5-1-2016

Using Rollback Avoidance to Mitigate Failures in Next-Generation Extreme-Scale Systems

Scott Levy

Scott Levy

Candidate

Computer Science

Department

This dissertation is approved, and it is acceptable in quality and form for publication:

Approved by the Dissertation Committee:

Patrick G. Bridges, Chair

Kurt B. Ferreira

Dorian Arnold

David Lowenthal

Using Rollback Avoidance to Mitigate Failures in Next-Generation Extreme-Scale Systems

by

Scott Levy

B.S., Electrical Engineering, Cornell University, 1995
J.D., Lewis & Clark Law School, 2004

DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

May 2016

Dedication

To Jane, for her steadfast support

"It seems very pretty," she said when she had finished it, "but it's RATHER hard to understand!" (You see she didn't like to confess, even to herself, that she couldn't make it out at all.) "Somehow it seems to fill my head with ideas–only I don't exactly know what they are! However, SOMEBODY killed SOMETHING: that's clear, at any rate"

Through the Looking Glass, Lewis Carroll

Acknowledgments

This document is the culmination of nearly six years of work. While I am credited as its sole author, neither this document nor the work that it summarizes would have been possible without significant contributions from many other people. I would like to begin by acknowledging my advisor, Patrick Bridges. Patrick's guidance throughout this process has been invaluable. I learned more from Patrick about the practice of computer science research and the art of clearly communicating detailed technical ideas than I can properly summarize in this constrained space.

I would also like to recognize the significant contributions of Kurt Ferreira. Throughout my tenure at Sandia, Kurt served as my *de facto* second advisor. He taught me how to navigate the technical details of working with large-scale systems and the process of publishing research papers. When I was in law school, the faculty would frequently tell us that their job was to teach us to think like lawyers. Kurt taught me to think like a computer scientist. For that, I will be forever grateful.

I would also like to acknowledge the contributions of the other two members of my committee, Dorian and Dave. I was fortunate to collaborate with Dorian on several projects (some related to this dissertation, some not) during my time as a student as UNM. He forced me to challenge my assumptions and to think carefully about the objectives of computer science research. I appreciate that Dave encouraged me to take risks, to build things and conduct experiments even when success was not guaranteed.

There are many others who contributed to the success of this undertaking. I was very fortunate to spend several years of my career as a Ph.D. student working as an intern at Sandia National Laboratories. I am grateful for the opportunity; it was an integral part of my education. In particular, I had the opportunity to collaborate with Patrick Widener on several occasions and my writing and research skills improved from the experience. Additionally, Ryan Grant, Simon Hammond, and Kevin Pedretti provided invaluable technical guidance on several occasions. At UNM, my interactions (both technical and not) with the members of the Scalable Systems Laboratory helped me navigate my graduate school experience. In particular, I would like to express my gratitude to: Matthew Dosanjh, Taylor Groves, Dewan Ibtesham, Donour Sizemore and Ricardo Villalon. The education that I received before I matriculated at UNM provided an integral foundation for this undertaking. I

was fortunate to be a student of many talented teachers, including Cheryl Ogden, John Garing, and Michael Blumm.

Finally, I would like to express my gratitude to my family. I would like to thank my parents for providing me with the opportunities that made this endeavor possible. They taught me perseverence and the importance of getting the details right; values that I hope are reflected in this document. I could not have even begun this journey without the guidance, encouragement, and confidence of my wife, Jane. Being married to a Ph.D. student (perhaps especially to *this* Ph.D. student) is no easy task. She handled it with aplomb. Thank you, Jane.

Using Rollback Avoidance to Mitigate Failures in Next-Generation Extreme-Scale Systems

by

Scott Levy

B.S., Electrical Engineering, Cornell University, 1995
J.D., Lewis & Clark Law School, 2004

Ph.D., Computer Science, University of New Mexico, 2016

Abstract

High-performance computing (HPC) systems enable scientists to numerically model complex phenomena in many important physical systems. The next major milestone in the development of HPC systems is the construction of the first supercomputer capable executing more than an exaflop, 10^{18} floating point operations per second. On systems of this scale, failures will occur much more frequently than on current systems. As a result, resilience is a key obstacle to building next-generation extreme-scale systems. Coordinated checkpointing is currently the most widely-used mechanism for handling failures on HPC systems. Although coordinated checkpointing remains effective on current systems, increasing the scale of today's systems to build next-generation systems will increase the cost of fault tolerance as more and more time is taken away from the application to protect against or recover from failure. Rollback avoidance techniques seek to mitigate the cost of checkpoint/restart by allowing an application to continue its execution rather than rolling back to

an earlier checkpoint when failures occur. These techniques include failure prediction and preventive migration, replicated computation, fault-tolerant algorithms, and software-based memory fault correction. In this thesis, I examine how rollback avoidance techniques can be used to address failures on extreme-scale systems. Using a combination of analytic modeling and simulation, I evaluate the potential impact of rollback avoidance on these systems. I then present a novel rollback avoidance technique that exploits similarities in application memory. Finally, I examine the feasibility of using this technique to protect against memory faults in kernel memory.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

High-performance computing (HPC) systems are a critical resource for scientists conducting cutting-edge research. Large dedicated machines enable scientists to numerically model complex phenomena that are otherwise difficult or impossible to study. These systems facilitate the development of scientific codes that are capable of high-fidelity simulations of a variety of important physical systems including climate and weather in the earth's atmosphere, combustion in next-generation engines, and the behavior of complex pathways in biological cells [14]. Detailed simulations of these phenomena enable scientists across many disciplines to make important scientific discoveries.

The next major milestone in the development of HPC systems is the construction of the first supercomputer capable executing more than an exaflop, 10^{18} floating point operations per second. Currently, the fastest supercomputers in the world are capable of executing a few tens of petaflops (10^{15} floating point operations per second) [5]. The first system capable of exaflops, an *exascale* system, is projected to be available as early as 2023 [85].

Resilience is a key obstacle to building next-generation extreme-scale systems [50,

62]. Observations on current systems show that although there are many sources of failure, hardware failures dominate [148, 149]. Building an exascale computer will likely require hundreds of thousands of processors and tens to hundreds of petabytes of memory [8]. Aggregating more and more components will increase the frequency with which these more powerful systems experience failure. Assuming that the failure rate of each individual node is identical, then the system mean time between failures (MTBF) is inversely proportional to the total number of nodes in the system [79, 139, 148]. As a result, next-generation systems could experience multiple failures per hour [50].

More frequent failure may mean that many current mitigation techniques will no longer be sufficient. Effective and efficient mechanisms for recovering from or avoiding failures may therefore be necessary for next-generation scientific applications to make meaningful forward progress on future systems.

1.1 Memory Faults in HPC Systems

Failures in current HPC systems are caused by many different fault types. Memory-related failures are one of the most frequently observed sources of node failure in large-scale distributed systems [149]. As a result, significant effort has been devoted to hardening applications against memory faults. The sheer volume of memory required to build next-generation extreme-scale systems combined with device trends, such as shrinking feature sizes, have the potential to increase the frequency of memory faults [96, 121, 122]. As a result, current projections suggest that a memory failure could happen as frequently as once per hour on next-generation systems [112].

Memory resilience issues may be exacerbated by attempts to address power concerns. Delivering power to next-generation systems is projected to be a significant challenge. Even accounting for advances in technology, scaling up today's systems to

reach exascale could require more than 100 megawatts (MW) of electricity [44, 62]. The monetary cost coupled with the technical challenges associated with delivering that much power make this approach infeasible. As result, the current power budget for exascale is 20 MW [62].

Memory in large-scale systems consumes a significant fraction of total system power. Using today's memory technology (e.g., DDR3) to construct an exascale system might require up to 50 MW to power the memory subsystem alone [72, 150]. Recent advances in memory technology (e.g., DDR4) are projected to reduce the memory power by as much as half [72, 147], but significant increases in memory efficiency will still be required to stay within the exascale power budget. However, gains in the power efficiency of memory devices frequently come at the expense of reliability [22, 72, 95].

Memory in HPC systems is typically partitioned between user-level applications and the kernel. Although the kernel occupies a small fraction of the memory used by the system, the consequences of memory failures in kernel memory are more severe than failures that occur in application memory [55]. A failure that causes the operating system to crash will limit the effectiveness of the many techniques that have been developed to protect against failures in application memory. Currently, most current HPC operating systems provide no memory protection beyond that provided by the hardware (e.g., error-correcting codes (ECC)). However, given the frequency of accesses to kernel memory there is evidence that failures in these regions of memory are more common than failures in other regions of memory [87].

1.2 Handling Failures with Checkpoint/Restart

A common approach to handling faults in HPC systems is checkpoint/restart. The basic idea is that each application process periodically saves its state, a *checkpoint*,

to a persistent storage facility. On today's systems, this frequently means writing to a global filesystem. When a failure occurs, the system must identify a set of checkpoints, one for each application process, that represent a *consistent global state* of the system. A consistent global state means that every message in the system that has been received by one application process has also been sent by another application process (i.e., there are no *orphan* messages). A set of checkpoints is *strongly consistent* if it represents a consistent global state and every message in the system that has been sent by one application process has also been received by another application process (i.e., there are no *lost* messages). After the system has identified a set of checkpoints that satisfy this condition, each application process restores its state from its checkpoint and resumes its computation.

1.2.1 Coordinated Checkpoint/Restart

Coordinated checkpointing is currently the most widely-used mechanism for handling failures on HPC systems. Coordinated checkpointing works by ensuring that all checkpoints are taken at the same logical time.[1] Although coordinated checkpointing remains effective on current systems, increasing the scale of these systems to build next-generation systems will increase the cost of fault tolerance as more and more time is taken away from the application to protect against or recover from failure. In particular, increasing the number of application processes increases contention for the parallel file system because each application process must simultaneously write its checkpoint to the global file system. As a result, committing checkpoints becomes more and more expensive as system scale increases. The combination of more frequent failures and more expensive checkpoints means that at the scales projected for the first exascale system, less than half of the system's time may be

[1] Logical time is a temporal abstraction that allows causality relationships to be reliably established in distributed systems, *see* [102].

available for advancing an application's computation [52, 126].

1.3 Reducing the Cost of Coordinated Checkpoint/Restart

Based on the dire predictions of the cost of coordinated checkpoint-restart on next-generation systems, many approaches have been proposed to reduce its performance impact. These include alternatives to coordinated checkpoint/restart and techniques that correct errors as they occur, reducing the need to rollback to a previous checkpoint.

1.3.1 Improving Checkpoint Write Performance

Contention for bandwidth to stable storage is one of the principal reasons that coordinated checkpointing is projected to scale poorly. As a result, a number of approaches to reduce the cost of writing checkpoints have been proposed. These approaches include techniques for improving global file system performance (e.g., the Parallel Log-structured File System (PLFS) [17]) and techniques for avoiding the global file system altogether (e.g., by writing checkpoints to node-local memory [43, 131, 169] or solid-state drives (SSDs) [18]).

1.3.2 Uncoordinated Checkpoint/Restart

Another approach to reducing the overhead of coordinated checkpoint/restart is to relax the requirement that all application processes commit checkpoints simultaneously. However, without coordinating the writing of checkpoints, the system cannot

guarantee that any set of the checkpoints taken represent a consistent global state of the machine. The resulting phenomenon is known as the *domino effect*; when one application process rolls back to an earlier checkpoint, communication dependencies force other processes to also roll back to resolve inconsistencies. In the worst case, the only way to reconstruct a consistent global state is to start the application again from the beginning [46].

There are a number of techniques for avoiding the domino effect. One widely-studied approach is to assume that the application is *piecewise deterministic* and augment checkpoint/restart by storing logs of messages that each process sends to (or, alternatively, receives from) its peers. In the execution of a piecewise-deterministic application, the only non-deterministic events are the receipt of messages; if the order of received messages is fixed, the application is deterministic. When a failure occurs, the failed process rolls back to its most recent checkpoint and resumes execution. As the failed process recovers, the messages that the process received during its original execution are replayed from the logs. If the failed application process is piecewise deterministic, this method guarantees that it will be restored to precisely the same state it was in when the failure occurred [46].

1.3.3 Rollback Avoidance

Rollback avoidance is the set of techniques that allow the application to continue its execution rather than rolling back to an earlier checkpoint when a failure occurs. Many such approaches have been proposed to reduce the performance impact of failures on checkpoint/restart systems. These include failure prediction and preventive migration [33, 66], replication-based approaches [45, 52, 59], fault-tolerant algorithms [29, 37, 38, 86, 97], and software-based memory fault correction [57, 110, 151]. The common principle underlying these approaches is that enabling an application

to continue executing (perhaps with some degradation) despite the occurrence or imminence of a failure will improve its performance.

1.4 Thesis Statement

In this thesis, I examine how rollback avoidance techniques can be used to address failures on extreme-scale systems. My hypothesis is that rollback avoidance techniques can be effectively used to address fault tolerance concerns on next-generation systems. I evaluate this hypothesis in the following ways:

- I develop and validate an analytic model of the impact of rollback avoidance on application performance. I use this model to evaluate the benefits of existing fault tolerance techniques and to project the benefits of future techniques on next-generation systems in Chapter 3. I also use this model to examine the impact of a novel rollback avoidance technique on application performance in Chapter 5.

