seeking help from manufacturer to repair the faulty components. For example, the root causes of hard errors can be misbehavior of fans, faulty cabinet power, and voltage faults. In recent times, the HPC community has seen a stellar growth in the capability of high-end systems. Machines such as IBM Blue Gene/Q and Cray XK7 have a peak performance that reaches to the tens of petaflops. Since the frequency of CPUs has been limited in recent years, these systems have increased processing power by increasing the number of cores. However, the increase in the number of system components required to build these machines has had a negative impact on the reliability of the system as a whole. Although the mean time between failure (MTBF) of individual components is high, the aggregate MTBF of the full system is low due to the large number of system components. If these trends persist, large systems in the near future may experience hard failures very frequently [7, 8].

Usually there are two approaches to mitigate such errors. One approach is to predict when and where the next hard error will happen based on past occurrence or knowledge about the hardware [9]. With accurate prediction mechanism set up, proactive action can be taken. For example, a parallel runtime system can shrink the number of processors the parallel application is running on in order to avoid the faulty hardware. The other approach is to take proper action right after the occurrence of the failure. For example, a node may go down due to a hard error. As a result, all the computation conducted on that node will be lost. After detecting the loss of node, a parallel runtime system can automatically perform local or global restart of the application to mitigate the loss as much as possible. In the scope of this thesis, we focus on the second approach to mitigate hard errors. We strive to reduce the performance overhead and memory overhead of the fault tolerance strategies.

Soft Error is caused by bit-flip fault. Soft error is transient and thus it cannot be reproduced. Soft errors are becoming more prevalent as feature sizes decrease along with supply voltages to chips [2]. As systems become more energy-efficient, chips may approach near-threshold operation to reduce the power required. Previous studies have shown a strong correlation between the increase in soft error rate and the decrease in device sizes and oper-

ating voltages [3]. This same study suggests that the soft error rate may reach the very high level, to the point that an undetected soft error may occur once per day in a single chip. The most insidious form of soft error is silent data corruption (SDC) [2]. For mission-critical applications, having a guarantee that the data was not silently corrupted may be very important. Even today's systems face a modest amount of soft errors. For example, ASC Q at LANL experienced on average 26.1 radioactivity-induced CPU failures per week [4].

Detection is the first step to mitigate soft errors. Existing approaches to detect soft errors include error correcting code (ECC), redundancy based techniques and algorithm based methods. A common practice to correct bit flip is re-execution: either restarting from the beginning or from last checkpoint. In this thesis we propose various techniques to mitigate soft errors at different levels. Our goal is the same as it is for mitigating hard errors: minimizing the performance overhead and memory overhead with minimal user intervention.

2.2 Study of System Logs from Supercomputers

To better understand the failure characteristics in supercomputers, in this section we summarize the findings made by the study of the system logs from Titan [10] and Blue Waters [11].

2.2.1 Titan

Titan is a Cray XK7 supercomputer located at the Oak Ridge Leadership Computing Facility (OLCF). The peak performance of Titan is 27 petaflops, according to the Top500 list [12]. Titan uses a hybrid architecture of CPUs and GPUs. Each node has an AMD 16-core Opteron CPU and an NVIDIA Tesla K20 GPU. There are 560,640 Opteron cores in total. Each node has a total of 32 GB of main memory. Titan features a 3D-torus Gemini interconnect.

A monitor system, *failure database* [13], keeps track of all the incidents on Titan. This database is automatically constructed by a program designed by the system administrators. The program uses SEC (Simple Event Correlator) to enforce correlation rules that insert records in the database after examining multiple anomalous outputs in the system. Each entry in the failure database represents an event that can be tracked to a particular component in the machine. Also each entry includes the identifiers of the jobs that it affects. These events may be related if the root cause of a failure has repercussions on other parts of the system. For example, a hardware malfunctioning may trigger an event on a monitor program as well as a event regarding to the failure in the user code running on that hardware. The

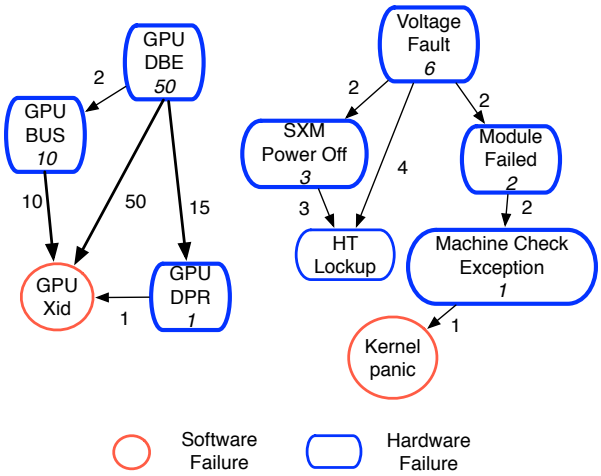


Figure 2.1: Failure propagation

severity of the different events in the failure database varies from warnings to catastrophic incidents. The job scheduler in Titan runs jobs with the assumption that any failure will be *fatal* for the job. Hence, a failure on the hardware assigned to a job will immediately terminate the job execution, regardless of the state of the rest of components assigned to the job.

A failure record in the Titan database includes the identifiers of the jobs affected by the failure, the time when the failure is reported, the category of the failure (hardware or software), the cause of the failure (system or user code), the failure type (a more detailed description of failure characteristics) and the nodes affected by the failure. There are more than $160,000$ entries in the original failure database of Titan for the entire year (2014). The massive amount of failure records is partly due to the redundant information in the database. For example, if a node fails when a job is running, then the node is marked for repair and it will not be assigned to any other jobs until the problem is resolved. However, before proper repair action is taken, the system keeps reporting the failure of that node.

To acquire accurate failure information from logs, we first filter the redundant entries in the failure database. If there are many entries regarding the same type of failure related to a node during the execution of a certain job, we only keep one such entry. Sometimes a failure may affect more than one node. For example, a GPU graphics engine fault may bring down all the related nodes running the same job or a voltage fault may bring down all the 4 nodes on the same blade. As a result, in our analysis if multiple nodes encounter the failure of the same type in the same time frame, we count them as one failure.

Next, we find the root cause of each failure and only keep that entry. For example, a hardware GPU DPR (dynamic page retirement) failure may be followed by a software GPU

Xid error on the same node. After confirming with Titan administrators, in such cases, the hardware failure is often the root cause. Thus we only keep the entry of the hardware failure. Figure 2.1 shows the instances of failure propagation we have observed. The number on an edges indicates the amount of instances that have shown the connection between the two failures. The number under a failure node indicates the number of instances that failure is proved to be the root cause. For example, 50 of the *GPU Xid* failures, 15 of the *GPU DPR* (dynamic page retirement) failures and 2 of the *GPU Bus* (GPU off the bus) failures are caused by *GPU DBE* (double bit error) failures. In 50 cases, *GPU DBE* failure is the root cause. This is because when *GPU DBE* leads to the failures of *GPU DPR* and *GPU Bus*, *GPU Xid* failures also come along. *Voltage Fault* may also have different subsequences, such as *Module Failed*, *SXM Power Off* and *HT Lockup*. Please note that *HT Lockup* and *Kernel panic* failures are almost always caused by other failures, thus we exclude them in our later discussion.

Finally, after careful consideration we ignore failures of certain types. According to Titan administrators, `heartbeat fault` failures are mostly false positive alarms since a network congestion may cause nodes not to respond to the heartbeat within the time limit. Hence all the `heartbeat fault` failure log entries are filtered out. We also filter all the user code related failures such as `out of memory` failures. There are 3884 user failures in total. Among them, 2615 are `out of memory` failures and 1269 are `GPU memory page fault` failures caused by user code.

Failure Category	Failure Type	Count	Percentage
GPU	GPU DBE	51	16.1%
	GPU DPR	66	20.8%
	GPU Bus	11	3.5%
	SXM power off	14	4.4%
	SXM warm temp	2	0.6%
Processor	Machine check exception Bank 0,2,6	31	9.8%
Memory	Machine check exception Bank 4, MCE	120	37.9%
Blade	Voltage fault	12	3.8%
	Module failed	10	3.1%

Table 2.1: Hardware-rooted failures

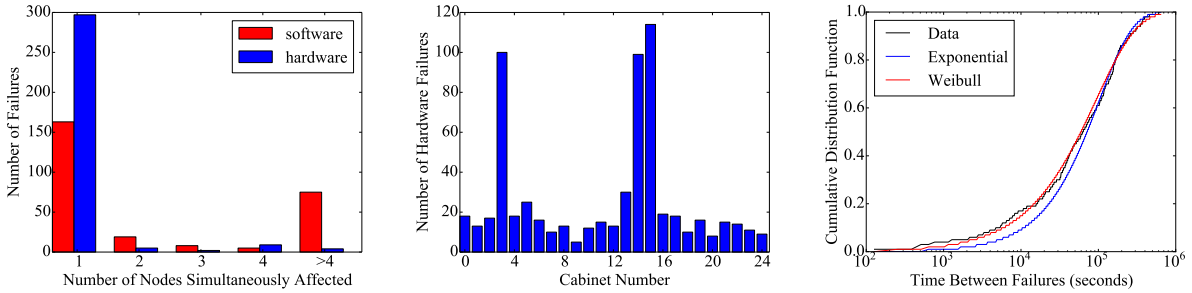
Table 2.1 shows the break down of hardware failures after the filtering process. There are 317 hardware failures in total. *GPU DBE* is GPU double bit error. *GPU DPR* refers to GPU dynamic page retirement. NVIDIA driver decides to retire a page after it has experienced

a single double bit errors or two single bit errors. *GPU Bus* indicates that the GPU has fallen off the bus. *SXM* related failures happen when there are GPU problems. *Voltage fault* and *Module failed* are blade level failures. When such failures happen, it usually brings down all the 4 nodes on the same blade at a time. As shown in Table 2.1, there are 45.4% GPU related hardware failures, 9.8% of failures are processor failures, 37.9% memory related failures and 6.9% blade-level failures.

Failure Category	Failure Type	Count	Percentage
GPU	GPU Xid	267	98.9%
Lustre	LBUG	3	1.1%

Table 2.2: Software-rooted failures

Table 2.2 shows the break down of software failures. Note here that for the Lustre related failures, only client side failures are included in the log we have studied. Thus there may be more Lustre software failures on the server side, which is not in the scope of this work. As can be seen from Table 2.2, GPU related software failures constitute 98.9% of all the software failures reported and there are only 3 Lustre client side software failures out of 270 software failures. We must emphasize that not all file-system software errors are reported to the failure database. Therefore, this proportion may not represent the actual distribution of software failures on Titan. It does, however, shows that GPU-related errors are significant in the system. According to [14], *GPU Xid* indicates a general GPU error, which can be indicative of a hardware problem, an NVIDIA software problem or a user application problem. In our filtering process, when there is a GPU Xid failure along with a hardware failure reported on the same node around the same time, we only keep the entry of the hardware failure. So the causes of the GPU Xid failures shown in Table 2.2 should be either software or user problems assuming all the hardware problems have been caught and recorded.



(a) (b) (c)

Figure 2.2: Failure pattern on Titan during 2014

Figure 2.2a shows the number of nodes affected by one failure. According to Figure 2.2a,

more than 93.7% hardware failures only affect one node. Out of 317 hardware failures in 2014, 9 failures affect 4 nodes. All these 9 failures are blade-level failures, which usually bring down all the 4 nodes on the same blade. As for the 270 software failures, 163 (60.4%) failures of them are single node failures. 29.6% of software failures affect more than 4 nodes.

Figure 2.2b shows the number of hardware failures in each cabinet. Unlike the previous analysis, if there are multiple nodes affected by the same failure, we count the occurrence on each node as one failure to study the spatial correlation. As can be seen in Figure 2.2a, cabinet 3, 14 and 15 have the most number of hardware failures. After a detailed look into the failure logs, we found that `module failed` type of failures brought down the entire cabinet 3, 14 and 15 at different times of the year. Especially right after cabinet 14 and 15 are down, Titan maintenance log points out that there was a LNET fallout due to 2-cabinet warm swap and other reasons.

We have tried to find the best distribution to describe mean time between hardware failures (MTBF) in Figure 2.2c. Both exponential distribution and Weibull distribution with shape parameter 0.82 fit the MTBF of Titan very well. The R-squared value for the goodness of exponential distribution is 0.91 while that value for Weibull distribution is 0.93. Using those distributions as a model for Titan's failures, the MTBF of the system is 22.78 hours.

2.2.2 Blue Waters

Next, we summarize the failure characteristics of Blue Waters based on the more comprehensive log analysis study done by Di Martino et al. [15, 16].

Hardware failures: according to a study performed on 261 days of Blue Waters system logs [15], hardware is not the main problem for system-wide reliability on Blue Waters. Even though hardware caused 52% failures, the repair time for hardware-related failures is only 23% of the total repair time. Overall, there are 1.5 million machine check exceptions on 22640 nodes during the 261 days period. However, the number of uncorrectable errors is only 28 because of the protection mechanism used. Also, 99.3% of hardware failures are contained in a single blade, which is consistent with the findings on Titan.

Software failures: during the period of 261 days, software-related failures lead to 74% system wide outage and contribute to 53% repair hours, which suggests that software failures are critical for the system reliability on Blue Waters. Failure of fail-over is the major cause of software-related system wide outage. Among all the software related failures, large-scale file system is the major source of failures. This is mainly because the mechanisms behind file system fail-over do not scale with the volume of data that needs to be recovered on Blue

Waters. Meanwhile, during recovery, it is difficult to keep the file system state constant.

Effect of system failures on applications: [16] studied 5 million applications that were run in 518 production days of Blue Waters. Among all these applications, 1.5% failed due to system problems and contributed to 9% of production hours. The probability of application failure due to system related issues increases by 20 times when scaling from $10,000$ to $22,000$ XE nodes. 37% of failed applications fail during fail-over operations, which indicates that system level resiliency mechanisms should interact with the workload management system and with application-level resilience to improve overall reliability. For examples, when a file system fail-over leads to applications failures, due to the unavailability of file system, checkpointing to file system will no longer be an effective resiliency scheme. If application-level resilience scheme coordinates with the system knowledge, other resilience scheme that does not rely on file system could be used.

According to [16], the mean node hours between failures (MNBF) of XE applications is higher than XK applications. This is mainly because GPU cores are equipped with less sophisticated error detectors and thus it is difficult to detect errors until they propagate and kill the application.

The chances of interconnect related issues being the root cause of failures increases as the applications scales to higher node counts. For example, for applications running on more than $10,000$ nodes, 66% failures are because of Gemini and LNet related issues. In contrast, for applications running on less than $10,000$ nodes, Gemini and LNet related issues account for only 23% failures.

2.2.3 Lessons learned from system log analysis

- **Automatic failure handling strategies are needed.** As applications in different fields reach new scalability milestones, failures become a norm rather than an exception as can be observed on Titan. The user code cannot assume that the underlying hardware is failure-free. Rollback-recovery strategies, such as checkpoint/restart must be considered as a first option to allow applications run through failures. Also on Blue Waters, among the 5 million applications being studied [16], 1.5% of them fail due to system failures and checkpoint/restart plays an important role in improving application resiliency to system problems. The abstraction behind checkpoint/restart is simple enough to be easily adapted to most programs, and powerful enough to run across architectures. Moreover, for certain types of failures, where the affected component can be isolated from the rest of the system, the job scheduler should avoid

terminating the job execution, and let the runtime system handle the problem. Smart runtime systems are known to be capable of dealing with this scenarios effectively [1]. Such a feature may save the resubmission time spent in the queue. At the same time, runtime systems must develop strategies for running malleable jobs (shrink and expand to use a different number of nodes) and maintain good progress rate despite failures.

- **Design fault-tolerance strategies that tolerate a single-node failure.** Figure 2.2a presents a distribution of the failures according to the number of nodes it affects on Titan. The majority of the incidents only affect a single node. Similarly, on Blue Waters, 99.3% of hardware failures are contained in a single blade. Fault-tolerance strategies should be optimized for the common case and should perhaps sacrifice generality for performance.

2.3 System Model: Charm++

In this thesis, we explore the role of runtime systems in reducing the amount of failures visible to users. The runtime we have chosen to implement various fault tolerance techniques is Charm++. Next, we briefly describe a few key concepts about programming in Charm++ that are relevant to this thesis.

Charm++ is a general purpose parallel programming framework used by many large scale applications [17–20]. It is based on the concept of over-decomposed units (defined by the programmer) whose execution is guided by the availability of data. Charm++ uses an adaptive runtime system (RTS) which is responsible for scheduling of the over-decomposed units and communication among them. In this way, it strives to perform a balanced division of labor among the programmers and the RTS.

Chares: The work and data units in Charm++ are implemented using specially designated C++ classes and objects, called chares. Programmers are encouraged to define different types of chares based on data and computation requirements of their applications. Further, the number of chares is typically much larger than the number of processors (e.g. 8×) so that the RTS can optimize application execution. Figure 2.3 (left) shows an example wherein the programmer's view of a Charm++ application consists of two types of chares with five objects in total (named *A1, A2, A3, B1, B2*). From execution model standpoint, chares are reactive entities, i.e. only when data/work is available for them, the RTS executes them. Chares are free to migrate to any processors via a packing and unpacking framework. With this framework, all the data required for migration can be serialized to or deserialized from

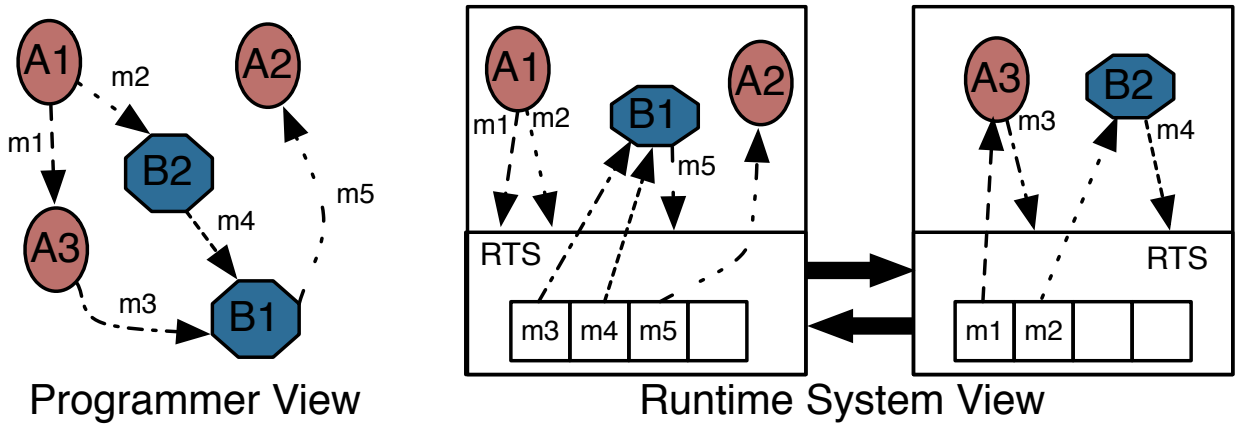


Figure 2.3: Charm++ execution model: applications are written using chares (C++ objects, e.g. A1, A2, B1) that communicate via possibly remote invocation of entry methods (member functions, e.g. m1, m2, m3). The Charm++ RTS mediates both communication of messages and scheduling of computation via entry method execution.

a stream of data. Figure 2.4 shows one example of the *pup* routine. The migratability of chares eases the development of fault tolerance strategies. For example, if a failure is going to occur on one processor, the runtime system can simply migrate all the chares on that processor to avoid the effects of the failure.

```
Chare Data
  int size;
  double temperature[size];

PUP Function(PUPer p)
  p | size;
  PUParray(p, temperature, size);
```

Figure 2.4: PUP function for serialization and migration.

Entry methods: Specially designated member functions, called entry methods, form the basic unit of execution of chares. They also provide the mechanism for communication, defined in terms of destination chares and designated entry methods. For example, in Figure 2.3 (left), when chare *A1* is executed, it send messages to other chares (*A3, B2*) by scheduling execution of member functions (*m1, m2*) on them. These messages are matched to their destination chares, and eventually leads to execution of these chares as either data or work has become available for them.

Charm++ RTS: The RTS is responsible for many tasks in Charm++: 1) distributing and redistributing chares to processes on which they are executed, 2) mediating communication

defined by programmers among chares, 3) executing entry methods on chares. On the right side of Figure 2.3, the RTS view of the program is shown for two processes. Chares *A1, A2, B1* are mapped to the first process, while chares *A3, B2* are on the second process. When chare *A1* requests invocation of entry method *m1* on chare *A3*, the RTS records the request and returns the control to chare *A1* to complete the execution of its entry method. Then, the RTS finds the processor on which chare *A3* resides and forwards the invocation request to that process. At the destination process, the request is enqueued into a message queue and leads to execution of the method *m1* on chare *A3* when the RTS find it suitable to do so.

2.4 Existing Fault Tolerance Strategies in Charm++ [1]

Three fault tolerance strategies are already available in Charm++: checkpoint/restart, message logging and proactive evacuation. In this section, we explain in detail how each strategy works and how they impact the execution of HPC applications. The primary limitation of existing approaches is that they only consider hard errors. To overcome this limitation, we explore the role of runtime system in protecting applications from soft errors in this thesis. We also attempt to address the performance and memory related overheads of the existing strategy.

2.4.1 Checkpoint/Restart

In *reactive* fault tolerance, the objective is to *overcome* a failure by providing a recovery mechanism after one component fails. We assume that the systems have a failure-detection mechanism. Typically, once a failure is detected, the runtime system starts a recovery protocol to bring the application execution back on track and make forward progress. With runtime system support, recovery can be *automatic*. That means, the user does not need to be aware that a failure has happened, although application execution may take slightly longer to complete.

Checkpoint/restart is arguably the most popular technique in HPC to provide fault tolerance. It is simple and effective in situations where failures are relatively rare. The fundamental principle of checkpoint/restart is to periodically save the state of the whole system. If a component crashes, the system rolls back to the most recent checkpoint and restarts

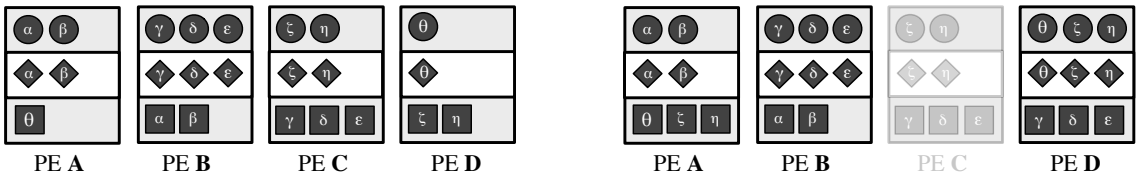
[1] Based on [1]

execution from there. Although the basic working of checkpoint/restart is straightforward, it can have many variants.

The checkpoint of a system can be *coordinated* if different components agree on when to store their state. The Chandy-Lamport algorithm for global snapshots [21] creates a collection of node checkpoints and in-flight messages that constitute a global checkpoint. An *uncoordinated* protocol, on the other hand, allows nodes to checkpoint at their own discretion. However, all node-checkpoints do not collectively make a consistent global checkpoint. If messages are not stored, then this scheme may suffer *cascading rollback*, a pathological situation where the rollback of one node may require the entire system to rollback several checkpoints. Coordinated checkpoint can be *blocking* or *non-blocking*, depending on whether the application has to stop execution during checkpoint, or the checkpoint process executes along with the application. A comparison between these two approaches can be found in [22].

With regards to generating a global checkpoint, the amount of data stored can differ according to what is included in the checkpoint [23]. In a *system-level* checkpoint, the whole state of the machine (including the complete address space of processes, CPU registers, file descriptors) is to be saved. This mechanism makes the application oblivious of the checkpoint. BLCR [24] is a library that implements this abstraction. Alternatively, *application-level* checkpoint makes the application an active participant of the checkpoint process. The user must write a checkpoint function and decide what to store in a checkpoint. That way, the amount of data to checkpoint may be dramatically reduced. Additionally, the knowledge of the programmer is used to insert the checkpoint calls at appropriate places. SCR [25] is a library that implements this method. The migratable-objects model encourages *runtime system based* checkpoint [26], where the runtime system provides an interface for the programmer to write the checkpoint methods. The runtime system may also participate more actively in deciding when to trigger a checkpoint. The migratable-objects model can also provide transparent checkpointing in which user's role in checkpointing is minimized. Note that checkpoint transparency and migratability are orthogonal functionalities.

A global checkpoint of the system, obtained with any of the mechanisms above, used to be stored only in the file system as a common practice in high performance computing. File system is a natural place to store the checkpoint, since it will survive the failure of a node. However, file system bandwidth does not cope with the increasing size of the supercomputers and the data size that needs to be checkpointed. Thus, file system becomes a bottleneck during checkpoint. Various alternatives have been explored to solve this problem. One popular choice is to store the checkpoint in local storage (either main memory, disk or solid-state drive). One such protocol is called *double in-memory* checkpoint/restart [26]. In this method, a processor (PE) stores copies of its checkpoint in its own memory and in the



(a) Double in-memory checkpoint/restart. The checkpoint of an object is stored in two places: the local memory of its host PE and the remote memory of the buddy PE.

(b) Recovery from failure in the migratable-objects model. After PE C fails, PE D takes over C's objects. PE A gets more remote checkpoints.

Figure 2.5: Memory footprint and failure recovery in double in-memory checkpoint/restart. Circles represent objects in application, rhombi are local checkpoints and squares are remote checkpoints.

memory of a *buddy* PE. Figure 2.5a illustrates the memory footprint of this approach. PEs A through D contain several objects each. The system uses a cyclic buddy assignment, where PE B is the buddy of A, C is the buddy of B, and so on. In the worst case, this mechanism triples the memory requirements of the application, but it is able to checkpoint rapidly, scale to large systems and is applicable to a wide range of HPC applications [27].

Double in-memory checkpoint/restart tolerates a failure by using spare PEs to substitute for the failed ones. For example, in Figure 2.5a, if PE C fails, a replacement PE will receive the checkpoint from PE D and the system can continue execution. Migratable object models like Charm++ empower this scheme in several ways. First of all, depicted in Figure 2.5b is the scenario in which spare PEs are not available. In such circumstances, there is no replacement for the failed PE C. The adaptive runtime system solves this situation by redistributing the objects of C to the rest of the PEs in the system. For this particular case, all the objects that were on C are moved to D. The buddy assignment is updated and the checkpoint placement corresponds to this new assignment.

The second way in which migratable objects improves this approach is by offering a load balancing framework in the case of no spare PEs. Once a failure hits the system and PEs are lost as a result, the system can re-balance if a PE ends up with a much higher load than the average. Finally, migratable objects provides the right environment for serialization methods to be written in a simple way. The runtime system naturally handles migration of the objects, because that is an intrinsic characteristic of the model.

An implementation of double in-memory checkpoint methods can be found in the Charm++ system [28]. These protocols also work for MPI applications through the AMPI extension [29]. Customized versions of these protocols are also available for a version of the runtime system specific to systems with multicore nodes [30].

Evaluation: To illustrate the full potential of the migratable-objects model in reducing the checkpoint overhead to a small level, we show the results of checkpoint and restart with two different types of applications. For weak-scaling runs, we use Jacobi3D and for strong-scaling experiments, LeanMD is used. LeanMD simulates particle motion by computing forces among them in an iterative manner. By placing the checkpoint calls at synchronization points between iterations, we manage to checkpoint only the fundamental data: attributes of particles including positions, velocities, etc. All other intermediate data structures are not stored as part of the checkpoint. Because of this, the size of checkpoints decrease drastically. Jacobi3D performs temperature relaxation on a 3D-grid of points using 7-point stencil. Like LeanMD, we only store the necessary data structures of Jacobi3D in the checkpoint. All temporary data structures and messages are not included. The top part of Figure 2.6 presents the checkpoint time for the two applications on Stampede [31]. These results show that the time to checkpoint is very low (measured in milliseconds) and that the checkpoint framework scales well.

The bottom part of Figure 2.6 shows the restart time, which includes time to notify all the active PE about the failure, synchronizing the rollback of all PEs and retrieving the checkpoint of the failed PE. The total restart time is constant for the weak-scaling experiment. The strong-scaling case shows that the restart time is initially dominated by the checkpoint transmission and thus improved performance is obtained. However, on higher process counts, the synchronization cost is the major factor.

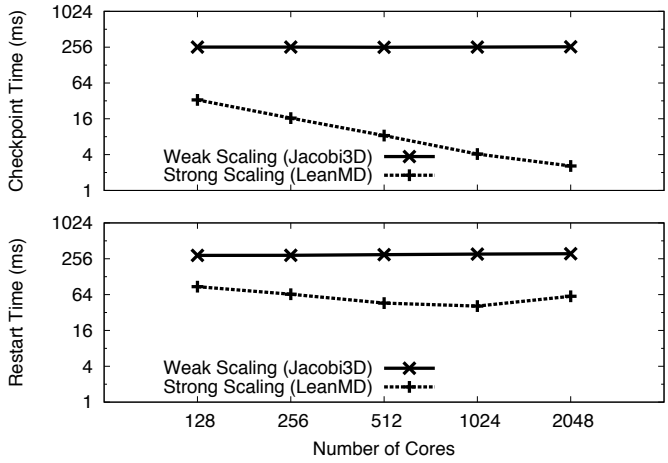


Figure 2.6: Fast checkpoint and restart time.

2.4.2 Message Logging

Although checkpoint/restart is a very popular alternative in HPC to provide fault tolerance, it embodies a fundamental disadvantage. A *global* rollback is needed for recovery, i.e. all the processors have to roll back to the latest checkpoint in case of failures. That downside becomes critical in an extreme-scale system; millions of PEs would have to roll back if one of them fails, resulting in a massive waste of time and energy.

Message logging is a technique that avoids global rollback by saving the messages an application sends and only rolls back the failed processor. It then requires only a *local* rollback and saves energy by having the rest of the system idle or making progress on their own [32]. It may save time too, because messages have no delay or contention during recovery. Additionally, it allows the checkpoint to be either coordinated or uncoordinated. In case of a failure of a particular processor A, all other processors that have stored messages sent to A will re-send those messages upon its failure. To catch up with the rest of the system, PE A will sort the re-sent messages and process them upon receiving all the messages. To guarantee a correct recovery, message logging requires storing information about non-deterministic events. Message reception is, in general, non-deterministic. Thus, every time a non-deterministic event occurs, a *determinant* is generated. A determinant will contain all the information required to ensure recovery reaches a consistent global state. This mechanism is based on the *piece-wise deterministic assumption* (PWD) [33], which states that logging determinants is enough to guarantee a consistent recovery. For example, a determinant could be formed by the tuple $\langle sender, receiver, ssn, rsn \rangle$. Both *sender* and *receiver* represent identification of the chares involved in communication. The *send sequence number* (ssn) is a unique identifier for each message, assigned by the sender chare. The receiver chare will generate a *receive sequence number* (rsn) upon reception of the message. The rsn totally orders the reception of the message and provides a strict sequence in which messages have to be processed during recovery. There are several message-logging protocols [34] that differ in the way they handle determinants. *Causal* message-logging makes the determinants travel with the messages that causally depend on them. More specifically, determinants are piggybacked on application messages until they are safely stored. A specific protocol in this family, called *simple causal* message-logging [35] has demonstrated scalability and low overhead. Strategies to decrease the memory overhead of the message log can be found elsewhere [36].

Parallel Recovery A migratable-objects model like Charm++ provides a fundamental advantage for message logging. A key observation, concerning many HPC applications, is that most codes are tightly coupled and if one PE fails, the rest of the system will remain

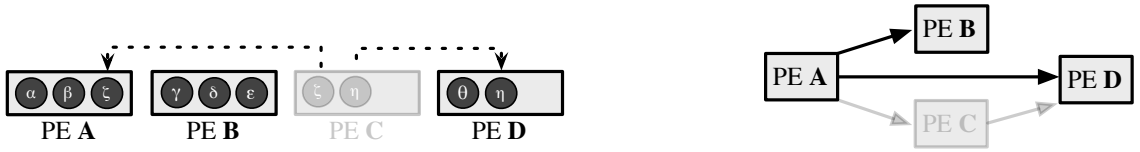
idle until the failed PE catches up with the execution of the application. Instead of waiting idle, surviving PEs can help accelerate recovery by receiving objects from the failed PE and perform what we call *parallel recovery* [37]. Objects living on a failed PE are distributed among other PEs for a speedup in recovery. Parallel recovery empowers message logging by increasing the progress rate during recovery. If failures are common, parallel recovery is able to recover faster and make progress even in the case where the MTBF is smaller than the checkpoint period.

2.4.3 Proactive Evacuation

Proactive evacuation *avoids* failures by migrating the objects from processors that are predicted to fail soon. This mechanism assumes that there is an agent in the system that predicts failures. Although failure prediction is a hard problem in HPC, there are many situations where measurements from different sensors can point to an impending failure [38–40]. If that is the case, the runtime system can receive a signal and proactively move away all the objects from the processor that is expected to fail. Other types of failures may not be predictable, but there is nothing that prevents combining a proactive approach with a *reactive* method such as checkpoint/restart and message logging strategy.

It is easy to see how the migratable-objects model makes a proactive approach for fault tolerance more effective. All that is needed for the *evacuation* of a processor is already available in the model. The objects can be naturally migrated from their current PE to other safer locations and the system should be flexible enough to cope with the update of data structures for a correct execution. Figure 2.7 shows two basic functionalities that a proactive fault tolerance approach should have: migration of tasks and reconstruction of data structures. In Figure 2.7a, the evacuation of PE C is illustrated. After the system receives the alarm of an impending fault in PE C, object ζ is moved to PE A and object η is moved to PE D. These migrations are naturally implemented in the migratable-objects model. Moreover, having multiple objects in one PE is not a problem, given the over-decomposition property of the model.

Figure 2.7b presents the modification of the spanning tree for collective communication operations. Initially, PE A is the root of the spanning tree with children PE B and PE C. PE D is a child of PE C. Once PE C is evacuated, the runtime system reconstructs this spanning tree by making the necessary adjustments so that collective operations can still be performed normally. The re-arrangement of the spanning tree only affects the parent PE A and children PE D of the evacuated PE C. PE C first sends tree modification message to



(a) Evacuation of a PE. All objects living on that PE are migrated upon the reception of an impending-fault signal.

(b) Spanning tree reconstruction. As soon as a PE is removed from the system, the spanning tree for collective communication operations is rearranged.

Figure 2.7: Operations in proactive fault tolerance.

its parent (PE A) and children (PE D). After receiving the message, parent and children store the changes but do not modify the current tree until all the outstanding collective communication operations are finished.

The major challenge of proactive fault tolerance is to keep the communication mechanism effective in the process of migration. The evacuation of a PE happens asynchronously with the execution of the program. The application does not stop to wait until a PE is evacuated. The runtime system must ensure that point-to-point communication works correctly during migration of objects. Scalable approaches for a correct message delivery in the face of asynchronous migration of objects can be found elsewhere [41]. Herein, we will describe what problems may arise and what data structures should be updated.

In Charm++ the mapping of migratable objects (chares) to PEs changes dynamically. The system assigns each object to a *home* PE, which always knows where the object is currently on. This data structure is called the *object-to-home* mapping. However, an object may not necessarily reside on the home PE, but on a different *host* PE. For instance, imagine an object η whose home PE is B, but whose host PE is C. If a message from PE A targets η and PE A does not know where η resides, it will send the message to B, the home PE of η. PE B knows that η lives on PE C, so it will forward the message to PE C. Additionally, PE B will send a control message to PE A to update its routing table. The next time PE A sends a message to η, the message will be directly targeted to PE C. When η migrates from PE C to PE D, the updated host information will be sent to its *home* PE B. Hence, during migration when PE A fails to send message to PE C for communication with η, PE A can always get the most recent location of η from PE B to reset the communication channel.

As we mentioned earlier, a proactive fault tolerance approach can be combined with other fault tolerance strategies and even with a load balancing framework. The migratable-objects model has the capability to rearrange the objects among the PEs to improve performance and speed up the application. It is natural to assume that a load imbalance will arise as a result of an evacuation. So, an equally natural decision is to run a load balancer after the

evacuation.

An implementation of the ideas described in this section is available in the Charm++ runtime system [28]. The utility of this approach for MPI applications has been demonstrated [42]. The implementation of AMPI allows the runtime system to migrate AMPI threads even when messages are in flight. This means a thread may have multiple outstanding MPI requests when it is migrated. If a thread migrates from PE C to PE D, the queue of requests is also packed on PE C and sent to PE D. On PE D, the queue is unpacked and the AMPI thread restarts waiting on the queued requests. However, almost all the outstanding send and receive requests are associated with a user-allocated buffer where the received data should be placed. Packing and moving the buffers would cause the buffers have different addresses on the destination PE. One way to solve this problem is by using the *isomalloc* technique proposed in PM^2 system [43]. This technique reserves a unique range of virtual address space for each thread. That way, there is no threat of memory violations after migration.

Evaluation: One of the most important features of an effective proactive fault tolerance approach is to provide a quick response mechanism. We investigated how rapidly a PE is evacuated by running Jacobi3D with 2,048 cores on Stampede. Table 2.3 presents the evacuation time when the data size per core ranges from 16 to 512 MB. The total time to migrate away all the objects on a PE can be measured in milliseconds and linearly depends on the data size. There are two datasets in the table, representing two important events in evacuation. The *local confirmation* stands for the moment when the failing PE has released all the objects. The *remote* confirmation represents the time when the failing PE has received a confirmation from all the destinations of the objects. A remote confirmation is expected to increase the local confirmation by a roundtrip through the network and the processing of the objects, which can be seen as the constant difference between the two datasets. The real evacuation time lays somewhere between the two, and it is constant regardless of the system size.

Data Size (MB)	16	32	64	128	256	512
Local ACK (ms)	11	26	50	105	206	428
Remote ACK (ms)	530	542	582	647	761	1054

Table 2.3: Fast evacuation time with different data sizes.

The ideal complement to a fast evacuation mechanism is a load balancing framework. Once a PE is evacuated, the additional objects assigned to the receiving PEs may cause

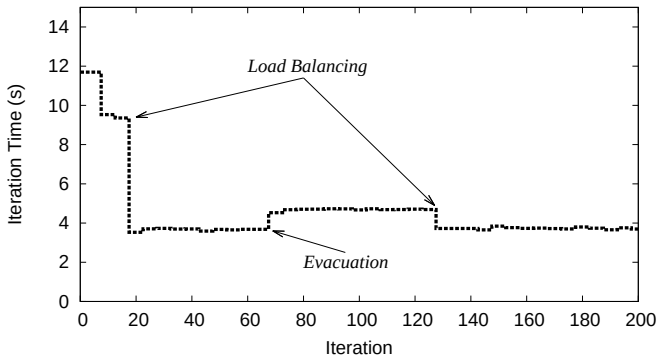


Figure 2.8: Effect of evacuation and load balancing on performance.

load imbalance. To even out the load in the collection of PEs, a load balancer looks for a redistribution of the objects to decrease the load excess on any PE. Figure 2.8 shows the interaction of evacuation and load balancing in NPB-BT multi-zone with 256 cores on Ranger [44]. This benchmark has an initial load imbalance that is later solved by calling a load balancer right before iteration 20. The effect of using a load balancer is significant. The average iteration time is drastically reduced, providing a speedup of 2.65. Then, at iteration 70 the system receives a warning of an impending failure and evacuates a PE. That creates a load imbalance in the system, which increases the iteration time by 22%. Finally, that loss in performance is solved by applying the load balancer once again and bringing down the iteration time to a level similar to the one before the evacuation (little over 1% overhead).

2.5 Memory Bottleneck

Double in-memory checkpointing succeeds in reducing the time to complete the checkpoint. However, the memory consumption increases tremendously by up to three times since at any time during execution: the application data, local checkpoint as well as remote checkpoint are all required to be present in memory. This issue becomes more prominent during checkpointing as the previous local checkpoint and remote checkpoint cannot be deleted until the current checkpoint completes. This high memory requirement may not be a problem for computation intensive applications like molecular dynamics. However, for data intensive applications such as graph processing, the high memory consumption of double in-memory checkpointing may prevent it from being widely adopted. As a result, design of fault tolerance strategies needs to take the memory limitation into consideration.

What makes the situation worse is that as the compute capacity of nodes continue to increase, DRAM is becoming a precious resource in today's supercomputer. Figure 2.9

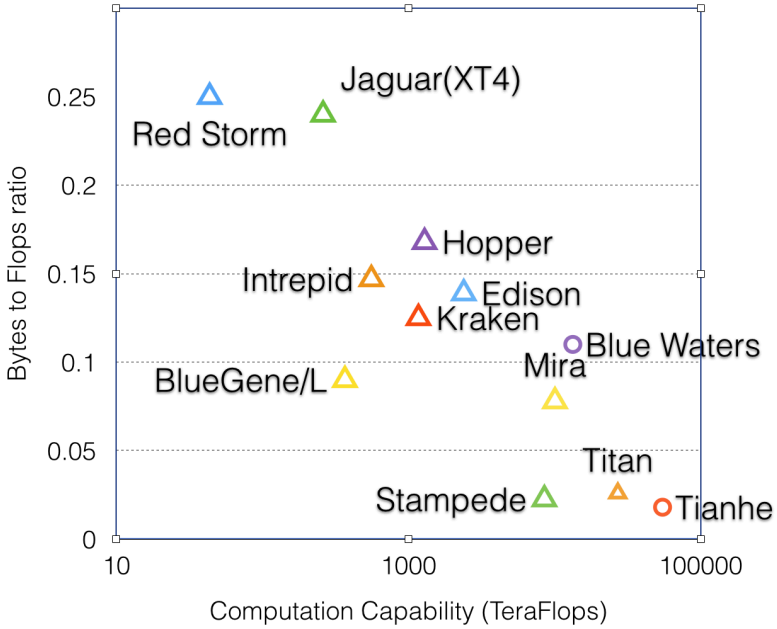


Figure 2.9: Trend shown by memory-to-flops ratio for HPC systems

compares the memory-to-flops ratio for a few supercomputers that have been widely used in recent time. The x-axis in this figure is the total computation capability of a given system while the y-axis represents the ratio of the memory capacity to the computation capability. The trend is easy to notice in Figure 2.9: as the capabilities of machines have increased, the memory-to-flops ratio has gone down from 0.25 to 0.03. Summit, the next generation supercomputer at Oak Ridge National Laboratory (ORNL) [45], is expected to have $3,400$ nodes, each with 44 TFlops computing power and 512 GB DRAM. For Summit, the memory-to-flops ratio is going to drop further to 0.01. In many cases, scientific applications are forced to run on high node counts due to limited available memory per node [46], [47]. As a result, applications face the challenge of increased communication cost, higher probability of failure, and consume more power.

Meanwhile, flash memory technique is being widely adopted in HPC systems. Comet, a cluster at SDSC, already has 250GB solid state disk (SSD), which is a type of non-volatile computer storage, available on each node. In comparison to hard drives, flash memory does not have mechanical limitations. Hence, the latency of SSD is several orders magnitude lower than that of spinning disks. As a result, SSD is more attractive when considering speed, noise, power consumption and reliability compared with traditional hard drives. In comparison to DRAM, SSD is much cheaper and consumes less power. The bandwidth of

SSD is limited by the bandwidth of SATA interface which is about 6Gb/s. NVM Express (NVMe), a logical device interface specification for accessing non-volatile storage attached via PCIe bus, exploits the parallelism of flash based storage devices and thus improves its bandwidth. With NVMe protocol SSD with 10GB/s bandwidth has been announced by Seagate. That being said, storage based on flash technique has its limitations as well. The first issues is block erasure, meaning that erase operations happen at the unit of block and thus increases rewrite overhead. The second issue is memory wear. By design, the number of write operations flash based storage can endure is limited in its lifetime. In this thesis, we take those limitations into consideration while exploring the use of flash memory techniques in designing fault tolerance strategies as well as ensuring the successful execution of data intensive applications.

CHAPTER 3

Semi-Blocking Checkpointing [1]

Current supercomputers consist of thousands of parts, from processor chips to routers and disks. Even when each individual component may be highly resilient, the net result of clustering too many parts into a single machine is an alarmingly low mean-time-between-failures (MTBF) of the system itself. Recent studies show supercomputers have a MTBF between 6 hours and several days [25,49]. However, predictions for Exascale forecast MTBF values from 2 to 60 minutes [8,25]. It will be hard for applications to make any progress in such circumstances without incorporating a fault tolerance mechanism.

In the HPC community, checkpoint/restart is the most popular fault tolerance technique. Its success comes from its simplicity: an application periodically saves its state and if one failure brings part of the system down, the application rolls back to the most recent checkpoint and restarts from there. Storing a global checkpoint in the file system (such as Lustre) is expensive and can be a bottleneck. One promising alternative is to use local storage (memory, SSD, local disks) [25,26,49]. During checkpoint, the application usually stops the execution until the checkpoint is safely stored, using what is called the *blocking* algorithm [22].

One drawback of the blocking algorithm is that it stalls the application while the checkpoint phase finishes. This delay accounts for most of the overhead of checkpointing. It is estimated that total checkpoint overhead can be more than 10% of the total execution time [49]. In popular implementations of the blocking algorithm, each node an application is running on must send its checkpoint to another node across the network [25,26]. At this point, the network usually gets saturated with the large amount of data it must transport before the application can make any progress. Further, a congested network transports data at a rate lower than the peak bandwidth may allow.

Recent studies [50, 51] estimate that the annual increase in memory size and network

[1] Based on [48]

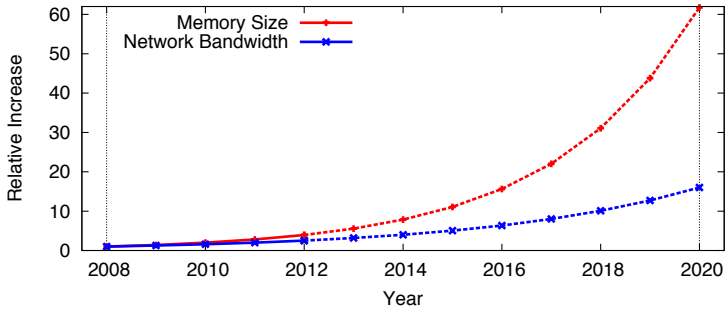


Figure 3.1: Disparity between network bandwidth and memory size.

bandwidth is 41% and 26%, respectively. Figure 3.1 shows the trends in both memory size and network bandwidth for the period between 2008 and 2020. The disparity between the two curves aggravates the problem of a saturated network during checkpoint.

In this chapter, we propose a solution for this problem. The key contributions presented in this chapter are as follows:

- We introduce, in Section 3.3, a semi-blocking checkpoint protocol that hides most of the checkpoint overhead. We also present a model to compute the optimal checkpoint interval for this protocol, which allows us to predict the benefit of our protocol for scenarios with different failure rates.

- We present an implementation of the semi-blocking checkpoint protocol in Section 3.4. It is based on the Charm++ runtime system and contains techniques to adaptively overlap checkpoint with application execution based on the communication and computation characteristic of the applications.

- We demonstrate that semi-blocking checkpoint protocol can hide the checkpoint overhead with real-world applications and improve the performance by up to 22% compared to using blocking checkpoint protocol.

3.1 Related Work

There are two main checkpointing methods in HPC: uncoordinated checkpointing and coordinated checkpointing. In uncoordinated checkpointing, each process independently saves its state. The benefit is that a checkpoint can take place when it is most convenient and thus no synchronization is required to initiate checkpointing. However, uncoordinated checkpointing is susceptible to rollback propagation, the domino effect which could cause systems to rollback to the beginning of the computation, resulting in the waste of a large amount of useful

work. Guermouche et al. [52] proposed an uncoordinated checkpointing without domino effect by logging useful application messages, which is applicable to send-deterministic MPI applications.

Coordinated checkpointing requires processes to coordinate their checkpoints in order to form a consistent global state. Coordinated checkpointing simplifies recovering from failures because it does not suffer from rollback propagations. BLCR [53] implements kernel-level checkpointing, but incurs excessive overhead for application at production level. Some multi-level approaches have been proposed recently to deal with failures at different frequencies of occurrence. FTI [49] is a multi-level coordinated checkpoint scheme using topology-aware RS encoding with about 8% checkpoint overhead. Moody et al. [25] propose a multi-level checkpoint scheme that is able to store the checkpoint in different places. Each place represents a different level and uses a Markov probability model to decide the checkpoint frequency of each level.

Kai Li et al. implemented concurrent checkpointing [54] for shared-memory multiprocessors using a forked process and buffers to overlap the writing of checkpoints to disk with the copying of checkpoints from memory. Our approach differentiates from theirs in that we provide techniques to reduce the interference of checkpoint for distributed memory clusters. Dong et al. leverage PCRAM [51] for checkpointing and propose the hybrid local/global checkpointing mechanism. Their approach can be incorporated with the semi-blocking algorithm by relaxing the stall of computation when taking global checkpoint. Vaidya et al. [55] proposed staggered consistent checkpointing to relieve the contention of writing checkpoint to one or a few stable storage devices while in our approach checkpoints are distributed among all the processors. Ouyang et al. [56] enhance the performance of checkpointing to SSD with aggregation of checkpoint buffer and staging IO. Our approach is different from theirs in that checkpoints are distributed among the SSD disks rather than stored in a central checkpoint server.

3.2 Background

Checkpoint/restart is widely used to provide fault tolerance in large-scale systems. In this scheme, each node saves its state to form a global checkpoint. If there is a failure on one node, all other nodes will rollback and restart from the previous checkpoint. This section presents two different algorithms to obtain a global coordinated checkpoint. These two methods are usually referred as *non-blocking* and *blocking*. We list the advantages and disadvantages of each. The traditional method to compute the frequency of checkpoints is also covered at the

end of the section.

One way to obtain a global checkpoint of an application is to get a global snapshot of different tasks that compose the application. A popular alternative is Chandy-Lamport algorithm [21], a non-blocking algorithm. This protocol does not require a global synchronization point in the application for the tasks to trigger a checkpoint. Instead, it works using a *checkpoint scheduler* to initiate a checkpoint by sending *marker messages* to every task. After receiving the marker message, a task stores its local state and sends a marker to every other task. Messages received through a channel after the local checkpoint has been taken but before having received the marker from the sender must be recorded. This non-blocking algorithm is totally asynchronous and runs in conjunction with the application. However, since it needs to store the in-flight messages as part of the checkpoint, it has a higher memory footprint and a non-trivial implementation [22].

The other way to implement global checkpointing is by using a blocking algorithm. In this case, once the checkpoint starts, the application must stop making progress until the checkpoint has finished. This method is totally synchronous. Usually, applications use global synchronization points to trigger the checkpoint in each task. This method does not require storing messages as part of the checkpoint, because the checkpoint is triggered at a synchronization point where there are no in-flight messages. Finding these synchronization points is not difficult in most scientific computing applications. Besides, these points may be carefully chosen by the programmer as those places where the state size of the application is minimal. Some runtime systems, like Charm++ [57] provide a flexible scheme, where programmers decide what to store as part of the checkpoint of a task.

There are multiple options for where to store the checkpoints of the tasks. Traditionally, NFS disks have been the choice. However, more recently, local storage have been used to keep the checkpoints [25, 26, 49]. One of the earliest such schemes in HPC is the double in-memory checkpoint/restart approach [26], where each node X has a *buddy* node Y that will hold the checkpoint of X in main memory. Each node will store its checkpoint in its own memory too. If node X crashes, its buddy Y will provide it with its checkpoint. All other nodes except the crashed node X will get their checkpoint from their own memory.

Since the nodes will checkpoint with a certain frequency, a natural question to ask is how often they need to checkpoint. The overhead of checkpointing the whole application is also related to how frequently the application checkpoints. Daly [58] investigates the optimum checkpoint interval to minimize the application execution time. The total execution time (T) is divided into:

$$T = T_{solve} + T_{dump} + T_{rework} + T_{restart} \tag{3.1}$$

where T_{solve} is the uninterrupted time to solve the problem, T_{dump} is the time to perform the checkpoint, T_{rework} is the time to recover the lost work due to a failure and $T_{restart}$ stands for the time required to resume execution after a failure.

The more frequently an application checkpoints, the more time an application will spend dumping checkpoints. However, the application will experience less rework time when a failure happens. So, there is always a balance between the checkpoint dumping time, mean time to failure and rework time. In Daly's first-order model, the optimum checkpoint interval τ is given by the following formula:

$$\tau = \sqrt{2\delta(M + R)} \tag{3.2}$$

where δ is the checkpoint dumping time, M is the mean time to failure and R is the restart time.

3.3 Semi-Blocking Algorithm to Reduce Checkpoint Overhead

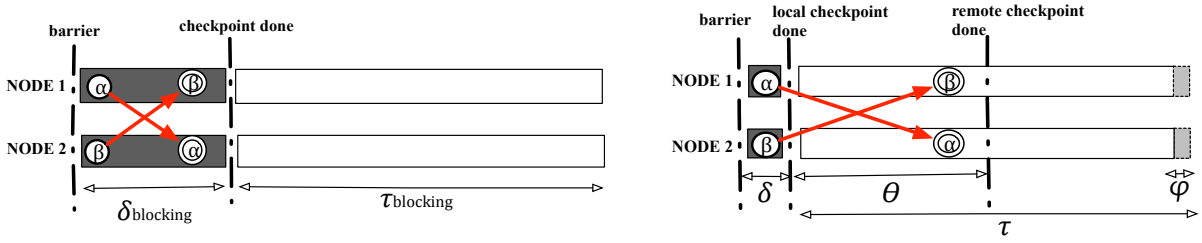
In this section we describe a semi-blocking checkpoint algorithm that strives a balance between the blocking and non-blocking protocols described in the previous section. Like the non-blocking approach, it allows interleaving the checkpoint process with application execution. However, similar to the blocking protocol, it does not need to store in-flight messages as part of the checkpoint. To facilitate the presentation of the algorithm, we summarize the most important parameters in Table 3.1. With the exception of *Benefit*, the unit of all other parameters is seconds.

3.3.1 Algorithm

The semi-blocking algorithm is based on the double in-memory application initiated checkpoint/restart algorithm [26] mentioned in the previous section. Figure 3.2a presents a sketch of the double in-memory blocking algorithm. The diagram presents nodes 1 and 2, where α is the application's data on node 1 and β is the application's data on node 2. Nodes 1 and 2 are buddies of each other. Upon reaching a global synchronization point (labeled *barrier* in the figure), the checkpoint mechanism kicks in. Each node saves its own checkpoint in memory and sends its checkpoint to its buddy. After receiving the checkpoint from the buddy, each node stores it in its main memory. The checkpoint phase has a duration of $\delta_{blocking}$ and consists of two parts: saving their own checkpoint in local memory and saving

Table 3.1: Parameters of checkpoint model

θ	Time to finish remote checkpoint
φ	Average interference of remote checkpoint to application
τ	Checkpoint interval (semi-blocking protocol)
$\tau_{blocking}$	Checkpoint interval (blocking protocol)
δ	Dump time of local checkpoint
$\delta_{blocking}$	Dump time of checkpoint (blocking protocol)
R	Restart time
M	Mean time between failures of the system
T_s	Workload of application
T	Total execution time (semi-blocking protocol)
$T_{blocking}$	Total execution time (blocking protocol)
$Benefit$	Performance improvement of the semi-blocking algorithm over the blocking algorithm



(a) Blocking Checkpoint

(b) Semi-Blocking Checkpoint

Figure 3.2: Checkpoint operation of blocking and semi-blocking algorithms.

their own checkpoint in the remote memory of the buddy. Clearly, it is the second part that consumes most of the time. That part involves almost no computation and, for applications with a non-negligible memory footprint, it may cause network congestion.

Figure 3.2b depicts the basic operations of the semi-blocking checkpoint protocol. The intuition behind the semi-blocking algorithm is to hide the second part of the checkpoint process, by interleaving the transmission of the checkpoints to remote memory with the execution of the application. The diagram shows that once the local checkpoint is saved in memory, the application resumes execution while the checkpoint traffic dribbles through slowly, preferably using the network when the application is not using it. Since the remote checkpoint runs in the background while the application continues executing, this method can substantially reduce the checkpoint overhead. With the semi-blocking algorithm, the checkpoint overhead can be reduced to the cost of just saving a local checkpoint if the checkpointing can perfectly overlap with application execution. This cost stands for a tiny

Local Variables
localCkpt[2]← [NULL,NULL], remoteCkpt[2]← [NULL,NULL]
Upon Global Synchronization
localCkpt[previous]← localCkpt[current], localCkpt[current]← $Data_{checkpoint}$
LocalCheckpointDone()
Upon Local Checkpoint Done
resumeComputation()
BuddyNode.recv($Data_{checkpoint}$)
Upon Receiving $Data_{checkpoint}$ from Buddy Node
remoteCkpt[previous]← remoteCkpt[current]
remoteCkpt[current]← $Data_{checkpoint}$
RemoteCheckpointDone()
Upon Remote Checkpoint Done
delete localCkpt[previous], delete remoteCkpt[previous]

Figure 3.3: Pseudocode of semi-blocking algorithm.

percentage of the total cost of checkpointing in the blocking algorithm. We observe a 22% reduction in the total execution time of iterative scientific applications because of the low cost to do checkpoints, which will be discussed in Section 3.6. Figure 3.3 shows the pseudo-code of the semi-blocking algorithm. In addition to storing the latest checkpoint, each node will also keep the previous checkpoint to ensure that the application can recover from failures during remote checkpointing. We use the names *current* and *previous* to refer to the latest and older checkpoints, respectively.

The benefits of the semi-blocking algorithm are obvious. It can potentially hide the checkpoint overhead by stalling the application only while the local checkpoint completes instead of waiting for the remote checkpoint to finish. Additionally, since time to perform one checkpoint becomes shorter, we can afford to checkpoint more frequently to reduce the amount of work lost in a failure. However, this algorithm has drawbacks that need to be addressed.

First, interleaving the remote checkpoint transmission with the application's messages may create interference and thus decrease the progress rate of the application. We explore an effective technique in Section 3.4 to decompose the checkpoint messages into chunks and inject them into the network at the appropriate time. This mechanism reduces the impact of remote checkpointing on the execution of the application.

Second, if a failure brings down a node during the remote checkpoint phase, it will require the system to rollback to an older checkpoint instead of the most recent one. Figure 3.4 shows the two types of failures that the semi-blocking protocol has to handle. In *Failure 1*, between the completion of the remote checkpoint and the start of the new checkpoint

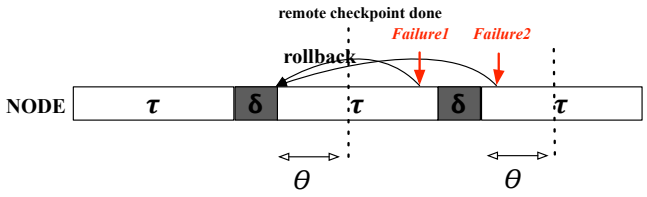


Figure 3.4: Rollback operation of the semi-blocking algorithm.

phase, the system rolls back to the first checkpoint. In case of *Failure 2*, which occurs in the middle of remote checkpoint transmission, still requires the system to rollback to the first checkpoint. This is because a global consistent state is not available until remote checkpoint is completed. The rest of this section presents a model that determines how these two types of failures affect the reliability of the system with different failure rates.

3.3.2 Model

In order to estimate the benefit of the semi-blocking checkpoint algorithm, we present a model that includes all the fundamental parameters. Two of them deserve special consideration. We use θ to denote the time in seconds to complete one remote checkpoint. Since the remote checkpoint runs concurrently with the application, it must hold that $\delta + \theta \geq \delta_{blocking}$. As we saw above, the larger θ is, the higher the chance a failure will require the system to rollback to an older checkpoint. To model the interference remote checkpoint may have on the execution of the application we use φ. It denotes the extra time in seconds per each checkpoint interval an application requires due to interference. The lower φ is, the better the semi-blocking algorithm is able to hide the checkpoint overhead.

The total time of a checkpointed workload with failures is divided into five parts:

$$T = T_s + T_{local} + T_{overhead} + T_{rework} + T_{restart} \tag{3.3}$$

where T_s is the pure computation time of the application without any checkpoints or rollbacks, T_{local} is the total time to dump local checkpoints in memory or local disk, $T_{overhead}$ is the total interference of remote checkpoint to applications, T_{rework} stands for the wall clock time required to bring the application back to the point it was right before the crash and $T_{restart}$ is the time to restart applications from checkpoint.

The dumping time of local checkpoints T_{local} is the product of the number of checkpoints and the dumping time of each checkpoint. The pure computation time in each checkpoint interval would be $\tau - \varphi$, thus,

$$T_{local} = \frac{T_s}{\tau - \varphi}\delta \tag{3.4}$$

$T_{overhead}$ is calculated in a similar way,

$$T_{overhead} = \frac{T_s}{\tau - \varphi}\varphi \tag{3.5}$$

The dotted lines in Figure 3.4 mark the end of remote checkpoint. Between them is one checkpoint interval τ plus the local checkpoint dumping time δ. Failures that happen during that period will rollback to the first checkpoint. The overlap period θ consists of the useful application work and the interference of the remote checkpoint φ. The useful application work of $\theta - \varphi$ is not checkpointed in the first checkpoint. So, rework time would be at least $\theta - \varphi$ if it happens right after the first remote checkpoint is done but at most $\theta - \varphi + \tau + \delta$ if it happens right before the second remote checkpoint is done. On the average, failures will occur half through the checkpoint interval. Thus,

$$T_{rework} = \frac{T}{M}\left(\frac{\tau + \delta}{2} + \theta - \varphi\right) \tag{3.6}$$

$T_{restart}$ depends on the number of failures and the restart time R, so

$$T_{restart} = \frac{T}{M}R \tag{3.7}$$

Hence, the total execution time including checkpoint and restart time for a workload of T_s is

$$\begin{aligned} T = T_s &+ \frac{T_s}{\tau - \varphi}\delta + \frac{T_s}{\tau - \varphi}\varphi \\ &+ \frac{T}{M}\left(R + \frac{\tau + \delta}{2} + \theta - \varphi\right) \end{aligned} \tag{3.8}$$

Similarly, for the blocking checkpoint, the total execution time $T_{blocking}$ of an application with T_s workload is

$$\begin{aligned} T_{blocking} = T_s &+ \frac{T_s}{\tau_{blocking}}\delta_{blocking} \\ &+ \frac{T_{blocking}}{M}\left(R + \frac{\tau_{blocking} + \delta_{blocking}}{2}\right) \end{aligned} \tag{3.9}$$

In order to minimize the equations 3.8 and 3.9 to find the optimum checkpoint interval for these two protocols we may use a standard numerical optimization technique. The checkpoint interval τ has strict bounds in both semi-blocking and blocking model, respectively, $\theta < \tau < M$ and $0 < \tau_{blocking} < M$.

To quantify the performance improvement using the semi-blocking checkpoint protocol compared to the blocking one, we calculate the $benefit$ of the semi-blocking protocol as,

$$Benefit = \frac{T_{blocking} - T}{T_{blocking}} \tag{3.10}$$

Here, T and $T_{blocking}$ are the total execution time of the application with the optimum checkpoint interval using the semi-blocking and blocking protocols, respectively.

3.4 Implementation

In this section, we describe our implementation of the semi-blocking checkpoint protocol. We present a technique to overlap the transmission of the remote checkpoint with the execution of the application. Two schemes to ensure that remote checkpointing is finished in a timely manner are also discussed. Finally, we discuss how to use solid state disk to reduce memory pressure.

3.4.1 A Runtime System for Multicore Clusters

We implemented the semi-blocking algorithm in the Charm++ [57] runtime system, which is based on a message-driven programming model. Applications are divided into fine-grain tasks using this model and tasks perform computation and communication through asynchronous method invocation associated with each message. Charm++ provides an *SMP* extension for machines with multicore nodes. The SMP version of Charm++ creates multiple *worker threads* per node. Additionally, each node has a dedicated *communication thread* to handle all the inter-node communication while cores on the same node communicate via shared memory. In this model, the worker threads do not need to bear the cost of communication and thus can provide high utilization. Typically, the communication thread is bound to a certain core in the node to avoid interference with worker threads. When a worker thread sends a network message, it enqueues the message into the outgoing message queue of the communication thread. Using the SMP mode of Charm++, we can achieve faster startup, reduction in memory consumption and optimized node-level collective communication operations.

Table 3.2: Overlap and interference associated with different communication to computation ratios

Case	Comm-Comp Ratio	θ	φ
1	0.2	6.6	0.6
2	1.6	13.5	0.8
3	3.1	17.3	1.5

3.4.2 Overlapping Remote Checkpointing and Computation

The success of semi-blocking checkpoint depends on the ability to overlap remote checkpoint transmission with the execution of the application. During remote checkpointing, each node sends checkpoint messages to its buddy; those messages are enqueued in the outgoing message queue after local checkpoint is finished. Thus, the transmission of application messages may be delayed due to transmission of checkpoint messages. To solve this problem, we add a separate checkpoint message queue on each communication thread. Worker threads enqueue the checkpoint messages to this new queue. The communication thread sends checkpoint messages only when there is no application message ready to be sent, to reduce the interference of the checkpoint message to applications. We call this strategy the *opportunistic* sending of checkpoint messages. Each checkpoint message is split into multiple small chunks for transmission to better overlap with the application execution.

Interference caused by remote checkpoint messaging to applications heavily depends on the communication profile of applications. We use a synthetic benchmark called FT_Test to analyze the sensitivity of the semi-blocking protocol to different conditions. FT_Test is a program that simulates a stencil computation. With FT_Test, it is possible to change the computation time per step, the message size, the communication topology of the objects and the checkpoint size. Table 3.2 presents three different levels of communication-computation ratios for FT_Test by changing the communication topology of the objects. We consider the time not spent in computation as communication time. We use the same checkpoint size of 512MB per node for the different communication-to-computation ratios. This amount of data requires 6.5s to checkpoint using the blocking protocol. As seen in Table 3.2, high communication to computation ratio increases both the overlap period θ and the interference of checkpointing to applications φ if using the semi-blocking algorithm. Figure 3.5 shows the benefit of the semi-blocking protocol for these three communication-to-computation ratios. As expected, the benefit is reduced when the communication-to-computation ratio of applications increases because of the increment in θ and φ . Even with a communication-to-computation ratio as high as 3.1, the semi-blocking protocol gives us more than 5% benefit compared to the blocking checkpoint protocol with an M value of 300s (M is the MTBF of

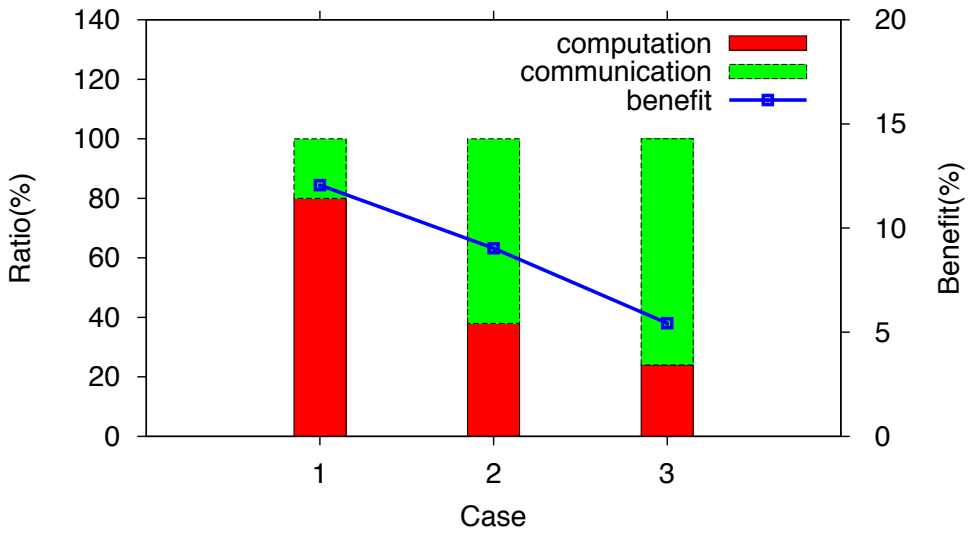


Figure 3.5: Effect of communication to computation ratio.

the system).

3.4.3 Opportunistic vs. Random Scheduling

To maximize the benefit of the semi-blocking protocol, we need to reduce the overlap period θ as well as interference φ. For a given application with certain communication to computation ratio, decreasing θ can help reduce the rework time when a failure happens. On the other hand, increasing θ may reduce the interference with the application. This is because with a longer period, it becomes easier to schedule checkpoint messages at optimal time. Finding a good overlap period is critical to the success of the semi-blocking protocol.

Opportunistic sending of checkpoint messages give us a fixed overlap period. However, we decided to test if this approach brings the most benefit that a semi-blocking algorithm can provide.For this purpose, we implemented *lottery scheduling* [59] to control the overlap period. Lottery scheduling is a randomized resource allocation mechanism used to control the relative execution rates of computations. It also supports resource management such as I/O bandwidth or memory. In this scheme, the allocations of shared resources to the clients are dependent by the number of lottery tickets they hold. Each time resources are granted to the client with the winning ticket. In the semi-blocking algorithm, transmission of remote checkpoint messages and application messages contend for the same NIC as the shared resource. By controlling allocations to use the NIC, we can achieve different overlap periods.

In FT_Test, the amount of messages sent is evenly distributed over computation. Thus, we

Upon Remote Checkpoint Done:
$appRatio \leftarrow \frac{\theta s}{\theta s+\tau c}$, $ckptRatio \leftarrow 1 - appRatio$
Send Network Message:
$choose \leftarrow rand()\%100$
if $choose \leq ckptRatio * 100$ **then**
  sendCheckpointMessage()
else
  sendApplicationMessage()
end if

Figure 3.6: Using lottery scheduling to control overlap period.

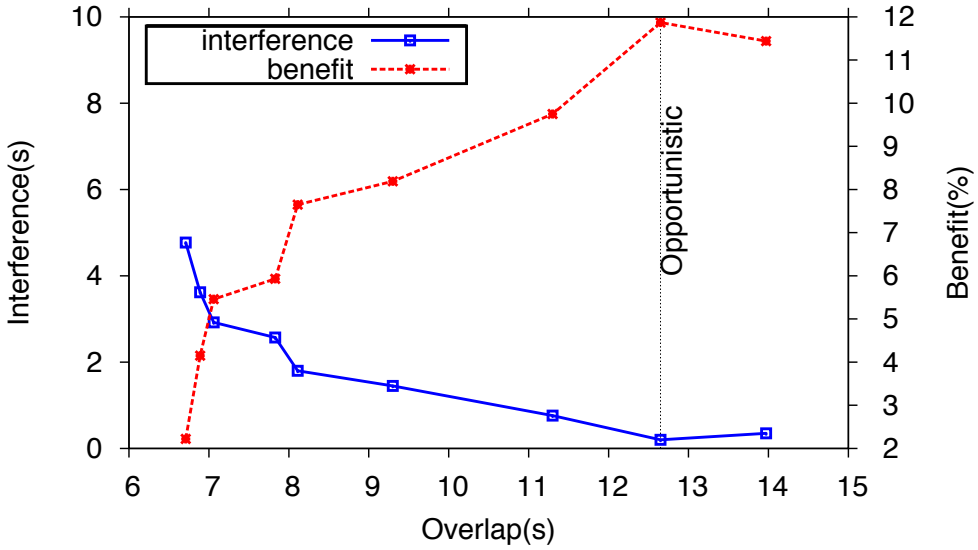


Figure 3.7: Interference and benefit of different overlap periods.

use the number of application messages sent to represent the amount of work that has been finished. The longer the overlap period is, the more application messages will be sent during the transmission of remote checkpoints. The number of application messages s and the number of checkpoint messages c in each checkpoint interval could be statistically obtained from previous checkpoints. Given an expected overlap period θ, the number of application messages sent during remote checkpoint is approximately $\frac{\theta}{\tau}s$. The number of application and checkpoint messages sent during remote checkpoint can be used as their lottery tickets. So the probability to send an application message during remote checkpoint is $\frac{\theta s}{\theta s+\tau c}$, while the probability to send a checkpoint message is $\frac{\tau c}{\theta s+\tau c}$.