- I describe and validate a simulation framework that was developed in collaboration with several colleagues. I then use this framework to examine the impact of rollback avoidance on applications using uncoordinated checkpointing with message logging in Chapter 4.

- I present a novel rollback avoidance technique that leverages memory content similarity. In Chapter 5, I describe a software library for extracting memory content similarity: the *Similarity Engine*. I then examine how the information collected by this library could be exploited to avoid rollback.

- I evaluate the viability of my similarity-based rollback avoidance technique for protecting kernel memory in Chapter 6. I examine snapshots of kernel memory

for two popular HPC operating systems to demonstrate that memory content similarity could also be used to avoid rollback due to failures in kernel memory.

Chapter 2

Related Work

This chapter examines the existing research literature that is related to the research contributions presented in this document. This examination is structured in the following way. Section 2.2 provides an overview of the current state of research on checkpoint/restart methods for fault tolerance. Section 2.3 describes the existing research on the development of rollback avoidance techniques. Sections 2.4 and 2.5 describe previous efforts to model and simulate, respectively, fault tolerance mechanisms on large-scale HPC systems. Section 2.6 describes earlier attempts to exploit similarities in memory contents. Finally, Section 2.7 examines how the contributions of this document are novel and distinct from the existing research.

2.1 Terminology

2.1.1 Binary Prefixes

Throughout this dissertation, I use the binary prefixes defined by the International Electrotechnical Commission (IEC) in the IEC 60027-2 standard to indicate binary

orders of magnitude For example, a KiB is a *kibibyte*, 2^{10} bytes. I extend this notation and apply it to indicate the number of processing elements used to run an application (e.g., 1 Ki processes is equivalent to 1024 processes).

2.1.2 Fault Tolerance

Discussing the current state of research on fault tolerance for extreme-scale systems requires first establishing a taxonomy of system misbehavior. Despite efforts at standardization, the terms used to describe system misbehavior are not always consistently defined in the literature (*cf.* [16, 70, 79, 100, 104, 123]). In this document, I adopt the terminology proposed by Gärtner [70]. Therefore, I define a *fault* as the lowest-level of misbehavior in the system. A common example of a fault is the case of a "stuck bit", where reading the value of a memory cell always yields the same value regardless of the value that has been written to it. A *failure* occurs when the systems deviates from its specified behavior. In this taxonomy, each failure is the manifestation of one or more faults. Thus, a computing system is *fault-tolerant* to the extent that it is able to prevent faults from leading to failures.

2.2 Checkpoint/Restart

The dominant approach to fault tolerance in today's largest systems is checkpoint/-restart.[1] The basic approach underlying all checkpoint/restart protocols is to periodically capture the state of the application (a *checkpoint*) and write it to some form of persistent storage. When a failure occurs, the checkpoint can be used to *restart* the application without necessarily requiring it to start over from the beginning.

[1] Additional detail on the evolution and development of checkpoint/restart-based fault tolerance techniques is available from Elnozahy et al. [46, 47].

One of the key challenges faced by checkpoint/restart protocols is identifying a set of checkpoints that represent a consistent global state. A *consistent global state* is one in which every message in the system that has been received by one application process has also been sent by another application process (i.e., there are no *orphan* messages). A set of checkpoints is *strongly consistent* if it represents a consistent global state and every message in the system that has been sent by one application process has also been received by another application process (i.e., there are no *lost* messages).

2.2.1 Coordinated Checkpoint/Restart

Coordinated checkpointing is currently the most widely-used mechanism for handling failures on HPC systems. Coordinated checkpointing works by ensuring that all checkpoints are taken at the same logical time. The most popular approach to ensuring this condition is *stop-and-sync*; stop-and-sync pauses each application process and waits for all in-flight messages to finish transmission before taking a checkpoint. The result is a set of checkpoints that represent a strongly consistent global state. Another less commonly used approach is the *Chandy-Lamport distributed snapshot* algorithm [127]. When this algorithm is used, checkpoints are initiated by a single application process. The initiating process transmits a *marker* on each of its communication channels. When a process receives a marker, it saves its current state if it has not already done so. If the process has already saved its state when a marker arrives on channel c, then it logs all of the messages that have arrived on c since the process saved its state. The resulting checkpoint is the combination of the saved state and the message logs for each application process. Every checkpoint taken in this manner is guaranteed to represent a globally consistent state [35].

Although coordinating when a checkpoint is taken guarantees the existence of a

consistent set of checkpoints, it introduces additional costs as well. At the end of each checkpoint interval, every application process attempts to write its checkpoint to persistent storage. In current systems, the persistent storage for checkpoints is typically a parallel file system. As a result, contention for file system resources reduces the bandwidth available to each process and the time required to commit a complete set of checkpoints increases. Significant effort has been dedicated to reducing the time required to store checkpoint data.

Increasing Write Performance

One approach to reducing the cost of checkpoints is to improve the speed at which data can be written to persistent storage. The Parallel Log-structured File System (PLFS) is an interposition layer that aims to improve checkpoint write bandwidth by arranging file system accesses to minimize contention for file system resources [17].

Another approach is to reduce contention by decentralizing the storage resources. *Diskless checkpointing* improves the speed at which checkpoints can be saved by storing the checkpoints in main memory rather than writing them to a parallel file system.[2] Plank et al. [128,129,131] proposed introducing m additional processes into the application for the purpose of storing checkpoint parity data. In this approach, each application process stores its checkpoint in a region of its main memory. The additional processors store parity checkpoints: the bitwise exclusive-or of all of the checkpoints taken by the application processes.

Alternatively, the process of computing the parity checkpoint can be tailored to the application to leverage algorithm-specific characteristics [128]. Thus, if any

[2]Although main memory is not persistent, reasonable assumptions can be made about the occurrence of faults in the system such that, with high probability, a process's checkpoint data will survive the failure of the process itself.

process fails, its checkpoint data can be reconstructed by the surviving processes. Silva and Silva [152] refined this basic approach by eliminating the processes dedicated to storing parity checkpoints. Specifically, they evaluated two approaches: *neighbor-based checkpointing*, where each process stores its checkpoint and a checkpoint from one of its neighbors; and *parity-based checkpointing*, where each process stores its own checkpoint in main memory and a designated processor also stores the parity checkpoint. Although these techniques can reduce the time required to write a checkpoint, diskless checkpointing may also significantly increase the total volume of memory required to run a given application.

Double checkpointing creates pairs of processors, called *buddy processors*, that each store duplicates of the other's checkpoints [43, 169]. When a failure occurs, there is a high probability that the buddy of the failed processor survives and is able to supply the lost checkpoint data.

The Scalable Checkpoint/Restart Library (SCR) [119] builds on these ideas to create a system of hierarchical distributed storage for checkpoint data. The lowest level of the hierarchy consists of node-local storage and is used to store the most recent checkpoint. While this level of the hierarchy provides the fastest access to storage resources, it only protects against failures that affect a small portion of the system. Higher levels of the hierarchy store older checkpoints using more distant storage resources that protect against failures that affect larger portions of the system.

Emerging storage technologies may also be able to improve checkpoint write performance. In particular, the speed and reliability of solid-state drives (SSDs) may allow for node-local or rack-local persistent checkpoint storage [18]. The principal technical challenge to this approach is that the capacity of SSDs is limited. As a result, the viability of SSDs for local checkpoint storage is dependent on the development of effective techniques for managing their limited storage capacity.

Reducing Checkpoint Volume

Another approach to reducing the time required to take a checkpoint is to reduce the volume of the data that must be saved in the checkpoint. In addition to application-directed checkpointing [153], a number of application-independent techniques for reducing the volume of checkpoint data have been proposed. *Incremental checkpointing* reduces the size of a checkpoint by only considering regions of memory that have been modified in the interval since the previous checkpoint was taken [7, 51, 56, 132].

Using conventional file compression techniques to reduce checkpoint volume has also been thoroughly studied [111, 120, 130, 132]. Building on this legacy, Ibtesham et al. have undertaken a thorough examination of the viability of checkpoint compression on modern systems [88, 89]. Their work includes an analytical model for identifying the circumstances when compression decreases checkpoint commit time. Similar techniques have been used across processors.

The mcrEngine [90] leverages semantic information in HDF5 checkpoints to aggregate and compress checkpoints from multiple processes. Similarly, Nicolae [124] has developed a framework that removes duplicate memory pages from system-level checkpoints.

Reducing Checkpoint Frequency

The overheads of coordinated checkpoint/restart may also be reduced by taking fewer checkpoints. On systems for which the time between failures is distributed according to the Weibull distribution, *lazy checkpointing* gradually increases the checkpoint interval as the time that has elapsed since the last failure occurred increases [159]. This technique is based on the observation that for Weibull-distributed failures, the probability of a failure decreases as a function of time. Because failures are increasingly unlikely as previous failures grow more distant in the past, the risk of lost work

also decreases and checkpoints can be taken less frequently without significantly degrading the application's time-to-solution.

2.2.2 Uncoordinated Checkpoint/Restart

Many alternative approaches to coordinated checkpoint/restart have been proposed due to projections about the prohibitive costs associated with this technique on next-generation systems. One prominent set of alternatives is uncoordinated checkpoint/restart protocols. The motivation is that relaxing the requirement that all checkpoints be computed at the same logical time can reduce contention for storage resources. However, when the checkpoints are not coordinated, each time one process rolls back to an earlier checkpoint, interprocess dependencies may force other processes to roll back as well. As a result, when a failure occurs and the failed process restarts from an earlier checkpoint, cascading rollbacks may require the application to start over from the beginning: the *domino effect* [135].

The most commonly-used antidote to the the domino effect is message logging (*see* [10]). For applications that are *piecewise-deterministic* (i.e., given a sequence of messages the application's computation is deterministic), logging all sent messages guarantees that when a failure occurs the failed process be restored to precisely the same state it was in when the failure occurred [46]. For example, when a failure occurs, the failed process can be restarted from its most recent checkpoint and all of the messages it received since that checkpoint was taken can be re-sent. Messages can be logged by either the sender [171] or the receiver [93, 157]. For sender-based logging, the logs are commonly kept in volatile memory [171]. However, for receiver-based logging, the logs must be stored in some form of persistent storage so that when a process fails its message logs are not lost. Additionally, messages can be pessimistically logged (i.e., logged immediately upon reception/transmission) or optimistically

logged (i.e., logged asynchronously after the message has been delivered) [93, 157].

An important challenge to logging application messages is the volume of data that potentially must be stored by the system. However, for many important HPC applications the order of messages that a given application process sends is independent of the order in which messages are received; such applications are *send-deterministic* [34]. Leveraging the fact that send-deterministic applications will send the same set of messages if they are re-executed from a checkpoint can reduce the number of messages that need to be logged by rolling back the senders of orphan messages when a failure occurs [74].

2.2.3 Hybrid Checkpoint/Restart Methods

In addition to coordinated and uncoordinated checkpoint/restart, many hybrid approaches have also been evaluated. These approaches seek to extract the benefits from coordination (e.g., tightly bounded recovery time) and independence (e.g., fast checkpoint writes) while avoiding the significant costs that each may incur in isolation.

Clustering

Clustering [27, 75, 140] uses coordinated checkpoint/restart within clusters of processes to minimize message logging costs, and message logging for inter-cluster message to minimize the fraction of the system that must be rolled back when a failure occurs. Clusters of processes can be identified by examining the volume of communication between processes [75, 140] or by considering the set of processes that may be affected by correlated failures [27].

Communication-induced Checkpointing

Communication-induced checkpointing exploits an application's communication pattern to relax the strict coordination requirements of coordinated checkpointing while avoiding the domino effect during failure recovery [9, 28, 92]. Including state information in messages that the application exchanges allows each process to determine when a checkpoint is required to ensure the existence of a set of checkpoints that represent a consistent global state.

Message Logging + Coordinated Checkpoint/Restart

Although message logging is most commonly associated with uncoordinated checkpoint/restart, it is also possible to use it with coordinated checkpoint/restart [48,138]. Because coordinated checkpoints limit how far the application will be forced to roll back when a failure occurs, message logs no longer need to be written to stable storage and may instead be maintained locally in memory. Additionally, when a coordinated checkpoint is complete, all of the messages that are successfully received before the checkpoint was taken can be discarded. This simplifies garbage collection and reduces the volume of logged messages. The addition of message logging to coordinated checkpoint/restart means that the whole system need not be forced to roll back to its previous checkpoint when a failure occurs. As a result, the system may be able may be able to take checkpoints less frequently.

2.3 Rollback Avoidance

Techniques for handling faults in large-scale systems can be grouped into three broad categories: *failure avoidance*, *failure effect avoidance*, and *failure effect repair* [32]. Failure avoidance techniques use prediction to forecast when a failure is likely to

occur and, based on this prediction, take action to minimize the impact of the failure on the application. For example, sophisticated analysis of system logs may provide enough advance notice to allow the processes threatened by an imminent failure to be migrated to safer hardware resources [66]. Failure effect avoidance techniques allow the application to continue to execute despite the occurrence of failures. This category includes fault-tolerant algorithms, replication, and software-based memory fault correction. For example, two matrices that are being multiplied together can be augmented with redundant data that allows the contents of the matrices to be reconstructed if a failure occurs [86]. Finally, failure effect repair techniques restore normal execution after the application has been compromised by a failure. The most widely-studied method in this category is checkpoint/restart and its variants. Rollback avoidance is the union of the first two categories: failure avoidance and failure effect avoidance.