The pseudo-code of using lottery scheduling to control the overlap period is shown in Figure 3.6. Each time, before sending a network message, a random number generator is used to randomly select a winning ticket. Then we locate whether the application or

checkpoint message is holding that ticket and find the message that is sent next. As seen in Figure 3.7, for a FT_Test with communication/computation ratio set to 1.6, the interference of opportunistic sending of checkpoint message is the minimum. Increasing the overlap period increases the overhead slightly while decreasing the overlap period increases the interference dramatically. Increasing or decreasing the overlap period both fail to bring us a higher benefit using the model in Section 3.3. So, opportunistic sending of the checkpoint message is better than lottery scheduling for the semi-blocking algorithm, empirically. Even though a reduced overlap period θ can reduce the rework time, it cannot offset the interference gained with the reduced θ.

3.5 Using SSD to Relieve Memory Pressure of Checkpointing

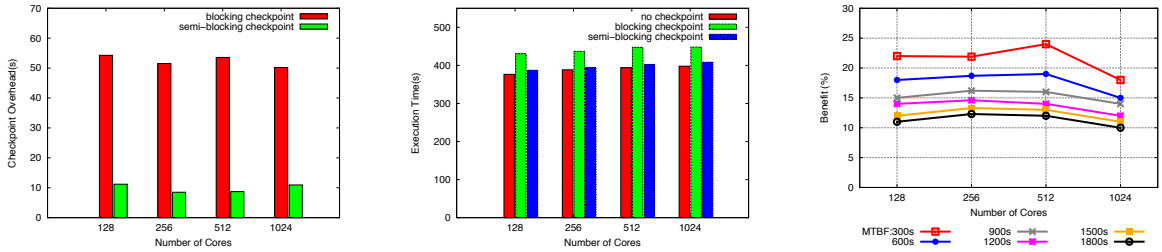
The local and remote checkpoints constitute a memory overhead for the presented semi-blocking protocol, but (somewhat surprisingly) this is tolerable for a large class of applications that have a smaller memory footprint at checkpoint. These include molecular dynamics applications, N-body codes, certain quantum chemistry (nanomaterials codes), etc. Even for applications with large memory footprint, we use SSDs to relieve the memory pressure of checkpointing.

Solid state disk (SSD) is becoming more and more promising to store the large amount of checkpoint data because of its good random access performance and low power consumptions. Considering the SSD bandwidth is not comparable to memory bandwidth nowadays, we need to carefully select what checkpoint data to be stored in SSD. We propose two strategies to checkpoint the data on SSD.

1. *Full SSD Strategy*: All the checkpoints will be saved to SSD which can fully relieve the memory pressure.

2. *Half SSD Strategy*: We reduce the writes to SSD by only storing the buddy's checkpoints in SSD. At restart, only checkpoints of the crashed node need to be read from SSD while other nodes can recover from the checkpoints in memory.

Instead of stalling every worker thread (core) to write to SSD, a dedicated IO thread is used on each node for asynchronous access to SSD. Each worker thread enqueues its IO requests to the queue for the IO thread and gets a notification when the IO request is completed by the IO thread. Access to SSD can thus be adaptively overlapped with useful computation. In Section 3.6 we will show the performance impact of using these two strategies for checkpoint and restart, and the performance gains provided by asynchronous access to SSD.



(a) Single Checkpoint (b) Checkpoint Overhead (c) Semi-Blocking Benefit

Figure 3.8: Weak scaling results - wave2D.

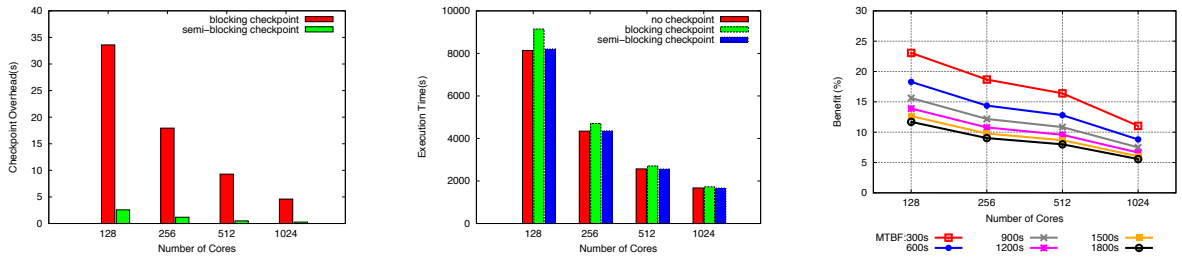
3.6 Experiments and Analysis

This section presents an evaluation of the semi-blocking checkpoint algorithm with two different applications. The first application is wave2D, which uses a finite differencing scheme to calculate pressure information over a discretized 2D grid. The second application, ChaNGa, is for N-Body based parallel simulations and is used in cosmology and astronomy [60]. We also present results related to the performance of restarting applications after a failure.

The experiments were performed on Trestles supercomputer at the San Diego Supercomputer Center. Trestles consists of 324 nodes with 32 cores per node. The theoretical peak performance of the system is 100 teraflops. Each compute node contains four sockets, each with a 8-core 2.4 GHZ AMD Magny-Cores processor. Each node has 64 GB of DDR3 RAM and 120 GB of flash memory(SSD).

3.6.1 Scalability

Figure 3.8a shows the overhead of one checkpoint based on a weak scaling test with wave2D using blocking and semi-blocking checkpoint protocol from 128 cores to 1K cores. The checkpoint size is 4 GB per node. Semi-blocking algorithm reduces the checkpoint overhead from 52 s to 10 s. The optimal checkpoint interval of the blocking algorithm is 372 s given an M value of 1800 s (M is the MTBF of the system) and $\delta_{blocking}$ of 52 s. This requires the wave2D application to checkpoint every 960 iterations. In Figure 3.8b, we show the checkpoint overhead to the execution of the applications using the two algorithms. The checkpoint overhead is reduced from 14% to 3% by using the semi-blocking algorithm. With the decreasing checkpoint overhead, the semi-blocking algorithm can afford to checkpoint more frequently to reduce the amount of rework time. So, the optimum checkpoint interval of the semi-blocking algorithm is different from that of the blocking algorithm.



(a) Single Checkpoint (b) Checkpoint Overhead (c) Semi-Blocking Benefit

Figure 3.9: Strong scaling results - ChaNGa.

Now, we compare the performance of the semi-blocking algorithm with the performance of the blocking algorithm using our model and considering both the checkpoint and rollback-recover overhead. As seen in Figure 3.8c, for different M values, the benefit of the semi-blocking checkpoint protocol is mostly constant from 128 cores to 1K cores. When M decreases, checkpoint and restart overhead for the blocking checkpoint protocol increases, hence the semi-blocking protocol shows more benefit. The benefit of the semi-blocking protocol varies from 10% for M of 1800 s to 22% for M of 300 s for checkpoint size of 4 GB/node.

ChaNGa is used to demonstrate the strong scalability of the semi-blocking checkpoint algorithm. We use a 100 million particle system. In one main step of ChaNGa, it first does domain decomposition of the particle space, then builds the Barnes-Hut trees, computes the gravitational forces, and finally updates the particles. Checkpoint is taken periodically after a certain number of steps. Figure 3.10 displays the view of communication bytes sent over time from our Projections performance analysis tool. There is less amount of communication data in the first two phases of one step: domain decomposition and tree building as seen in the figure. Sending more checkpoint messages in these phases can help us incur less interference to the application. With the opportunistic sending of the checkpoint message, our scheme can identify such phases without application knowledge.

Figure 3.9a shows the checkpoint overhead of one checkpoint based on a strong scaling test of ChaNGa application using the blocking and semi-blocking algorithms separately. Checkpoint size per node decreases for a strong scaling test, so the blocking checkpoint time is reduced from 33 s on 128 cores to 5 s on 1K cores. The semi-blocking checkpoint time decreases from 2.6 s to 0.27 s, almost hiding the checkpoint overhead.

In Figure 3.9b, we display the checkpoint overhead of the execution of ChaNGa. The optimum checkpoint interval is adjusted to the blocking checkpoint time. The blocking checkpoint overhead decreases from 12% on 128 cores to 5% on 1K cores because of the de-

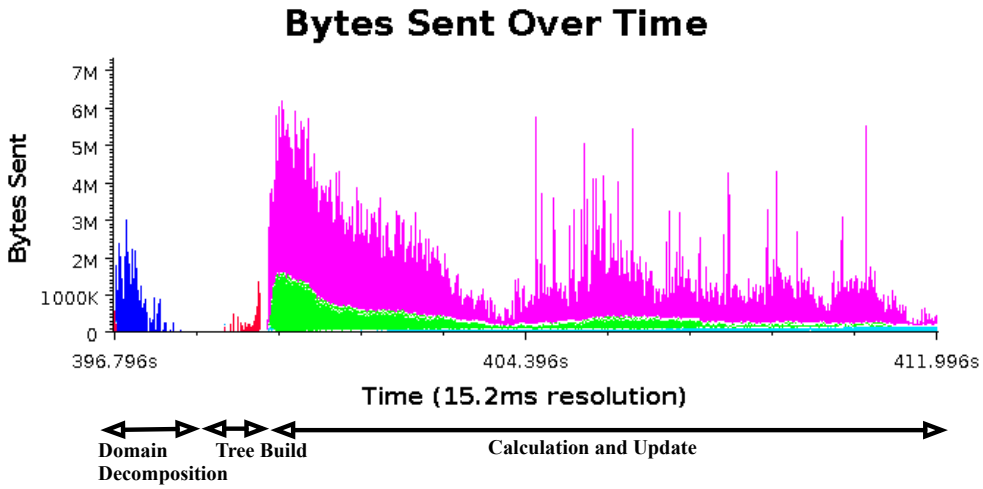


Figure 3.10: Bytes sent in one step of ChaNGa.

creased checkpoint size per node. With the semi-blocking algorithm, the checkpoint overhead is below 1%. Of course with such low checkpoint overhead of the semi-blocking algorithm, applications can benefit more from frequent checkpoints so as not to lose lots of wall clock cycles when failure happens.

Figure 3.9c depicts the percentage benefit of the semi-blocking algorithm to the blocking algorithm at their own optimal checkpoint intervals. Semi-blocking algorithm achieves the largest benefit on 128 cores where the blocking checkpoint overhead is at its maximum in a strong scaling experiment. Even when running on 1K cores with M of 1800 s, the semi-blocking algorithm has over 6% benefit compared to the blocking algorithm. Given that the memory consumption is only 763 MB per node when running on 1K cores, we expect more benefit of the semi-blocking algorithm for applications with larger memory consumption.

3.6.2 Virtualization Analysis

Over-decomposition and asynchronous communication in Charm++ can greatly help overlap communication and computation of applications. In Charm++, programs are broken up into objects called chares. Usually, there are more chares than the number of processors. The number of chares divided by the number of processors is called *virtualization ratio*. With high virtualization ratio, the communication of the checkpoint or application message of one chare can be overlapped with the computation of other chares on the same core which can further hide the checkpoint overhead, so we can expect more benefits.

We use the wave2D benchmark to illustrate the benefit of high virtualization ratio for

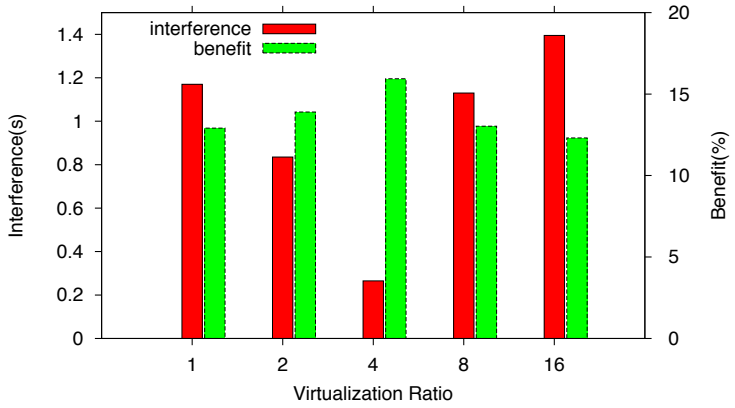


Figure 3.11: Effect of virtualization.

the semi-blocking algorithm. The checkpoint size is 0.9 GB per node. The interference of remote checkpoint per checkpoint interval decreases from 1.4 s with 1 chare per core to 0.3 s with 4 chares per core in Figure 3.11. Correspondingly, the benefit of the semi-blocking protocol to the blocking version calculated from the model increases from 12.3% to 15.9% when M is 300 s as expected. However, when the virtualization ratio is increased to 8 and beyond, there is extra overhead to schedule the work of multiple chares, so there is more interference.

3.6.3 Checkpoint and Restart with SSD

As discussed in Section 3.5, half SSD and full SSD schemes can be used depending on the memory consumption of applications.

Figure 3.12 shows the checkpoint timing penalty using SSD with checkpoint data size ranging from 0.45 GB to 2.23 GB per node for the wave2D benchmark. We compare the performance of half and full SSD scheme with asynchronous (half-aio, full-aio) and synchronous IO access (half-sio, full-sio) respectively. Using full SSD scheme with asynchronous IO access saves us more than half the time of writing checkpoint data to SSD with synchronous IO. In Figure 3.13, we show the restart time of in-memory checkpointing, half and full SSD checkpointing with asynchronous IO access. Half SSD scheme has almost negligible overhead compared to the in-memory checkpointing, while full SSD scheme has around one second overhead. During restart, objects first need to get their checkpoints either from their own or their buddy node's local disk or memory, and then restore the application and process data from the checkpoints. With asynchronous IO access and high virtualization ratio, the restoring of one object can be overlapped with the process of getting checkpoints for another

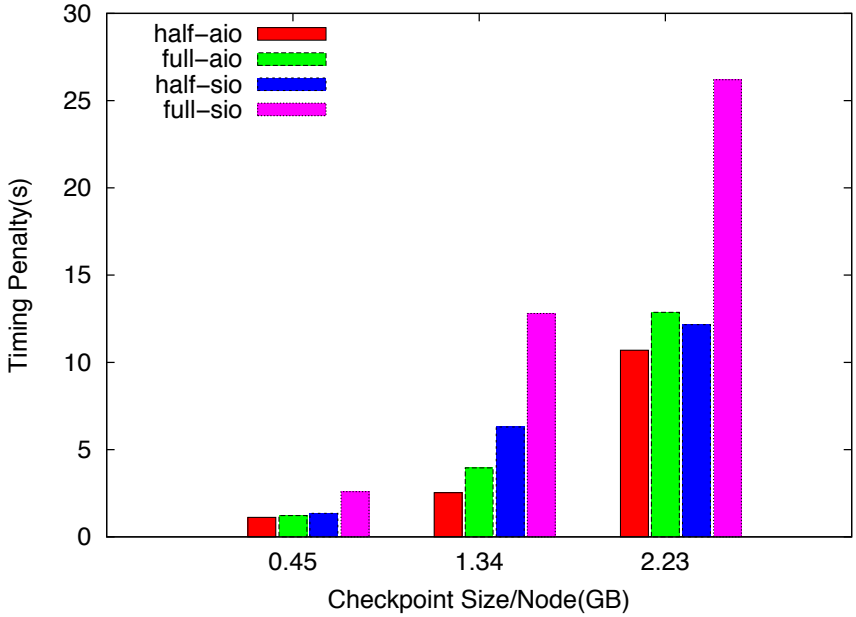


Figure 3.12: Penalty of checkpointing to SSD.

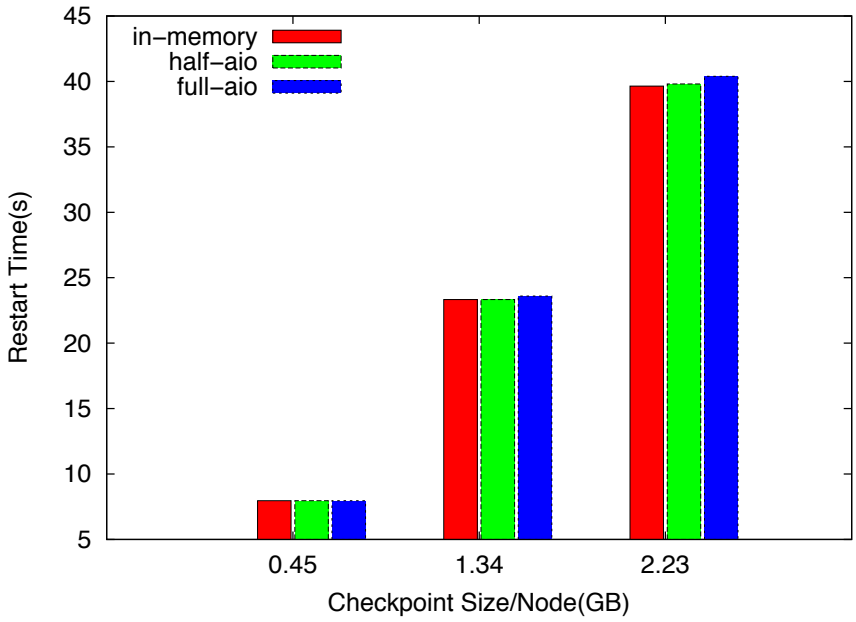


Figure 3.13: Low overhead of restarting from SSD.

object. Thus we see the restart time is not affected much by checkpointing to SSD.

3.7 Conclusion

In this chapter, we have presented a semi-blocking checkpoint algorithm to provide low-overhead resilience for HPC applications. Our algorithm succeeds in hiding checkpoint overhead by overlapping checkpoint transmission with execution of the application. We also provided a model for the algorithm to find out the optimum checkpoint interval and to compute the expected benefit of the algorithm under various scenarios.

The presented experimental results are very encouraging. Both strong and weak scalability of the semi-blocking algorithm have been demonstrated with different applications. We have showed that, on 1024 cores, the semi-blocking algorithm reduces the checkpoint overhead from $50s$ to $10s$ for a stencil computation. A strong scaling test using a cosmology application showed that the checkpoint overhead is almost negligible. Using our model and a range of different failure rates predicted at Exascale, we have also showed that the semi-blocking checkpoint algorithm may reduce the total execution time by 22% in comparison to the traditional blocking checkpoint algorithm.

CHAPTER 4

Replication Enhanced Checkpointing for Soft and Hard Error Protection [1]

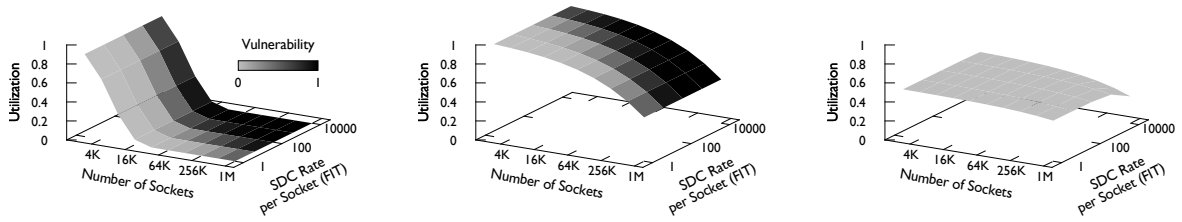
In Chapter 3, we described a semi-blocking checkpoint algorithm to reduce the overhead of protecting applications from fail-stop failures. Other than fail-stop failures, soft error is another challenge that prevents correct execution of HPC applications. Soft errors are becoming more prevalent as feature sizes decrease along with supply voltages to chips. The most insidious form of soft error is silent data corruption (SDC) [2]. For mission-critical applications, having a guarantee that the data was not silently corrupted may be very important. To address this concern, in this chapter, a general solution to protect applications from both hard and soft errors is introduced.

4.1 Overview

As discussed in the previous chapter, one of the common approaches to tolerate intermittent faults is by periodically checkpointing the state of the application to disk and restarting when needed. However, as hard failure rates increase along with machine sizes, this approach may not be feasible due to high overheads. If the data size is large, the expense of checkpointing to disk may be prohibitive, and may incur severe forward-path overheads. At the possible cost of memory overhead, recent libraries for checkpointing have successfully explored alternative local storage resources to store checkpoint data [25, 26, 49].

Although checkpoint/restart strategies may be effective for hard faults, SDC can not be detected using them. Detection of SDC is a difficult problem and most traditional fault

[1] Based on [61]



(a) No fault-tolerance protection (b) Hard-error checkpoint-based protection (c) ACR with protection from SDC and hard error

Figure 4.1: Overall system utilization and vulnerability to SDC with different fault tolerance alternatives (for a job running 120 hours). ACR offers holistic protection using scalable mechanisms against SDC and hard errors.

tolerance approaches applied to HPC fail at addressing this problem. One possible solution, which has been shown to be effective in the past, is to use redundant computation to detect SDC and correct them. This approach is beneficial because it is universally applicable and is well-established as a solid approach. However, due to its perceived high cost, it is only recently been explored for HPC. Analytical studies have shown that if the failure rate is sufficiently high, introducing redundancy to handle errors may actually increase the overall efficiency in the HPC realm [62, 63]. Other work has shown that redundancy is an effective methodology for detecting SDC in HPC applications [64].

In order to better grasp the problem, we model the system utilization and system vulnerability (the probability of finishing execution with an incorrect result) as the number of sockets increase and SDC rate increases. Figure 4.1a shows the results of varying these parameters without any fault tolerance. Note that, as the socket count increases from 4K to 16K, the utilization rapidly declines to almost 0. With hard-error resilience (shown in Figure 4.1b) using checkpoint/restart, the utilization increases substantially, but still drops after 64K sockets. However, since checkpoint/restart cannot detect SDC, the vulnerability remains very high. To mitigate both these problems, we present ACR: a scalable, automatic checkpoint/restart framework that can detect and correct both SDC and hard errors. As shown in Figure 4.1c, by using our framework the system vulnerability disappears and the utilization remains almost constant; the utilization penalty, which seems significant at small scale, is comparable to other cases at scale.

We believe, to reach the exascale computing realm effectively, we must develop holistic solutions that cover all the types of failures that may occur in the system. Although hard failures may be detectable by external system components, soft errors remain elusive. Hence, software solutions that address both problems in an integrated fashion are needed. ACR does exactly this, and also utilizes a novel mechanism to interact with applications for

checkpointing based on observed failure rates.

By developing this framework and empirically evaluating it under various failure scenarios, we make the following contributions:

- We present a novel design for an automatic checkpoint/restart mechanism that tolerates both SDC and hard errors, and can adaptively adjust the checkpointing period (§4.2, §4.3, §4.4).
- We present a distributed algorithm for determining checkpoint consensus asynchronously, and show empirically that it causes minimal application interference (§4.2).
- We present three distinct recovery schemes in ACR that explore the tradeoff between performance and reliability. Using a model we have developed, we analytically study for these schemes the interaction between hard-error recovery and soft-error vulnerability at large scales (§4.2, §4.5).
- We demonstrate use of topology-aware mapping to optimize communication, and empirically show that this results in significant speedup during checkpointing and restart (§4.4, §4.6).
- We evaluate ACR by showing for five mini-applications, written in two programming models, on 131,072 cores that the framework is highly scalable and adapts to dynamic behavior (§4.6).

4.2 Automatic Checkpoint Restart

In this section, we describe the Automatic Checkpoint/Restart (ACR) framework, a low-overhead framework that aims to provide protection from both SDC and hard errors to applications. To handle failures efficiently, ACR automatically checkpoints at an adaptive rate on the basis of failure history. If failures occur, based on the type of error, ACR enacts corrective measures and performs an automatic restart.

4.2.1 Replication-enhanced Checkpointing

ACR uses checkpointing and replication to detect SDC and enable fast recovery of applications from SDC and hard errors. When a user submits a job using ACR, a few nodes are marked as spare nodes and are not used by the application, but only replaces failed nodes

when hard errors occur. The rest of the nodes are equally divided into two partitions that execute the same program and checkpoint at the same time. We refer to these two partitions as replica 1 and replica 2. Each node in replica 1 is paired with exactly one unique node in replica 2; we refer to these pairs as *buddies*. Logically, checkpointing is performed at two levels: local and remote. When a checkpoint is instigated by the runtime, each node generates a local checkpoint by invoking a serialization framework to save its current state. Programmers are required to write simple functions that enable ACR to identify the necessary data to checkpoint. Local checkpoint of a node in one replica serves as remote checkpoint of the buddy node in another replica.

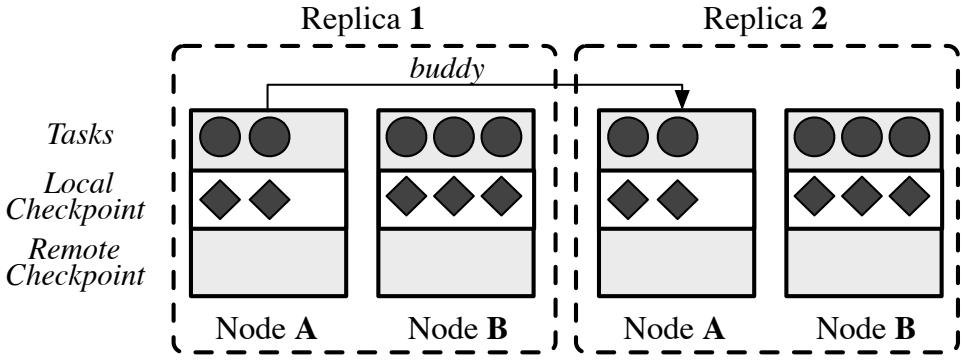


Figure 4.2: Replication enhanced checkpointing. The buddy of a node is the corresponding node in the other replica.

Hard Error Protection: We assume hard errors follow a fail-stop model, i.e. a failed node does not perform any communication. We call the replica containing the crashed node the *crashed* replica and the other replica the *healthy* replica. After a failure is detected, the buddy node (in the healthy replica) of the crashed node sends its own local checkpoint to the new node (from the spare pool) that replaces the crashed node. Since the paired buddy nodes perform exactly the same work during forward-path execution, the crashed node can be restarted using the checkpoint of its buddy node on the new node. Every other node in the crashed replica rolls back using the checkpoint stored locally.

Detection and Correction of Silent Data Corruption: In order to detect SDC, every node in the replica 1 sends a copy of the local checkpoint to its buddy node in replica 2. Upon receiving the remote checkpoints from their buddy nodes in replica 1, every node in replica 2 compares the remote checkpoint with its local checkpoint using the same serialization framework used to pack the checkpoint. If a mismatch is found between the two checkpoints, ACR rolls back both the replicas to the previous safely stored local checkpoint, and then resumes the application. Note that the remote checkpoint is sent to the replica 2 only for

SDC detection purposes, and hence ACR does not store the remote checkpoint. Figure 4.2 shows the stable state of nodes during application execution when using ACR.

4.2.2 Automatic Checkpoint Decision

An important feature of replication-enhanced checkpointing is its ability to reduce recovery overhead in the face of hard errors, which is enabled by automatic checkpointing. When a hard failure occurs, if an immediate checkpoint can be performed in the healthy replica to help the crashed replica recover instead of using the previous checkpoint, the crashed replica can quickly catch up with the progress of the healthy one. Moreover, as online failure prediction [65] becomes more accurate, checkpointing right before a potential failure occurs can help increase the mean time between failures visible to applications. ACR is capable of scheduling dynamic checkpoints in both the scenarios described.

Upon reception of a checkpointing request, ACR can not simply notify every task to store its state. This may lead to a situation where an inconsistent checkpoint is stored, causing the program to hang. For example, in an iterative application, assume task a receives the checkpoint decision at iteration i, and after restart it will wait to receive a message from task b to enter iteration $i+1$. However when task b receives the checkpoint decision, it is already at $i+1$ and has already sent out message c. This in-flight message c will not be stored in the checkpoint anywhere. Thus after restart, task b will be unaware that it needs to re-send the message to a and task a will hang at iteration i. This scenario is possible when there is no global synchronization at each iteration and each task progresses at different rates during application execution. ACR ensures the consistency of checkpointing with minimal interference to applications using the following scheme based on a similar idea used in Meta Balancer [66].

Periodically, each task reports its progress to ACR through a function call. In an iterative application, for example, this function call can be made at end of each iteration. In most cases, when there is no checkpointing scheduled, this call returns immediately. ACR records the maximum progress among all the tasks residing on the same node as shown in Figure 4.3 (Phase 1). If checkpointing is required, either due to a failure in one of the replicas or based on an observation of the failure history, ACR proceeds to find a safe checkpoint iteration.

Using the local progress information, ACR begins an asynchronous reduction to find the maximum progress among all the tasks. In the mean time, tasks whose progress has reached the local maximum are temporarily paused to prevent tasks from going beyond

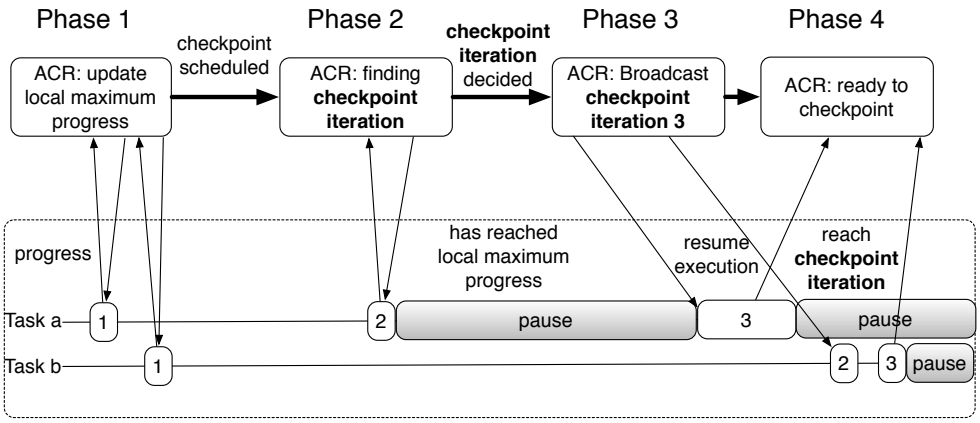
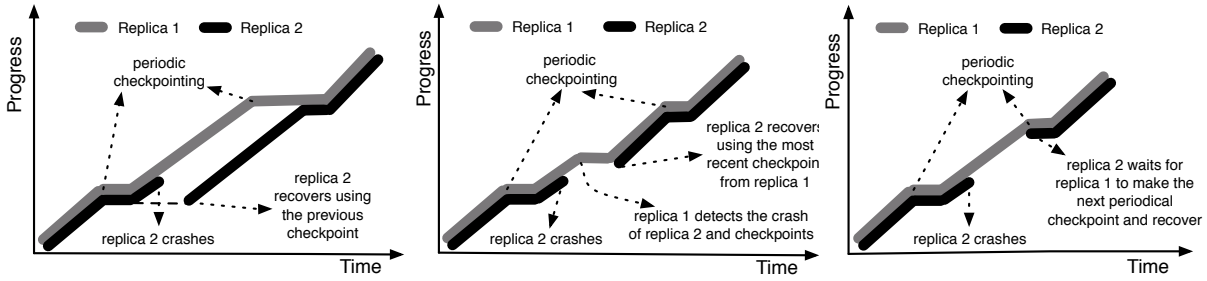


Figure 4.3: Initialization of automatic checkpointing.

the possible checkpoint iteration (Figure 4.3, Phase 2). Once ACR finds the maximum progress in the system, the checkpoint iteration is decided accordingly. Each task compares its progress with the checkpoint iteration; if its progress has reached the checkpoint iteration, the task is considered ready for the checkpoint and transitions to the pause state if it was in the execution state or remains in the pause state if it was already in the pause state. Otherwise, the computation task will continue or resume execution until it reaches the checkpoint iteration (Figure 4.3, Phase 3). Eventually, when all the tasks get ready for the checkpoint, checkpointing is initiated (Figure 4.3, Phase 4). The more frequently the progress function is invoked, the sooner ACR can schedule a dynamic checkpoint.

Adapting to Failures: It has been shown that a fixed checkpoint interval is optimal if the failures follow a Poisson process [58]. However, a study of a large number of failure behaviors in HPC systems [67] has shown that a Weibull distribution is a better fit to describe the actual distribution of failures. An important point to note in this study is that the failure rate often decreases as execution progresses. Dynamically scheduling checkpoints has shown benefits in such scenarios in comparison to a fixed checkpoint interval in an analytical study [68, 69]. Hence, it is important to fit the actual observed failures during application execution to a certain distribution and dynamically schedule the checkpoints based on the current trend of the distribution. To support such adaptivity, ACR provides a mode in which each checkpoint interval is decided based on the distribution of the streaming failures. This is enabled by the automatic checkpointing and automatic recovery in ACR.



(a) Strong resilience recovery scheme (b) Medium resilience recovery scheme (c) Weak resilience recovery scheme

Figure 4.4: Recovery in different resilience levels of ACR. Strong resilience rolls back immediately after a hard error. Medium resilience forces an additional checkpoint and restarts from there. Weak resilience waits until the next checkpoint to restore.

4.2.3 Interaction of Hard Error Recovery and Vulnerability to Silent Data Corruption

Detection and correction of SDC using replication enables ACR to recover from hard failures in different ways. These choices offer *novel trade-offs* that have not been encountered in any framework for HPC. Three resilience schemes may be used in ACR depending on the reliability and performance requirements of an application.

1) Strong Resilience: In this scheme, the crashed replica is rolled back to the previous checkpoint. The restarting process (on the spare node) is the only process in the crashed replica that receives the checkpoint from the other replica, and hence minimal network traffic is generated. Every other node in the crashed replica rolls back using its own local checkpoint. Figure 4.4a shows the progress chart in which replica 2 is recovered using strong resilience. When a hard error is encountered in replica 2, the crashed replica restarts using the previous checkpoint. Having reached the next checkpoint period, replica 1 waits for replica 2 to resume application execution.

The advantage of using this scheme is 100% protection from SDC. The execution of applications in the two replicas is always cross-checked. Additionally, restarting the crashed replica is very fast because only one message is sent from the healthy replica to the restarting process. However, the amount of rework being done is large, and it may slow down the application progress.

2) Medium Resilience: This scheme attempts to reduce the amount of rework by forcing the healthy replica to immediately schedule a new checkpoint when a hard error is detected in the crashed replica as shown in Figure 4.4b. The latest checkpoint is sent from every node of the healthy replica to their buddy nodes in the crashed replica, which may incur

relatively higher overhead in comparison to the strong resilience scheme. Moreover, any SDC that occurred between the previous checkpoint and the latest checkpoint will remain undetected. On the positive side, this scheme avoids rework on the crashed replica and hence the two replicas reach the next checkpoint period at similar times in most cases. However, if a hard failure occurs in the healthy replica before the recovery of crashed replica is complete, application needs to rollback either to the previous checkpoint or the beginning of execution. Since the healthy replica schedules the next checkpoint in a very short period, the probability of such a rollback is very low.

3) Weak Resilience: In this scheme, the healthy replica does not take any immediate action to restart the crashed replica when a hard error is detected. Instead, it continues execution until the next checkpoint and thereafter sends the checkpoint to the crashed replica for recovery. This scheme leads to a "zero-overhead" hard error recovery since in most cases the healthy replica does not incur any extra checkpoint overhead to help the crashed replica recover, and the crashed replica does not spend any time in rework. The only exception is when hard failure occurs in the healthy replica before it reaches the next checkpoint time. If the failure happens on the buddy node of the crashed node (though the probability is very low [30, 63]) application needs to restart from the beginning of the execution. Otherwise application needs to restart from the previous checkpoint. Typically, the system is left unprotected from SDC for the entire checkpoint interval. Figure 4.4c shows the event chart for this resilience scheme. Assuming a large rework time, Figure 4.4 suggests that this scheme should be the fastest to finish application execution.