2.3.1 Failure Prediction

Several techniques have been proposed that seek to proactively avoid faults that may occur in the future. These approaches monitor the state of the system to predict where and when a fault will next occur. When a fault is predicted to affect a hardware resource, all of the application processes that depend on that resource can be migrated to other hardware resources.

Several methods have been proposed for predicting faults. One approach to predicting faults is system log analysis. Fu and Xu [63] have proposed using a neural network trained with failure data extracted from system logs to forecast future faults. By extracting spatial and temporal correlation data from the system log, they can accurately predict when and where faults are likely to occur in the future. Gainaru et al. have proposed treating system log events as temporal signals [65]. Using this

approach, they are able to identify likely future faults by looking for anomalies in the signals (e.g., changes in frequency or amplitude) generated by the system log. The authors subsequently combined this approach with data mining techniques to correlate outliers in the signals they analyze [66].

Another approach to fault prediction is to monitor the state of the hardware using sensors that collect data such as fan speeds, voltage levels and chassis temperature. Wang et al. have proposed using the data generated by these hardware sensors to predict and avoid failures [161]. For example, when the chassis temperature exceeds an established threshold, failure of the processes that rely on hardware in that chassis is likely. Litvinova, Engelmann, and Scott have extended this technique to more effectively predict failures by considering temporal trends in the data generated by the hardware sensors [113].

2.3.2 Replicated Computation

All fault tolerance requires some form of redundancy [70]. One approach to introducing redundancy for fault tolerance is to explicitly replicate computation. In this approach, the application performs each step of its computation two or more times. Periodically, the states of the computations are compared to determine whether an error occurred. Ensuring that each computation will arrive at the same answer in a failure-free environment requires that the application guarantee that for a given set of inputs, each thread of computation will end up in the same state.

The ROAR project replicates application processes on multiple threads of a single processor [146]. Periodically, the states of the replicated threads are compared; divergence among the threads indicates that an error occurred. Similarly, EDDI uses the compiler to replicate individual instructions [136]. To minimize interference, each set of instructions uses a unique set of registers and memory locations. At pre-

determined synchronization points, the values computed by each set of instructions are compared to determine whether an error has occurred.

These methods protect against errors that are transient or affect a small portion of the system (e.g., a single register or the state of a single process). To protect against faults that may affect an entire node, several methods of replicating application processes have been proposed. A common approach to process replication is to use features of MPI to facilitate communication between replicated processes. This approach requires that the application be *piecewise deterministic* (i.e., the application's computation is deterministic for given a sequence of messages).

rMPI [52] and MR-MPI [49] exploit the MPI profiling interface (PMPI) to protect against fail-stop faults. The PMPI interface allows these methods to ensure that messages are delivered to the intended application process as well as its replica. When a single process fails, the state of the replica is up-to-date and can be used to continue the computation. To protect against undetected failures, RedMPI [60] compares the contents of messages sent by each application process and its replica(s). If a process receives a different message from the application process and its replica, then an otherwise undetected error has occurred.

2.3.3 Software Methods for Memory Failure Detection and Correction

Large-scale systems commonly incorporate some form of hardware protection (e.g., error-correcting codes) to protect against frequent memory failures. Numerous software techniques for augmenting or replacing this hardware protection have also been evaluated.

Shirvani et al. have proposed a software library that can be used in place of

hardware memory protection [151]. By modifying an application to access memory through their API, they track the application's memory usage and interpose the computation of codewords. Periodically, their library uses these codewords to validate the contents of the application's memory. Similarly, Yoon et al. have proposed a technique that allows for software control of error detection [167]. They propose new hardware that enables software to dynamically adjust the degree to which memory is protected against errors.

Not all memory errors can be detected by existing hardware. When undetected errors occur, they may invalidate the results generated by an application. This phenomenon, known as *silent data corruption*, is particularly troubling because the end user has no way of knowing that something untoward has befallen their application. LibSDC attempts to mitigate the effects of silent data corruption by computing checksums of memory pages and configuring them as read-only [58]. This allows it to determine whether the contents of the page have changed since it was last explicitly written. When the application attempts to write to a read-only page, a segmentation fault occurs and LibSDC restores the access protections of the page to allow the application to write to the memory normally. After the page is written, the memory is no longer protected against silent corruption. As a result, LibSDC periodically re-protects memory. Berrocal et al. have proposed an algorithm-agnostic technique that identifies memory corruption by tracking the evolution of the values of the application's data [19]. This approach makes predictions about future values of the application's data. Significant differences between the prediction and the actual value suggest that the application's memory has been corrupted.

2.3.4 Algorithm-Based Fault Tolerance

In addition to the algorithm-independent techniques described above, numerous methods for exploiting algorithm characteristics have been proposed. For example, Huang and Abraham introduce a method for adding fault tolerance to matrix operations [86]. By augmenting matrices with checksum vectors before performing a given matrix operation (e.g., multiplication), their algorithm is able to determine whether the result of the operation is correct or whether an error occurred. Chen and Dongarra extend this idea and propose an encoding that also allows data lost due to failures to be recovered by the surviving processes [38]. In addition to these fault tolerance techniques that are entirely contained within an algorithm, Bridges et al. have proposed a method by which the generalized minimal residual method (GMRES) can be modified to cooperate with the operating system to protect against memory errors [29].

2.4 Modeling Fault Tolerance

Young [168] used a simple model of application execution with coordinated checkpoint/restart to derive the value of the optimal checkpoint interval. The application is divided into a series of alternating intervals in which either the application is executing or a checkpoint is being saved. Daly [42] extends this model by introducing restart time, i.e., the time that elapses from the point of a failure until the application is ready to resume execution. Oldfield et al. [126] further extended this model to consider the impact of filesystem performance characteristics on application execution time. Similarly, Wingstrom [33, 164] presented a model of waste time (i.e., time not available for application computation) based on Young's original model. This model includes restart time but does not account for failures that occur during a restart interval.

As discussed in Section 2.3, many approaches for avoiding rollback have been proposed. In addition, several approach-specific models have been developed to evaluate the performance impact of these approaches on applications running on extreme-scale systems. For example, Ferreira et al. constructed a probabilistic model for process-level replication in the context of high-performance computing [52]. Using this model, they showed where in the exascale design space replication outperforms traditional coordinated checkpoint restart to a parallel filesystem.

Cappello et al. used the Wingstrom model of waste time to evaluate the performance impact of proactive migration and preventive checkpointing [33]. Gainaru et al. incorporated precision and recall into the model to evaluate the effectiveness of using signal processing to predict failure [66]. Aupy et al. used the model to examine the impact of fault prediction on preventive checkpointing [15].

These models do not, however, account for the performance impact of uncoordinated checkpoint/restart. Few general models of the impact of uncoordinated checkpoint/restart on application execution time exist (*see e.g.,* Bosilca et al. [23]). Moreover, the models that do exist do not account for the impact of communication dependencies of each application process. This is an important omission for two reasons. First, uncoordinated checkpoint/restart allows decisions about the timing of checkpoints to be made locally.[3] As a result, when one process decides to take a checkpoint, it may delay its communication with other processes.[4] Like operating system noise [53, 82], these delays may propagate and impact application execution time in a way that it is difficult to capture in an analytic model. Second, when a

[3]For some bulk synchronous parallel (BSP) applications, it may be possible to implicitly coordinate their checkpoints [68]. However, applications that implicitly coordinate the timing of their checkpoints may not be able to realize the full benefit of uncoordinated checkpoint/restart because they may still experience significant contention for bandwidth to persistent storage.

[4]This phenomenon has a much smaller impact when coordinated checkpoint/restart is used because delays due to checkpointing activities are coordinated across processes. As a result, inter-process timing is largely preserved.

failure occurs, the surviving processes can continue to make progress unless and until they have a dependency on the failed process. In many current bulk synchronous parallel (BSP) applications, the amount of progress that the surviving processes are able to make is likely to be small. However, important applications (*e.g.*, S3D [77]) are emerging for which global synchronization is infrequent [36, 69].

2.5 Simulating Fault Tolerance on Large-scale Systems

Fault tolerance for HPC has been a very active area of research, but few tools exist that project behavior beyond small-scale systems. Simulating fault tolerance techniques requires an appropriate level of detail about the communication of the target application. Without an accurate representation of application communication, simulators cannot accurately account for the performance of some fault tolerance techniques (e.g., asynchronous checkpointing). Too much detail, on the other hand, unnecessarily reduces simulator performance. The application simulators for fault tolerance that do exist tend to fall to either extreme; either they are not communication-accurate or they simulate communication in greater detail than necessary.

Riesen et al. present a simulator that models the impact of node failure on application performance in the context of traditional coordinated checkpoint/restart [139]. This simulator can also account for process replication. Tikotekar et al. propose a similar approach [158]. They present a simulator that models coordinated checkpointing and can also simulate fault prediction and process migration. While these tools have been shown to be effective for their stated purposes, they are not communication-accurate. As a result, they are unable to account for fault tolerance

techniques whose performance may be influenced by communication patterns.

At the other extreme is xSim [21]. xSim builds on the MPI profiling interface and interposes itself between the application and the MPI library. As a result, the simulator is able to run unmodified HPC applications. Scaling is achieved by oversubscribing the nodes of the system used for validation. While this provides a tremendous amount of detail about the performance of the application, it imposes a significant cost. Due to limits on the degree of oversubscription, large-scale systems are required to simulate systems that approach extreme-scale. Moreover, as the size of the simulated system grows and the degree of oversubscription therefore increases, the time required to simulate the system grows dramatically. Lastly, this oversubscription could place significant limits on the size of the problem that can be solved as the memory for each simulated node must exist in the memory of one physical node.

Boteanu et al. present a fault tolerance extension to an existing simulator in [24]. However, they target a datacenter environment where each job is a discrete unit that is assigned to a single processing element.

Finally, SST/macro [6, 91] is a coarse-grained, lightweight simulator designed to simulate the performance of existing and future large-scale systems. By collecting traces of application execution, SST/macro is able to simulate the application's computation and communication patterns at scales and on hardware that does not yet exist.

2.6 Memory Content Similarity

2.6.1 Memory De-Duplication in Virtualization

Memory content similarity has been most thoroughly explored in the context of data de-duplication. The preponderance of the relevant research on memory de-duplication has been in virtualization. The Disco VMM [31] introduced transparent memory sharing to reduce virtual machine memory consumption by exploiting memory content similarity. By intercepting disk requests that DMA data into memory, the Disco VMM consolidated read-only pages (e.g., text segments of applications, read-only pages in the buffer cache[5]) containing data from the disk across virtual machines. In some cases, this approach allowed the Disco VMM to significantly reduce memory consumption. For example, transparent memory sharing allowed the VMM to reduce the total memory consumed by 8 VMs, each running the same guest OS and workload, by more than half.

More recently, VMware ESX server incorporated a broader approach to memory de-duplication. Instead of intercepting disk requests, Waldsburger proposed identifying all pages in a virtual machine by their contents. When any two pages are found to have the same contents, the pages are consolidated using copy-on-write (COW). Applying this approach to systems running as many as 10 identical VMs running the SPEC95 benchmark on Linux, the VMware ESX server is able to reduce memory consumption by nearly 60%.

[5]Although the function of the buffer cache has since been folded into the page cache, this term reflects the time period in which the paper was written

2.6.2 Other Uses of Memory De-Duplication

In addition to virtualization, content duplication has been effectively exploited in other domains. In the context of data storage, reducing storage requirements in primary and archival data storage applications by eliminating duplicate data blocks has been widely studied, *see e.g.,* [134, 166, 170]. Similarly, Nicolae [124] developed a technique for eliminating duplicate memory pages in checkpoint data before it is written to persistent storage. Kernel Shared Memory (KSM) allows duplicate memory to be consolidated in Linux with or without virtualization [12].

2.6.3 Internode De-Duplication

Xia and Dinda have advocated for broadening the scope of sharing in virutalization to consider internode sharing. To evaluate the feasibility of this approach, they consider the prevalence of duplicate pages between nodes running several HPC applications. For some workloads (notably HPCCG), they observe that significant inter- and intra-node sharing opportunities exist. Inspired by these results, Xia and Dinda constructed a service, ConCORD, that tracks duplicate memory pages in distributed systems [165]. By providing an interface by which duplicate memory pages can be identified, ConCORD facilitates the development of new system services, e.g., collective checkpointing. Similarly, *SBLLmalloc* has been used to demonstrate that memory consumption can be significantly reduced by consolidating duplicate pages in the application memory of several HPC applications [20]. In several cases, this approach yields memory savings in excess of 50%.