4.2.4 Control Flow

Figure 4.5 presents a study of application execution using ACR with different reliability requirements. In each scenario, execution begins with the division of the allocated nodes into two replicas each of which performs the same application task.

Figure 4.5(a) presents the scenario in which an application only requires support for handling hard errors. No periodic checkpointing is needed in this scenario. When a node crash is detected at *T2*, replica 2 schedules an immediate checkpoint, and sends the checkpoint to replica 1. This allows replica 1 to continue on the forward path without rollback.

Figures 4.5(b,c,d) present the general scenario in which both silent data corruptions and hard errors may occur in the system. In all these scenarios, periodic local checkpointing is performed (e.g. at time *T1*, *T3* etc.). When a hard failure occurs at time *T2*, in the strong resilience scheme shown in Figure 4.5(b), replica 2 sends its SDC-free local checkpoint at *T1*

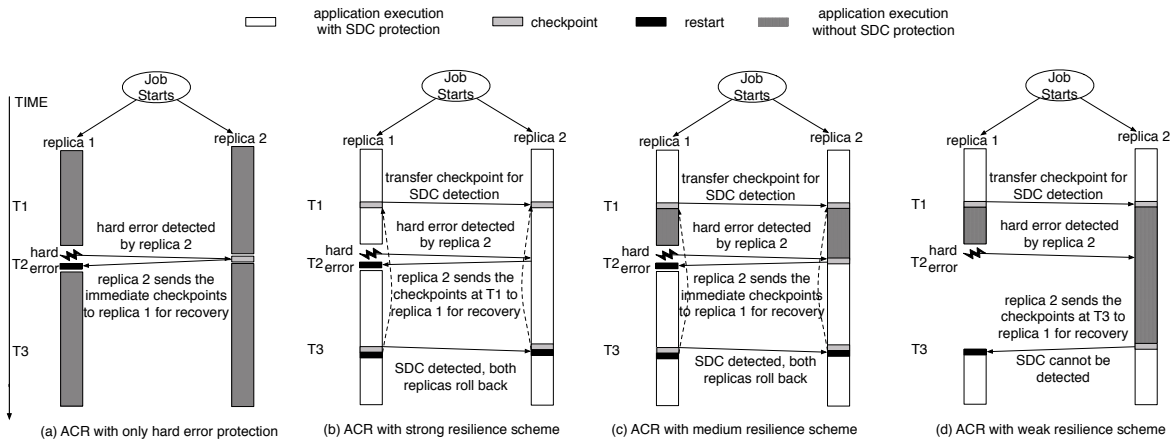


Figure 4.5: The control flow of ACR with different reliability requirements.

to the restarting process in replica 1 to help it recover. The application is fully protected from SDC in this scenario. However, in Figure 4.5(c) with medium resilience, an immediate checkpoint is performed in replica 2 when a failure occurs at time *T2*. Replica 1 is recovered using this new checkpoint. As such, at time *T3*, only SDC that occurred after *T2* will be detected. In weak resilience scheme of Figure 4.5(d), replica 2 continues execution until the next scheduled checkpoint time *T3*, and then sends this checkpoint to replica 1 for recovery. Although this scheme incurs zero-overhead in the case of a hard failure, the application is not protected from SDC from time *T1* to *T3*.

4.3 Design Choices

During the design process of ACR, we evaluated alternative methods for different components of the framework, and selected the ones most suited to our needs. In this section, we present those design choices and their trade-offs relative to the alternatives.

1) Ensuring consistent states. To enable the recovery of a crashed replica from a hard error using information from the healthy replica, it is necessary that the processes in the two replicas are interchangeable, i.e. for every process in replica 1 there is a process in replica 2 that has the same application state. ACR makes use of coordinated checkpointing to ensure this consistency, and hence does not require any communication or synchronization between replicas unless a hard error occurs.

An alternative approach to maintain the consistent state between replicas is by cloning messages. Libraries such as rMPI [63] and P2P-MPI [70], which provide replication-based reliability to MPI applications, provide reliability support by ensuring that if an MPI rank

dies, its corresponding MPI rank in the other replica performs the communication operations in its place. This approach requires the progress of every rank in one replica to be completely synchronized with the corresponding rank in the other replica before and after the hard error. Such a fine-grained synchronization approach may hurt application performance, especially if a dynamic application performs a large number of receives from unknown sources. In fact, in such scenarios the progress of corresponding ranks in the two replicas must be serialized to maintain consistency. For a message-driven execution model in which the execution order is mostly non-deterministic, this approach is certainly not optimal.

2) Who detects silent data corruption? In ACR, the runtime is responsible for transparently creating checkpoints and their comparison to detect SDC. Algorithmic fault tolerance is an alternative method based on redesigning algorithms using domain knowledge to detect and correct SDC [71]. Use of containment domains [72] is a programming-construct methodology that enables applications to express resilience needs, and to interact with the system to tune error detection, state preservation, and state restoration. While both these approaches have been shown to be scalable, they are specific to their applications. One may need to have in-depth knowledge of the application domain and make significant modifications to the code in order to use them. In contrast, a runtime-based method is universal and works transparently with minimal changes to the application. Hence, we use this strategy in ACR.

3) Methods to detect silent data corruption. Similar to *ensuring consistent states*, an alternative method to detect SDC is to compare messages from the replicas [64]. If a message is immediately sent out using the corrupted memory region, early detection of SDC is possible using this scheme. However, a major shortcoming of message-based error detection is the uncertainty of error detection – if the data affected by SDC remains local, it will not be detected. Moreover, even when corruption has been detected, it may be difficult to correct the corrupted data on the source process if the corruption was not transient or was used in the computation. Checkpoint-based SDC detection does not suffer from any of these issues; given the synergy with the hard-error recovery method, it is the appropriate choice for ACR.

4) Redundancy model. Based on dual redundancy, ACR requires re-executing the work from the last checkpoint if one SDC is detected. Alternatively, triple modular redundancy (TMR) is a popular method to provide resilience for applications that have real-time constrains. In TMR, the results processed by the three redundant modules are passed by a voting system to produce a single output and maintain consistency. The trade off to consider between dual redundancy and TMR is between re-executing the work or spending another

33% of system resources on redundancy. We have chosen the former option assuming good scalability for most applications and relatively small number of SDCs. Dual redundancy, as a fault tolerance alternative, requires to invest at least 50% of the system's utilization. This is a considerable price to pay upfront to recover from SDCs, but it is a general-purpose solution. Additionally, it has been shown that replication outperforms traditional checkpoint/restart for scenarios with high failure rates [63].

5) Checkpointing level. Checkpointing can be performed either at the kernel or user level. Kernel-level checkpointing like BLCR [24] dumps all the system state and application data during checkpointing. As a result it can quickly react to a failure prediction. In contrast, user-level checkpointing such as SCR [25] is triggered by the application at a certain interval. Compared to kernel-level checkpointing, it can reduce the checkpoint size since the process state and buffered messages are not stored. ACR performs user-level checkpointing but with the simplicity and flexibility advantages of the kernel-level scheme. The checkpointing in ACR is equivalent to the combination of LOCAL and PARTNER levels in SCR. Each checkpoint is invoked by the runtime at a safe point specified by the user in order to store the minimal state needed. But we allow the interval between checkpoints to be dynamically adjusted to the observed failure rate without user involvement.

4.4 Adaptation and Optimizations

An adaptation of ACR to Charm++ has been performed to validate it on real systems executing applications of various types. In order to support ACR, we have made use of some existing features in Charm++ and added new ones. Important optimizations to boost performance and reduce overheads have been performed.

4.4.1 Implementation Details

Replication. To support replication, we have augmented Charm++ with support for transparent partitioning of allocated resources. On a job launch, ACR first reserves a set of spare nodes (§ 4.2) to be used in event of failure. The remaining nodes are divided into two sets that constitute the two replicas. The application running in each replica is unaware of the division and executes independently in each replica. In addition to support for regular intra-replica application communication, we have added an API for inter-replica communication that is used by ACR for various purposes.

Checkpointing. Charm++ supports checkpointing to either memory or file system using simple user specified *pup* functions. The *pup* functions use the Pack and UnPack (PUP) framework to serialize/deserialize the state of the application to/from the chosen medium using special PUPer(s).

Handling SDC. We have augmented the PUP framework to support a new PUPer called the *checker*. This PUPer compares the local checkpoint of a node with the remote checkpoint sent to the node by its buddy, and reports if silent data corruption has occurred. *PUPer::checker* also enables a user to customize the comparison function based on their application knowledge. For example, since the floating point math may result in round-off errors, a programmer can set the relative error a program can tolerate. One may also guide the PUP framework to ignore comparing data that may vary between different replicas, but are not critical to the result.

4.4.2 Optimizations

Simultaneous transmitting checkpoints for comparison or during restart using the weak and medium resilience schemes may saturate the network and result in congestion. We have implemented the following two techniques to reduce network congestion, and hence improve the performance of ACR.

Checksum. A simple but effective solution to network congestion problem is use of a *checksum* to compare the checkpoints. ACR uses the position-dependent Fletcher's checksum algorithm [73] to calculate the checksum of a checkpoint, which is then transmitted to the buddy for comparison. While the use of checksums reduces the load on the network, it increases the computation cost. Instead of a single instruction required to copy the checkpoint data to a buffer if the full checkpoint is sent, 4 extra instructions are needed to calculate the checksum. Assuming a system that has the communication cost per byte of β and computation cost of γ per byte, the difference in cost of the two schemes is $(\beta - 4\gamma) \times n$. Hence, using the checksum shows benefits only when $\gamma < \frac{\beta}{4}$.

Topology-aware mapping. ACR implements topology-aware task mapping to reduce network congestion during restart and checkpoint comparison (if checksum is not used). Consider the default mapping of replicas onto 512 nodes of Blue Gene/P running in shared-memory mode (Figure 4.6(a)). Only the mapping for the front plane ($Y = 0$) is shown for ease of understanding. Replica 1 is allocated the block of nodes that constitute the left half of the allocations, whereas replica 2 is allocated the right half. During checkpointing, node i of replica 1 sends a message to node i of replica 2. Using the default mapping, the nodes

are laid out such that the path taken by checkpoints sent by nodes in each column overlaps with the path traversed by checkpoints sent by nodes in their row in every other column. In Figure 4.6(a), this overlap is represented by tagging each link with the number of messages that pass through them during checkpointing. Even if the torus links are considered, the overlap on links exist albeit in lower volume. In effect, the links at the bisection of replica 1 and replica 2 become the bottleneck links; the loads on these bottleneck links are determined by the number of columns. On BG/P, the default mapping is $TXYZ$ in which ranks increase slowest along Z dimension; hence the two replicas are divided along the Z dimension and the load on bottleneck links is proportional to the length of Z dimension.

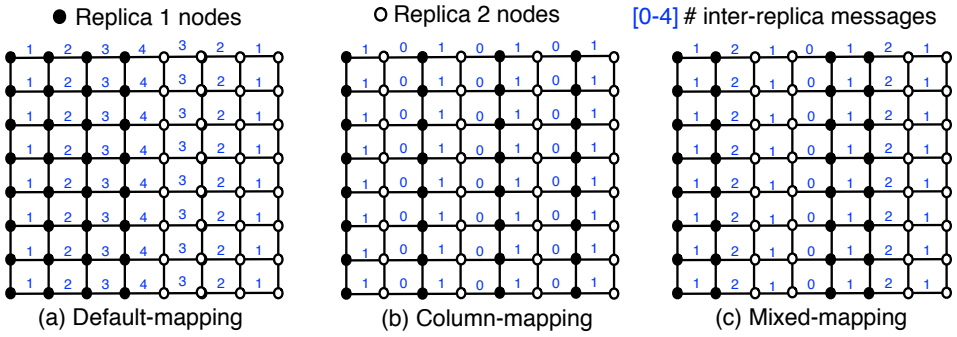


Figure 4.6: Mapping schemes and their impact on inter-replica communication: the number on the links is the number of checkpoint messages that will traverse through those links.

The excess load on the bottleneck link can be reduced by using an intelligent mapping that places the communicating nodes from the two replicas close to each other. Consider the *column*-mapping of the two replicas in Figure 4.6(b) that alternatively assigns the columns (and the corresponding Z planes that are not shown) to replica 1 and replica 2. This kind of mapping eliminates the overlap of paths used by inter-replica messages, and is best in terms of network congestion. However, providing disjoint mapping for a replica may interfere with application communication and result in slow progress for communication-intensive applications. Additionally, placing the buddy nodes close to each other increases the chances of simultaneous failures if the failure propagation is spatially correlated. In such scenarios, one may use *mixed*-mapping in which chunks of columns (and the corresponding planes) are alternatively assigned to the replicas as shown in Figure 4.6(c).

4.5 Modeling Performance and Reliability

A fundamental question when using checkpoint/restart is how often to checkpoint. Frequent checkpoints will imply less work to be recovered in case of a failure, but it will incur high overhead because of the more time spent on checkpoints. This section presents a model to help understand the performance and reliability difference for the three resilience schemes in ACR. The presented model represents a system with a number of parameters and defines several equations to compute relevant values: optimum checkpoint period, total execution time and probability of undetected silent data corruptions. Additionally, the model allows us to understand how ACR will scale and perform in different scenarios. Similar study has also been done by Pradhan et al. on the performance and reliability trade-off of the roll-forward checkpointing scheme [74].

The model extends the theoretical framework presented in the literature [58] by incorporating SDC in the equations, and three different levels of resilience recovery schemes. We assume failures follow the Poisson process. Parameters used in the model are listed in Table 4.1. These parameters include application-dependent parameters (W, δ,R_H,R_S), system-dependent parameters (M_H, M_S, S), and the output of the model (τ, T, T_S, T_M, T_W).

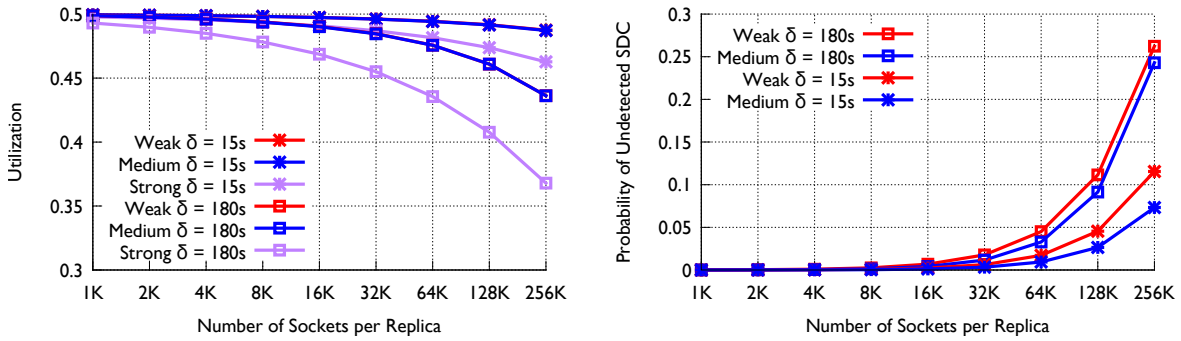
	Description		**Description**
W	Total computation time	τ	Optimum checkpoint period
δ	Checkpoint time	S	Total number of sockets
R_H	Hard error restart time	T	Total execution time
R_S	Restart time on SDC	T_S	T strong resilience
M_H	Hard error MTBF	T_M	T medium resilience
M_S	SDC MTBF	T_W	T weak resilience

Table 4.1: Parameters of the performance model.

Since total execution time is the main variable of interest which we are trying to minimize, we use the following equation to describe the different components:

$$T = T_{Solve} + T_{Checkpoint} + T_{Restart} + T_{Rework}$$

where T_{Solve} is the useful computation time, $T_{Checkpoint}$ is the time spent exclusively on checkpointing, $T_{Restart}$ is the time spent in restarting applications for execution after detecting any type of error, and T_{Rework} stands for the time spent in re-executing the work after both SDC and hard errors. The total checkpointing time is simply the product of the individual



(a) Utilization (b) Probability of undetected SDC

Figure 4.7: The utilization and vulnerability of the different recovery schemes for different checkpoint size. Strong resilience scheme detects all the SDCs but results in a loss of 65% utilization. Weak resilience scheme has the best utilization but is more likely to have undetected SDC for a large δ. Medium resilience scheme reduces the likelihood of undetected SDC with little performance loss.

checkpoint time and the number of checkpoints:

$$\Delta = T_{Checkpoint} = \left(\frac{W}{\tau} - 1\right)\delta$$

The total restart time is similarly the product of individual restart time and the number of restarts:

$$R = T_{Restart} = \frac{T}{M_H}R_H + \frac{T}{M_S}R_S$$

In order to represent the three different levels of resilience defined in Section 4.2, we define an equation for each level. The total execution time for strong resilience level (T_S) uses the fact that a hard error will require the system to rollback immediately to a previous checkpoint. The medium resilience level (whose total execution time is T_M) will checkpoint right after the hard error, so on average, half that checkpoint interval the system is unprotected against SDC. Finally, the weak resilience level (T_W represents the total execution time) will leave the whole checkpoint period unprotected against SDC. The equations for these variables are presented below. As discussed in Section 4.2.3, the application may need to rollback to the previous checkpoint when a hard failure occurs in the healthy replica using the weak resilience scheme. P is the probability for more than one failure in a checkpoint period. Note that this is a loose upper bound on the probability to rollback; we assume that at least one of the multiple failures happens in the healthy replica.

$$P = 1 - \exp(-\frac{\tau+\delta}{M_H})(1 + \frac{\tau+\delta}{M_H})$$

$$T_S = W + \Delta + R + \frac{T_S}{M_H}\left(\frac{\tau+\delta}{2}\right) + \frac{T_S}{M_S}(\tau+\delta)$$

$$T_M = W + \Delta + R + \frac{T_M}{M_H}\delta + \frac{T_M}{M_S}(\tau+\delta)$$

$$T_W = W + \Delta + R + \frac{T_S}{M_H}\left(\frac{\tau+\delta}{2}\right)P + \frac{T_W}{M_S}(\tau+\delta)$$

Using these formulae, we calculate the optimal checkpoint interval for the three resilience schemes and use the best total execution time for further analysis.

Performance and Protection: We define the *utilization* of the system as the portion of the time that is devoted to do useful work: $\frac{W}{T}$. The complement of the utilization is the overhead of the fault tolerance approach. This overhead includes the checkpoint time, restart time, rework time and the utilization loss due to replication. Figure 4.7a shows the utilization of different schemes with different checkpoint time from 1K sockets to 256K sockets per replica. The checkpoint time projected for exascale machine ranges from seconds to minutes [8]. Thus, we choose δ to be 180s and 15s to represent large and small checkpoints respectively (since one of the dominant factors in δ is the checkpoint size [§ 4.6]). We assume a mean time between hard errors M_H of 50 years (equivalent to the MTBF of Jaguar system [75]) and SDC rate of 100 FIT [76]. For δ of 15s, the efficiency for all the three resilience schemes is above 45% even on 256K sockets. When δ is increased to 180s, the efficiency of the strong resilience scheme decreases to 37% while that of the weak and medium resilience schemes is above 43% using 256K sockets. Note that in weak and medium resilience schemes, the system is left without any SDC protection for some period of time. Hence, based on the application, one may have to sacrifice different amount of utilization to gain 100% protection from SDC.

Figure 4.7b presents the probability of occurrence of SDC during the period in which the framework does not provide any protection to silent data corruptions using medium and weak resilience schemes for a job run of 24 hours. The results suggest that for low socket count (up to 16K sockets), the probability of an undetected error is very low for the two types of applications we considered. It is also worth noting that even on 64K sockets, the probability of an undetected SDC for the medium resilience scheme is less than 1% (using δ = 15s). These scenarios may be sufficient to meet the resilience requirements of some users with minimal performance loss. However, the probability of an undetected SDC is high on 256K sockets, and users will have to choose the strong resilience scheme with some performance loss to execute a fully protected experiment. For both the cases, the medium resilience scheme decreases the probability of undetected SDC by half with negligible performance

loss.

4.6 Evaluation

We have used various mini-applications including a stencil-based state propagation simulation, a molecular dynamic simulation, a hydrodynamics simulation using an unstructured mesh, and a conjugate gradient solver to evaluate ACR.

Jacobi3D is a simple but commonly-used kernel that performs a 7-point stencil-based computation on a three dimensional structured mesh. We evaluate our framework using a Charm++ based and an MPI-based implementation of Jacobi3D. *HPCCG* is distributed as part of the MPI-based Mantevo benchmark suite [77] by Sandia National Laboratories. It mimics the performance of unstructured implicit finite element methods and can scale to large number of nodes. *LULESH* is the Livermore Unstructured Lagrange Explicit Shock Hydrodynamics mini-app [78]. It is a mesh-based physics code on an unstructured hexahedral mesh with element centering and nodal centering. *LeanMD* [79], written in Charm++, simulates the behavior of atoms based on short-range non-bonded force calculation in NAMD [80]. *miniMD* is part of the Mantevo benchmark suite [77] written in MPI. It mimics the operations performed in LAMMPS. In contrast to the first four benchmarks, the molecular dynamic mini-apps have low memory footprint. Moreover, owing to their implementations, checkpoint data in these programs may be scattered in the memory resulting in extra overheads during operations that require traversal of application data.

Benchmark	Configuration (per core)	Memory Pressure
Jacobi3D (BGP)	64*64*128 grid points	high
Jacobi3D (Blue Waters)	256*256*256 grid points	high
HPCCG	40*40*40 grid points	high
LULESH	32*32*64 mesh elements	high
LeanMD	4000 atoms	low
miniMD	1000 atoms	low

Table 4.2: Mini-application configuration.

In our experiments, the MPI based programs were executed using AMPI [81], which is Charm++'s interface for MPI programs. Most of the experiments were performed on Intrepid hosted at Argonne National Laboratory. Intrepid is an IBM Blue Gene/P with a 3D-torus based high speed interconnect. In addition, we also performed the experiments using the Jacobi3D benchmark on Blue Waters at National Center for Supercomputing Applications. Blue Waters is composed of both Cray XE and XK nodes with a 3D-torus based

high bandwidth network. The configuration of our experiments can be seen in Table 4.2. The Charm++ and MPI implementation of Jacobi3D used the same configuration in our experiments.

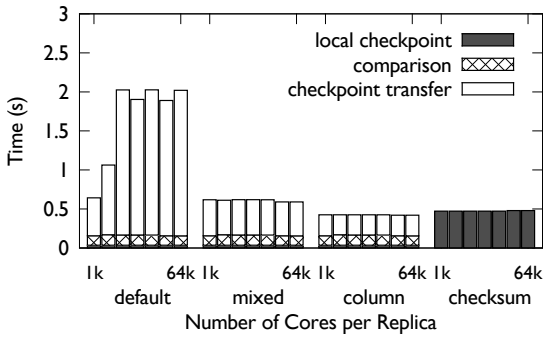
To produce an SDC, our fault injector injects a fault by flipping a randomly selected bit in the user data that will be checkpointed. On most existing systems, when a hard error such as a processor failure occurs, the job scheduler kills the entire job. To avoid a total shutdown, we implement a *no-response* scheme to mimic a 'fail-stop' error. When a hard fault is injected to a node, the process on that node stops responding to any communication. Thereafter, when the buddy node of the this node does not receive heartbeat for a certain period of time, the node is diagnosed as dead.

4.6.1 Forward Path

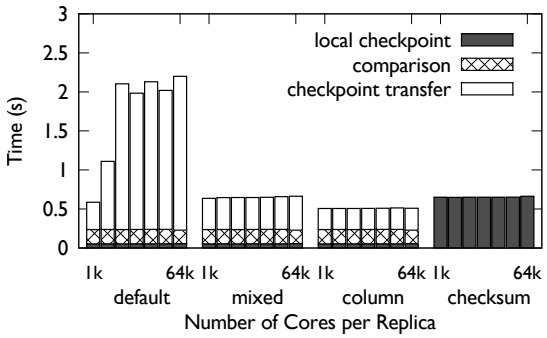
In this section, we analyze the overheads ACR incurred in a failure-free case. These overheads include the time spent in local checkpointing, transferring the checkpoints, and comparing the checkpoints. For these experiments, the system size is varied from 2K cores to 128K cores, i.e. there are 1K to 64K cores assigned for each replica.

Figures 4.8 presents a comparison of the overheads using the default method and with the proposed optimizations for all the mini-apps described. To easily view the change in overheads we graph four of the mini-apps which have higher memory usage on the top and two of the molecular dynamic simulation on the bottom.

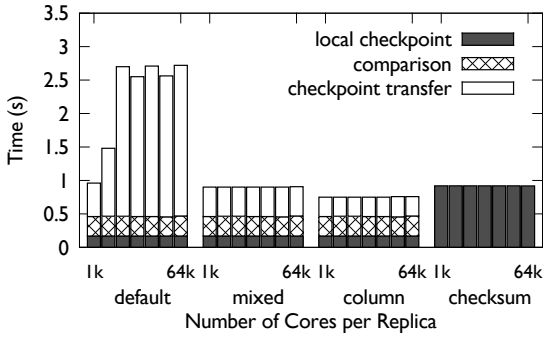
Using the default mapping method, we observe a four-fold increase in the overheads (e.g., from 0.6s to 2s in the case of Jacobi3D) as the system size is increased from 1K cores to 64K cores per replica. By analyzing the time decomposition, we find that the time for inter-replica transfer of the checkpoints keeps increasing while the time spent on local checkpointing and comparison of checkpoints remains constant. An interesting observation is the linear increase of the overheads from 1K to 4K cores and its constancy beyond 4K cores. This unusual increase and steadiness is a result of the change in the length of the Z dimension in the allocated system which determines the load on the bisection links between replica 1 and replica 2 (Section 4.4.2). As the system size is increased from 1K to 4K cores per replica, the Z dimension increases from 8 to 32, after which it becomes stagnant. Beyond 4K cores, only the X and Y dimensions change but they do not have any impact on the performance. We make use of the mapping schemes proposed in Section 4.4.2 to eliminate the dependence of overheads of the default method on the length of Z dimension. Figure 4.8 shows that column and mixed mappings help reduce the inter-replica communication time



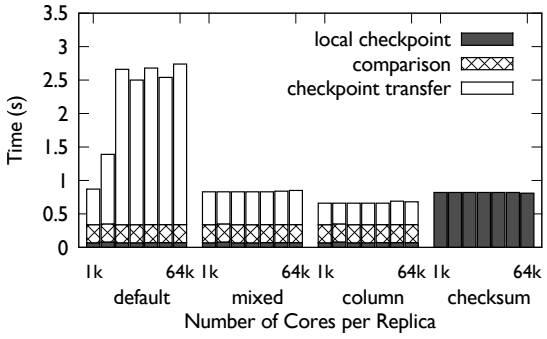
(a) Jacobi3D Charm++



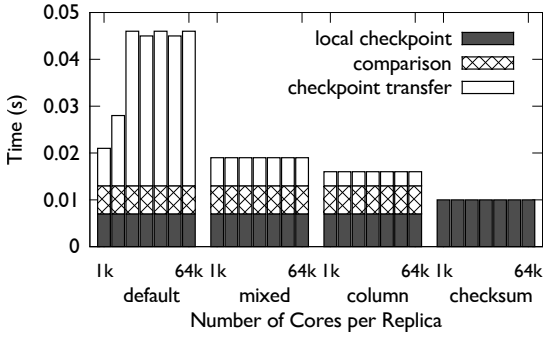
(b) Jacobi3D AMPI



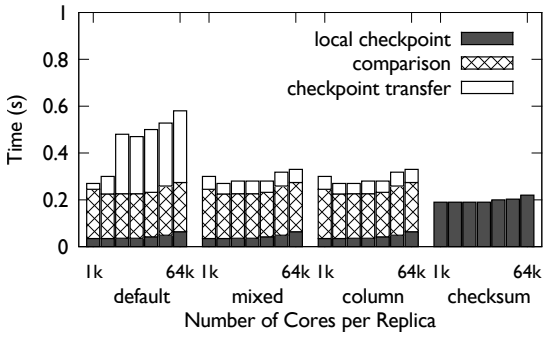
(c) LULESH



(d) HPCCG



(e) LeanMD



(f) miniMD

Figure 4.8: Single checkpointing overhead on BGP. Our framework incurs minimal overheads and provides scalable error detection.

significantly, enabling the full checkpoint-based error detection method to incur a constant overhead. Moreover, no significant performance difference was found for applications using column or mixed mappings when compared to the default mapping.

In contrast, the overheads incurred using checksum based error detection method remain constant irrespective of the mapping used. Most of the time is spent in computing the checksum with trivial amount of time being spent in checksum transfer and comparison as expected since the checksum data size is only 32 bytes. Note that, due to extra computation cost one has to pay for computing checksum, overheads for it are even larger than the column-mapping for high memory pressure applications. Compared to the other three memory consuming mini-apps, LULESH takes longer time in local checkpointing since it contains more complicated data structures for serialization.

Figure 4.8e and 4.8f present the checkpointing overheads for the molecular dynamic mini-apps. While the general trend of results for these mini-apps is similar to the high-memory-pressure mini-apps, the effect of small size of checkpoints and scattered data in memory results in some differences. First, gains in eliminating the overhead due to use of optimal mappings are lower in comparison to the high memory pressure mini-apps. Secondly, only 20% of the time is spent in remote checkpoint transfer with optimal mapping while for the first four mini-apps checkpoint transfer costs around 50% of the time. Thirdly, the checksum method outperforms other schemes though the absolute time is now in $100 - 200$ms range.

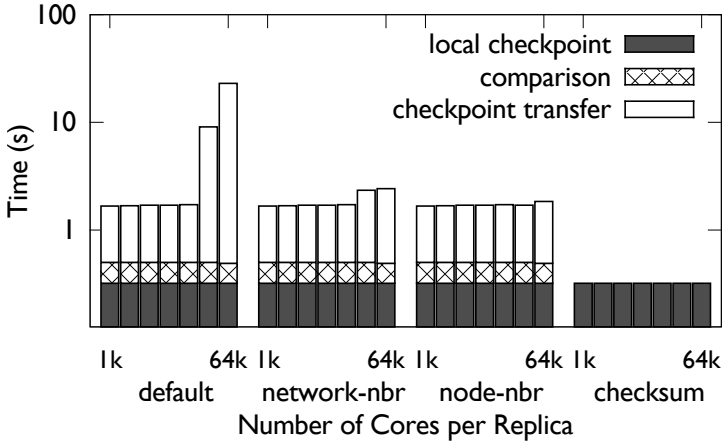
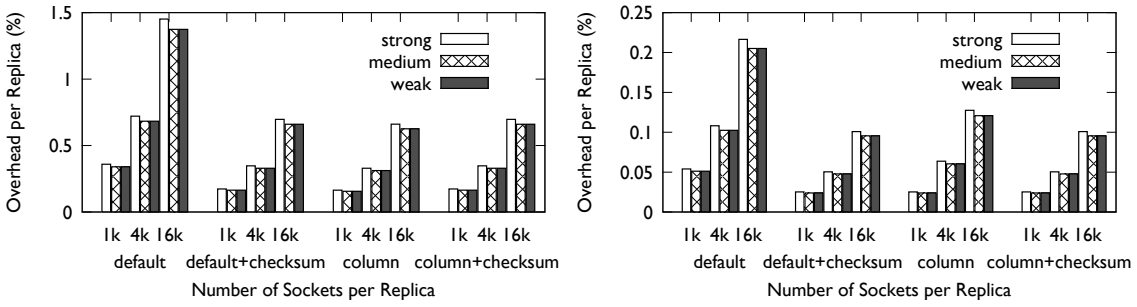


Figure 4.9: Single checkpointing overhead for Jacobi3D on Blue Waters.

Figure 4.9 shows the time to perform one checkpoint for the Jacobi3D benchmark on Blue Waters. Before the topology aware scheduling (in production since 01/13/16) was introduced on Blue Waters for scheduling jobs, we encountered long delays in checkpointing to buddy across partitions due to the arbitrary allocation obtained. With the new topology



(a) Jacobi3D Charm++ (b) LeanMD

Figure 4.10: ACR forward path overhead.

aware scheduling scheme, the checkpointing overhead has been reduced significantly. Since Blue Waters has higher available memory per node in comparison to Intrepid, we ran the Jacobi3D benchmark with a larger problem size as specified in Table 4.2. On Blue Waters, the allocation assigned to a given job may not be a regular grid, which prevents use of mapping schemes developed for Intrepid. Instead, to suit the needs of Blue Waters, we developed two new similar schemes - *network-nbr* and *node-nbr*. In the node-nbr scheme, using the ordering of the node provided by the job scheduler, alternate nodes are assigned to different partitions. Assuming that the job scheduler orders the node to provide good performance, this scheme is likely to reduce network contention. However, since two nodes are available per Gemini router on Blue Waters, the node-nbr scheme assigns two nodes attached to the same router to different partitions in many cases. This is not a good scenario since failure of one node on a router is highly likely to affect the other node. Hence, the alternate scheme we developed (network-nbr) ensures that alternate nodes from different routers are assigned to different partitions. In Figure 4.9, it can be seen that use of topology aware schemes allows the ACR checkpoint time to weak scale as the node count is increased to thousands of nodes.

Figure 4.10 shows the checkpoint overhead of ACR for Jacobi3D and LeanMD when checkpointing at the optimal checkpoint interval according to the model in Section 4.5. The MTBF for hard error used in the model is 50 years per socket while the SDC rate per socket is estimated as $10,000$ FIT. The low checkpoint overhead enables us to checkpoint more often to reduce the rework overhead. The optimal checkpoint interval for Jacobi3d and LeanMD is 133s and 24s on 16K cores with default mapping. Use of either checksum or topology mapping optimization can bring further down the low checkpointing overhead (1.5%) of default mapping by 50%. Overhead of using strong resilience scheme is slightly higher than overhead of weak and medium resilience schemes; this is because applications using strong

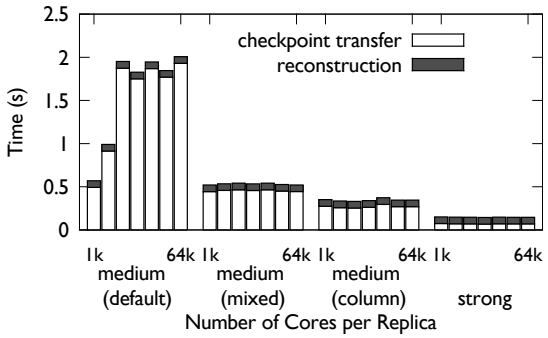
resilience scheme need to checkpoint more frequently to balance the extra rework overhead on hard failures. As the failure rate increases with the number of sockets in the system, forward path overhead also increases.

4.6.2 Restart from Errors

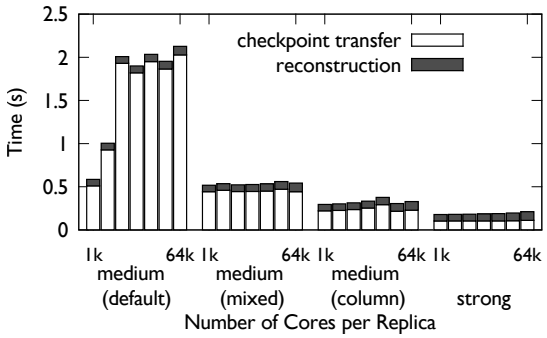
Figure 4.11 presents the restart overhead of strong and medium resilience schemes with different mappings. The restart overhead for hard errors includes the time spent on getting the checkpoints from the replica and the time to reconstruct the state from the checkpoint. After an SDC is detected, every node rolls back using local checkpoint without checkpoint transfer, so the restart overhead for SDC is equivalent to the reconstruction part of restarting from hard errors. The only difference between medium and weak resilience is whether an immediate checkpoint is needed (we found the overhead of scheduling an immediate checkpointing to be negligible). Thus the restart overhead is the same for both cases, hence the restart overhead for only medium resilience scheme is presented.