2.6.4 Exploiting Similar Memory Pages

Memory de-duplication research has considered consolidating only duplicate pages. The Difference Engine [76] introduced the idea that similar pages could also be consolidated. In this context, two pages are similar if the difference between them can be represented by an `xdelta` patch file that is smaller than 2 KiB. By relaxing the requirement that only duplicate pages be consolidated, the authors show that under some e-commerce workloads, the Difference Engine can extract significantly more memory savings than VMware ESX server. Moreover, they show that the Difference Engine can reduce memory consumption by more than 50% even for VMMs hosting a single VM. The data presented in [76] were collected by modifying a Xen VMM and using it to host virtualized workstations running workloads consisting of a mix of web and database server and compilation benchmarks.

2.7 Chapter Summary

This chapter surveyed the existing literature on fault tolerance, modeling rollback avoidance, simulation of fault tolerance mechanisms, and exploiting memory content similarity. Despite the extensive body of research on addressing failures in large-scale distributed systems, fault tolerance remains an important issue for next-generation systems.

In this thesis, Chapter 3 begins by introducing a model for predicting the impact of rollback avoidance and coordinated checkpointing on large-scale distributed systems. As discussed in Section 2.4, several models of application performance have been proposed, *see e.g.,* [33, 42, 168]. My model, an extension of the model proposed by Young and refined by Daly, also accounts for the impact of rollback avoidance. Unlike models that are specific to a particular rollback avoidance technique (e.g.,

fault prediction [15, 66]), my model provides a framework for directly comparing rollback avoidance techniques with a common set of assumptions.

Chapter 4 considers how rollback avoidance may impact the performance of applications when uncoordinated checkpoint/restart is used for fault tolerance. While high-quality models of coordinated checkpoint/restart exist, no such models exists for uncoordinated checkpoint/restart. As a result, I present a novel simulation framework based on `LogGOPSim` that allowed me to conduct an inquiry into the relationship between rollback avoidance and uncoordinated checkpoint/restart. This is not the first simulator capable of modeling large-scale application performance (*cf.* [21,24,91,139]), but unlike earlier approaches it considers how communication dependencies in the application interact with uncoordinated checkpoint/restart activities.

Finally, Chapters 5 and 6 present a novel technique for exploiting memory content similarity to improve application, and potentially also operating system, fault tolerance. Although memory content similarity has been exploited for several purposes (e.g., reducing memory consumption [20,31], eliminating redundant checkpoint data [124]), this thesis is the first to examine how similarities in memory can be exploited for fault tolerance. Additionally, this approach can be used to complement many of the fault tolerance techniques discussed in Section 2.1.

Chapter 3

Modeling the Impact of Rollback Avoidance and Coordinated Checkpoint/Restart on Application Performance

3.1 Introduction

As described in Chapter 1, rollback avoidance is the set of techniques that enable an application to continue executing (perhaps with some degradation) despite the occurrence or imminence of a failure. These techniques typically rely on reducing checkpointing costs by significantly reducing the frequency with which the application is forced to roll back to a previous checkpoint. Analysis of these techniques is difficult, however, for two reasons: (i) checkpoint/restart costs vary non-linearly with system mean time to interrupt (MTTI) [42]; and (ii) these techniques frequently can mitigate only a subset of system failures (e.g., memory corruption) [29]. As a result,

the suitability of these techniques in exascale systems is not always clear.

To address this problem, this chapter presents a general model for analyzing the performance impact of techniques for avoiding failure. The contributions of this chapter include:

- A general conceptual model that captures the key features of rollback avoidance techniques: the ability to avoid rollback and the overhead of doing so;

- Two analytic models of the impact of failure avoidance on application performance, one when failure avoidance is used in conjunction with coordinated checkpointing and one when failure avoidance is used as replacement for coordinated checkpointing;

- Case studies mapping the performance of both replication and fault prediction techniques to this model; and,

- An analysis of when rollback avoidance techniques are viable either on their own or in concert with coordinated checkpoint/restart in next-generation HPC systems.

The remainder of this chapter is organized as follows. Section 3.2 introduces the analytical models and Section 3.3 provides validation of these models against a previously validated simulator. Sections 3.4 and 3.5 describe two case studies using these models: process replication and failure prediction, respectively. Section 3.6 uses these models to analyze the effectiveness of failure avoidance both with and without checkpointing and how the performance of these techniques drive the requirements of future systems, and Section 3.7 summarizes the chapter.

3.2 An Analytical Model of Rollback Avoidance

Several models exist for specific rollback avoidance techniques (e.g. [52,66]), but not for rollback avoidance in general. A general analytical model of rollback avoidance is important for understanding the strengths, limitations, and tradeoffs involved with these techniques. Because these techniques are frequently used in concert with other techniques, for example checkpoint/restart, understanding their general behavior tradeoffs can provide important guidance on the design of next-generation systems. It can also guide research on current and future rollback avoidance techniques, providing information about the best way to address resilience challenges.

3.2.1 Developing a Model of Rollback Avoidance

I begin by a developing a general conceptual model of rollback avoidance. In this model, rollback avoidance is stochastic and can be defined in terms of two characteristics: (i) the ability to avoid rollback; and (ii) the overhead costs. When a failure occurs, a rollback avoidance technique prevents the application from experiencing the failure with probability p_a, the *rollback avoidance probability*. I assume that failures are exponentially distributed. The inter-occurrence time of each of the failures that result in rollback despite the best efforts of these techniques is a sum of exponentially distributed variables. Because the number of consecutive of failures for which rollback is avoided is geometrically distributed, the resulting sequence of failure inter-occurrence times is also exponentially distributed [156, p. 320]. The mean of the resulting distribution is the effective system MTTI (M') and is shown in Equation 3.1. The derivation of this expression is shown in Appendix A.1.

$$M' = \frac{\Theta}{1 - p_a} \tag{3.1}$$

where

$\Theta =$ native system MTTI

$p_a =$ rollback avoidance probability

Overhead is modeled as an extension of the application's solve time. *Rollback avoidance overhead* (o_a) describes the overhead cost as a fractional increase in solve time due to a rollback avoidance technique. This fraction represents the expected increase in solve time due to the overhead imposed by rollback avoidance, including preparation for future failures, migration to avoid imminent failures, and application degradation (e.g., slower rate of convergence) due to partial correction. However, it does not include time spent recovering from a failure for which rollback could not be avoided. I assume that the application is otherwise unperturbed. The resulting expression of the application's solve time (T_s') is shown in Equation 3.2.

$$T_s' = T_s(1 + o_a) \tag{3.2}$$

where

$T_s =$ application's native solve time

$o_a =$ overhead of rollback avoidance

To understand how these parameters map to rollback avoidance techniques, consider two examples: repairing memory errors and proactive process migration. When an ECC error is detected in memory, the memory controller raises a machine check exception (MCE) in the processor. Current HPC operating systems terminate the offending process or reboot the entire node in response to a MCE. However, some proposed techniques allow detected memory errors to be corrected by either using application knowledge [29,73] or by leveraging redundant information in the application's memory [110]. In these cases, p_a is the probability that a memory error can be

corrected such that the application can continue without experiencing a failure and rolling back. The overhead, o_a, of these techniques is the expected value of the sum of the additional time that is necessary to prepare for failure and the time required to recover from memory errors as they occur.

Another approach to rollback avoidance is to proactively migrate processes away from hardware that is predicted to fail. A common approach is to continuously examine system event logs looking for sequences of log entries that are believed to be indicative of impending failure [65]. Health monitoring data can also be used to determine when a particular node is likely to fail based on environmental observations (e.g., temperature, input voltage, etc.) [161]. In both cases, p_a is equal to the *recall* of the prediction mechanism. Recall captures the fraction of failures in the system that are correctly predicted. The overhead, o_a, of these techniques is comprised of two principal components: (i) the cost of the prediction mechanism, including gathering and processing the information necessary to make predictions; and (ii) the cost of migrating processes following the prediction of a failure, including costs due to false positives.

3.2.2 Augmenting Coordinated Checkpointing

This general conceptual model facilitates an analytical model of the impact of rollback avoidance techniques on application performance when used in conjunction with traditional coordinated checkpointing. Specifically, Daly's model[1] of application performance [42] can be extended to account for rollback avoidance by modifying the system's mean time to interrupt (MTTI) and the application's solve time to account for the impact these techniques have on application performance. The resulting ex-

[1]This conceptual model could be used to extend any accurate application performance model. I chose Daly's because it is accurate and widely accepted.

pression for application runtime (T_w) is shown in Equation 3.3.[2] Because rollback avoidance effectively increases the system's MTTI, the optimal checkpoint interval also increases.

$$T_w(p_a, o_a) = M' e^{R/M'} \left(e^{(\tau'_{opt}+\delta)/M')} - 1 \right) \frac{T'_s}{\tau'_{opt}} \tag{3.3}$$

where

R = time to required to restart a failed node

M' = system MTTI with rollback avoidance (see Equation 3.1)

δ = checkpoint commit time

τ'_{opt} = optimal checkpoint interval computed using M'

T'_s = solve time of the application (see Equation 3.2)

3.2.3 Replacing Coordinated Checkpointing

It is also instructive to consider how effective rollback avoidance would need to be in order to be a viable replacement for coordinated checkpoint/restart. Combining the conceptual model of rollback avoidance and Daly's model of coordinated checkpoint/restart yields the model of rollback avoidance in the absence of check-

[2] Although T_w is undefined for $p_a = 1.0$, the model is correct in the limit:

$$\lim_{p_a \to 1.0} M' e^{R/M'} \left(e^{(\tau'_{opt}+\delta)/M')} - 1 \right) \frac{T'_s}{\tau'_{opt}} = T'_s$$

The mathematical basis for this assertion is presented in Appendix A.2.1.

point/restart shown in Equation 3.4.[3].

$$T_w(p_a, o_a) = M' e^{R/M'}(e^{T_s'/M'} - 1) \tag{3.4}$$

where

R = time to required to restart a failed node

M' = system MTTI with rollback avoidance (see Equation 3.1)

T_s' = solve time of the application (see Equation 3.2)

3.3 Validation

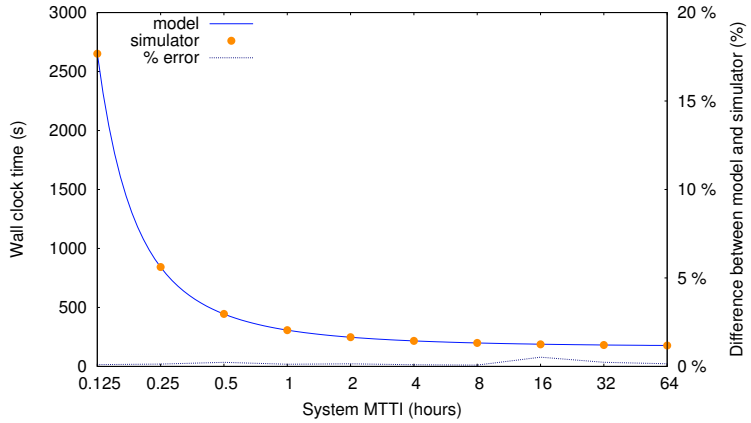
This section validates the accuracy of the two models introduced in the preceding section. It accomplishes this by comparing the application runtime predicted by the model to the application runtime predicted by a validated and freely available simulator [52,115,139]. Because the simulator did not originally account for rollback avoidance, I modified it to ignore failures on simulated nodes with probability p_a. Following a node failure, the interarrival time of the next failure is the sum of the interarrival times of the failures that are successfully avoided and the interarrival time of the next failure that cannot be avoided.