Figure 4.11 shows that the strong resilience scheme incurs the least restart overhead for all the mini-apps. Two factors help strong resilience scheme outperform the other two schemes- i) the checkpoint that needs to be sent to the crashed replica already exists, and ii) only the buddy of the crashed node has to send the checkpoint to the spare node. Since there is only one inter replica message needed to transfer checkpoints in the strong resilience scheme, we found that mapping does not affect its performance. In comparison, for the medium and the weak resilience schemes, every node in the healthy replica has to send the checkpoint to its buddy in the crashed replica. The simultaneous communication from all the nodes results in network congestion similar to what we saw during checkpointing phase (§ 4.6.1): the time increase comes from the checkpoint transfer stage as shown in Figure 4.11. We make use of topology aware mapping to address the congestion problem and bring down the recovery overhead from 2s to 0.41s in the case of Jacobi3D for the medium resilience schemes. Similar results were found for the other benchmarks with relatively large checkpoints.

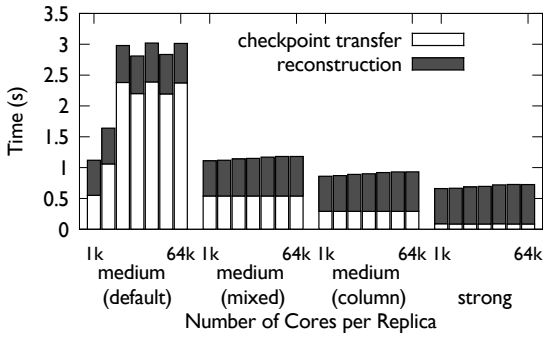
For LeanMD, which has a small checkpoint, the overheads are presented in Figure 4.11e. Note that unlike checkpointing, the restart of a crashed replica is an unexpected event. Hence it requires several barriers and broadcasts that are key contributors to the restart time when dealing with applications such as LeanMD whose typical restart time is in tens of milliseconds. Figure 4.11e shows these effects with a small increase in reconstruction time as the core count is increased. Further inspection confirms that the extra overheads can be attributed to the synchronization costs.



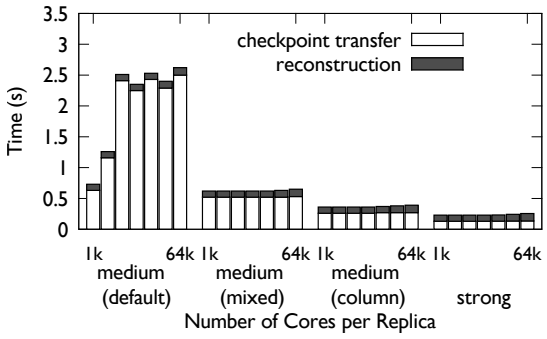
(a) Jacobi3D Charm++



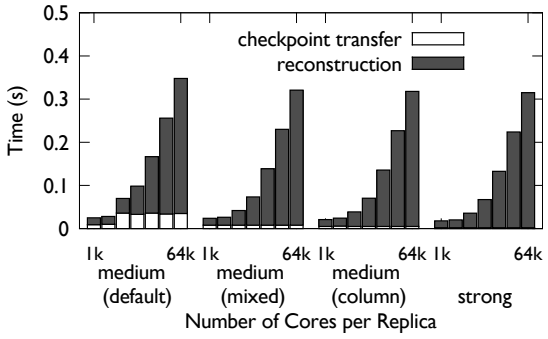
(b) Jacobi3D AMPI



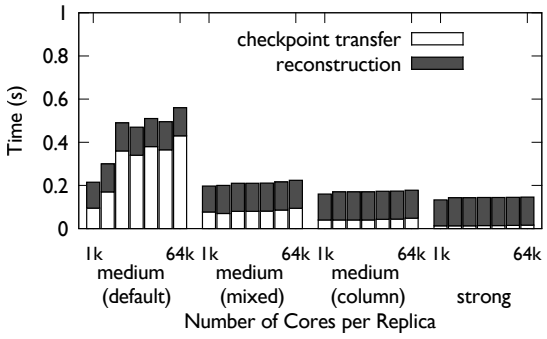
(c) LULESH



(d) HPCCG

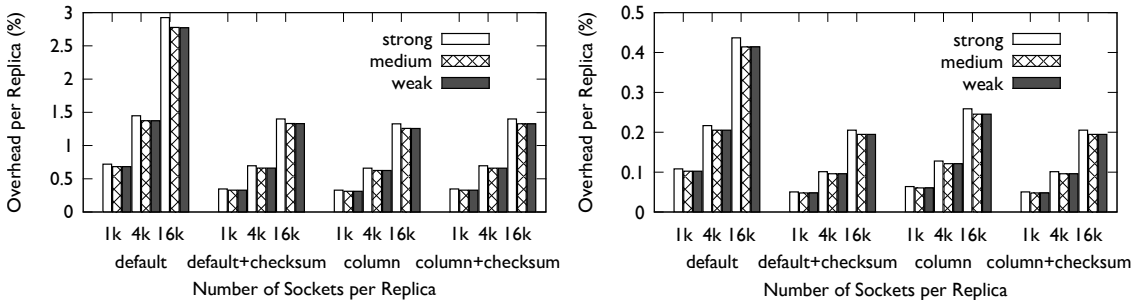


(e) LeanMD



(f) miniMD

Figure 4.11: Single restart overhead on BGP. Strong resilience scheme benefits because of the smaller amount of checkpoint data transmitted.



(a) Jacobi3D Charm++ (b) LeanMD

Figure 4.12: ACR overall overhead.

Figure 4.12 shows the overall overhead of ACR which includes the restart and checkpointing overhead for Jacobi3D and LeanMD at their optimal checkpoint interval. It follows similar trend as shown in Figure 4.10; the overall overhead is larger than checkpointing overhead alone because of the time spent in recovering from hard failure and SDC. Although restarting is faster using strong resilience as shown in Figure 4.11, the extra checkpointing overhead and extra time spent re-executing the work lost due to hard failures makes it worse compared to weak and medium resilience no matter which optimization techniques are used. Regardless, *the overhead of strong resilience is less than* 3% *for Jacobi3D and around* 0.45% *for LeanMD. Using optimizations, the overall overhead is further reduced to* 1.4% *and* 0.2%. As such, ACR performs well in comparison to other libraries such as SCR with overhead of 5% [25].

4.6.3 Adaptivity

As discussed in Section 4.2.2, ACR can dynamically schedule checkpoint based on the failure behavior. In order to test ACR's capability to adapting to the change of failure rate, we performed a 30 minutes run of Jacobi3D benchmark on 512 cores of BGP with 19 failures injected during the run. The failures are injected according to Weibull process with a decreasing failure rate (shape parameter is 0.6). Figure 4.13 shows the timeline profile for this run. The red part is the useful work done by application. Black lines mean a failure is injected at that time and white lines indicate that a checkpoint is performed. As can be seen in the figure, more failures are injected at the beginning and the failure rate keeps decreasing as time progresses. ACR changes the checkpoint interval based on the current observed mean time between failures. Accordingly, it schedules more checkpoints in the beginning (checkpoint interval is 6s) and fewer at the end (checkpoint interval increases to

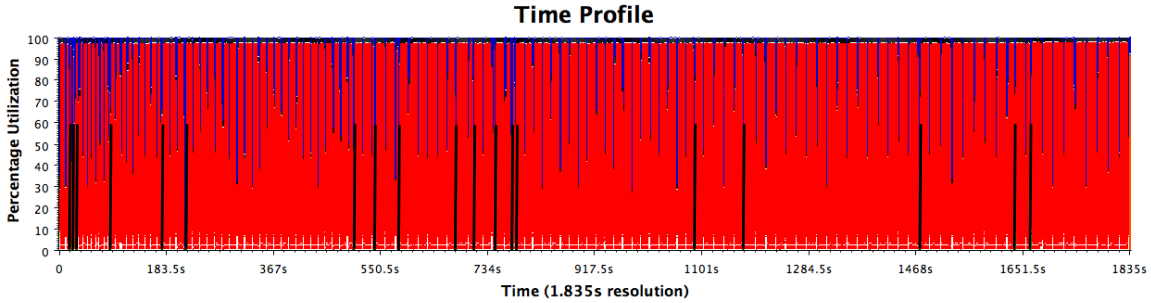


Figure 4.13: Adaptivity of ACR to changing failure rate. Black lines show when failures are injected. White lines indicate when checkpoints are performed. ACR schedules more checkpoints when there are more failures at the beginning and fewer checkpoints towards the end.

17s).

4.7 Summary

This chapter introduced ACR, an automatic checkpoint/restart framework to make parallel computing systems robust against both silent data corruptions and hard errors. ACR uses 50% of a machine's resources for redundant computation. Such investment is justified in making ACR a general purpose solution for silent data corruptions and in having a resilient solution that outperforms traditional checkpoint/restart in high failure-rate scenarios. ACR aims to automatically recover applications from failures and automatically adjust the checkpoint interval based on the environment. ACR supports three recovery schemes with different levels of resilience. We built a performance model to understand the interaction of SDC and hard errors and explore the trade-off between performance and reliability in the three schemes.

We described the design and implementation of ACR in an established runtime system for parallel computing. We showed the utility of topology aware mapping implemented in ACR, and its impact on the scalability. ACR was tested on a leading supercomputing installation by injecting failures during application execution according to different distributions. We used five mini-apps written in two different programming models and demonstrated that ACR can be used effectively. Our results suggest that ACR can scale to $131,072$ cores with low overhead.

CHAPTER 5

Automatic Targeted Protection Against Silent Data Corruption

In the previous chapter, we showed that redundancy and duplication can be used to tolerate SDCs [61]. However, due to duplication, the maximum efficiency achievable in these approaches in comparison to the base version is 50%. This chapter aims at addressing this limitation of duplication-based approaches by exploiting datatype-specific methods for detection and correction SDCs at low cost.

Our work is motivated by the following observation made in [82]: an application execution on a system is reliable if it satisfies two conditions – 1) it computes the data correctly, i.e. the execution performs all the work in the right order and 2) it computes the correct data, i.e. when the expected work is performed, no errors are made. In a typical HPC application, different types of data may be critical for carrying out these two steps and for determining their correctness. Additionally, different methods may be more effective and efficient for protecting different types of data. We explore these possibilities and show that significant reliability and performance improvements can be obtained by using datatype-specific methods.

Broadly speaking, as shown in Figure 5.1, we divide the data of an application into two categories: *control* data and *field* data. Control data, as its name suggests, is used to determine the flow of a program, e.g. an iteration counter, number of message receives, or rank of neighboring process. A corruption in control data can make the program perform incorrect computation, communicate with wrong processes, or even crash. Hence, SDCs in them should be detected and corrected as soon as possible. On the other hand, field data is used in most of the computation work and typically leads to incorrect output. Application-specific methods for ensuring reliability of field data may be sufficient and faster.

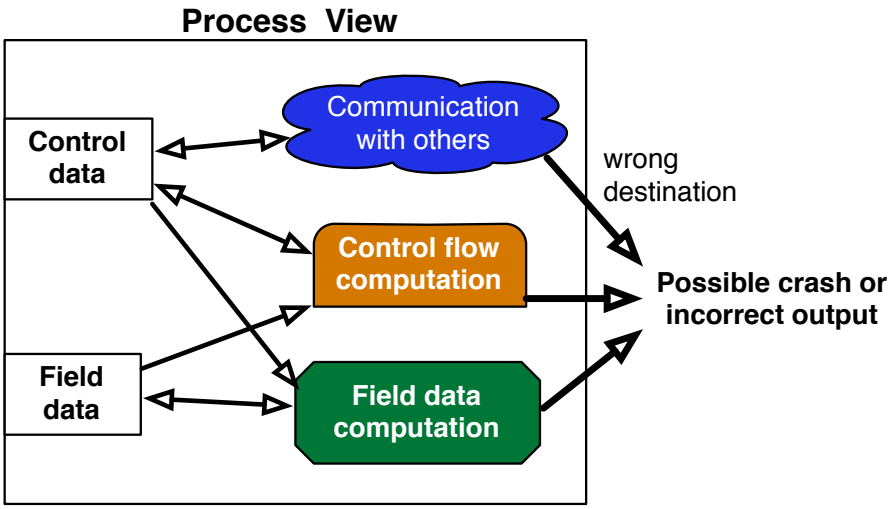


Figure 5.1: Different types of data and their effect on reliability.

Building upon the distinction between the control data and the field data, we have developed a framework, called FlipBack, that provides automated targeted protection against SDC. The main contributions of our work that are described in detail in the rest of this chapter are:

- We present the design of a novel framework that automatically adjusts the protection mechanism based on the data and calculation types (§ 5.2, § 5.3, § 5.4).
- We demonstrate important roles played by the compiler and the runtime system in improving applications reliability (§ 5.2, § 5.3).
- We evaluate FlipBack with two proxy applications and show that 80% to 100% soft error coverage can be provided by the framework with additional 6% to 20% performance overhead.

5.1 Related Work

Software instruction duplication: SWIFT [83] is a fully software-based instruction duplication technique that leverages unused instruction level parallelism to schedule duplicated computation. SWIFT provides good failure coverage except when bit flips occur between verification and actual use of data or when bit flips change the opcode of non-store instructions to store. However, it doubles the number of dynamic instructions and thus leads to

significant cost in terms of performance and power. Shoestring [82] enhances SWIFT with symptom-based detection and only applies instruction duplication to the code segments that are more vulnerable to silent data corruptions. Compared to FlipBack which is a more generic approach based on the characteristics of HPC applications, Shoestring relies on hardware-specific knowledge. HAUBERK-NL [84] duplicates the non-loop computation in GPU programs and applies value-ranging check to protect the loop portions. IPAS [85] uses machine learning techniques to find the instructions that are more likely to cause silent data corruptions and only protect those instructions by duplication. However, it requires extra training time that can be substantially more costly than a compiler and runtime-based approach.

Redundancy techniques: Redundancy techniques have been deployed in both hardware and software. IBM S/360 [86] and HP NonStop systems [87] use large scale modular redundancy to provide fault tolerance. Watchdog [88] uses an extra processor to detect errors by monitoring the behavior of the main processor. ACR [61] enhances the traditional checkpointing scheme with replication to detect and correct both hard failures and silent data corruptions. RedMPI [64] compares messages received by the original and replicated MPI processors in order to detect silent data corruptions.

Anomaly based detection: Based on the characteristics of application data, different techniques have been proposed to compare the application data with expected values in order to detect silent data corruptions. Detection based on temporal and spatial similarity of application data has been proposed and explored in various ways. Yim et al. [89] computes a histogram of application data to detect outliers in conjunction with temporal and spatial similarity. Sheng et al. [90] have designed a detector that can select the best-fit prediction method during execution and automatically adjust the detection range according to the false-positive events observed. Bautista-Gomez et al. [91] use multivariate interpolation to detect and correct silent data corruptions in stencil applications. Subasi et al. [92] uses support vector regression to explore spatial features of neighbor data for detection of silent data corruption. All these different anomaly based detection techniques can be easily plugged into FlipBack to improve its failure coverage.

Algorithm based fault tolerance: Specially designed algorithms can protect different computation kernels with low cost and high detection coverage. Examples of such algorithms are [93] for sparse linear algebra, [94] for iterative solvers and [95] for LU factorization. However, this approach is not applicable to every application and requires significant development time. In comparison, FlipBack is a generic approach that can be used for a wide variety of of HPC applications.

5.2 Runtime Guided Replication

FlipBack is built upon the concept of datatype specific protection from silent data corruptions. It views the application data as a collection of control data and field data. For each of these data types, it deploys different mechanisms to protect them against silent data corruptions. In this section, we describe how *runtime guided replication* can protect the data and computation related to program control flow. In Section 5.3, we discuss the method that is used to protect transient data followed by Section 5.4 in which techniques to protect field data are introduced.

5.2.1 Motivating Examples

Figure 5.2 shows two code snippets written in Charm++ that represent common use cases for HPC programs. Figure 5.2a consists of a stencil computation in which relaxation-updates are applied on grid-based data. The control flow of this program is as follows: at the beginning of every iteration (*beginNextIter*), each chare (C++ object) checks if the iteration count (*iterCount*) has reached the total iterations desired (*totalIter*) and if the program should exit. If not, each chare sends the ghost data to its neighbors to begin the next iteration. At this point, all chares wait to receive messages from their neighbors. When a message arrives, the ghost data received is processed (*receiveMessage*). When all the ghost messages are received, computation is triggered (*invokeComputation*) and the control is passed to the *beginNextIter* function once computation is done.

In stencil computation described above, certain pieces of data used for calculation play an important role to ensure that the program flow is correct. For example, if the value of *msgCount* is modified due to bit flips, the program may hang. This is because some processors may now expect to receive more number of ghost messages than they should, and thus lead to a logical error. Similarly, if a bit flip happens during the computation of *iterCount*, the program may either execute fewer or more iterations than originally specified. Finally, if the calculation of the message destination is corrupted, the program may either compute using the wrong ghost data or may hang because of the incorrect delivery of messages. We refer to variables like *msgCount* and *iterCount* as *control variables* since their correctness effects the program's control flow significantly.

Although the molecular dynamics simulation program shown in Figure 5.2b has a different control flow structure in comparison to stencil computation, we still observe that certain calculations are more vulnerable than others. For example, the calculation of the destination where each particle should migrate to (*id*) is essential for correctness. If corruption affects

```
void Stencil::beginNextIter()
{
  iterCount++;
  if(iterCount >= totalIter){
    mainProxy.done(); //program exits
  }else{
    for(int i = 0; i < totalDirections; i++)
    {
      ghostMsg * m1 = createGhostMsg(dirs[i]);
      copy(m1->data, boundary[i]);
      int sendTo = myIdx+dirs[i];
      stencilProxy(sendTo).receiveMessage(m1);
    }
  }
}
void Stencil::receiveMessage(ghostMsg * m)
{
  msgCount++;
  processGhostMsg(m);
  if(msgCount == numMsgExpected){
    msgCount = 0;
    thisProxy(self).invokeComputation();
  }
}
void Stencil:invokeComputation()
{
  //computation routine
  for(int i = 0; i < size; ++i){
    temperature[i] = ...
  }
  thisProxy(self).beginNextIter();
}
```

(a) Stencil computation

```
void Cell::compute()
{
  for(each p in particles){
    computeTotalForce(p);
  }
  cell(self).startMigration();
}
void Cell::startMigration()
{
  for(each p in particles){
    int id = findOwner(p);
    if(id != myID){
      cell(id).migrateParticl(p);
    }
    deletedParticles.push_back(p);
  }
}
```

(b) Molecular dynamics simulation

Figure 5.2: Code snippets from scientific applications.

such calculations, particles may move to the wrong chare and soon silent data corruptions will propagate through the entire system.

What we learn from these code examples is that protection of the computation involving control variables and message destinations is necessary to ensure that the program executes the right computation. Hence, to protect these code regions from silent data corruptions, we *conduct recomputation* using the same inputs. The results thus obtained are compared with the control variables and message destinations obtained from the original computation. If there is a mismatch, we detect that silent data corruption has happened.

5.2.2 Compiler Slicing Pass

Re-executing all the computation is very expensive since it doubles the execution time of the program and potentially doubles the memory requirement as well. In order to reduce the recomputation time and memory overhead, FlipBack introduces a *slicing pass* into the standard compiler backend to limit the recomputation scope. A program slice is part of the program that affects the values computed at some points of interest. In the slicing pass, FlipBack first identifies the instructions that modify the value of either a control variable or variables that affect communication. We refer to those instructions as *critical* instructions. Next, FlipBack finds all other instructions that directly or indirectly affect the *critical* instructions. In this way, all the instructions whose execution can potentially affect the computation of control variables are discovered.

Having identified the program slice for control variables and message destinations, recomputation is performed only on those instructions. We have found that this reduces the overheads of recomputation significantly. Moreover, only control data and messages need to be duplicated for recomputation. Typically, the memory space used by control data and messages is much less than the other data used in scientific computation. Thus, the slicing pass not only reduces the computation time but also relieves memory pressure.

Algorithm 1 shows how the slicing pass works. First the slicing pass finds the slicing criterions: instructions that either assign values to control variables or send messages (line $1 - 5$). Using the example shown in Figure 5.2a, the code on lines 3, 5, 12, 18, 21, 22, and 31 will be selected as slicing criterions after this step. Next, in order to find all the instructions that may affect the instructions in the slicing criterions, we conduct backward data flow analysis and control flow analysis. To better understand the process of data flow analysis, let us look at line 12 in Figure 5.2a. According to Algorithm 1, first we need to find the values used in that instruction (line 10), which are *sendTo* and *m1*. The definitions

```
   Input:
   f: the targeted function to perform slicing on
   c: set of control variables
   Output: slices: the program slice for recomputation
   // search for slicing criterions
 1 foreach Instruction I in f do
 2 |   if Defs(I) ⊂ c or I sends messages then
 3 |   |   criterions.push(I);
 4 |   end
 5 end
 6 while !criterions.empty() do
 7 |   I←criterions.top(); criterions.pop();
 8 |   if !I.processed() then
 9 |   |   slices.push(I);
   |   |   // data flow analysis
10 |   |   foreach Values I' in Uses(I) do
11 |   |   |   foreach Instruction I'' in Defs(I') do
12 |   |   |   |   if I'' may lead to I then
13 |   |   |   |   |   criterions.push(I'');
14 |   |   |   |   end
15 |   |   |   end
16 |   |   end
   |   |   // control flow analysis
17 |   |   foreach BasicBlock B that may lead to I do
18 |   |   |   criterions.push(B.getTerminator());
19 |   |   end
20 |   end
21 end
```

Algorithm 1: Slicing pass to find recomputation region.

of *sendTo* and *m1* are at line 11 and 9 respectively in Figure 5.2a. Since line 12 is in the execution path from either line 11 or 9 to the end of the function, those two new lines are added to the *criterions* and *slicing* set. Similarly, the definition of i in line 7 is also included in the *slicing* set due to the data flow analysis. Control flow analysis can be illustrated using line 21 in Figure 5.2a. Since branch instruction at line 20 acts as the terminator instruction of the basic block that leads to the execution of line 21, it is also included in the *slicing* set.

Note that although line 10 in Figure 5.2a should be part of the slice according to Algorithm 1, we prune it since the content of the message depends on non-control variables that can be protected using other methods. A limitation of the presented slicing tool is that inter-procedure and pointer analysis is not supported.

```
1  //mark control and field data
2  addControl(&msgCount);
3  addControl(&iterCount);
4  addField(&particles);
5  ------------------------------
6  //annotate important members in messages
7  class ghostMsg
8  {
9    control int direction;
10   double * data;
11 }
```

Figure 5.3: Annotations of the control and field variables.

5.2.3 Runtime Support

The ideas on which FlipBack is based are broadly applicable. However, as a proof of concept, we have chosen Charm++ to implement a prototype of our ideas for two reasons. First, the notion of chares is highly suitable for containing faults. Since a chare only modifies the data that it owns, if the silent data corruptions can be detected and corrected within a chare, their propagation to the rest of the application data can be avoided. Second, since the runtime system manages what work is executed on which chares at what time, it provides opportunities to insert hooks for FlipBack to detect and correct SDCs.

The runtime system (RTS) component of FlipBack has multiple roles. It is responsible for local checkpoint/recovery, recomputation and verification. When a chare is created, FlipBack automatically creates a corresponding "shadow" chare that is invisible to users for recomputation. Shadow chares only execute the program slice obtained from the slicing compiler pass. Thus, another compiler pass is needed after the original slicing pass to prevent shadow chares from executing instructions that do not belong to the slice by adding conditional instructions. FlipBack allows users to mark control and field data using a simple API as shown in Figure 5.3. The RTS creates shadow chares using default constructors and then copies the value of control variables from the original chare. As for the field data, RTS ensures that shadow chares and the original chares point to a common copy of the variables.

Algorithm 2 shows the interaction between the original and shadow chares in the runtime system. Broadly speaking, the execution model is transaction based at the level of entry method executions. Actions that affect other chares can only be committed once it is confirmed that there are no silent data corruptions. As shown in Algorithm 2, when a message M targeted at the original chare is received, the RTS first checkpoints the control variables in both the original chare and the shadow chare (line $1-2$). Next, the RTS invokes the entry

```
   Input: o: the original full fledged chare
   s: the shadow chare
   // RTS receives a message M for o
 1 checkpointControl(o);
 2 checkpointControl(s);
 3 restart←true;
 4 while restart do
       // buffering outgoing messages
 5     o.invoke(M); s.invoke(M);
 6     if compareControl(o, s) and compareMsgs(o, s) then
 7         restart←false;
 8         sendMsgs(o); deleteMsgs(s);
 9     end
10     else
11         restartControl(o); restartControl(s);
12     end
13 end
```

Algorithm 2: Workflow in RTS.

method associated with M on the original chare. During the execution of the original chare, all the outgoing messages are buffered. Then, the RTS invokes the same entry method on the shadow chare and buffers all the outgoing messages as well. As mentioned earlier, when the entry method is invoked on the shadow chares, only the instructions that are part of the sliced set will be executed.

In the end, control variables and buffered messages obtained from the original chare and the shadow chare are compared (line 6). If there is mismatch between the data from the two sources, the RTS rollbacks both the shadow chare and the original chare to the beginning of the entry method execution using the checkpoints of the control variables, and then repeats the executions. The buffered messages are not sent out until the control variables and buffered messages from the original and shadow chares match with each other.

To enable faster comparison of messages, the RTS also allows users to annotate important members in each message. As shown in Figure 5.3, users can annotate members, such as the integer data *direction*, as a control variable. This results in creation of a customized function that is used by the RTS to only compare the *direction* from the two messages. This scheme is coherent with the slicing pass methodology where the shadow chares do not execute line 10 in Figure 5.2a since it is not needed for comparison. Data corruptions in the *data* field of the *ghostMsg* are protected using techniques for field data described in a later section.

We find that the message driven execution model like Charm++ greatly eases the process of local recovery by separating the control and computation flow into different entry methods.

As can be seen in Figure 5.2a, actions before and after the computation routine *invokeComputation* are encapsulated in entry methods *receiveMessage* and *beginNextIter*. Hence, there is no need to checkpoint the field data *temperature* for the local recovery of computations in *receiveMessage* and *beginNextIter*. As for the entry method *invokeComputation*, we protect it using the techniques described in Section 5.3-5.4.

5.3 Selective Instruction Duplication

Runtime guided replication is not adequate for dealing with certain categories of SDCs. In particular, since the comparison is made only after the entry method finishes execution, errors that are "felt" only during the execution are left unprotected. For example, it is difficult to protect loop variable i in the entry method *invokeComputation* shown in Figure 5.2a using runtime guided replication since its lifetime ends before the completion of the entry method. However, bit flips in i can lead to severe consequences: the program may either crash due to out-of-bound access or cause SDCs because incorrect data is used. FlipBack uses *selective instruction duplication* to protect such transient calculations. More specifically, FlipBack selectively duplicates the integer arithmetic instructions in the code region that are not part of the sliced set to be replicated by shadow chares.

To enable duplication of integer arithmetic instructions, we perform another compiler pass in FlipBack, primarily to protect address calculations used for memory access. In this pass, we first identify all the instructions that calculate the addresses used in load and store instructions. Next, we find the duplication path for those instructions using *use-def* and *def-use* chains, and duplicate every instruction in this path. This step is similar to what has been shown in the previous work [82, 83]. Our current implementation limits duplication path to be within a basic block. Finally, comparison instructions are added at synchronization points, i.e. before store and branch instructions to check for SDCs.

Our approach for instruction level duplication is different from the previous work [82, 83] in that FlipBack performs local recovery immediately if SDC is detected. Figure 5.4 shows a simplified version of the LLVM intermediate representation (IR) code after the selective instruction duplication pass. Line 3 duplicates the original instruction in line 2. Line 4 compares the two execution paths. If there is mismatch, we perform re-computation again in line 7. At line 10, we decide which computation result should be used based on whether SDC has occurred or not. Afterwards, we replace all the future uses of instruction 1 with instruction 7. Note that the underlying assumption we make is that the probability of bit flips occurring at the same place in a short duration is very low.

```
1  ;label 0
2  %1 = add i, 1
3  %2 = add i, 1
4  %3 = icmp eq %1, %2
5  br %3, label %4, label %6
6  ;label 4
7  5 = add i, 1
8  br label %6
9  ;label 6
10 %7 = phi [%1, label %0], [%5, label %4]
```

Figure 5.4: Illustration of the LLVM IR code after selective instruction duplication pass.

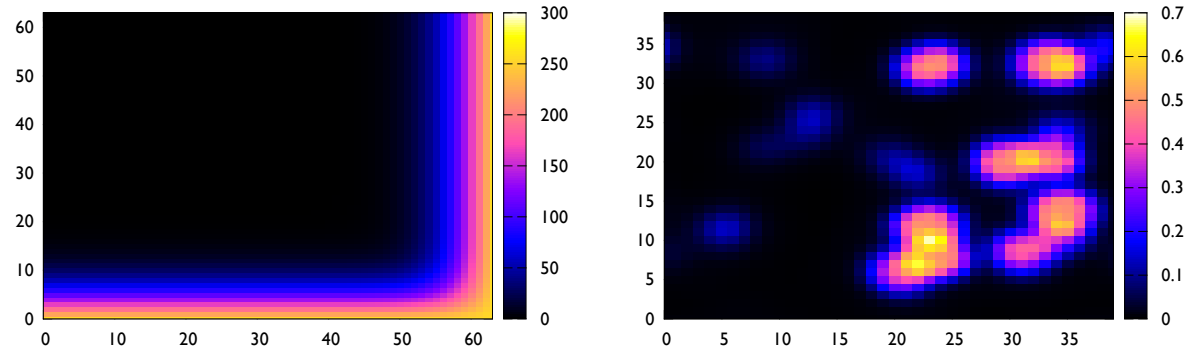


Figure 5.5: Continuity in scientific data.

5.4 Adaptive Protection for Field Data

FlipBack provides several detection routines to protect field data that are typically computed using floating point calculations against SDCs. In general, scientific HPC applications simulate phenomena that occur in the real world, such as particle motion, heat-propagation, climate changes, and fluid dynamics. As a result, the continuity of data found in nature is also observed in a correctly executing scientific program unless bit flips silently change its output. For example, Figure 5.5 shows a heat map for data values arranged as a 2D grid in two applications that perform stencil-based property update and molecular dynamics based on Car-Parrinello method [19]. It is easy to see that gradual changes are observed in data values as a sweep is made through the grid.

FlipBack leverages the aforementioned continuity in the data values and reports anomalies when such a continuity is not observed. Currently, we provide different detectors that exploit different types of data similarity as discussed below. At the beginning of each run, FlipBack tests the application data with all the detectors and selects the one that predicts values most accurately. We refer to this initial stage as the testing phase in the rest of this section.

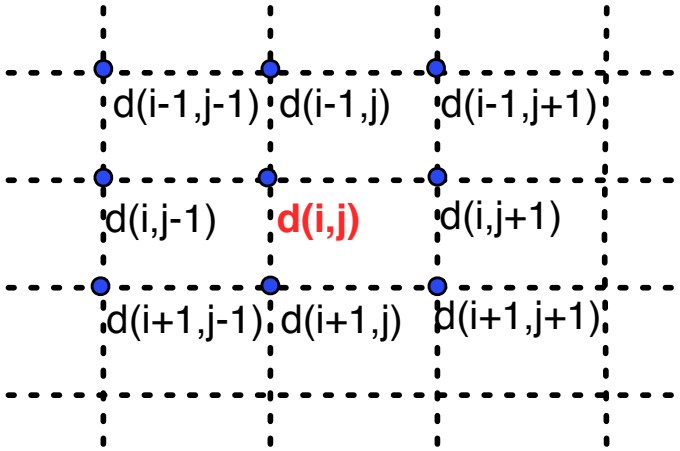


Figure 5.6: Representation of data in 2D grid.

Spatial data similarity: This detector predicts the data value at each point based on the value at its neighbor points. The predicted value is compared with the real value to check if the difference between the real value and the predicted value is within a user-defined range. If the difference is out of the range, FlipBack reports the anomaly to users. In a typical HPC application, data is arranged as a multi-dimensional structured or unstructured mesh. Depending on the application, data values may show higher similarity or better correlation along one dimension in comparison to other dimensions. Thus, in the testing phase, FlipBack first finds the best dimension to use for prediction and then predicts the data values along that dimension.

In order to combine the information from multiple dimension for better prediction, FlipBack utilizes a method that explores the relationship between neighbors points in one dimension through an orthogonal dimension. In the data grid shown in Figure 5.6, to predict the value at point (i, j), FlipBack first studies the relationship of each j^{th} data point with its left $(j-1)$ and right $(j+1)$ neighbors in rows $i-1$ to $i+1$. More specifically, FlipBack first calculates r_{i-1} and r_{i+1}:

$r_{i-1} = \frac{d_{i-1,j-1} - d_{i-1,j}}{d_{i-1,j-1} - d_{i-1,j+1}}$, $r_{i+1} = \frac{d_{i+1,j-1} - d_{i+1,j}}{d_{i+1,j-1} - d_{i+1,j+1}}$

Then it uses r_{i-1} and r_{i+1} to interpolate the corresponding r_i in row i. Next, FlipBack predicts the value at point (i,j) as $p_{i,j} = d_{i,j-1} - (d_{i,j-1} - d_{i,j+1}) * r_i$. Finally, it compares $p_{i,j}$ with $d_{i,j}$ to detect if an anomaly has been found.

Temporal data similarity: This detector studies the temporal evolution of the application data. Assume that the current detection time step is t and data from previous two detection steps is stored (steps $t-k$ and $t-2k$ where detection is done every k time steps).

First, FlipBack computes and records δ as the data difference from time step $t-2k$ to $t-k$. Next, FlipBack uses δ and d^{t-k} (the data value at time step $t-k$) to predict the data value at time step t. If the difference between the predicted value and the real value (d^t) is beyond a predefined range, FlipBack reports the anomaly to users.

Spatial & temporal data similarity: For certain applications, it is hard to predict the expected data values by use of either spatial or temporal locality only. For example, in computation fluid dynamics, when external force is applied at one end of the mesh, accurate prediction of the expected data value based on data from previous time steps is difficult. At the same time, depending on the effect of the force, spatial similarity may not be present in the data. In such scenarios, FlipBack attempts to use both spatially and temporally close data to detect anomalies.

In this method, for each data point d, the first step is to compute if spatial or temporal similarity exists. If neither spatial or temporal similarity exists, the method checks if there is a similarity between the temporal updates made to spatially neighboring data values, i.e. if the update to the value of d since the last time step is similar to the change in the value(s) of the neighboring point(s) of d. For example, in a 1D grid, this step checks if the difference $d^t(i) - d^{t-k}(i)$ is similar to either $d^t(i-1) - d^{t-k}(i-1)$ or $d^t(i+1) - d^{t-k}(i+1)$. Next, if needed, we check if there is a temporal similarity between the difference in the neighboring values: if the difference between the value of d and its neighbor is similar at time step t and $t-k$. For example, in a 1D grid, this step checks if the difference $d^t(i) - d^t(i-1)$ is similar to $d^{t-k}(i) - d^{t-k}(i-1)$. If none of these steps find the data to be as expected, an anomaly is reported to the user.

5.5 Evaluation

We use LLFI [96], a fault injection tool based on LLVM [97], to inject bit flip induced soft errors. LLFI works at the level of LLVM intermediate representation code and injects fault into live registers. The original version of LLFI developed by Wei et al. is for injecting faults into sequential programs only. In order to use it for our experiments, we extend LLFI to work with parallel programs. The parallel version of LLFI randomly selects a processor and injects failure to it during the execution. In this way, we mimic the occurrence of a bit flip on arbitrary processors.