Figure 3.1 shows a comparison between the model of the impact of failure avoidance and coordinated checkpointing (Equation 3.3) and the modified simulator. To minimize the modifications to the simulator, the simulator's computation of the

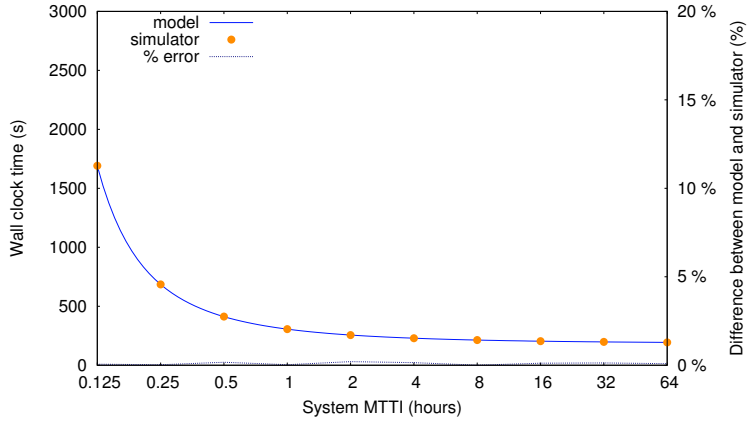
[3]As with the previous model, T_w is undefined when $p_a = 1.0$. However, the model is again correct in the limit:

$$\lim_{p_a \to 1.0} M' e^{R/M'}(e^{T_s'/M'} - 1) = T_s'$$

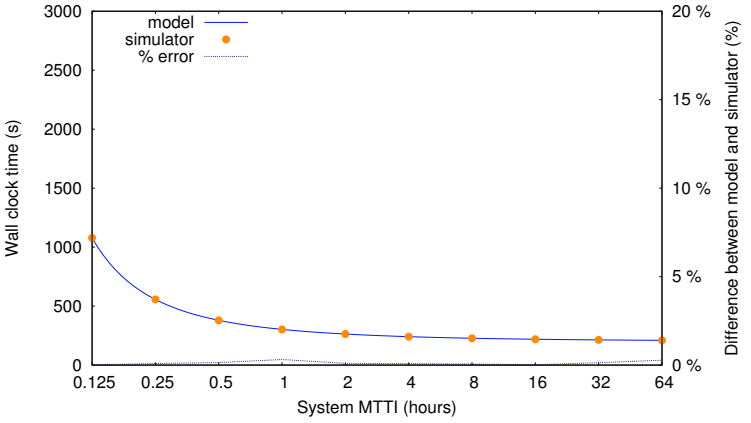
The mathematical basis for this assertion is presented Appendix A.3.2



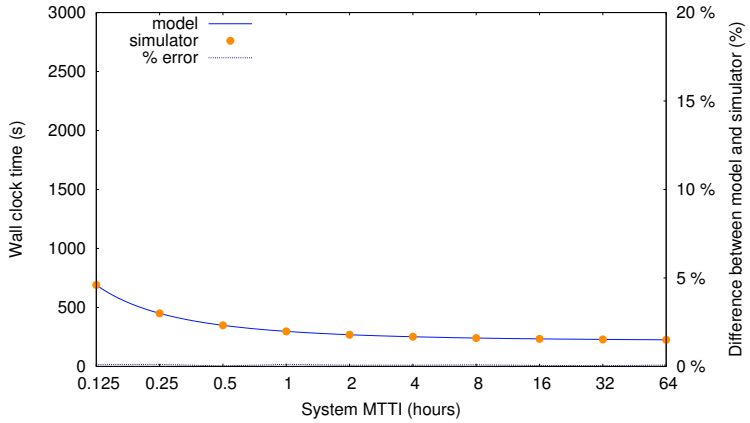
(a) $p_a = 0.00$, $o_a = 0.00$



(b) $p_a = 0.25$, $o_a = 0.10$



(c) $p_a = 0.50$, $o_a = 0.20$

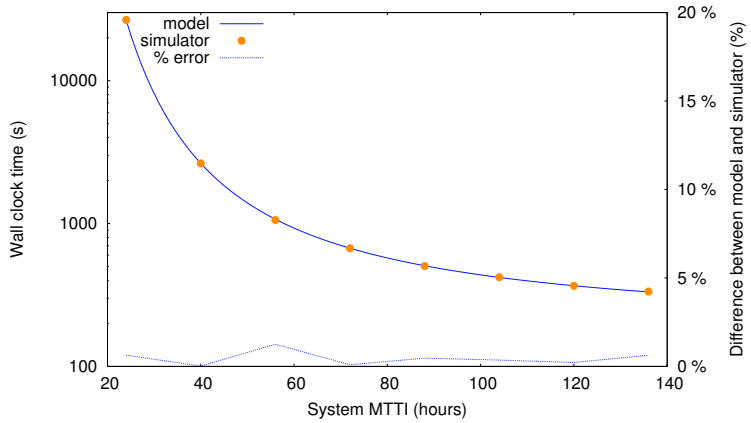


(d) $p_a = 0.75$, $o_a = 0.30$

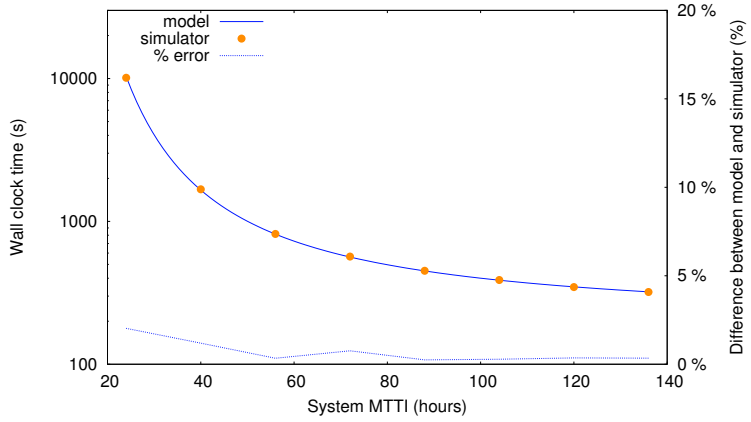
Figure 3.1: *Validation of the model for augmenting C/R.* These figures show the results of validating the analytic model of the performance impact of augmenting coordinated checkpoint/restart with rollback avoidance on application performance against an existing simulator [52, 139]. I modified the simulator to account for rollback avoidance. The model and the simulator use identical values for the solution time ($T_s = 168$ hours), the checkpoint commit time ($\delta = 5$ minutes) and node MTBF ($\Theta_n = 5$ years). Both use the optimal checkpoint interval (τ_{opt}). The subfigures of this figure compare the results across several values of p_a and o_a. The values predicted by the model and the simulator error differ by less than 1% in all cases.

optimal checkpoint interval is unmodified. As a result, for the purposes of this comparison, I modified the model such that it computed the checkpoint interval based on the system MTTI without rollback avoidance. Each of the three subfigures show the results for a different pair of values for o_a and p_a. In each case, the application runtime predicted by the model closely matches (within 1%) the value predicted by the simulator.

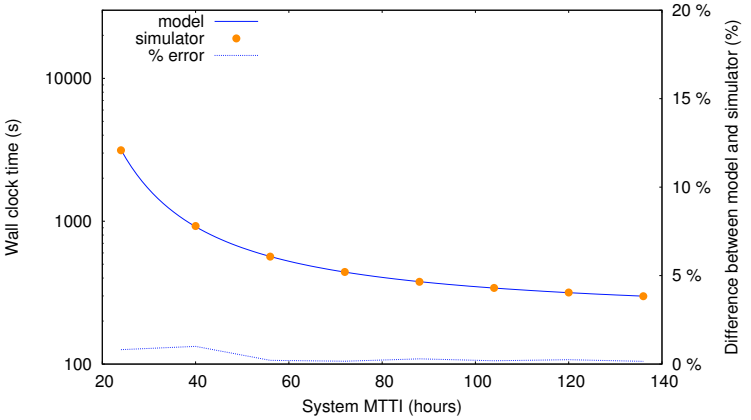
Figure 3.2 shows a comparison between the model of the impact of rollback avoidance (Equation 3.4) and the modified simulator. Once again, each of the three subfigures show the results for a different pair of values for o_a and p_a. In each case, the application runtime predicted by the model closely matches (within 2%) the value predicted by the simulator.



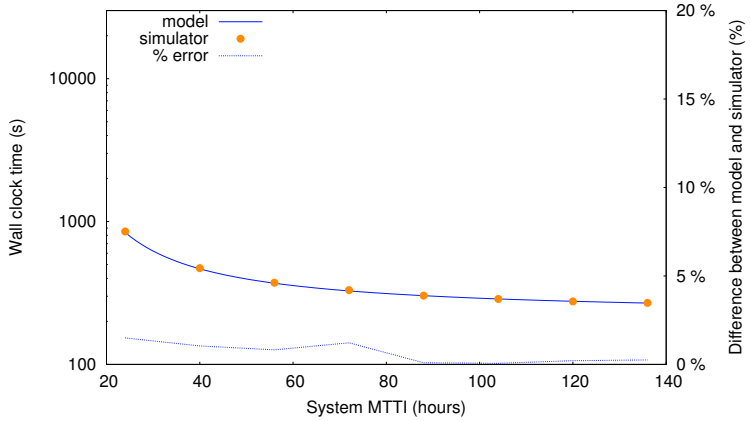
(a) $p_a = 0.00$, $o_a = 0.00$



(b) $p_a = 0.25$, $o_a = 0.10$



(c) $p_a = 0.50$, $o_a = 0.20$



(d) $p_a = 0.75$, $o_a = 0.30$

Figure 3.2: *Validation of the model for replacing C/R.*. Validation of the analytic model of the impact of replacing coordinated checkpoint/restart with rollback avoidance on application performance against an existing simulator [52, 139]. I modified the simulator to account for rollback avoidance. The model and the simulator use identical values for the solution time ($T_s = 168$ hours) and node MTBF ($\Theta_n = 5$ years). The subfigures of this figure compare the results across several values of p_a and o_a. The values predicted by the model and the simulator error differ by less than 2% in all cases.

3.4 Case Study: Process Replication

To demonstrate the power of these analytical models, I use it to examine an existing process replication library, *r*MPI [52]. *r*MPI is a user-level MPI library that facilitates process replication in HPC systems by ensuring that each process and its replica receive all application messages even if one of the processes fails.

3.4.1 Model Parameters

Modeling *r*MPI requires appropriate values for the overhead (o_a) and the avoidance probability (p_a). Naively, the overhead is equal to the overhead of replicating messages. Because each process and its replica run simultaneously, the application's

time-to-solution could be accurately modeled. However, this underestimates the cost of replication. To more completely account for the cost of replicating every process, o_a must be at least 1.0. In this case, the model will not yield raw execution time; it measures resource usage in terms of the number of node-hours required for the computation.

The runtime overhead of rMPI is generally less than 5%, depending on the application [52]. Because this overhead affects each process and its replica, the overhead is captured by setting $o_a = 1.10$. The probability of avoiding the effects of a failure by using rMPI is given by the birthday problem [52]. On average, the number of faults that will occur before the application observes a node failure (i.e., a process and its replica are down simultaneously) is given by Equation 3.5.

$$F(n) \approx \sqrt{\frac{\pi n}{2}} + \frac{2}{3} \tag{3.5}$$

where n is the total number of nodes that comprise the system. The probability of correcting any single failure can be derived from this expression and is shown in Equation 3.6.

$$p_a(n) = \frac{3\sqrt{\pi n} - \sqrt{2}}{3\sqrt{\pi n} + 2\sqrt{2}} \tag{3.6}$$

The remainder of the model parameters are duplicated from the evaluation of rMPI. A summary of the model parameters is shown in Table 3.1.

3.4.2 Model Performance

Given the model and this set of parameters, the efficiency of process replication can be compared against the efficiency of coordinated checkpoint/restart. Figure 3.3 shows the system efficiency as a function of system size both with and without process replication. The key observation is that this figure closely matches the data collected in the evaluation of rMPI (cf. Figure 7 in Ferreira et al. [52]). For small systems, the

Parameter	Description	Value
T_s	SOLVE TIME	168 hours
Θ_n	NODE MEAN TIME TO INTERRUPT (MTTI)	5 years
δ	CHECKPOINT COMMIT TIME	15 minutes
R	RESTART TIME	15 minutes
o_a	ROLLBACK AVOIDANCE OVERHEAD	1.10
p_a	PROBABILITY OF ROLLBACK AVOIDANCE	*see* Equation 3.6

Table 3.1: Model parameters for examining the performance of process replication.

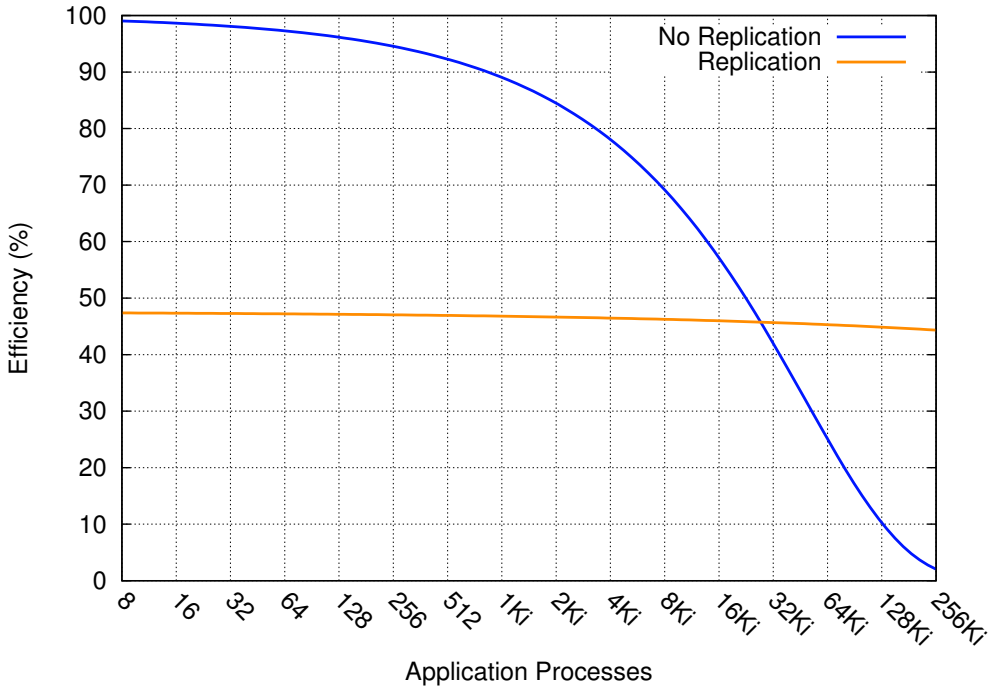
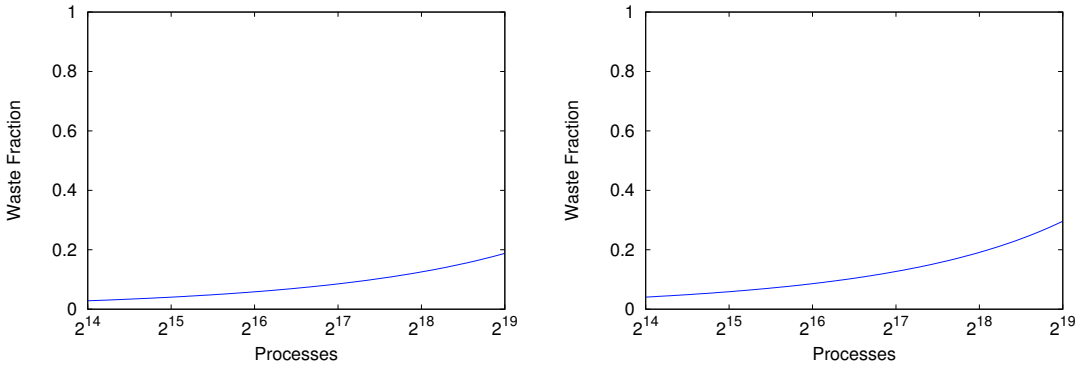


Figure 3.3: *Comparison of application efficiency with and without process replication.* The results obtained using our approach-independent model closely match existing data on process replication (*see* Figure 7 of Ferreira et al. [52]). These data were collected using the parameters in Table 3.1.

overhead of replication is prohibitive. However, as system size increases and failures become more likely, replication is more efficient than the baseline approach.