All experiments presented in this section were conducted on Catalyst, a cluster at Lawrence Livermore National Laboratory. It consists of Intel E5-2695 processors with two sockets per node and 12 cores per socket. We use two proxy applications to evaluate the failure coverage

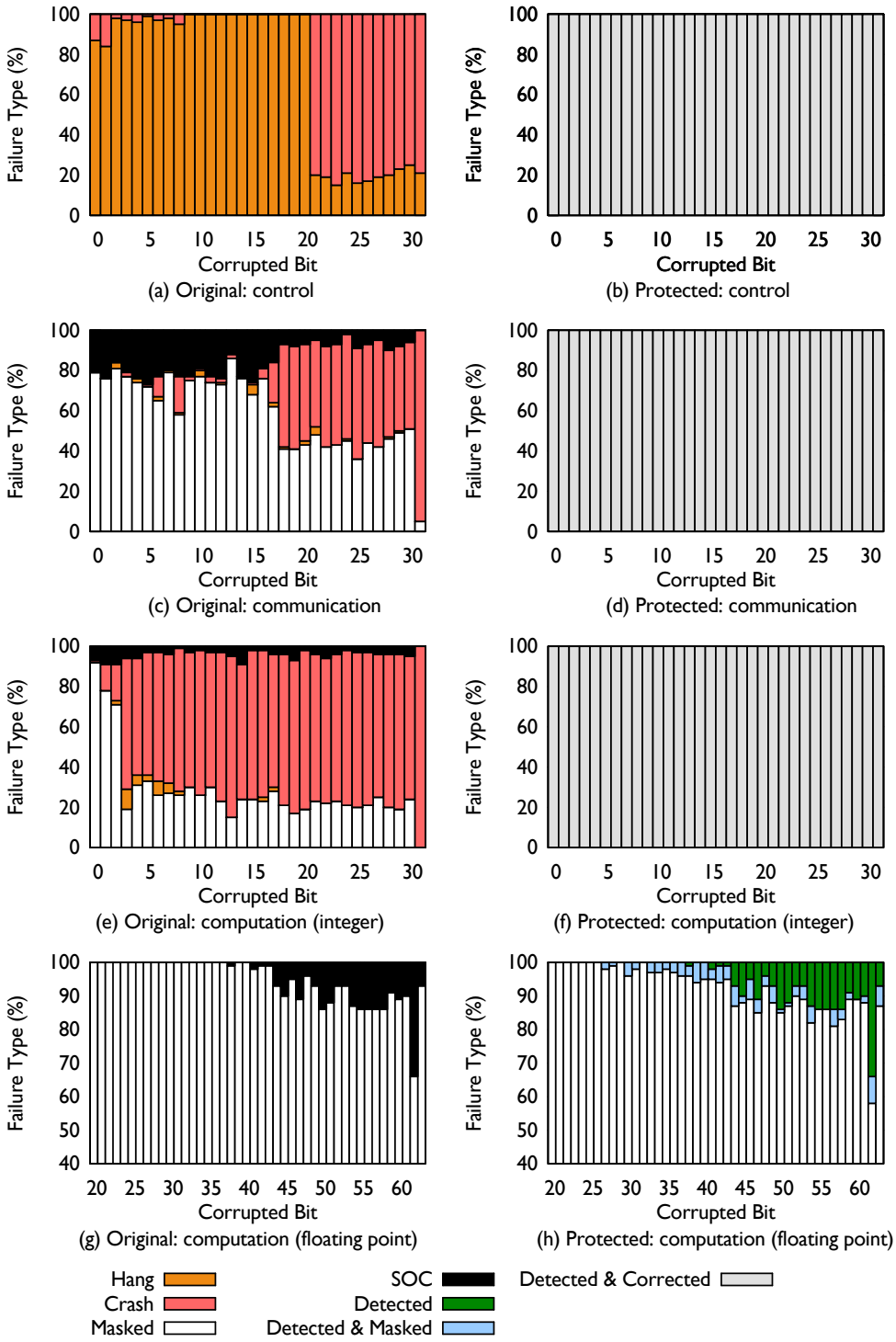


Figure 5.7: Effect of bit flips on **Miniaero: 3d-sod dataset** with and without FlipBack protection.

and performance of FlipBack: Miniaero and Particle-in-cell (PIC). Miniaero [98] is a proxy application from Sandia National Laboratory which solves the compressible Navier-Stokes equations using explicit RK4 method. Particle-in-cell (PIC) is a proxy application from the PRK benchmark suite [99], in which particles are distributed within a fixed grid of charges. In each time step, PIC calculates the impact of the Coulomb potential at neighboring grid points on the motion of every particle. We also use a micro-benchmark, Stencil3D, that performs 7-point stencil-based computation on a 3D-structured mesh to further evaluate FlipBack.

Both Miniaero and PIC provide verification routines that we use to determine if an execution generates correct output at the end of the run. For Stencil3d, output of a given run is compared with the output generated from the failure-free run. If the maximum difference between the two sets of outputs is less than 1%, we deem the execution to be correct.

Silent data corruptions in different parts of the programs manifest themselves in different ways, thus we categorize the failure injection experiments into four sets and study them one by one. The first set impacts communication routines in which message creation and reception occurs. The second set corresponds to the integer calculations in the computation routine. The third one relates to floating point calculations in the computation routine. The last set includes all the remaining code regions which we refer to as the *control* set since the computation in this set can easily affect the program flow. For every bit position that can be flipped, we run the proxy application 100 times, and in each run LLFI selects when and to which register a fault is injected. As a result, we perform $3,200$ to $6,400$ failure injection experiments in total for each set that contains either 32-bit data or 64-bit data.

5.5.1 Miniaero

Figure 5.7 shows how Miniaero reacts to bit flips with and without the protection provided by FlipBack using the *3d-sod* data set distributed with MiniAero. As can be seen in Figure 5.7(a), the program hangs in more than 80% of the cases when failures are injected to the control data related to computation routines and if there is no soft error protection. The primary reason for the hang is the incoherent behavior of different processors caused by the bit flips. When bit flips happen in the higher bits, Miniaero crashes most of the time. This occurs when control data is used to calculate the array access indices and a out-of-bound access causes the crash. Figure 5.7(b) shows the program behavior when the same bit flip pattern is injected into executions protected by FlipBack. It can be seen that with the runtime guided replication and selective instruction duplication in FlipBack, all

the bit flips are captured as soon as they occur and automatic local recovery is performed. As a result, the program runs smoothly without any crash or hang.

Figure 5.7(c) shows the results of the runs in which bit flips are injected into the communication routines. The main work performed in the communication routines of Miniaero is sending and receiving of messages that contain the ghost region. For 10% to 15% cases in which bit flips happen in the communication routines, silent output corruptions (SOC) occur due to sending of wrong data to the neighbors. Silent output corruptions (SOC) means that the output obtained is incorrect. Also, when a large fraction of bit flips happen on higher bits, the execution crashes because of out-of-bound access to the data. Most of remaining executions are not affected by bit flips, i.e. the bit flip is masked. This can happen if the value of the corrupted data is similar to the value of the correct data, and thus the final result is still within the acceptable error tolerance. In Figure 5.7(d) with FlipBack, we are able to detect and correct all the bit flips in communication routines.

Figure 5.7(e) shows the behavior of Miniaero when failures are injected into the integer calculations in computation routines. In majority of these cases, the program crashes due to out-of-bound accesses. Incorrect results are observed due to use of corrupted data for around 10% cases. As before, when used, FlipBack detects and corrects all such corruptions.

In Figure 5.7(g,h), we inject bit flips to floating point calculations of the computation routines in Miniaero. In these plots, note that x-axis, which represents the position of corrupted bit, starts from 20 while y-axis starts from 40. The bit flip injections not shown in Figure 5.7(g,h) are masked, i.e. all bit flips inserted to bits in position 0 to 20 are masked. According to Figure 5.7(g), the higher the position of bit that gets flipped, the higher the chances that silent output corruptions is observed. The most accurate detector for the 3d-sod data set found by FlipBack during the test phase is the *spatial similarity* detector using the y dimension. With that detector, FlipBack is able to detect all the anomalies that lead to incorrect output as shown in Figure 5.7(h). However, in a few experiments, FlipBack detects anomaly but the final result satisfies the verification test shown as *detected & masked* in Figure 5.7(h). This is because over time, the corrupted data slowly converges to a reasonable value using the uncorrupted data from the neighboring points in the data grid.

Although the *spatial similarity* detector works well with the 3d-sod data set, FlipBack found it unsuitable for another data set distributed with Miniaero, the *flat-plate* data set. The best detector for the flat-plate data set is the one that combines both spatial and temporal similarity. Figure 5.8 shows the effect of bit flips on the floating point calculations of the computation routines when Miniaero is executed with the flat-plate data set. It can be seen in Figure 5.8(b) that FlipBack is able to detect more than 80% of the bit flips that

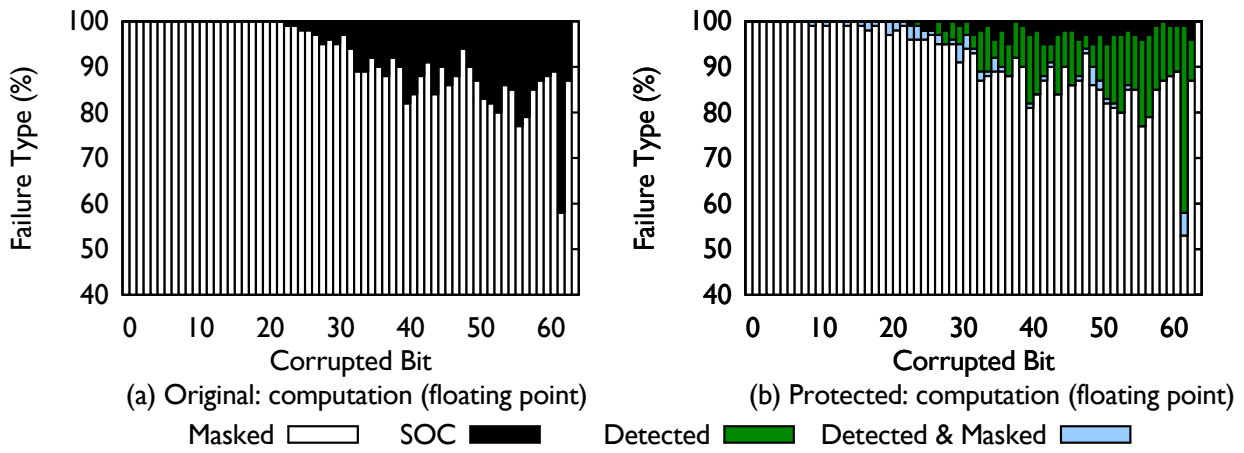


Figure 5.8: Effects of bit flips on **Miniaero: flat-plate dataset** with and without FlipBack protection.

lead to incorrect results.

5.5.2 Particle-in-cell

Figure 5.9 shows the behavior of the Particle-in-cell proxy application when bit flips are inserted with and without protection provided by FlipBack. Figure 5.9(a,c,e) shows that PIC's execution behavior is similar to Miniaero when FlipBack is not used. The only major deviation from the behavior shown by Miniaero is when bit flips are inserted to communication routines. In this case, the bit flips are mostly masked. FlipBack is able to detect and correct all bit flips in the control and communication routines, and in the integer calculations performed in computation routine as shown in Figure 5.9(b,d,f).

For the field data in PIC, the *temporal similarity* detector provides the best results. Figure 5.9(g) shows that PIC is much more sensitive to bit flips in the field data in comparison to Miniaero: more than 70% bit flips lead to incorrect output and 2% of bit flips cause the program execution to crash. The crashes happen because in PIC, each particle uses its position to calculate the grid indices it should interact with. If the position data is corrupted, the calculation of the grid indices is wrong and thus out-of-bound accesses occur and cause crashes. FlipBack detects all the bit flips that lead to SOC as well as bit flips that temporarily cause anomalies but are eventually masked as the simulation continues.

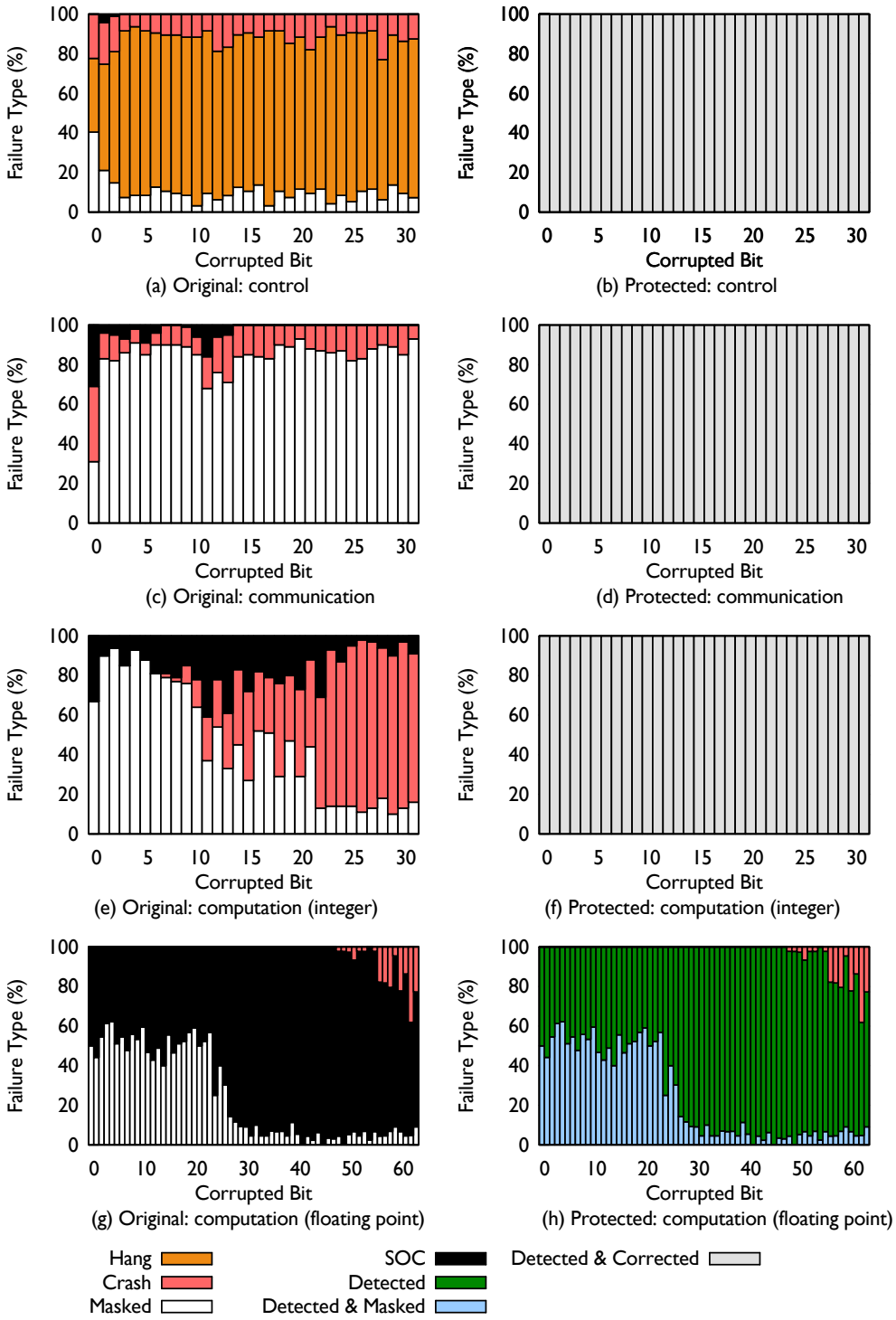


Figure 5.9: Effect of bit flips on **Particle-in-cell** with and without FlipBack protection.

5.5.3 Stencil3d

Figure 5.10 compares how Stencil3d reacts to bit flips with and without using FlipBack. The program behavior is similar to the previous two application with two exceptions. First, bit flips in Stencil's communication routines are more likely to lead to crashes. Second, bit flips during integer calculations of the computation routine causes fewer crashes. Nonetheless, FlipBack is able to detect and correct all the bit flips in the control routine, communication routine and integer calculations part in computation routine as shown in Figure 5.9(b,d,f).

The *spatial similarity* detector along both x and y dimensions works the best with the field data in Stencil3d. Figure 5.10(g) shows that bit flips injected into higher bits of the floating point data are much more likely to induce incorrect results. Among all these bit flips that lead to incorrect results, FlipBack is unable to catch only 1.8% bit flips as shown in Figure 5.10(h).

5.5.4 Performance

Figure 5.11 shows the performance overhead of using FlipBack to protect Miniaero, PIC and Stencil3d from silent data corruptions. Both the proxy applications and the micro-benchmark are strong scaled from 16 cores to 256 cores in these experiments to evaluate FlipBack's performance. Since we are interested in studying the impact of each protection mechanism provided by FlipBack on the application performance, the presented result aggregates the overheads as we move from left to right and add different features. As can be seen from Figure 5.11(a), for Miniaero proxy application, the overhead added by runtime guided replication ranges only from 2.6% to 4.1% as the application is scaled from running from 16 to 256 cores. Adding selective instruction duplication incurs an additional 2% to 4% overhead. The performance degradation caused by field data detector is negligible: less than 1%. Overall on 256 cores, FlipBack increases the execution time of Miniaero by 6% even when all the protection mechanisms enabled.

For PIC, runtime guided replication increases the execution time by around 10% when compared to the baseline case in which no protection mechanism is used as shown in Figure 5.11(b). Selective instruction duplication introduces an additional $2-5\%$ overhead. Adding a detector to protect field data increases the total overhead of FlipBack to $9-18\%$. An interesting observation for PIC is that as the parallel efficiency of the program drops from 128 cores to 256 cores, the performance overhead introduced by FlipBack decreases. This is because FlipBack utilizes the idle time such as the time spent on communication to

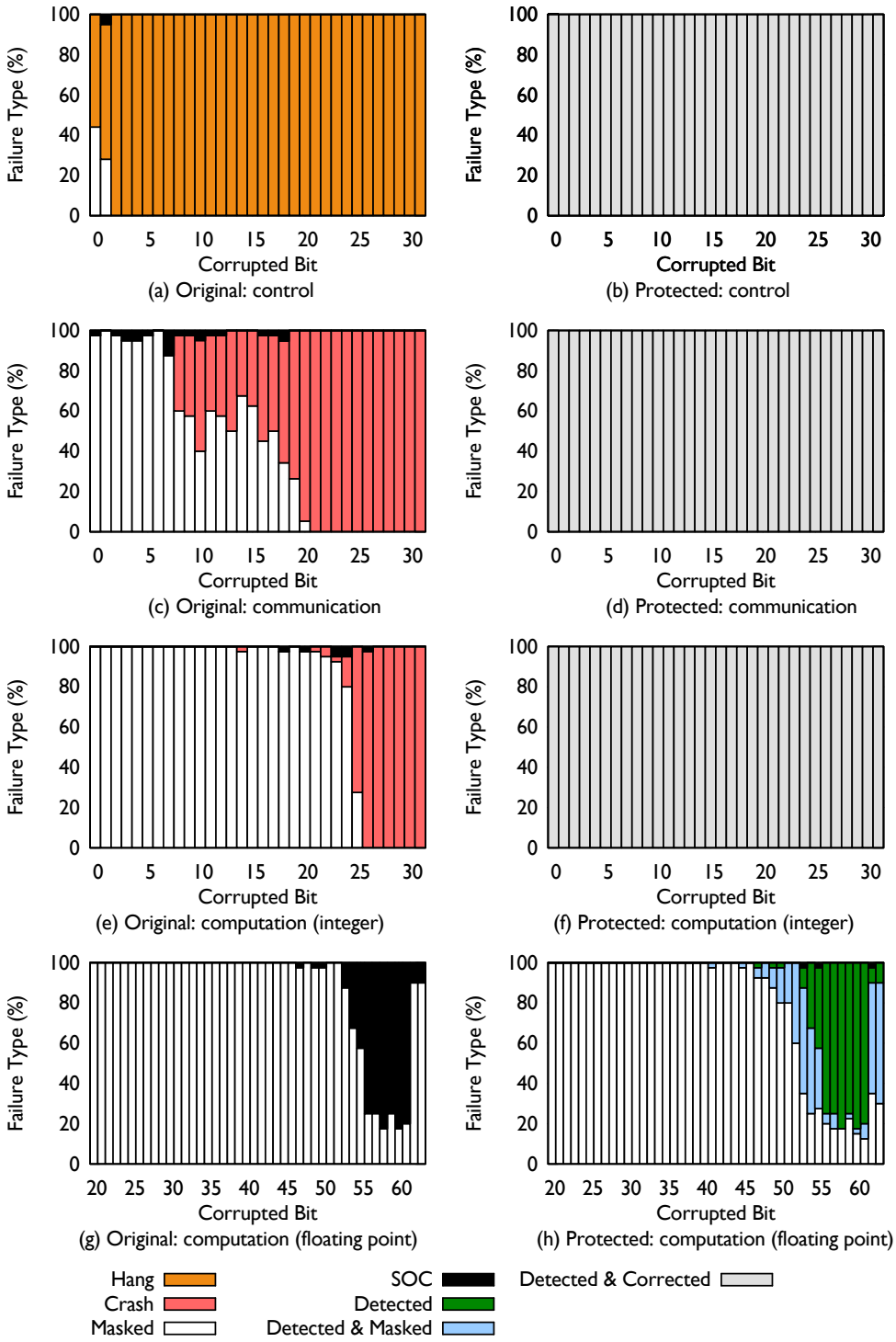


Figure 5.10: Effect of bit flips on **Stencil3d** with and without FlipBack protection.

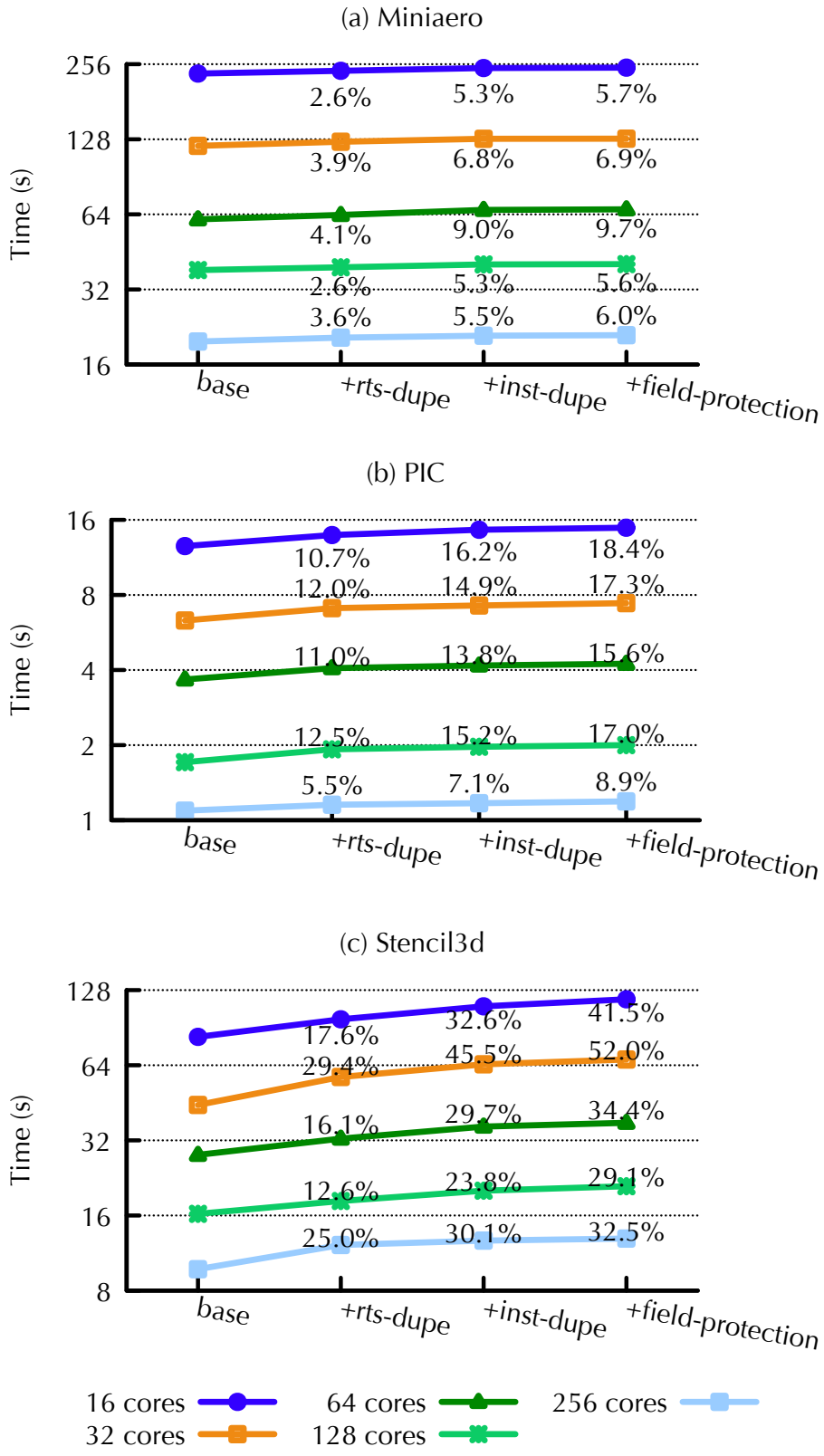


Figure 5.11: Performance with different levels of protection: x-axis is functionality aggregated from left to right.

perform the replicated work.

Figure 5.11(c) shows the performance of Stencil3d when executed with FlipBack. Similar trends as the two proxy applications are observed for Stencil3d. However, the relative amount of overhead due to FlipBack is higher for Stencil3d. The higher overhead is because Stencil3d is a synthetic micro-benchmark that performs much less floating point computation in comparison to the other applications. As a result, the ratio of the computation that is replicated to the total work is high.

5.6 Discussion

Comparison with traditional checkpoint/restart strategy: In the previous section, we showed that when bit flips that definitely cause programs to crash or hang occur, FlipBack can detect them and perform local recovery with 100% coverage. Hence, such failures are completely hidden from users. Although use of FlipBack results in an increase in the total execution time when there are no failures, the recovery overhead using FlipBack is almost minimal. It only needs to either re-execute an entry method or re-run a few instructions for recovery.

Alternatively, one may choose to use traditional checkpoint restart in which if the program crashes or does not make progress for a long period of time, the application state is rolled back to the last checkpoint and restarted from there. This scheme may work if the bit flips induced soft errors are rare, though it will be difficult to be certain if the results are correct. Nevertheless, as the soft error rate increase, the recovery overhead with the checkpoint restart strategy will be very high and prevent applications from achieving sustainable scalable performance. Moreover, as applications scale, the increase in checkpoint time will also reduce performance further.

Figure 5.12 shows the expected overhead of using checkpoint/restart strategy for various soft-error induced crash/hang rates and time to checkpoint. It can be seen that the overhead increases linearly as the soft error rate increases. An increase in time to checkpoint also leads to a significant increase in the total checkpoint/restart overhead. With the worst failure rate (100K crashes/hangs in 1 billion seconds), even if one checkpoint takes only $300s$, the total overhead is almost 30%.

In contrast to the overhead of checkpoint restart, the overhead of using runtime guided replication and selective instruction duplication provided by FlipBack has been observed to be less than 10% for applications that we have tested, e.g. Miniaero and PIC. Besides, FlipBack can also protect applications from bit flips that cause incorrect results which

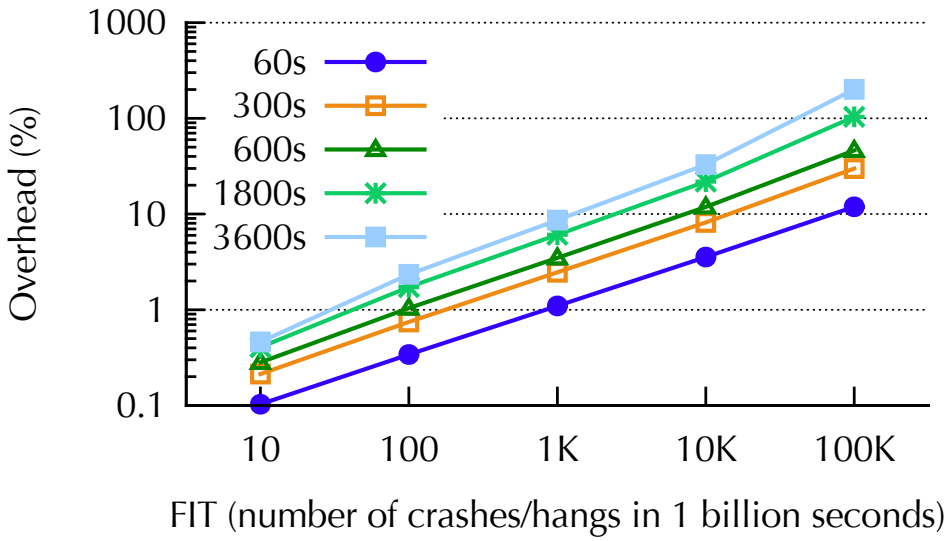


Figure 5.12: Modeling the overhead of checkpoint/restart strategy for different crash/hang rates and time to checkpoint.

is not possible with the checkpoint restart strategy. Moreover, FlipBack is a much more scalable solution. The overheads introduced by FlipBack do not scale with the number of cores; additionally, FlipBack leads to very low overhead during bit flip recovery that may lead to crashes or hangs.

Advantage of runtime guided replication: Runtime guided replication in FlipBack improves fault coverage in two ways over the software based instruction duplication approach. First, runtime guided replication approach is able to detect SDCs that occur in memory by comparing the control variables and messages between the original and shadow chares. In contrast, with instruction duplication technique, incorrect values will be loaded from the memory for both the original and duplicated instruction. Second, store instructions is a single point-of-failure for the software based instruction duplication approach. Even if store instructions are duplicated, the second one will overwrite the previous store instruction. In contrast, for runtime guided replication, the values being stored are compared at the end of entry method between the original and shadow chares, and thus bit flips in store instruction can be detected.

5.7 Summary

As the frequency of bit flips increases due to decreasing feature size and lower operating voltage, efficient methods to protect HPC applications from SDCs will be needed. As a step towards achieving this goal, this chapter introduced FlipBack, a framework that self adapts the protection mechanism based on the data and computation characteristics. We have shown that combined use of compiler techniques and runtime system can significantly improve applications reliability by detecting and correcting SDCs. We evaluated FlipBack with two proxy applications and a micro-benchmark and showed that FlipBack can achieve 100% soft error coverage with only $6 - 20\%$ performance overhead for the proxy applications.

CHAPTER 6

Relieving Memory Pressure

So far we have discussed the design and implementation of various runtime techniques to protect HPC applications from failures. In this chapter, we focus on another challenge that prevents smooth execution of applications: memory limitation. Due to slow growth in capability of memory system relative to computational capability of nodes, applications with large memory footprint are forced to run on large node counts. The advent of non-volatile memory (NVM) can potentially relieve this type of memory pressure. Non-volatile memory, such as flash memory, solid state disk (SSD), provides a very large but slower pool of memory at a lower energy cost per bit. Compared to spinning disk, the latency of NVM is several orders of magnitude lower.

Large size non-volatile memory is expected to become available in the future supercomputers. Comet, a cluster at San Diego Supercomputer Center, already has 250 GB SSD per node available to users. Summit at ORNL will have 800 GB NVM per node. However, how can one make the best use of non-volatile is still an open question. On one hand, NVM can be used as a swap space. Traditionally, supercomputers have disabled virtualization of memory and use of swap space due to lack of local disks. With NVM, this trend can be reversed and use of virtual memory can be re-enabled to utilize NVM as a swap space. Data-intensive applications that require more memory than the DRAM capacity can memory-map their data to NVM with the operating system implicitly handling the data buffering and movement. On the other hand, non-volatile memory can serve as an external storage layer for out-of-core algorithms. Traditionally, most out-of-core algorithms [100] make explicit I/O calls to load and store data between external storage and application data buffer.

Both approaches have their limitations in different aspects. Using NVM as a swap space relieves users from manually managing the data movement between NVM and DRAM. In this case, whenever application attempts to read the data that is not available in physical memory,

a page fault is generated. In response to the page fault, the operating system (OS) moves the data to the physical memory from NVM. However, application progress is stalled because of the lack of data. Although operating systems attempt to perform prefetching to prevent such stalls, it is usually hard for the OS to predict the data access pattern [101]. Besides, the OS does not know if some pages are more heavily used than others. Thus, it is impossible for the OS to decide the optimal pages that should be swapped out. Meanwhile it has been pointed out that the memory-map runtime (mmap) in Linux will rapidly lose performance as concurrency increases and as memory within the system becomes more constrained [47]. Moreover, mmap uses the Least Recently Used (LRU) replacement policy to track page usage, which may not work for applications with complicated data reuse attributes. In other words, it is hard obtain the best performance by just using a "one-fits-all" policy.

On the other hand, if we treat NVM as an external storage space, application writers need to manage data movements on their own by explicitly calling IO operations at right places in their codes. In such scenarios, placement of IO operations is tailored for each application based on the data access pattern. For example, an application can issue a read request for a piece of data much before its actual use. The read request can then be overlapped with the computation over other data that is already available in DRAM. When it is time to swap out some data due to space limitation, programmers can choose to swap out the data that is not of immediate use. As a result, the amount of data movements can be significantly reduced. On the flip side, this approach requires careful arrangement of data prefetch and buffering for each application and thus hinders productivity and adaptivity.

In parallel programming, the best features of the two approaches described above – automated movement of data between NVM/DRAM and application-aware scheduling of data movement – can be combined by leveraging a parallel runtime system. In this chapter, we make a parallel runtime system, Charm++ [17], aware of the existence of NVM and augment it to perform data management automatically. Using our improvements, applications running on top of the Charm++ can conduct out-of-core calculations transparently without user intervention. The runtime system makes optimal data movement decisions based on task dependences as well as data attributes. To the best of our knowledge, our approach is a first of its kind approach that enables automatic parallel out-of-core programming, while striving for both efficiency and productivity. In the rest of this chapter, we discuss different aspects of our approach in detail while exploring answers to the following questions:

- How can a runtime system leverage the computation and data dependences to reduce data movements and overlap I/O activities with application execution?
- On multicore systems, many processors post external data requests required for their

computation concurrently. How can the runtime system balance the data requests to keep every processor busy?

- If users have the flexibility to annotate the access attributes of each data element being accessed (e.g. as readonly or writeonly), how can the runtime system utilize this information to improve performance?

6.1 Related Work

Techniques for optimizing virtual memory polices for out-of-core computation have been extensively studied in the past. Park et al. [102] study the use of virtual memory or file interface for data intensive applications. According to their study, using virtual memory for out-of-core computation can ease programming difficulty and its overhead can be reduced with custom replacement policies. However, their work does not address the concurrency issue virtual memory is facing as the number of cores on a socket increases. HiPEC [103] provides a way for applications to choose specific virtual memory page replacement strategy for optimal data scheduling. They show that for applications with repetitive access pattern, Most Recently Used (MRU) replacement policy improves the performance by two times compared to LRU policy. However, in HiPEC, application writers still need to carefully schedule tasks by themselves in order to minimize the impact of IO activities.

Many specific out-of-core algorithms [100] have also been proposed in order to reduce the amount of data transferred between external storage and DRAM by improving data reuse. However, these algorithms are designed for stand-alone applications with specific access patterns. Mills et al. [104] propose a framework that allows scientific applications to dynamically adapt to memory pressures. Different page replacing strategies are also provided to the application users. Though their approach relieves application writers from manually managing data scheduling and caching, it only works well with scientific applications that use block-based algorithms.

Recent work has also explored ways in which non-volatile memory (NVM) can be used either for out-of-core computation or to improve performance of the programs that require external persistent storage. Mogul et al. [105] discuss the necessary changes in operating system to manage both DRAM and NVRAM. However, without application specific knowledge, it is hard for operating system to make optimal choices for managing page migration between DRAM and NVRAM. Mnemosyne [106] is an interface for programmers to use with persistent memory. The focus of Mnemosyne is to make byte-addressable persistent data structure possible while the proposed work focuses on achieving transparent transition from

in-core programs to out-of-core programs as persistent memory becomes more common.

Ko et al. [107] propose a new swap scheme for flash memory storage device targeted to optimize garbage collection performance, so that the number of erasures can be reduced. CFLRU [108] is a new page replacement policy proposed for flash memory. CFLRU prioritizes swapping of clean pages rather than dirty pages to reduce the replacement cost. Our approach also incorporates this idea by taking data effects into consideration. NVMalloc [109] explores new ways to use local and remote SSD storage as a secondary memory partition which is built on memory-mapped IO interface. However, this approach is expected to be limited by the performance of mmap as the concurrency increases. Further, good performance is only shown when applications allocate read-only data on SSD.

DI-MMAP [110] presents the modifications needed in mmap interface in order to use NVRAM for out-of-core computation and achieves significant improvements over the original mmap interface. However, DI-MMAP requires Linux kernel modifications while our approach can be run on any system without kernel updates. SSDAlloc [111] is a memory manager and runtime library that allows applications to use SSD as memory extension. Our approach complements this effort and focuses on the challenges for parallel applications.

6.2 Design and Implementation

In this section we describe the design and implementation of the Charm-HMC framework that automatically enables out-of-core computation. We start by presenting the basic design of Charm-HMC and various optimizations that have been done to obtain good performance. Thereafter, we discuss the user interface and minimal modifications required to application codes for enabling efficient execution with Charm-HMC.

6.2.1 System Design

There are two key components in the design of Charm-HMC, both of which perform specific tasks. The *progress engine* is a module added to Charm++'s main scheduler for tracking IO requirements of different chares and requesting the *IO Thread* to fulfill them. As suggested by its name, the *IO Thread* decides which IO requests are processed in what order. In addition, these two components coordinate to ensure load balance among work that can be performed by different cores at a given time.

Parallel applications written using Charm++ progress via message transmission and function (entry method) invocation. When a message is received by the Charm++ RTS, the

```
Input:
t: computation task
a: number of extra data objects DRAM can contain
availData: list of data units with reference count of zero
1  while not exiting do
2      if !pendingTasks.empty() then
3          t ← pendingTasks.top();
           // number of data objects that are in NVRAM
4          n ← t.data.num();
           // look for space in DRAM
5          canUse ← a > n? n:a;
6          n − = canUse; a − = canUse;
7          if availData.size() >= n then
8              pendingTasks.pop();
9              foreach d in availData[0:n) do
10                 write(d); d.setState(INNVM);
11             end
12             foreach d in t.data do
13                 read(d); d.setState(INMEM);
14             end
15             notify worker thread;
16         else
               // once more computations tasks complete to clear the space,
                  IO thread will be awaken for another trial
17         end
18     end
19 end
```

Algorithm 3: Workflow of IO thread

computation function registered with the message is invoked on the chare (C++ object) targeted by the message. Thus, parallel application execution in Charm++ can also be viewed as a collection of multiple computation tasks (invocations of computation functions) interacting via messages. The scheduling unit in Charm++ is a worker thread that is generally mapped to an individual hardware resource. Typically, on a node, several pthreads are created to serve as worker threads. These threads share address space and a communication thread which drives all external interaction. Each worker thread maintains a scheduler for all the incoming messages; these messages are processed in either FIFO order or based on priority if applicable.

When external storage such as NVRAM is available and applications require more data than the DRAM capacity, Charm-HMC allows application to allocate data using its API (Section 6.2.2). Data allocated in this way is special because their location is decided by

Charm-HMC, i.e. they can either be in NVRAM or in DRAM. When such allocations are done, before a computation task is executed, the *progress engine* checks if the data required by the computation task is in DRAM. In a naive scheme, if the data is not available in DRAM, the progress engine will halt and begin the transfer of required data from NVRAM to DRAM. However, this will result in the worker thread being idle very often due to unavailability of data.

To reduce excess waits on worker threads, we add another type of thread to Charm++ - *IO thread*. Like the communication thread, one IO thread is created per node to take care of the data movements between DRAM and NVRAM for all worker threads. Now, when progress engine wants to execute a task with data in NVRAM, instead of halting the worker thread for the transfer of data from NVRAM to DRAM, it enqueues the tuple containing the computation task as well as all the dependent data units into a request queue to IO thread. After enqueuing the request, the progress engine and the worker thread continues onto the next task in their scheduling queue without waiting for the IO to finish the request.

When the IO thread has read all the data units requested by the progress engine from NVRAM and written it to DRAM, it pushes the computation task back into the worker threads's scheduling queue. To facilitate management of data in DRAM, Charm-HMC maintains a reference count for each data unit. When a computation task is ready to be executed, i.e. all data required by it is in the DRAM, the reference count for each data unit required by it is incremented. When the computation task is executed, all the corresponding reference counts are decremented. The reference count serves as the guidance for the IO thread to decide which data units to swap out. A data unit can be swapped out only when its reference count is zero. This is necessary so that the data unit that is being used or will be used in near future will not be moved back and forth between NVRAM and DRAM.

Currently, we rely on users to annotate the dependences between data units and computation tasks. Each data unit can be in one of the three states: *INNVM*, *INMEM* or *TOMEM*. State INNVM or INMEM means that the data object is either in NVRAM or DRAM, respectively. State TOMEM is a transient state, which means that the data object is currently being read from NVRAM to DRAM. This state is useful especially when a data unit is needed by multiple computation tasks. For such cases, all but one computation tasks avoids issuing the repetitive read request for the same data unit. Each data unit also maintains a list of the computation tasks that depend on it. As soon as the state of one data unit turns from *TOMEM* to *INMEM*, IO thread checks if any of the computation tasks in the list has all the data units in DRAM. If one or more such computation task is found, IO thread pushes the computation task to the queue of the corresponding worker thread and increments the reference count to those data units.

Algorithm 3 describes the work flow of the IO thread. First the IO thread fetches one task from the top of its queue (line 3). Next, IO thread searches for memory space for that computation task (line 4 to 7). If there is not enough space in DRAM, IO thread tries to make space by evicting other data units (line 7 to 11). Frequent allocations and deallocations incurs extra overhead. In our current implementation, we assume each data object is of the same size so that memory buffers can be reused.

If there are not enough data objects that can be evicted to make space for the new data request (line 16), IO thread aborts the current trial and waits until more data objects can be evicted. Note that we only commit the IO transactions when all the data units of a pending task can be placed in DRAM to avoid deadlocks (line 8 to 14). After all the data objects are fetched from NVRAM to DRAM, IO thread notifies the corresponding worker thread so that the associated computation task can be executed. More data units may become available as computation tasks finishes. At that point, IO thread can start a trial to bring data required by new pending tasks to DRAM.

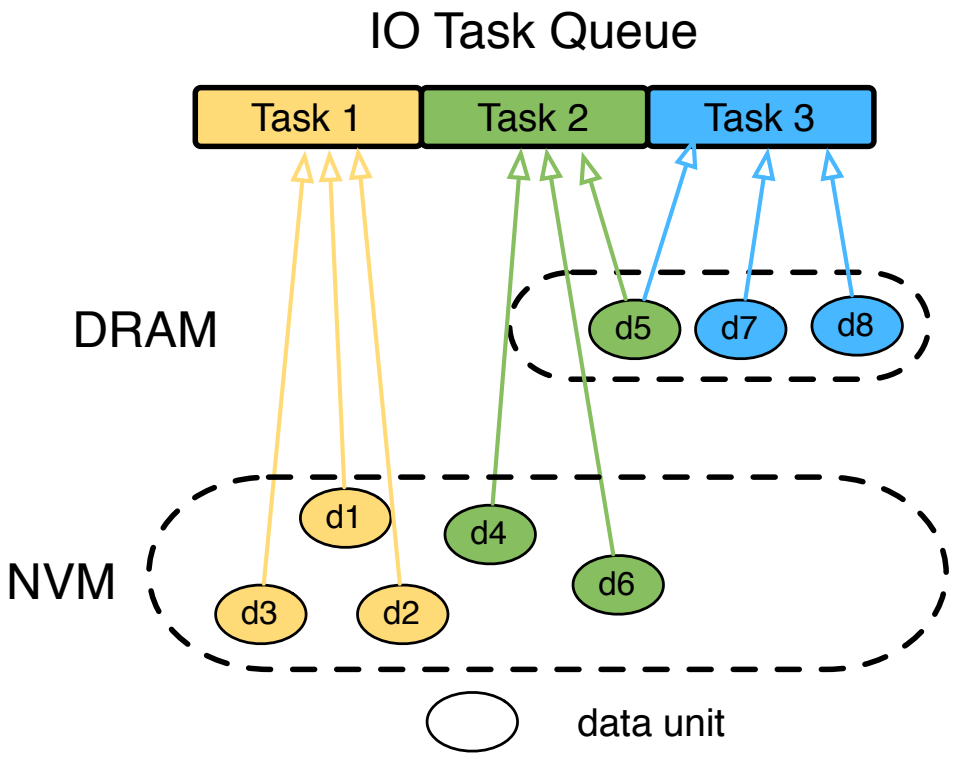


Figure 6.1: Example state of data objects and computation tasks

IO Thread Scheduler: CHARM-HMC supports three types of queues that can be used to optimize scheduling of IO requests. The choice of queue is made based on application

requirements to reduce the idle time caused by data IO between NVRAM and DRAM. The default queue is a FIFO queue, in which IO thread schedules IO requests in the order in which different worker threads submit the requests.

The second type of queue prioritizes the IO requests from computation task that depend on the least number of data units in NVRAM. Compared to the default FIFO queue, prioritized scheduling of IO based on data requirement can reduce the amount of data movements as shown in the example in Figure 6.1. In Figure 6.1, each pending computation task depends on 3 data units. Task 1, 2 and 3 have 3, 2 and 0 data units in NVRAM separately. Task 2 and task 3 both depend on data units 5. Let us assume that in this case DRAM can only contain 3 data units at a time.

If IO thread schedules tasks in the FIFO order, i.e. task 1, 2 and then 3, the total number of data writes and reads is 16. First, IO thread needs to move data units $d5$, $d7$ and $d8$ to NVRAM in order to bring data units $d1$, $d2$ and $d3$ in DRAM . After task 1 is finished, IO thread writes data units $d1$, $d2$ and $d3$ to NVRAM and reads data units $d4$, $d5$ and $d6$ from NVRAM to DRAM in order to schedule task 2. Similarly when it is time to schedule task 3, IO thread needs to write data units $d4$ and $d6$ and then read data units $d7$, $d8$. In contrast, if IO thread chooses to schedule task 3 first since it depends on the least number of data units in NVRAM, task 3 can be scheduled right away. After task 3 is done, between task 2 and 1, task 2 depends on less number of data units in NVRAM. Thus task 2 gets scheduled next. IO thread writes data units $d7$ and $d8$ to NVRAM and reads data units $d4$ and $d6$. After task 2 is done, IO thread writes data units $d4$, $d5$ and $d6$ and then reads data units $d1$, $d2$ and $d3$ in order to schedule task 1. As a result, the total number of data movements is reduced to 10. In Section 6.3.1, we show how this type of prioritized scheduling can help improve performance with a real computation kernel.

The third type of queue schedules IO tasks based on the priority specified by users. This type of queue is especially useful when the execution of some computation tasks can improve data reuse and thus reduce the number of writes. For example, in LU factorization, we assign the computation tasks that corresponds to the left part of the matrix higher priority. This is often referred to as the left-looking algorithm [112]. The disadvantage of this algorithm is that the degree of concurrency is small. However, with Charm-HMC, we can seek a balance between data reuse and the available parallelism based on DRAM capacity. More details and explanation of this scenario is presented in Section 6.3.2 with the LU example.

Balancing Inequality in Data Fulfillment: While trying to minimizing IO activities between DRAM and NVRAM, making sure that pending tasks from different worker threads are fulfilled in a balanced way poses a significant challenge. Figure 6.2(a) shows the usage

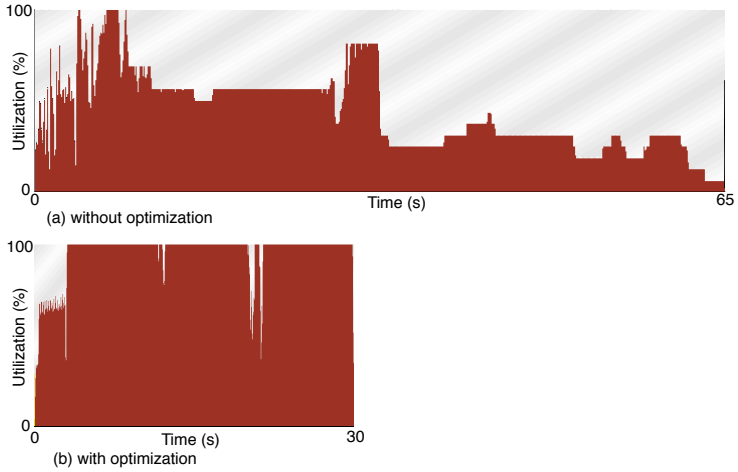


Figure 6.2: Processor utilization before and after the use of shared queue.

profile of a matrix matrix multiplication program running on 16 cores when only 1/6 of the total required memory is available. As can be seen in Figure 6.2(a), the average utilization of the cores is below 50%, since half of the worker threads are left idle waiting for data to be brought from NVRAM to DRAM. After a closer look, we found that the worker threads that are kept busy usually have more than one computation task whose data is in DRAM and are ready for execution at a time.

The imbalance in the number of ready tasks for different workers raises the following question: what causes IO thread to favor some worker threads in comparison to other? This is because computation tasks from the same worker thread may partially depend on the same data units. Once IO thread brings all the data units needed by one computation task, other computation tasks from the same work thread are prioritized for IO scheduling since they depend on less number of data units in NVRAM when compared to computation tasks from other worker threads. For example, in the case of matrix multiplication program, computation tasks from the same worker thread are responsible for the computation of the data in the same column and thus they all depend on the data units from that column.

To solve the aforementioned problem, we relax the mapping between computation tasks and worker thread in Charm++: a computation task can now be executed on any worker thread within node instead of being restricted to the thread that generates the task. One thing to note here is that even with this relaxation, we still need to guarantee that computation tasks associated with the same chare are executed one at a time to eliminate data race.

Algorithm 4 shows how this optimization works. Once IO thread determines that a computation task can be executed, it enqueues that computation task to the "ready" queue of

```
   Input:
   Q: queue of "ready" chares, shared by all worker threads
   t: computation task
   // IO thread
 1 if t.ready() then
 2 |   c ← t.owner;                                  // chare associated with t
 3 |   c.pendingQ.push(t);
 4 |   if !c.inUse and c.pendingQ.size() == 1 then
 5 |   |   Q.push(c);                          // push c to Q if not already in
 6 |   end
 7 end
   // Worker thread
 8 if no local work and !Q.empty() then
 9 |   c ← Q.top();
   |   // make c invisible to avoid data race
10 |   Q.pop(); c.inUse ← true;
11 |   t ← c.pendingQ.top(); c.pendingQ.pop();
12 |   process t;
13 |   if c.pendingQ.size()>0 then
14 |   |   Q.push(c);                                  // push c back to Q
15 |   end
16 |   c.inUse ← false;
17 end
```

Algorithm 4: Optimization to balance load in out-of-core computation

the chare it is associated with (line 3). If the length of the "ready" queue is one and the computation task from that chare is not being executed by any worker thread, IO thread determines that it is safe to push the chare to the shared work queue Q (line 4, 5). Each worker thread actively polls from the shared work queue if there is no unprocessed local work (line 8,9). When a worker thread acquires an active chare from the shared queue, it removes the chare from the shared work queue so that other threads do not execute any computation task from that chare concurrently (line 10).

Next the same worker thread acquires one computation task from the top of the "ready" queue of that chare (line 11). Once the computation task is done, worker thread pushes the chare back to the shared work queue if there are still more computation tasks in its "ready" queue (line 12 to 16). Please note that we use locks to protect the shared data structures used among threads. For simplicity, lock related code is not shown in Algorithm 4. After applying this optimization, processor utilization improves significantly as shown in Figure 6.2(b). The total execution time decreases by more than 50% from 65s to 30s.

stencil.ci *(interface file)*

entry void compute()[**readwrite** : **temperature**];

stencil.C

IOHandle<double> temperature;
temperature.**resize**(blockDimX*blockDimY*blockDimZ);

Figure 6.3: Modifications in stencil program for out-of-core computation

6.2.2 User Interface

Figure 6.3 shows the modification needed in application codes in order to use Charm-HMC. Charm-HMC gives the users flexibility to express that certain application data can be placed either in NVRAM or DRAM. All the scalar and array data types are supported. As seen in Figure 6.3, since the double array *temperature* contributes to the majority of the memory consumption and thus it should be placed in either DRAM or NVRAM for out-of-core computation, its type is declared as *IOHandle<double>* rather than *double **. To change the size of that array, a *resize* function is provided. Based on DRAM capacity, Charm++ runtime system decides where the data unit should be placed. If there is not enough space in DRAM, Charm-HMC creates a file on NVRAM and dumps the contents of the data object into the file through direct I/O. Users express the dependence between data units and computation task in the interface file. In the example shown in Figure 6.3, the execution of computation task *compute()* depends on the *temperature* array. During computation, it is the responsibility of Charm-HMC to make sure that the *temperature* is in memory before the execution of *compute()* method.

Users can also express the privileges each computation task has over the data units it depends on. As shown in Figure 6.3, *compute* method declare *readwrite* privilege on *temperature* date units. The types of data privileges supported are: *readonly*, *writeonly*, *readwrite* or *buffer* (for temporary data). Expressing data privileges helps reduce data movements. For example, data units annotated with *writeonly* or *buffer* privilege do not need to be read from NVRAM before each use. Instead Charm-HMC just needs to ensure there is enough space to allocate buffers for them. More importantly, after each use, Charm-HMC can categorize data units into clean (with privilege *readonly* and *buffer*) or dirty list (with privilege *writeonly* and *readwrite*). We prioritize to evict data units from the clean list first to reduce the number of writes. In Section 6.3.1 and 6.3.2, we will provide more experimental details on how data privileges can help improve performance.

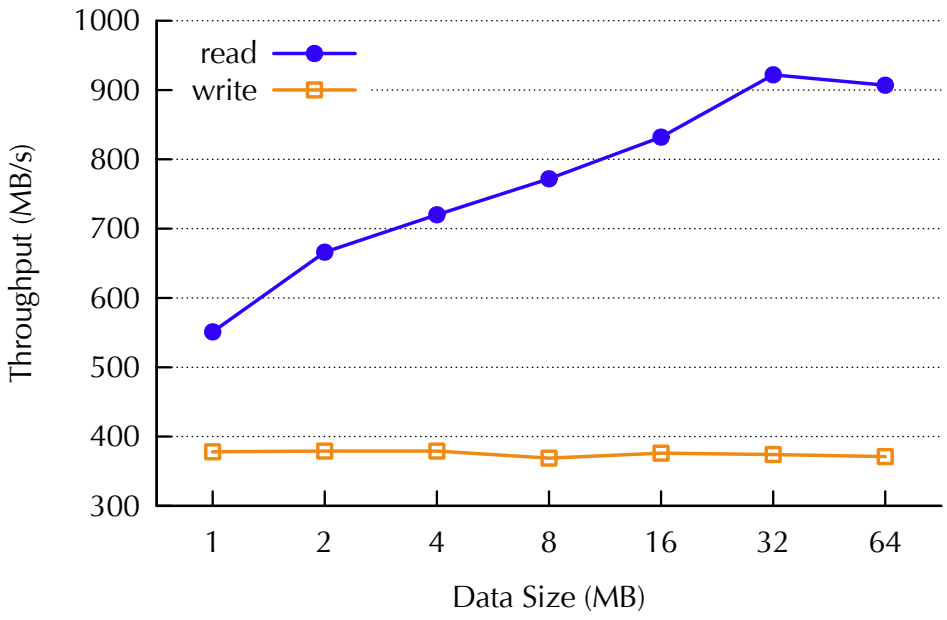


Figure 6.4: Read and write performance of Comet SSD with direct IO.

6.3 Case Studies

We use three computation kernels to evaluate Charm-HMC: matrix matrix multiplication, LU factorization and stencil computation kernel which performs 7-point stencil based computation on a 3D structured mesh. In the matrix matrix multiplication program, the two input matrices as well as the output matrix are declared as IOHandle type, which implies that they can be placed in either DRAM or NVM. As for the LU factorization kernel, the matrix to be factorized is declared as IOHandle type. In the stencil kernel, the internal mesh data is declared as IOHandle type so that it can be freely moved between NVM and DRAM. However, data that serves as the ghost layer of the internal mesh always stays in DRAM. We make this design choice because the amount of ghost data is relatively small and thus there is enough space to place it in memory. Moreover, this design helps avoid extra IO transactions whenever there is communication for ghost update.

All the experiments for this chapter have been performed on Comet, a cluster at San Diego Supercomputing Center. Each node of Comet has 2 sockets with 12 Intel E5-2680 processors each. Each node is also equipped with 128 GB of DRAM and 250 GB of SSD. Since we focus on the IO activities between NVM and DRAM in this chapter, all the experiments are conducted on one node to exclude the effect of network communication.

In Figure 6.4, we show the file read and write performance on the SSD of Comet with *O_DIRECT* enabled to eliminate the effect of file system cache. As can be seen from Fig-

short form	explanation
f	Tasks are scheduled in FIFO order
a	Tasks with the least number of data units in NVM have higher priority
p	Tasks are scheduled based on user defined priority
s	Shared queue is used to balance computation tasks
d	Prioritize to evict the data that has not been modified since last read

Table 6.1: Explanations of the short forms of the runtime options used in experiments

ure 6.4, the throughput for write operations is around 380 MB/s while the throughput for read operations can be up to 900 MB/s.

We compare Charm-HMC against using Linux memory mapped interface for out-of-core computation. Instead of declaring certain variables as IOHandle, we mmap each of them to a file on NVM and rely on operating system to take care of the data management and buffering. Comet is a machine designed specially for data intensive applications and thus has adequate DRAM capacity. In order to evaluate our approach for general use cases where access to large amount of DRAM is not possible, we artificially "reduce" the memory capacity using Linux *mlock* interface. For example, if we want to evaluate how an out-of-core matrix multiplication program that requires 6 GB of memory performs when only 3 GB memory is available, we use mlock to "lock" the remaining memory and only leave 3 GB for the program to use. Table 6.1 lists different options that were described earlier and used in the experiments. We refer to them using the short forms in the rest of the chapter.

6.3.1 Matrix Matrix Multiplication

The total memory requirement for the matrix multiplication program used in our experiment is 6 GB with each matrix taking 2 GB memory space. Figure 6.5 shows the performance comparison between Charm-HMC and mmap. The normalized computation time is calculated based on the computation time when DRAM has enough capacity for all three matrices at the same time. The percentage of available memory varies from 83% to 6.25%, i.e. available DRAM for the program to use decreases from 5 GB to 375 MB. As can be seen from Figure 6.5, even when only 1 GB memory (17%) is available, the performance degradation is only 18% using Charm-HMC on 16 cores. In contrast, the approach with mmap increases computation time by more than 4 times. A series of optimizations, **optimal queuing strategy**, **work sharing for load balancing** and **expression of data privileges** helps improve

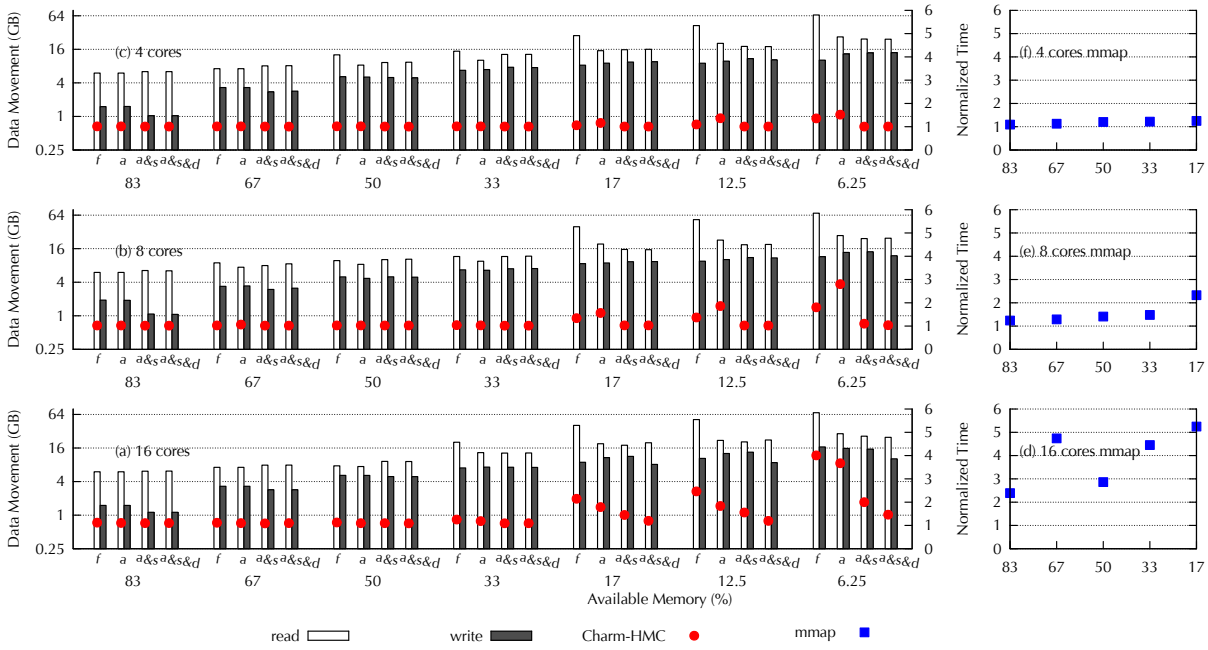


Figure 6.5: Performance and data movement of using CHARM-HMC for out-of-core computation using **matrix matrix multiplication** program (2 GB/matrix). The base time used to calculate the normalized time is the execution time when all the memory required is available.

the performance by more than two times when only 384 MB memory is available. Another general trend that can be observed from Figure 6.5 is that as the number of cores increases, the performance improvement brought by various optimizations becomes more significant. This is because the contention for IO resource increases rapidly with the increase in core count. The proposed optimizations greatly help relieve such memory contention. Next, we discuss each optimization strategy in detail.

Effect of queuing policy: As can be seen on Figure 6.5, using the queue that prioritizes the task that has the least number of data objects in NVM (represented by label "a") reduces the amount of data being read, especially as the available memory decreases. When the processor count is higher, e.g. 16, the reduction in data movements does help improve the performance of out-of-core execution by $8-25\%$. However, on 8 and 4 processors, we observe an increase in computation time when switching from the FIFO queue to the queue based on data availability. This happens when IO thread starts favoring work requests from a particular processor since most of its dependent data is already in DRAM. Figure 6.6 shows the timeline view of how computation is performed with different queuing strategies on 8 processors when only 6.25% memory is available. With FIFO queue shown in Figure 6.6(a)

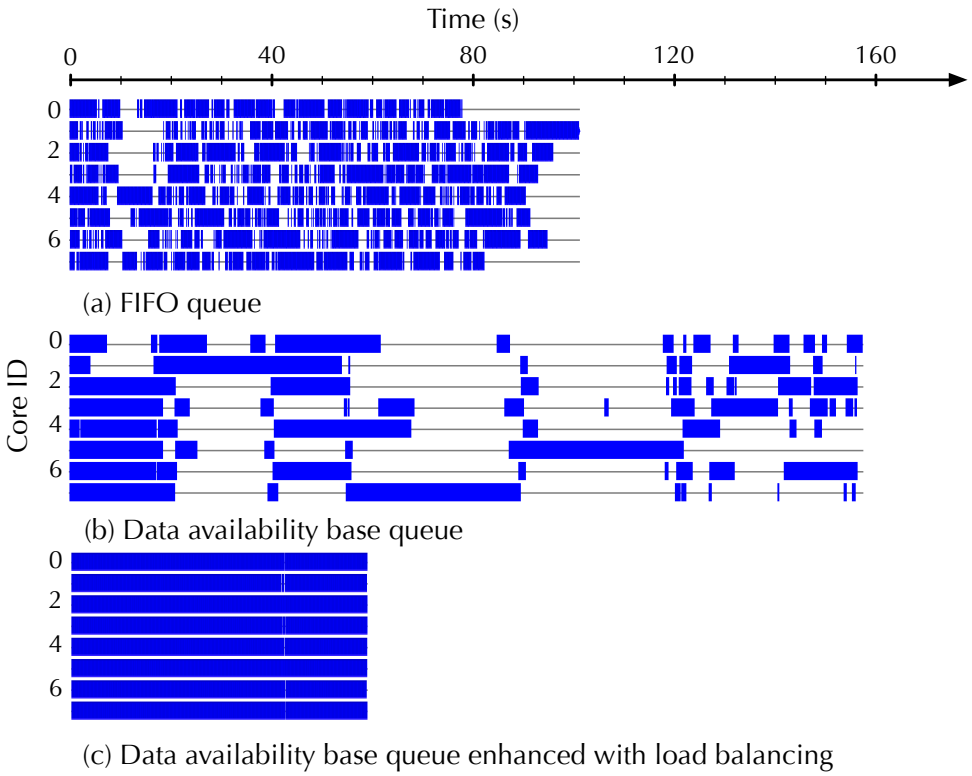


Figure 6.6: Timeline profile of various optimizations using Charm-HMC on 8 cores with 6.25% available memory for matrix multiplication program.

the processor utilization is not high due to dependence on I/O activities. Figure 6.6(b) shows the timeline view when the queue based on data availability is used. As can be seen from Figure 6.6(b), for some periods of time, only one processor has work to do while others are left idle since IO thread favors the computation tasks from that processor due to data availability.

Effect of work sharing: Sharing of the "ready" computation tasks helps keep every processor busy. This optimization is especially useful in conjunction with the previous optimization that has the side effect of causing load imbalance. As seen in Figure 6.5, after applying the shared queue optimization (represented by label "a&s"), the execution time decreases by half when only 384 MB memory is available even though the total amount of data movements does not change much. Figure 6.6(c) also shows this decrease in execution time. Since the computation tasks from the overloaded processor can be freely executed on any other processor, the total processor utilization increases a lot according to Figure 6.6(c).

Effect of eviction mechanism: As discussed in Section 6.2, with user annotating the effect of each data object, Charm-HMC can prioritize eviction of the data objects that have not

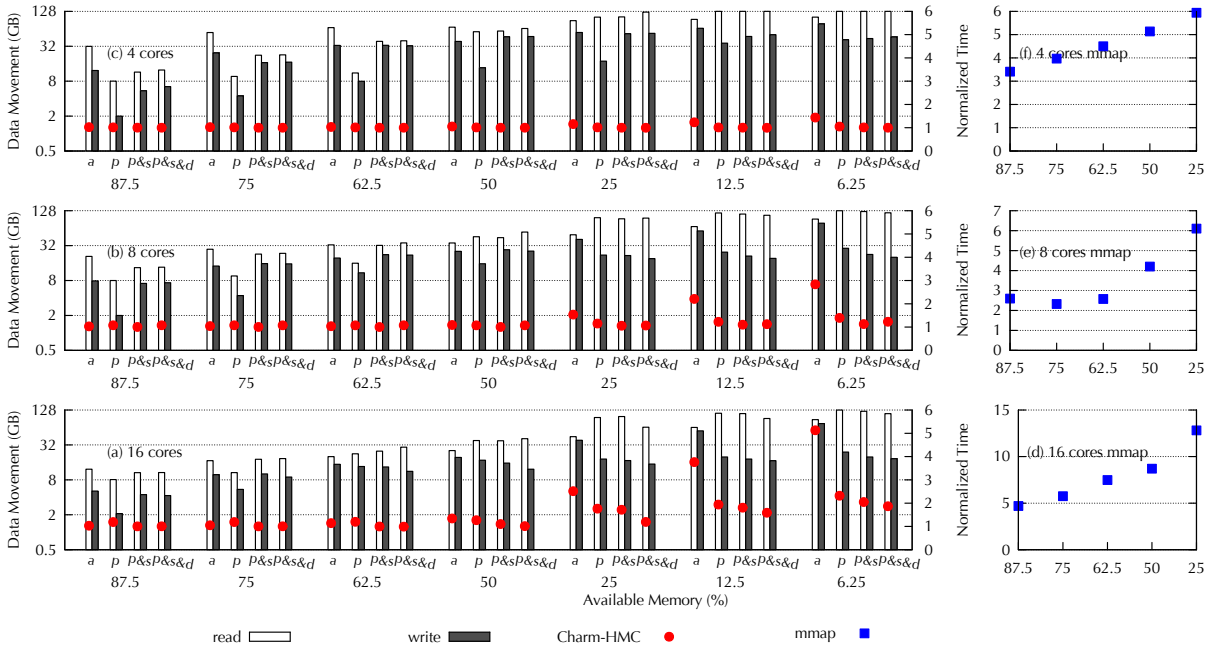


Figure 6.7: Performance comparison of the runtime support for out-of-core computation with mmap using **LU factorization** program (8 GB/matrix). The base time used to calculate the normalized time is the execution time when all the memory required is available.

been changed since last read to reduce the amount of data being written. According to Figure 6.5, when this optimization is used (represented by label "a&s&d"), the amount of data needs to be written is reduced, especially on higher core counts when the available memory is low. For example, when the available memory is less than 17% on 16 cores, this optimization alone can reduce the execution time by 20%.

6.3.2 LU Factorization

Figure 6.7 shows the comparison of the data movements as well as execution time for the out-of-core LU kernel between Charm-HMC and mmap. As can be seen from Figure 6.7, even when only 25% of the memory required for the LU kernel is available, the normalized execution time with Charm-HMC is approximately 1.2. In contrast, with mmap, performance degrades by more than 4 times. The optimal queuing policy for the matrix multiplication kernel does not work well for LU factorization. For LU factorization, if the tasks corresponding to the lower left part of the matrix are given higher priority, the data reuse ratio can be significantly improved and thus the number of writes is reduced [112]. As a result the best queuing strategy is the one that follows the priority assigned by users to

reduce the number of writes by improving data reuse.

Effect of queuing policy: The benefit of using priority based queue increases as the available memory decreases. When the available memory decreases to 1 GB (12.5% of the total memory required), with the help of the priority queue (represented by label "p"), the execution time is reduced by half. The improvement and the good performance is because of the reduction in the number of writes. As can be seen from Figure 6.7, even though the amount of data that needs to be read increases, the data to be written decreases significantly (by two thirds) when this optimization is used. Since the write operation is much more expensive than read, the overall performance improves significantly.

Effect of work sharing: Similar to the trend we have observed for the matrix matrix multiplication kernel, the use of shared queue further helps reduce the execution time, though it has no impact on the amount of data movement. When the available memory is less than 50%, this optimization alone (represented by label "p&s") can help reduce the execution time by $10 - 40\%$. The effectiveness of this optimization becomes more significant as the number of cores used on one node increases since the contention for IO resource become severe.

Effect of eviction mechanism: By carefully selecting the data objects for eviction (represented by label "p&s&d"), the total execution time of LU is reduced by $10 - 30\%$ since less data needs to be written as can be seen in Figure 6.7.

Stress Test

So far we have evaluated Charm-HMC using relative small problem sizes by limiting the available memory a program can use. In this way we have illustrated the scenario of out-of-core computation. We are also interested in studying how Charm-HMC behaves when the problem size is more than the available memory of a system such as Comet. Hence, we test Charm-HMC using a matrix of size 128 GB for LU factorization. Without using Charm-HMC, it is not possible to execute LU factorization of a matrix this large on one node of Comet since the execution would simply crash with "out-of-memory" error message.