3.5 Case Study: Fault Prediction



(a) Good Predictor $P = 0.82$, $R = 0.85$ (b) Bad Predictor $P = 0.4$, $R = 0.7$

Figure 3.4: *Impact of failure prediction on the fraction of waste time.* Waste time is all of the application's execution time that is not spent doing useful work. Impact of failure prediction on the fraction of waste time (i.e., time spent not doing useful work). These result closely match existing modeling and simulation data for exponentially distributed failures (*cf.* [15] : compare the good predictor results to the red line in Figure 3(b); compare the bad predictor results to the red line in Figure 4(b) [15]). These data demonstrate that the model is capable reproducing key results in the field while being general enough to account for additional important scenarios.

Techniques for predicting the occurrence of failures have been widely studied [64–67, 142]. Accurately predicting failures before they occur may allow the system to take corrective action that could prevent the application from being compromised. The benefit of this family of approaches is typically characterized by *recall*: the fraction of the total number of failures that can be predicted. In terms of the model, p_a is equal to the method's recall. There are two principal costs of failure prediction: (i) *runtime overhead* (o_{rt}), the costs associated with processing system information

(e.g., collecting and analyzing system log files); and (ii) *false positive overhead (o_{fp})*, the costs associated with unnecessarily initiating proactive response (e.g., needlessly migrating a process). As shown in Equation 3.7, the false positive overhead can be expressed in terms of the recall and precision of a given prediction method.

$$o_{fp} = \frac{(1-P)R}{PM}c \tag{3.7}$$

where

$P =$ precision of the prediction method

$R =$ recall of the prediction method

$c =$ average time required for a proactive response

$M =$ system MTTI

As a result, the overhead of failure prediction can be expressed as shown in Equation 3.8.

$$o_a = \frac{(1-P)R}{PM}c + o_{rt} \tag{3.8}$$

where

$o_{rt} =$ runtime overhead of the prediction method expressed as a fraction of total execution time

Figure 3.4 compares the results of the model to existing data on the impact of fault prediction on application performance [15]. Specifically, it compares two specific cases of predictor performance. Figure 3.4(a) shows the fraction of waste time (i.e., time spent by the system doing something other than directly advancing the application's computation) for a "good" predictor ($P = 0.82$, $R = 0.85$) as a function of system size. Figure 3.4(b) shows the same result for a "bad" predictor ($P = 0.4$, $R = 0.7$). These results demonstrate that the model closely matches

existing modeling and simulation data on the the impact of fault prediction (*cf.* [15]: compare the red lines in Figure 3(b) to the good predictor results (Figure 3.4(a)), and the red lines in Figure 4(b) to the bad predictor results (Figure 3.4(b))). The results are similar when the model is used to reproduce the other figures presented by Aupy et al. that assume exponentially distributed failures. Although the rollback avoidance model is technique-independent, it yields results that closely match the results of this technique-specific model.

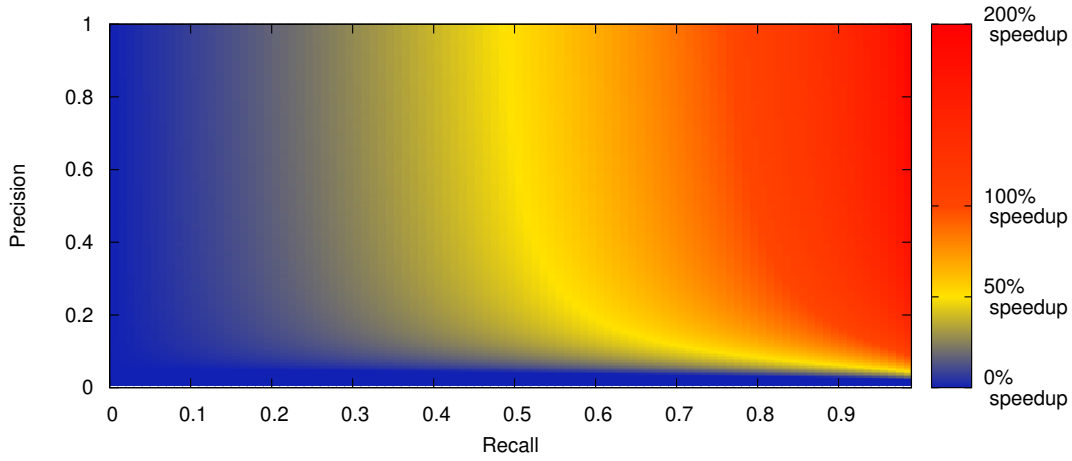
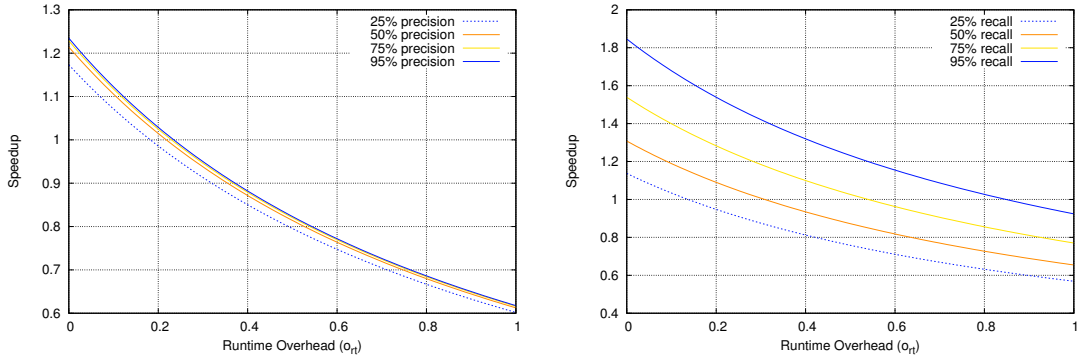


Figure 3.5: *Impact of precision and recall on application speedup.* This figure shows that the performance impact of fault prediction is much more strongly influenced by recall than precision. These results were generated using a solve time (T_s) of 168 hours, a system MTTI (M) of 45 minutes, a checkpoint commit time (δ) of 15 minutes, and a restart time (R) of 10 minutes. Speedup is calculated relative to the execution time with no rollback avoidance.

The rollback avoidance model also allows the claim—first articulated by Aupy et al. [15]—that recall is more important than precision for fault prediction to be more closely examined. Figure 3.5 shows the relative impacts of precision and recall on application performance. For the data in this figure, the system MTTI (M) is



(a) Precision (b) Recall

Figure 3.6: *Comparison of the relative impact of precision and recall on application speedup.* These figures confirm that precision has much less impact on application performance than either recall or runtime overhead.

fixed at 45 minutes, a value that is well within the range currently projected for the first exascale machine. The proactive response cost is fixed at two minutes. While this value is longer than the prediction window of current techniques, it is also shorter than most current proactive measures (e.g., checkpointing) would require (*cf.* [65]). The data in this figure show that above a modest threshold (e.g., 50%—many modern methods are above 90%), precision has relatively little influence on application execution time.

Figure 3.6(a) illustrates the impact of precision relative to runtime overhead. The data in this figure were collected with the recall value fixed at 40%—a value achieved by many current methods. Each of the lines in this figure corresponds to a different precision, ranging from 25% to 95%. These four lines almost entirely overlap one another. Although the existing literature is largely silent on the costs of predicting failures, this figure shows that runtime overhead has a much larger effect on application runtime than precision does. Moreover, in this configuration, once the runtime overhead exceeds 20% the benefits of failure prediction disappear and application execution time increases. Finally, the impact of runtime overhead

is significant. Below 20% the slopes of these lines are nearly -1; any increase in the runtime overhead results in a commensurate increase in execution time. As a result, efforts to improve precision that result in increases in the runtime overhead will yield little benefit.

Figure 3.6(b) shows the impact of recall relative on runtime overhead. The data in this figure were collected with the precision value fixed at 95%. Similar to the previous figure, each of the lines in this plot correspond to a different recall value. If we consider horizontal slices of this figure, increasing the recall value may be fruitful even if it results in relatively large increases in the runtime overhead. For example, a method for which the runtime overhead is 0% and the recall is 50% would yield the same speedup as a method for which the runtime overhead is 17.8% and the recall is 75%. In other words, any method that increases the recall value from 50% to 75% and imposes a runtime overhead of less than 17.8% will yield a benefit. This figure shows that innovative techniques that increase recall—even at the cost of runtime overhead—may increase the overall benefit of failure prediction.

3.6 Analysis & Discussion

3.6.1 Designing Rollback Avoidance for Exascale

Due to the projected overheads of coordinated checkpoint/restart at extreme-scale, significant effort has been devoted to developing new rollback avoidance techniques. In particular, considerable attention has been paid to application-specific techniques.

This section uses the model to explore the projected design space of the first exascale system to determine where new application-specific techniques may offer the greatest benefits. The analysis begins by examining how the relationship between the avoidance probability and the runtime overhead affects application performance.

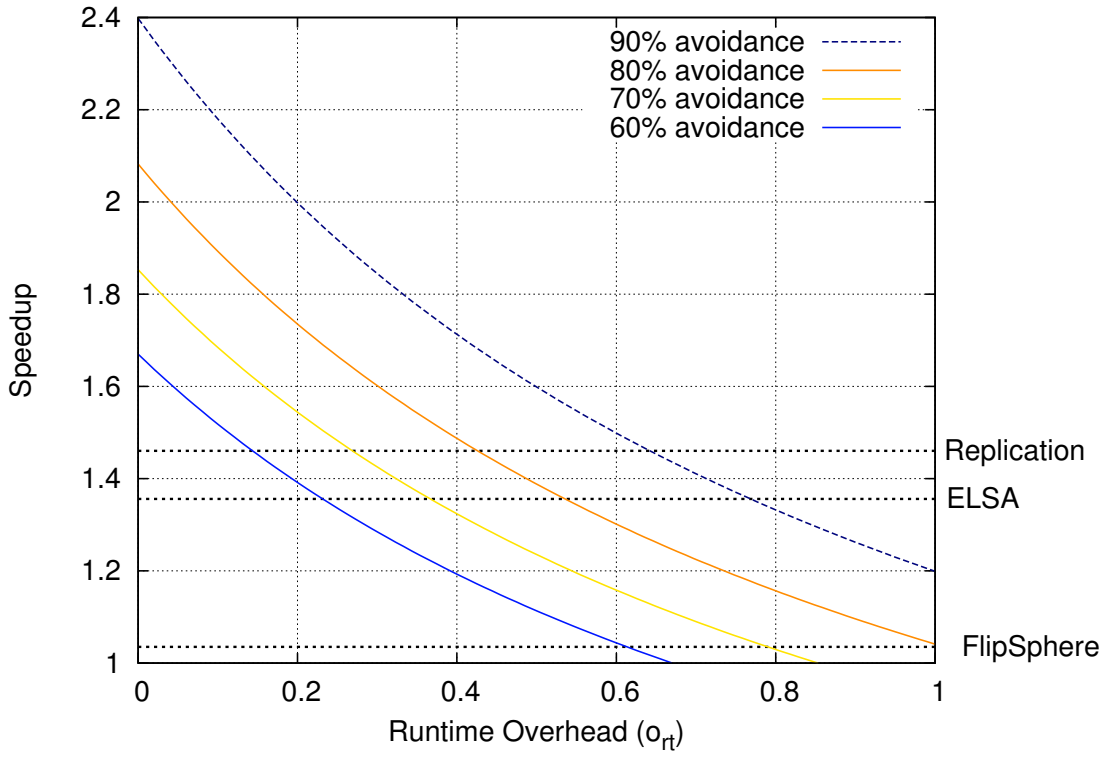
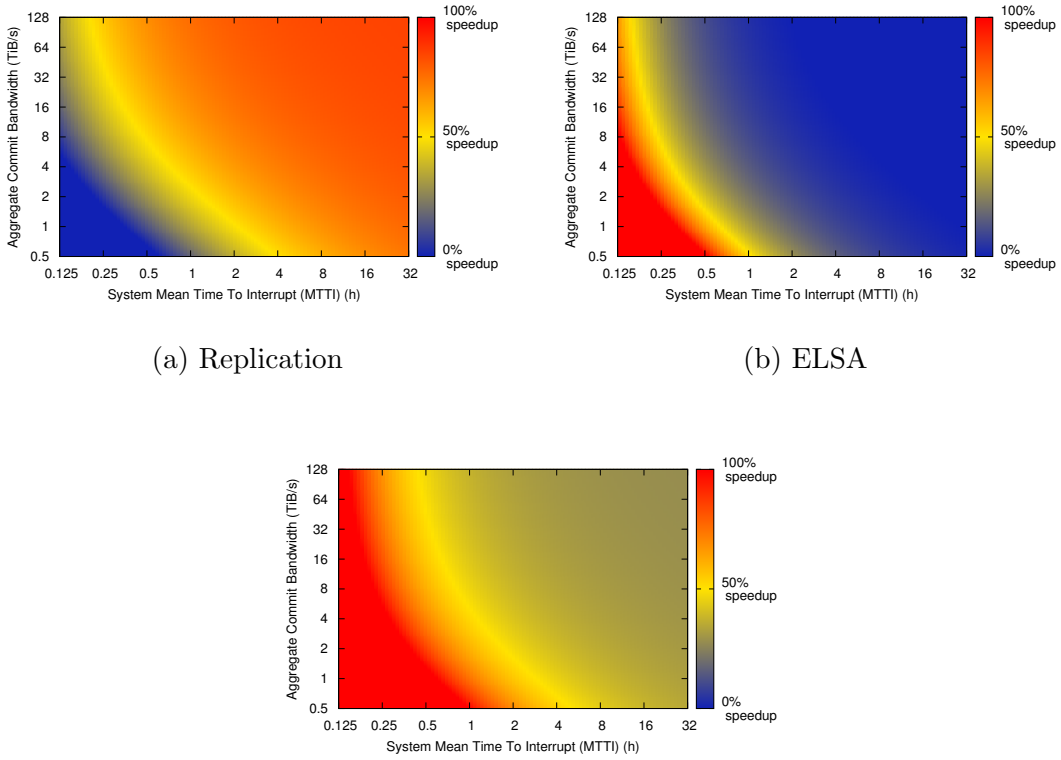