Figure 6.8(b) shows the performance of using Charm-HMC for LU factorization of the 128 GB matrix. We limit the actual memory usage by the program from 64 GB down to 8 GB as can be seen on the x axis. In comparison, Figure 6.8(a) shows the performance of the execution without Charm-HMC for problem size ranging from 8 to 72 GB. As can be seen from Figure 6.8, GFlops obtained from out-of-core computation is comparable to that obtained from in-core computation even when only 12.5% memory is available. Also,

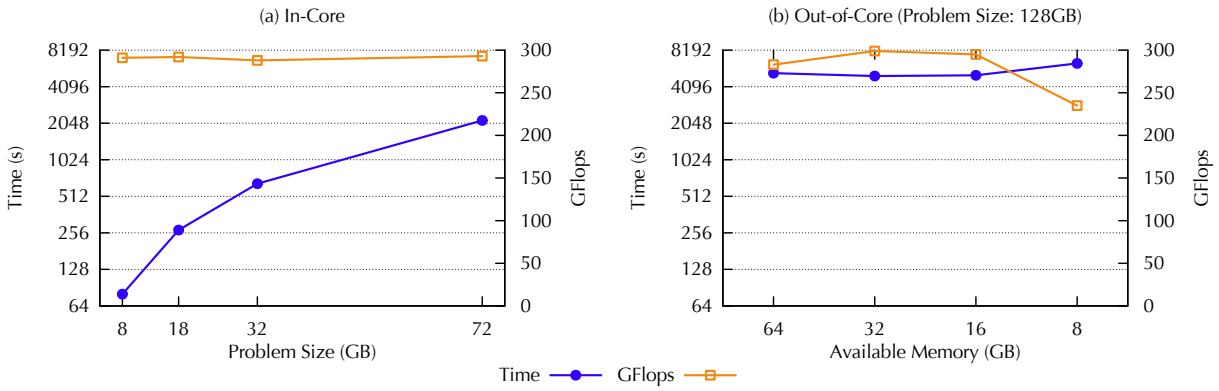


Figure 6.8: Comparison of the out-of-core computation enabled by Charm-HMC with in-core performance.

the execution time for out-of-core computation increase gradually from the in-core execution time. This proves that by using Charm-HMC for LU factorization, NVM can be treated naturally as an extension of DRAM with reasonable performance degradation.

6.3.3 Stencil Computation

The total mesh size used in stencil experiments is 8 GB. We limit the available memory used to allocate mesh from 7 GB to 1 GB for the out-of-core experiments. As can be seen in Figure 6.9, 6.10 and 6.11, the amount of data movement per step with Charm-HMC matches with the theoretical expectation. Since all the mesh data needs to be accessed once every step, data that resides on NVM needs to be brought to DRAM at least once and correspondingly the same amount of data needs to be written from DRAM to NVM in order to make space. Thus the amount of data movement in read and write operations when 7 GB memory is available is 1 GB each given that 1 GB of the mesh data is always in NVM. Similarly, when the available memory decreases to 1 GB, the data movement is around 7 GB for either read or write operations.

The execution time of out-of-core computation with Charm-HMC is limited by the low data reuse ratio. Unlike the data dependence pattern in matrix multiplication and LU factorization kernel, in stencil computation, all the data needs to be processed exactly once in every iteration before being reused due to the synchronization requirement. When only one internal gauss-seidel iteration is performed per step in Figure 6.9, the total computation time per step on 16 cores is 1s. Given the throughput of SSD on Comet (Figure 6.4), it takes at least 4s to read and write 1 GB of data. Thus it is not surprising to see that the normal-

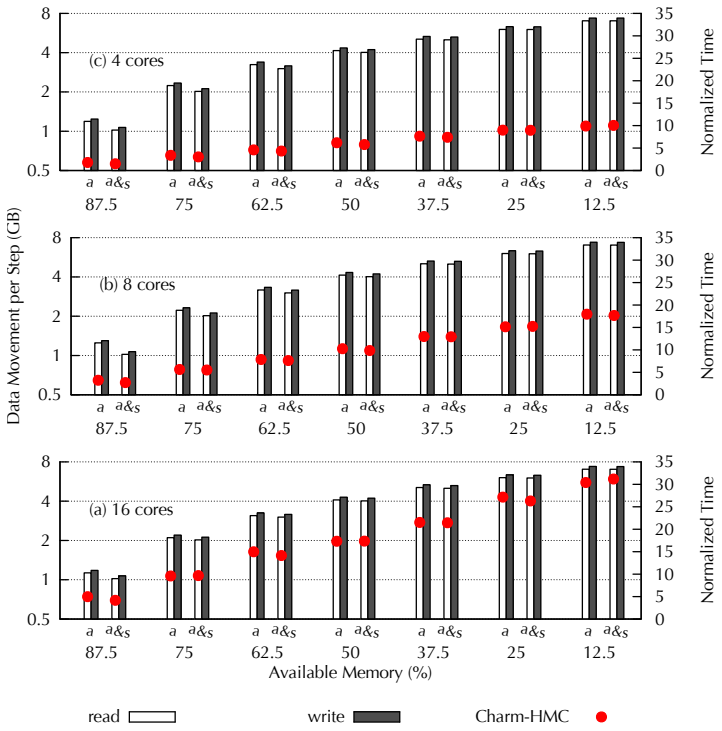


Figure 6.9: Performance and data movement using CHARM-HMC for out-of-core **stencil** computation with one internal gauss-seidel iteration.

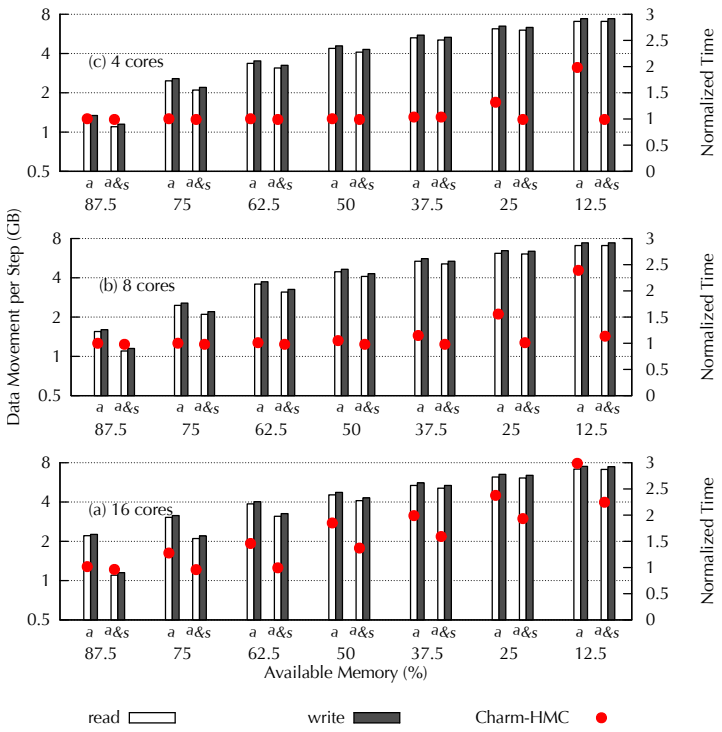


Figure 6.10: Performance and data movement using CHARM-HMC for out-of-core **stencil** computation with 20 internal gauss-seidel iteration.

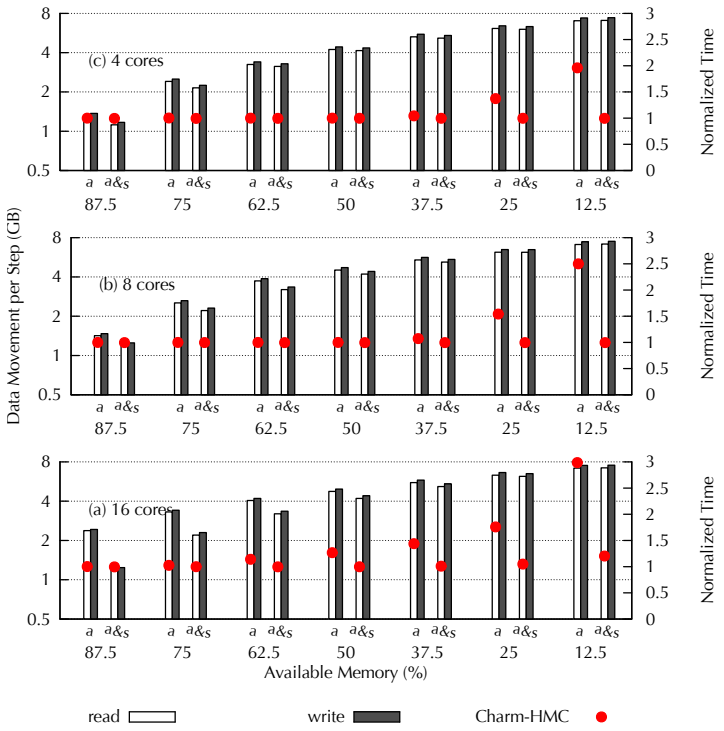


Figure 6.11: Performance and data movement using CHARM-HMC for out-of-core **stencil** computation with 40 internal gauss-seidel iteration.

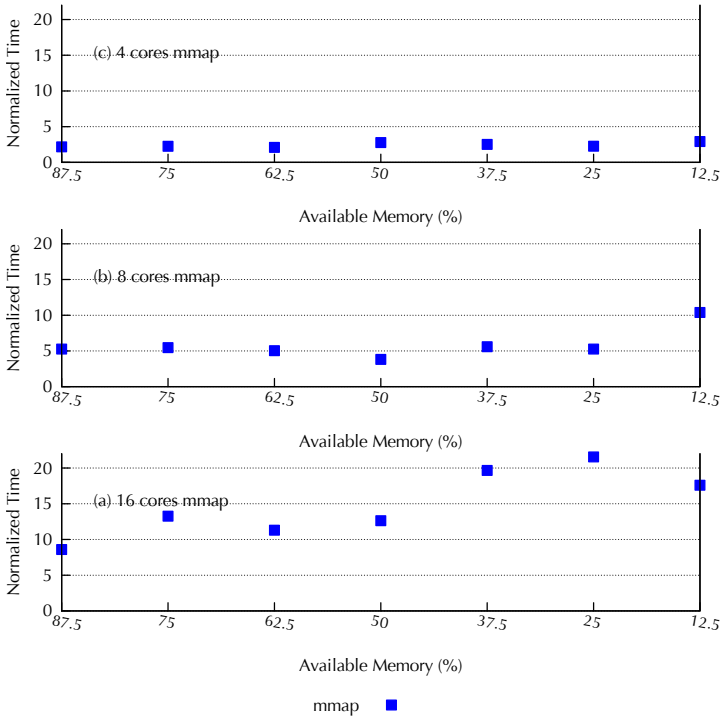


Figure 6.12: Performance using mmap for out-of-core **stencil** computation with 40 internal gauss-seidel iteration.

ized execution time is almost 5 times when 87.5% memory is available on 16 cores. Future advancements in SSD techniques may help hide this type of excess I/O overhead. For example, SSDs with 10 GB/s throughput have already been announced. With newer generation SSDs, Charm-HMC can potentially overlap application execution and I/O transactions even when available memory is low in a better way.

Nevertheless, the limited amount of computation performed in the artificial stencil kernel does not leave much scope to overlap computation with the I/O work. Practical executions of this method often execute multiple internal gauss-seidel iterations, thus we increase the number of internal gauss-seidel iterations performed in each step. In Figure 6.10 and 6.11, the number of internal gauss-seidel iteration is increased to 20 and 40. As can be seen in Figure 6.10, Charm-HMC is able to reduce the normalized execution time to 1 on 4 and 8 cores even when 12.5% memory is available. When the internal gauss-seidel iteration is increased to 40 in Figure 6.11, even on 16 cores with 12.5% available memory, the normalized execution is around 1.

Figure 6.12 shows the performance of stencil with mmap. The number of internal gauss-seidel iterations per step for these runs is 40. It can be seen that out-of-core computation with mmap is much worse than Charm-HMC. On 16 cores, the normalized execution time is almost 20 when 12.5% memory is available.

Effect of work sharing: The benefit of sharing "ready" computation tasks is observed only when the number of internal gauss-seidel iterations per step is 20 or 40. As can be seen in Figure 6.10 and 6.11, up to 2.5× speed-up can be observed with the use of work sharing queue (represented by label "a&s") especially when the percentage of available memory is low. When there is only one internal gauss-seidel iteration per step, the performance is limited by the slow I/O transmission time rather than the load imbalance. Thus no improvement is observed.

6.4 Conclusion

In this chapter, we introduced Charm-HMC, an enhancement to the Charm++ parallel runtime system that automatically manages heterogeneous memory resources. We found that the runtime approach can greatly ease out-of-core programming while achieving good performance at the same time. Various queuing strategies provided by Charm-HMC can satisfy the needs of different applications in order to reduce data movements. Charm-HMC also addresses the load imbalance issues caused by data availability. Last but not the least, Charm-HMC effectively explores how data dependences and data privileges can help

improve performance. We evaluated CHARM-HMC with three different computation kernels and showed that even when only 25% of the requested memory is available, CHARM-HMC can help applications achieve good performance as if the applications are running with all the memory required in DRAM.

CHAPTER 7

Conclusion

This thesis has explored the role of parallel runtime systems in mitigating failures during execution of HPC applications. Design and implementation of fault tolerance strategies that can protect applications from both hard and soft failures with low cost have been presented. The proposed methods make use of both runtime system support and compilation techniques. It is to be noted that the proposed runtime system based approaches are adaptive and user-oblivious, i.e. they can be automatically adjusted for faster application progress rate. Last, but not the least, this thesis has proposed runtime system based techniques to mitigate the effects of inadequate memory resources, which is not only beneficial for the memory-consuming fault tolerance strategies but also data-intensive applications.

7.1 Contributions

Below we briefly summarize the contributions of this thesis.

- A semi-blocking checkpoint algorithm that significantly reduces checkpoint overhead has been proposed and implemented. By leveraging an adaptive parallel runtime system, the proposed algorithm can automatically schedule the checkpoint communication while optimizing for the application communication.
- A replication enhanced checkpointing mechanism, ACR, is presented to protect applications from both hard and soft errors. Depending on their performance and resilience requirements, ACR provides three recovery schemes for users to choose from.
- A framework that automatically adjusts checkpoint interval based on execution configuration and environment has been presented. For example, this framework can

automatically schedule more checkpoints when the observed failure rate increases in order to reduce the recovery overhead. The presented framework can also be used alongside a failure predictor so that checkpoint can be automatically scheduled just before a failure is predicted to occur.

- FlipBack, a framework that automatically adapts the soft error protection mechanism based on data and computation characteristics, is presented. FlipBack leverages both runtime system capabilities and compilation techniques to improve application resilience against soft errors. Compared to ACR, FlipBack is a more specialized solution that works based on the characteristics of HPC applications and thus reduces the overhead.

- We explore how runtime systems can take advantage of non volatile memory to extend the memory capacity for smooth execution of HPC applications. Using the proposed out-of-core execution framework, Charm-HMC, we show that the time spent on data movements can be easily overlapped with useful computation and thus comparable performance can be obtained for out-of-core execution vis-a-vis in-core computation.

7.2 Future Work

Going forward, several useful extensions and new work can be performed based on this thesis. Here, we list a few such promising ideas.

Data correction post anomaly detection: In FlipBack, spatial and temporal data similarity is used to detect silent data corruptions for the field data of HPC applications. However, the task of correcting such bit flips after detection is yet another challenge. Local recovery is one solution, but it incurs the cost of checkpointing the field data before each computation task. One interesting related study would be to explore the relationship between recovery cost and the local checkpoint frequency. Another possible solution for bit flips is to correct them, rather than merely detecting them, using spatial and temporal smoothness. However, this will most likely introduce inaccuracy and loss of data precision. It will be useful to study the effects of changes in data precision on the stability and correctness of numerical algorithms.

Exploring the memory hierarchy on accelerators: Charm-HMC explores the use of NVM as extension of DRAM for out-of-core computation. We believe a similar idea can be applied to manage the high bandwidth memory (HBM) on next generation accelerators such as Intel Xeon Phi KNL. With simple modifications, Charm-HMC should be able to auto-

matically schedule the computation tasks performed on accelerators and help minimize the data movements between HBM and DRAM.

Concluding remark: We expect the issues associated with resilience to increase in importance within High Performance Computing, and hope that this dissertation has made a significant contribution that can be part of the eventual solution.

REFERENCES

[1] E. Meneses, X. Ni, G. Zheng, C. L. Mendes, and L. V. Kale, "Using migratable objects to enhance fault tolerance schemes in supercomputers," in *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[2] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, "The soft error problem: An architectural perspective," in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*. IEEE, 2005, pp. 243–247.

[3] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: probabilistic soft error reliability on the cheap," in *Architectural support for programming languages and operating systems*, ser. ASPLOS XV. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1736020.1736063 pp. 385–396.

[4] S. Michalak, K. Harris, N. Hengartner, B. Takala, and S. Wender, "Predicting the number of fatal soft errors in los alamos national laboratory's asc q supercomputer," *Device and Materials Reliability, IEEE Transactions on*, vol. 5, no. 3, pp. 329 – 335, sept. 2005.

[5] "How To Kill A Supercomputer," http://www.hpcwire.com/2016/02/24/how-to-kill-a-supercomputer-tips-from-an-expert, 2016.

[6] Z. Zheng, L. Yu, Z. Lan, and T. Jones, "3-dimensional root cause diagnosis via co-analysis," in *Proceedings of the 9th international conference on Autonomic computing*. ACM, 2012, pp. 181–190.

[7] F. Cappello, "Fault tolerance in petascale/ exascale systems: Current knowledge, challenges and research opportunities," *IJHPCA*, vol. 23, no. 3, pp. 212–226, 2009.

[8] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick, "Exascale computing study: Technology challenges in achieving exascale systems," 2008.

[9] A. Gainaru, F. Cappello, M. Snir, and W. Kramer, "Fault prediction under the microscope: A closer look into hpc systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 77.

[10] "TITAN, Oak Ridge National Laboratory," https://www.olcf.ornl.gov/titan.

[11] "Blue Waters, National Center for Supercomputing Applications," http://www.ncsa.illinois.edu/enabling/bluewaters.

[12] "Top500 supercomputing sites," http://top500.org, 2013.

[13] D. M. et al, "Restoring the cpa to cnl," in *Cray User Group Conference*, 2008.

[14] "Nvidia xid errors," http://docs.nvidia.com/deploy/xid-errors/.

[15] C. Di Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons learned from the analysis of system failures at petascale: The case of blue waters," in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*. IEEE, 2014, pp. 610–621.

[16] C. Di Martino, W. Kramer, Z. Kalbarczyk, and R. Iyer, "Measuring and understanding extreme-scale application resilience: A field study of 5,000,000 hpc application runs," in *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*. IEEE, 2015, pp. 25–36.

[17] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale, "Parallel Programming with Migratable Objects: Charm++ in Practice," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. New York, NY, USA: ACM, 2014.

[18] J. Phillips, G. Zheng, and L. V. Kalé, "Namd: Biomolecular simulation on thousands of processors," in *Workshop: Scaling to New Heights*, Pittsburgh, PA, May 2002.

[19] N. Jain, E. Bohm, E. Mikida, S. Mandal, M. Kim, P. Jindal, Q. Li, S. Ismail-Beigi, G. Martyna, and L. Kale, "Openatom: Scalable ab-initio molecular dynamics with diverse capabilities," in *International Supercomputing Conference*, ser. ISC HPC '16 (to appear), june 2016.

[20] H. Menon, L. Wesolowski, G. Zheng, P. Jetley, L. Kale, T. Quinn, and F. Governato, "Adaptive techniques for clustered n-body cosmological simulations," *Computational Astrophysics and Cosmology*, vol. 2, no. 1, pp. 1–16, 2015.

[21] K. M. Chandy and L. Lamport, "Distributed snapshots : Determining global states of distributed systems," *ACM Transactions on Computer Systems*, Feb. 1985.

[22] D. Buntinas, C. Coti, T. Hérault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello, "Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant mpi protocols," *Future Generation Comp. Syst.*, vol. 24, no. 1, pp. 73–84, 2008.

[23] M. Schulz, "Checkpointing," in *Encyclopedia of Parallel Computing*, 2011, pp. 264–273.

[24] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (BLCR) for linux clusters," in *SciDAC*, 2006.

[25] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *SC*, 2010, pp. 1–11.

[26] G. Zheng, L. Shi, and L. V. Kalé, "FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI," in *2004 IEEE Cluster*, San Diego, CA, September 2004, pp. 93–103.

[27] G. Zheng, X. Ni, and L. V. Kale, "A Scalable Double In-memory Checkpoint and Restart Scheme towards Exascale," in *Proceedings of the 2nd Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS)*, Boston, USA, June 2012.

[28] Sayantan Chakravorty, Celso Mendes and L. V. Kale, "Proactive fault tolerance in large systems," in *HPCRI Workshop in conjunction with HPCA 2005*, 2005.

[29] O. Lawlor, M. Bhandarkar, and L. V. Kalé, "Adaptive mpi," Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, Tech. Rep. 02-05, 2002.

[30] E. Meneses, X. Ni, and L. V. Kale, "A Message-Logging Protocol for Multicore Systems," in *Proceedings of the 2nd Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS)*, Boston, USA, June 2012.

[31] "Stampede, Texas Advanced Computing Center," https://portal.tacc.utexas.edu/user-guides/stampede.

[32] E. Meneses, O. Sarood, and L. V. Kale, "Assessing Energy Efficiency of Fault Tolerance Protocols for HPC Systems," in *Proceedings of the IEEE 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, New York, USA, October 2012.

[33] R. Strom and S. Yemini, "Optimistic recovery in distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 3, pp. 204–226, 1985.

[34] L. Alvisi and K. Marzullo, "Message logging: pessimistic, optimistic, and causal," *International Conference on Distributed Computing Systems*, pp. 229–236, 1995.

[35] E. Meneses, G. Bronevetsky, and L. V. Kale, "Evaluation of simple causal message logging for large-scale fault tolerant HPC systems," in *16th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems in 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2011).*, May 2011.

[36] E. Meneses, "Scalable message-logging techniques for effective fault tolerance in HPC applications," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, 2013.

[37] S. Chakravorty and L. V. Kale, "A fault tolerance protocol with fast fault recovery," in *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*. IEEE Press, 2007.

[38] S. Fu and C.-Z. Xu, "Exploring event correlation for failure prediction in coalitions of clusters," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, ser. SC '07. New York, NY, USA: ACM, 2007, pp. 41:1–41:12.

[39] A. Gainaru, F. Cappello, M. Snir, and W. Kramer, "Fault prediction under the microscope: A closer look into hpc systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 77:1–77:11.

[40] Z. Lan, J. Gu, Z. Zheng, R. Thakur, and S. Coghlan, "A study of dynamic meta-learning for failure prediction in large-scale systems," *J. Parallel Distrib. Comput.*, vol. 70, no. 6, pp. 630–643, June 2010.

[41] O. S. Lawlor and L. V. Kalé, "Supporting dynamic parallel object arrays," *Concurrency and Computation: Practice and Experience*, vol. 15, pp. 371–393, 2003.

[42] S. Chakravorty, C. L. Mendes, and L. V. Kalé, "Proactive fault tolerance in mpi applications via task migration." in *HiPC*, ser. Lecture Notes in Computer Science, vol. 4297. Springer, 2006, pp. 485–496.

[43] G. Antoniu, L. Bouge, and R. Namyst, "An efficient and transparent thread migration scheme in the PM^2 runtime system," in *Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP) San Juan, Puerto Rico. Lecture Notes in Computer Science 1586*. Springer-Verlag, April 1999, pp. 496–510.

[44] "Ranger, Texas Advanced Computing Center," https://portal.xsede.org/tacc-ranger.

[45] "SUMMIT, Oak Ridge National Laboratory," https://www.olcf.ornl.gov/summit.

[46] I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," in *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2012, pp. 1222–1230.

[47] B. Van Essen, R. Pearce, S. Ames, and M. Gokhale, "On the role of nvram in data-intensive architectures: an evaluation," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 2012, pp. 703–714.

[48] X. Ni, E. Meneses, and L. V. Kalé, "Hiding checkpoint overhead in HPC applications with a semi-blocking algorithm," in *IEEE Cluster 12*, Beijing, China, September 2012.

[49] L. Bautista-Gomez, D. Komatitsch, N. Maruyama, S. Tsuboi, F. Cappello, and S. Matsuoka, "FTI: High performance fault tolerance interface for hybrid systems," in *Supercomputing*, Nov. 2011, pp. 1 –12.

[50] S. L. Graham, M. Snir, and C. A. Patterson, Eds., *Getting Up to Speed, The Future of Supercomputing*. The National Academies Press, 2006.

[51] X. Dong, N. Muralimanohar, N. P. Jouppi, R. Kaufmann, and Y. Xie, "Leveraging 3D PCRAM technologies to reduce checkpoint overhead for future exascale systems," in *SC*, 2009.

[52] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello, "Uncoordinated checkpointing without domino effect for send-deterministic mpi applications," in *IPDPS*, 2011, pp. 989–1000.

[53] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (BLCR) for Linux clusters," *Journal of Physics Conference Series*, vol. 46, pp. 494–499, Sep. 2006.

[54] K. Li, J. Naughton, and J. Plank, "Low-latency, concurrent checkpointing for parallel programs," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 5, no. 8, pp. 874 –879, aug 1994.

[55] N. H. Vaidya, "Staggered consistent checkpointing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 10, no. 7, pp. 694–702, 1999.

[56] X. Ouyang, S. Marcarelli, and D. Panda, "Enhancing checkpoint performance with staging io and ssd," in *Storage Network Architecture and Parallel I/Os (SNAPI), 2010 International Workshop on*, may 2010, pp. 13 –20.

[57] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of OOPSLA'93*, A. Paepcke, Ed. ACM Press, September 1993, pp. 91–108.

[58] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Generation Comp. Syst.*, vol. 22, no. 3, pp. 303–312, 2006.

[59] C. A. Waldspurger and W. E. Weihl, "Lottery scheduling: flexible proportional-share resource management," in *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, ser. OSDI '94. Berkeley, CA, USA: USENIX Association, 1994. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267638.1267639

[60] F. Gioachin, A. Sharma, S. Chakravorty, C. Mendes, L. V. Kale, and T. R. Quinn, "Scalable cosmology simulations on parallel machines," in *VECPAR 2006, LNCS 4395, pp. 476-489*, 2007.

[61] X. Ni, E. Meneses, N. Jain, and L. V. Kale, "Acr: Automatic checkpoint/restart for soft and hard error protection," in *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. IEEE Computer Society, Nov. 2013.

[62] C. Engelmann, H. H. Ong, and S. L. Scott, "The Case for Modular Redundancy in Large-Scale High Performance Computing Systems," in *International Conference on Parallel and Distributed Computing and Networks (PDCN) 2009*. ACTA Press, Calgary, AB, Canada, Feb. 2009. [Online]. Available: http://www.csm.ornl.gov/~{}engelman/publications/engelmann09case.pdf pp. 189–194.

[63] K. Ferreira, J. Stearley, J. H. Laros, III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold, "Evaluating the viability of process replication reliability for exascale systems," in *Supercomputing*. New York, NY, USA: ACM, 2011, pp. 44:1–44:12.

[64] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell, "Detection and correction of silent data corruption for large-scale high-performance computing," in *Supercomputing*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389102 pp. 78:1–78:12.

[65] Z. Lan, J. Gu, Z. Zheng, R. Thakur, and S. Coghlan, "A study of dynamic meta-learning for failure prediction in large-scale systems," *J. Parallel Distrib. Comput.*, vol. 70, no. 6, pp. 630–643, June 2010.

[66] H. Menon, N. Jain, G. Zheng, and L. V. Kalé, "Automated load balancing invocation based on application characteristics," in *IEEE Cluster 12*, Beijing, China, September 2012.

[67] B. Schroeder and G. Gibson, "A large scale study of failures in high-performance-computing systems," in *International Symposium on Dependable Systems and Networks (DSN)*, 2006.

[68] M. Bougeret, H. Casanova, M. Rabie, Y. Robert, and F. Vivien, "Checkpointing strategies for parallel jobs," in *Supercomputing*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 33:1–33:11.

[69] Y. Ling, J. Mi, and X. Lin, "A variational calculus approach to optimal checkpoint placement," *Computers, IEEE Transactions on*, vol. 50, no. 7, pp. 699–708, 2001.

[70] S. Genaud, C. Rattanapoka, and U. L. Strasbourg, "A peer-to-peer framework for robust execution of message passing parallel programs," in *In EuroPVM/MPI 2005, volume 3666 of LNCS*. Springer-Verlag, 2005, pp. 276–284.

[71] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou, "Algorithm-based fault tolerance applied to high performance computing," *JPDC*, vol. 69, no. 4, pp. 410–416, 2009.

[72] J. Chung, I. Lee, M. Sullivan, J. H. Ryoo, D. W. Kim, D. H. Yoon, L. Kaplan, and M. Erez, "Containment domains: a scalable, efficient, and flexible resilience scheme for exascale systems," in *Supercomputing*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389075 pp. 58:1–58:11.

[73] "Fletcher checksum algorithm wiki page," http://en.wikipedia.org/wiki/Fletcher's_checksum.

[74] D. K. Pradhan and N. H. Vaidya, "Roll-forward and rollback recovery: Performance-reliability trade-off," in *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on*. IEEE, 1994, pp. 186–195.

[75] J. Vetter, "Hpc landscape application accelerators: Deus ex machina?" Invited Talk at High Performance Embedded Computing Workshop, Sep. 2009.

[76] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *Device and Materials Reliability, IEEE Transactions on*, vol. 5, no. 3, pp. 305–316, 2005.

[77] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," Sandia National Laboratories, Tech. Rep., September 2009.

[78] "Lulesh," http://computation.llnl.gov/casc/ShockHydro/.

[79] L. Kale, A. Arya, A. Bhatele, A. Gupta, N. Jain, P. Jetley, J. Lifflander, P. Miller, Y. Sun, R. Venkataraman, L. Wesolowski, and G. Zheng, "Charm++ for productivity and performance: A submission to the 2011 HPC class II challenge," Parallel Programming Laboratory, Tech. Rep. 11-49, November 2011.

[80] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kalé, and K. Schulten, "Scalable molecular dynamics with NAMD," *Journal of Computational Chemistry*, vol. 26, no. 16, pp. 1781–1802, 2005.

[81] C. Huang, G. Zheng, S. Kumar, and L. V. Kalé, "Performance Evaluation of Adaptive MPI," in *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006*, March 2006.

[82] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: probabilistic soft error reliability on the cheap," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1. ACM, 2010, pp. 385–396.

[83] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: Software implemented fault tolerance," in *Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society, 2005, pp. 243–254.

[84] K. S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, and R. Iyer, "Hauberk: Lightweight silent data corruption error detector for gpgpu," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011, pp. 287–300.

[85] I. Laguna, M. Schulz, D. F. Richards, J. Calhoun, and L. Olson, "Ipas: intelligent protection against silent output corruption in scientific applications," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. ACM, 2016, pp. 227–238.

[86] L. Spainhower and T. A. Gregg, "Ibm s/390 parallel enterprise server g5 fault tolerance: A historical perspective," *IBM Journal of Research and Development*, vol. 43, no. 5.6, pp. 863–873, 1999.

[87] W. Bartlett and L. Spainhower, "Commercial fault tolerance: A tale of two systems," *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, no. 1, pp. 87–96, 2004.

[88] A. Benso, S. Di Carlo, G. Di Natale, and P. Prinetto, "A watchdog processor to detect data and control flow errors," in *On-Line Testing Symposium, 2003. IOLTS 2003. 9th IEEE*. IEEE, 2003, pp. 144–148.

[89] K. S. Yim, "Characterization of impact of transient faults and detection of data corruption errors in large-scale n-body programs using graphics processing units," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014, pp. 458–467.

[90] S. Di and F. Cappello, "Adaptive impact-driven detection of silent data corruption for hpc applications."

[91] L. Bautista-Gomez and F. Cappello, "Detecting and correcting data corruption in stencil applications through multivariate interpolation," in *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*. IEEE, 2015, pp. 595–602.

[92] O. Subasi, S. Di, L. Bautista-Gomez, P. Balaprakash, O. Unsal, J. Labarta, A. Cristal, and F. Cappello, "Spatial support vector regression to detect silent errors in the exascale era," in *Proceedings of the 2015 IEEE/ACM International Symposium on Cluster Cloud and Grid Computing*. IEEE, 2016.

[93] J. Sloan, R. Kumar, and G. Bronevetsky, "Algorithmic approaches to low overhead fault detection for sparse linear algebra," in *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*. IEEE, 2012, pp. 1–12.

[94] F. Oboril, M. B. Tahoori, V. Heuveline, D. Lukarski, and J.-P. Weiss, "Numerical defect correction as an algorithm-based fault tolerance technique for iterative solvers," in *Dependable Computing (PRDC), 2011 IEEE 17th Pacific Rim International Symposium on*. IEEE, 2011, pp. 144–153.

[95] P. Du, P. Luszczek, and J. Dongarra, "Algorithm-based fault tolerance method for soft error resilience in high-performance linpack," in *IEEE Cluster*, 2011.

[96] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the accuracy of high-level fault injection techniques for hardware faults," in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*. IEEE, 2014, pp. 375–382.

[97] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[98] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving Performance via Mini-applications," Sandia National Laboratories, Tech. Rep. SAND2009-5574, 2009.

[99] R. V. der Wijngaart, A. Kayi, J. Hammond, G. Jost, T. John, S. Sridharan, T. G. Mattson, J. Abercrombie, and J. Nelson, "Comparing runtime systems with exascale ambitions using the parallel research kernels," ser. International Supercomputing Conference ISC, 2016.

[100] J. S. Vitter, "Algorithms and data structures for external memory," *Foundations and Trends® in Theoretical Computer Science*, vol. 2, no. 4, pp. 305–474, 2008.

[101] A. D. Brown, T. C. Mowry, and O. Krieger, "Compiler-based I/O prefetching for out-of-core applications," *ACM Transactions on Computer Systems (TOCS)*, vol. 19, no. 2, pp. 111–170, 2001.

[102] Y. Park, R. Scott, and S. Sechrest, "Virtual memory versus file interface for large, memory-intensive scientific applications," in *Supercomputing, 1996. Proceedings of the 1996 ACM/IEEE Conference on*. IEEE, 1996, pp. 53–53.

[103] C.-H. Lee, M. C. Chen, and R.-C. Chang, "Hipec: high performance external virtual memory caching," in *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*. USENIX Association, 1994, p. 12.

[104] R. T. Mills, A. Stathopoulos, and D. S. Nikolopoulos, "Adapting to memory pressure from within scientific applications on multiprogrammed cows," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. IEEE, 2004, p. 71.

[105] J. C. Mogul, E. Argollo, M. A. Shah, and P. Faraboschi, "Operating system support for nvm+ dram hybrid main memory." in *HotOS*, 2009.

[106] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," *ACM SIGPLAN Notices*, vol. 46, no. 3, pp. 91–104, 2011.

[107] S. Ko, S. Jun, Y. Ryu, O. Kwon, and K. Koh, "A new linux swap system for flash memory storage devices," in *Computational Sciences and Its Applications, 2008. ICCSA'08. International Conference on*. IEEE, 2008, pp. 151–156.

[108] S.-y. Park, D. Jung, J.-u. Kang, J.-s. Kim, and J. Lee, "Cflru: a replacement algorithm for flash memory," in *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*. ACM, 2006, pp. 234–241.

[109] C. Wang, S. S. Vazhkudai, X. Ma, F. Meng, Y. Kim, and C. Engelmann, "Nvmalloc: Exposing an aggregate ssd store as a memory partition in extreme-scale machines," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 2012, pp. 957–968.

[110] B. Van Essen, H. Hsieh, S. Ames, R. Pearce, and M. Gokhale, "Di-mmap: a scalable memory-map runtime for out-of-core data-intensive applications," *Cluster Computing*, vol. 18, no. 1, pp. 15–28, 2015.

[111] A. Badam and V. S. Pai, "Ssdalloc: hybrid ssd/ram memory management made easy," in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*. USENIX Association, 2011, pp. 211–224.

[112] J. J. Dongarra, S. Hammarling, and D. W. Walker, "Key concepts for parallel out-of-core lu factorization," *Parallel Computing*, vol. 23, no. 1, pp. 49–70, 1997.