Figure 3.7: *Application speedup as a function of overhead and probability of rollback avoidance.* This figure shows the potential application speedup (i.e., the relative reduction in application execution time) for several hypothetical rollback avoidance techniques as a function of the overhead imposed by the technique. For comparison, the dashed horizontal lines correspond to the speedup achieved by existing techniques. These results were generated using a solve time (T_s) of 168 hours, a system MTTI (M) of 45 minutes, a checkpoint commit time (δ) of 5 minutes, a restart time (R) of 10 minutes, and no runtime overhead ($o_a = 0.0$). Speedup is calculated relative to the execution time with no rollback avoidance.

For many application-specific methods—fault-tolerant algorithms, for example—the overhead represents how much longer the computation takes to converge than if no error had occurred. Figure 3.7 shows the application speedup for several avoidance probabilities as a function of overhead. By way of comparison, the dashed horizontal lines in this figure show the speedups that can be achieved using existing application-



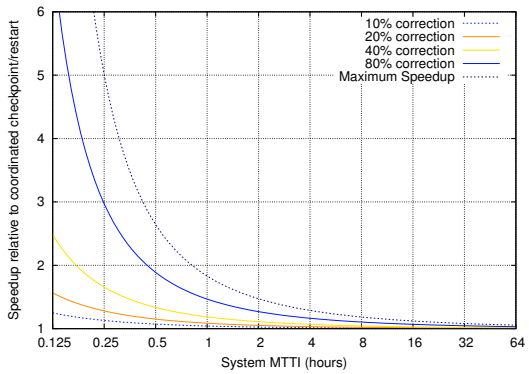
(a) Replication

(b) ELSA

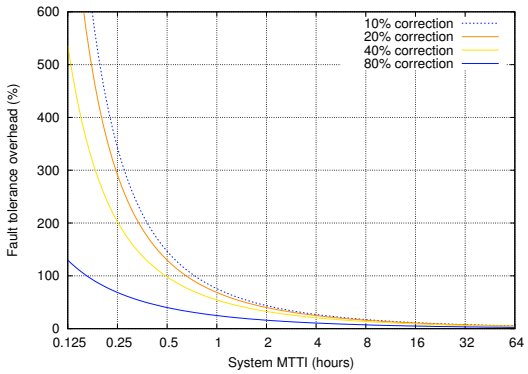
(c) FlipSphere

Figure 3.8: *Comparison of a strawman rollback avoidance technique to three existing techniques.* This figure shows heatmaps that show the potential benefit of a strawman rollback avoidance technique ($p_a = 0.80$, $o_a = 0.10$) relative to three existing application-independent techniques. Each of these heatmaps covers the range of MTTI (M) and an aggregate checkpoint commit bandwidth (β) that is currently projected for exascale. This figure assumes a system comprised of 128Ki nodes, each of which has 8 GiB of memory. These figures collectively show that this aggressive strawman would only improve application performance over a fraction of the exascale design space.

independent techniques. Even if they impose relatively large overheads, there is room for application-specific techniques to outperform existing techniques if they can avoid a high percentage of rollbacks due to all sources of failure. For the application-specific methods that can only avoid rollbacks that result from a subset of system failures,



(a) Speedup relative to coordinated checkpoint/restart



(b) Overhead of fault tolerance

Figure 3.9: *Effect of rollback avoidance probability and system reliability on application speedup.* This figure shows the effect of a range of rollback avoidance probabilities on application speedup as a function of system reliability. These results were generated using a solve time (T_s) of 168 hours, a checkpoint commit time (δ) of 5 minutes, a restart time (R) of 10 minutes, and no runtime overhead for rollback avoidance ($o_a = 0.0$). Speedup is calculated relative to the execution time with no rollback avoidance. "Maximum Speedup" represents the maximum increase in application performance that can be achieved by eliminating all wasted time.

achieving such high avoidance probabilities will be challenging.

Figure 3.7 represents a single point in the exascale design space. To evaluate the potential benefits over the entire design space, I consider a strawman. The strawman is able to avoid 80% of all rollbacks while imposing a 10% overhead. This is an aggressive strawman given that many application-specific techniques only protect against rollbacks caused by memory corruption, which is a fraction of all rollbacks. Given this strawman, I compare its relative performance to three existing techniques:

- *Replication.* I consider the case where each process is replicated exactly once; the replication degree is 2. Additionally, I assume a perfectly strong scaling application (i.e., running the same application with process replication requires

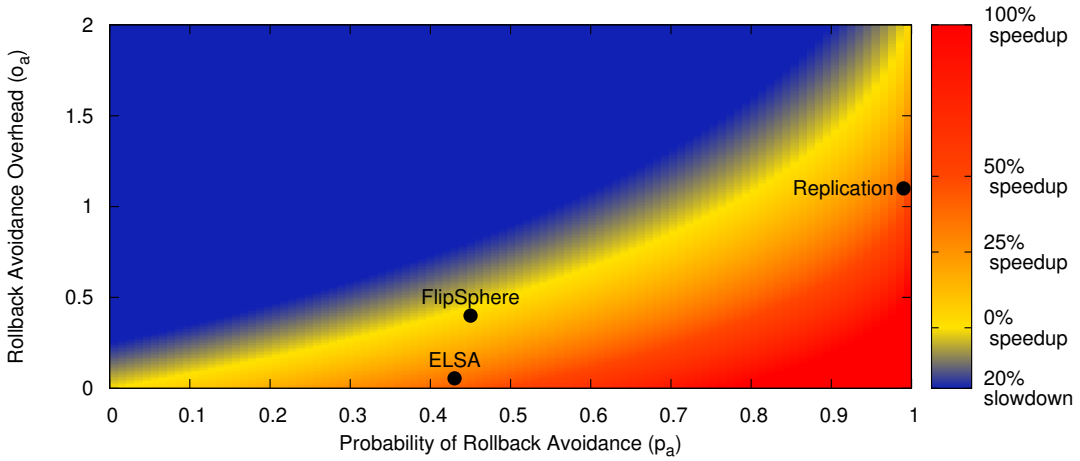


Figure 3.10: *Impact of overhead and probability of rollback avoidance on application speedup.* These results were generated using a solve time (T_s) of 168 hours, a system MTTI (M) of 45 minutes, a checkpoint commit time (δ) of 5 minutes and a restart time (R) of 10 minutes. Speedup is calculated relative to the execution time with no error correction. For comparison, the performance of several rollback avoidance techniques are shown: *r*MPI, process-level replication for HPC [52] ($p_a = 0.99$, $o_a = 1.1$); ELSA, log-based prediction library [65] ($p_a = 0.43$, $o_a = 0.05$); and FlipSphere, software-based error correction library for HPC [57]($p_a = 0.45$, $o_a = 0.4$).

twice as much time).

- *Fault prediction (ELSA).* ELSA is a fault prediction toolkit with a precision of 93% and a recall of 43% [65]. Although the runtime overhead of ELSA has not been publicly documented, I assume that it is no more than 5%.

- *FlipSphere.* FlipSphere is a software-based memory error correction library. It protects 90% of application memory and imposes an overhead of 40%. Because FlipSphere only protects against rollbacks that are due to memory corruption, its protective benefit is less than the fraction of memory that it protects. I generously assume that 50% of all rollbacks are due to some form of mem-

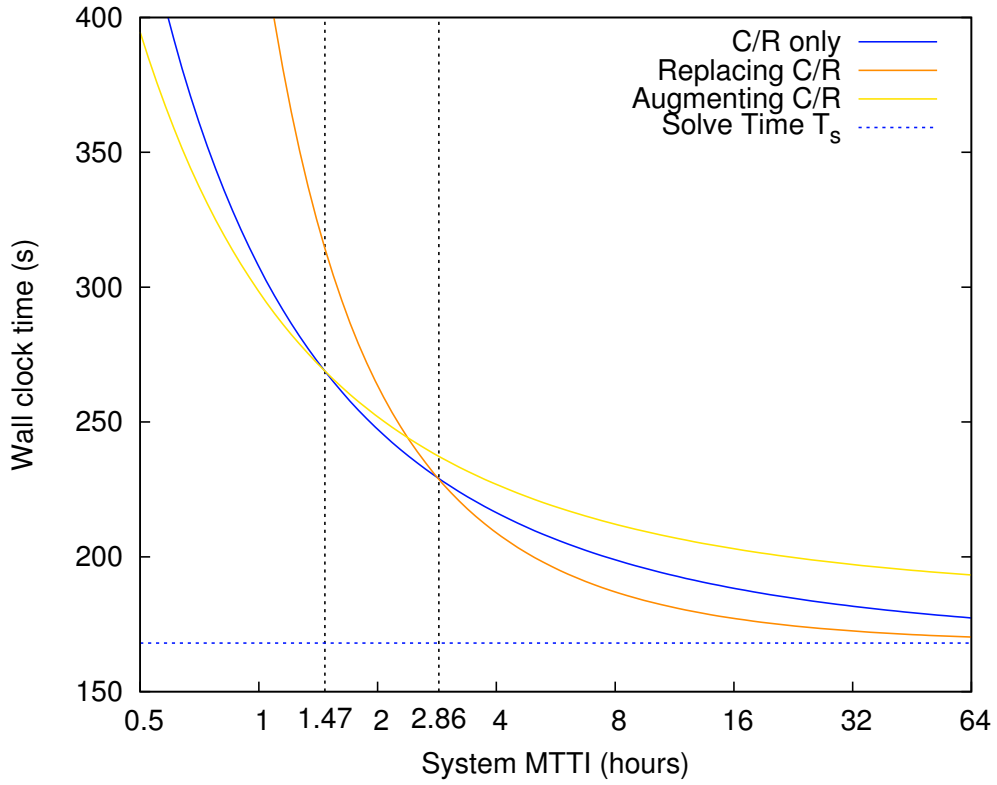


Figure 3.11: *Impact of fault tolerance techniques on application performance as a function of system MTTI.* This figure compares three different approaches to fault tolerance: (i) Coordinated Checkpointing only; (ii) Augmenting Checkpointing with Rollback Avoidance ($p_a = 0.25$ / $o_a = 0.10$); and Rollback Avoidance only ($p_a = 0.99$ / $o_a = 0.00$). The results in this figure were generated using a solve time (T_s) of 168 hours, a checkpoint commit time (δ) of 5 minutes, and a restart time (R) of 10 minutes.

ory corruption. As a result, the analysis in this section uses $p_a = 0.45$ for FlipSphere.

Figure 3.8 shows the results of the comparison of the strawman to these three techniques. In these heat maps, the blue regions are where the strawman provides little or no improvement, the red regions are where the strawman offers significant benefits. These figures indicate that the benefits of this strawman depend on where

in this design space the first exascale system appears. For systems with small MTTI and small checkpoint bandwidths, replication will outperform the strawman. For systems with large MTTI and large checkpoint bandwidths, fault prediction will outperform the strawman. However, somewhere between these two extremes in the design space there appears to be a significant opportunity for rollback avoidance techniques such as the strawman to provide significant benefits.

3.6.2 Replacing Coordinated Checkpoint/Restart

In principle, rollback avoidance could also be used as a replacement for coordinated checkpointing. However, for any rollback avoidance to be an effective replacement, its probability of avoiding rollback would have be very close to 1.0. To see why this is, consider a technique that avoids 90% of rollbacks on a system with an MTTI of 1 hour. In this case, the effective MTTI given by Equation 3.1 is 10 hours. Given exponentially distributed failures, the probability of completing a 168-hour job without encountering a failure (which is the criteria for success in this case) is approximately 5.0×10^{-8}. Figure 3.11 illustrates this phenomenon more concretely. Even a hypothetical rollback avoidance technique that avoids 99% of rollbacks and imposes no overhead cannot compete with coordinated checkpoint/restart in all circumstances. In particular, for values of system MTTI below one hour this hypothetical replacement technique is no longer competitive with traditional coordinated checkpointing. Therefore, the model shows that rollback avoidance alone is unlikely to perform well on exascale systems.

3.6.3 Assessing the Impact of Model Parameters

The model for predicting application performance also allows for a careful exploration of the design space for rollback avoidance techniques. Examining the characteristics

that have the greatest impact on application can inform the design, development, and refinement of current and future rollback avoidance techniques. This section considers the impact of three of the model's parameters: (i) probability of rollback avoidance (p_a); (ii) rollback avoidance overhead (o_a); and (iii) system MTTI (M).

Probability of Rollback Avoidance

Combining rollback avoidance techniques and coordinated checkpoint/restart has the potential to improve application performance by reducing the frequency with which the application is forced to roll back to a previous checkpoint. I begin by considering the impact that the probability of rollback has on application performance. Figure 3.9(a) shows the decrease in application runtime (i.e., speedup) as a function of the system MTTI for several values of p_a. To isolate the impact of p_a, this figure examines the impact of approaches that impose no overhead (i.e., $o_a = 0.0$). To put the system MTTI into context, a system MTTI of 0.5 hours corresponds to a system comprised of 65,536 (64Ki) nodes, each with an MTTI of 3.75 years. Similarly, a system MTTI of 8 hours corresponds to a system comprised of nodes whose MTTI is 60 years. These two values roughly represent the range of node MTTIs that are currently projected for the first exascale system [50].

The most striking feature of this figure is how unreliable the system must be before even very good rollback avoidance techniques (e.g., able to avoid rollback 80% of the time) significantly improve application performance. For systems with an MTTI greater than 8 hours, there is very little speedup in application runtime. This is an especially stark result given that these figures assume zero overhead. However, for unreliable systems (e.g., systems with an MTTI that is less than 2 hours) rollback avoidance yields significant benefits, in some case reducing the application runtime by more than 81%.

The reason for this behavior is due to a phenomenon that is closely related to Amdahl's Law [11]. An application that runs on a system with an MTTI of eight hours operates with an efficiency of 85%. In other words, 85% of the system time required to run the application is used to perform useful work. A direct consequence of this fact is that the maximum possible speedup is approximately 18% (1.00/0.85). The dashed curve in Figure 3.9 shows the maximum speedup as a function of system MTTI. This curve represents the maximum application speedup that could be achieved if *all* wasted time (e.g., writing checkpoints, restarting nodes, redoing lost work) was eliminated. Until the system MTTI drops below approximately 8 hours, even heroic efforts to eliminate wasted system time will yield only modest gains in application speed.

Figure 3.9(b) shows the overall overhead of fault tolerance. Even though each of these avoidance probabilities achieves significant speedup over coordinated checkpoint/restart, the overhead of fault tolerance is still significant. Moreover, once the fault tolerance overhead exceeds 100%, dual process replication will outperform these less protective strawman approaches.

Rollback Avoidance Overhead

Figure 3.10 considers the impact of the overhead of rollback avoidance techniques on application performance. It shows the relationship between the probability of avoiding rollback and the overhead that avoidance imposes on the application. This figure considers a case where the efficiency of the application with checkpoint/restart is low: the system MTTI is low (45 minutes) and the checkpoint commit time is high (15 minutes). Although these values are well within the range projected for the first exascale system, they represent a system in which less than half of the application's runtime is available for useful computation. This figure also demonstrates the importance of minimizing the overhead of rollback avoidance for techniques that only avoid

a modest fraction of failures. For example, a technique that imposes a 20% overhead will not improve application performance unless it is able to avoid more than 23% of all failures. In contrast, reducing the overhead to 10%, shows improvement when avoiding just 12% of failures. For techniques that avoid a large fraction of failures (e.g., more than 70%), overhead has a lesser impact; even when overhead consumes a significant fraction of the application's system time, there is application speedup.

System MTTI

As observed earlier, system MTTI has an (unsurprisingly) large impact on the effectiveness of rollback avoidance mechanisms. To isolate the effects of system MTTI and to reduce the number of dimensions in the configuration-space, consider three representative approaches to fault tolerance: (i) coordinated checkpoint/restart only; (ii) coordinated checkpoint/restart augmented with a rollback avoidance technique for which the probability of failure avoidance is 25% and the overhead is 10%; and (iii) a rollback avoidance technique that is able to avoid 99% of all failures with 1% overhead. All other parameters are taken from Table 3.1. Figure 3.11 shows the performance of these three approaches as a function of system MTTI, ranging from 30 minutes to 64 hours. The approach that yields the best application performance depends on the which of three different regions of system MTTI that a given system is operating in. In the configuration shown in this figure, augmenting coordinated checkpoint/restart with rollback avoidance is the most effective approach for unreliable systems, those with an MTTI below approximately 81 minutes. For reliable systems, those with an MTTI above approximately 2.9 hours, the rollback avoidance technique by itself is most effective. Coordinated checkpoint/restart is the preferred approach in a small range of system MTTI values between these two extremes.

3.7 Chapter Summary

This chapter introduced and validated two analytical models for evaluating the impact of rollback avoidance on application performance in large-scale systems. These models allow rollback avoidance to examined both in conjunction with coordinated checkpointing and in isolation. Using these models, I examined the impact of failure avoidance techniques based on system characteristics. In particular, I showed that for reliable systems, using rollback avoidance to augment checkpointing yields only modest performance gains. However, as systems grow in size and failures occur more frequently, rollback avoidance can yield significant improvements. I also showed that even very effective rollback avoidance techniques are unlikely to replace coordinated checkpointing unless future systems are much more reliable than currently projected. More broadly, these models allow for an exploration of system and application parameters for which rollback avoidance can potentially provide significant benefits.

Chapter 4

Simulating the Impact of Rollback Avoidance and Uncoordinated Checkpoint/Restart on Application Performance

4.1 Introduction

As described in Chapter 1, coordinated checkpoint/restart is currently the dominant approach to fault tolerance on today's largest systems. Uncoordinated checkpoint/restart with message logging has emerged as a potential alternative. One of the key benefits of uncoordinated checkpoint/restart is that relaxing the coordination requirement reduces contention for bandwidth to persistent storage thereby allowing checkpoints to be written more quickly.

Analytic models that can accurately account for the impact of communication dependencies on uncoordinated checkpoint/restart do not exist. Therefore, I developed

a simulation framework to examine the impact of uncoordinated checkpoint/restart and rollback avoidance on application performance in collaboration with Kurt Ferreira, Dorian Arnold, Bryan Topp, and Torsten Hoefler. The approach to simulation is motivated by two observations: (1) simulation can be computationally expensive, and simulation efficiency is maximized by considering only the features of the computing environment that are relevant to the performance impact of checkpoint/restart; and (2) the coarse-grained operation of checkpoint/restart (on the order of minutes to hours) allows the overheads and complexities of cycle-accurate simulation to be avoided. Based on these observations, we hypothesize that like operating system noise [53, 82], resilience mechanisms (e.g., writing checkpoints, restarting after a failure or redoing lost work) can be modeled as CPU detours. A CPU detour is a number of CPU cycles that are used for something other than the application.

This chapter presents an approach to efficiently simulating checkpoint/restart-based fault tolerance for large-scale HPC systems. Using this approach, it introduces a framework for simulating the performance impact of coordinated and uncoordinated checkpoint/restart protocols on existing and hypothetical extreme-scale systems. This framework allows the impact of rollback avoidance and uncoordinated checkpoint/restart on application performance to be examine. Specific contributions in this chapter include:

- A survey of system, application, failure, and resilience characteristics required for accurate and efficient simulation of workloads running on extreme-scale systems;
- A simulation framework, based on extensions to `LogGOPSim` [83];
- An evaluation of the predictive performance of the simulation approach against an existing analytic model of coordinated checkpoint/restart; and
- An examination of the impact of rollback avoidance and uncoordinated check-

point/restart on application performance

The organization of this chapter is as follows: the next section discusses the relevant system, application, failure and resilience characteristics that must be accounted for by the simulation framework. This section also offers background on checkpoint/restart protocols and shows how they factor into the considerations. Section 4.3 provides an overview of `LogGOPSim`, the simulator that serves as the basis of the framework. Section 4.4 describes how the simulator models failures and rollback avoidance. Section 4.5 validates the accuracy of the predictions of the simulation framework. Section 4.6 presents the results of using the simulator to examine the impact of rollback avoidance on uncoordinated checkpoint/restart. Finally, Section 4.7 summarizes the contributions of this chapter.

4.2 Considerations for Resilience at Scale

To enable efficient, large-scale simulations of resilience techniques, this section identifies the relevant hardware and software characteristics that impact simulation performance. It also examines how system features, application behavior, fault-tolerance mechanisms, and failures impact application performance.

4.2.1 Hardware Characteristics

One of the objectives of this chapter is to develop a simulation framework that will enable the evaluation of resilience techniques on current and future systems. As a result, the simulator must be able to accurately and efficiently model the impact of faults and fault tolerance on application performance given the: (a) temporal scale, (b) spatial scale, and (c) architectural features of next-generation extreme-scale systems.

Temporal Scale

Faults and fault tolerance mechanisms typically operate at large time-scales (for example, minutes, hours or even weeks). As discussed in Chapter 1, projected mean-time-to-interrupt (MTTI) on the first exascale machines are on the order of hours. Additionally, many of the target applications are long running, and the behaviors of the applications as well as the systems are expected to be dynamic. As a result, simulating resilience requires a simulator that can model relatively long periods of application execution.

Spatial Scale

Currently, the largest HPC systems are comprised of tens of thousands of nodes. If current predictions hold, the first exascale system may be nearly an order of magnitude larger. As a result, the simulator must be capable of modeling the behavior of systems that are much larger than any that are currently available.

Architectural Features

The first exascale system is not projected to appear until sometime after 2020 [154]. In the intervening span of years, improved interconnect and persistent storage technologies are likely to emerge. The simulator must therefore also be able to evaluate the impact of these advances on resilience mechanisms.

4.2.2 Application Characteristics

The simulator must be capable of accounting for the performance aspects of an application's behavior. Prior research and experience has shown that it may be sufficient

to do this at the coarse granularity of the target application's computation, specifically: its *communication graph*, a description of how processes communicate with each other; its *computation time*, the time between communication events; and its *dependencies*, a partial ordering of all communication and computation events. The next section examines the interplay of these characteristics and resilience mechanisms.

4.2.3 Impact of Checkpoint/Restart Mechanisms

Despite the proliferation of checkpoint/restart-based resilience mechanisms [9,56,71, 117,118,128,131,152], effective methods for evaluating the true costs of each of these approaches on exascale systems do not exist [163]. Given the large temporal and spatial scales of the simulated systems that we wish to consider, effective simulation demands the elimination unnecessary detail. Existing work on modeling and simulation of coordinated checkpointing provides a guidepost on the required components and level of details [26,42,139].

In a failure-free environment, the impact of coordinated checkpointing can be accurately modeled by considering the following application and system characteristics:

- *checkpoint time*, the amount of time that checkpointing activities prevent the application from executing. The checkpoint time can be broken down into the time required for the following phases:
  - *coordination* phase
  - *checkpoint calculation* phase, during which time the checkpoint data are computed; the *checkpoint commit* phase, during which the checkpoint data are written to stable storage; and the *resumption* phase, during which the

system resumes normal application execution.

- *checkpoint interval*, time between consecutive checkpoints
- *work time*, the amount of time that the application would execute in the absence of checkpointing activities.

For approaches like uncoordinated checkpointing that lack explicit coordination, the simulator also needs to consider application characteristics such as the communication characteristics described earlier in this section. Consider a simple uncoordinated checkpointing strategy where each process generates checkpoints strictly according to local policies. Communication dependencies may cause the checkpointing activities of one process to perturb the behavior and performance of other processes. For example, if the recipient of a message is currently busy generating a checkpoint then reception of the message may be delayed until the checkpoint is complete. Further, all actions that are dependent on the reception of the message will also be delayed. Additionally, many asynchronous resilience techniques incorporate some form of message logging [46] to mitigate recovery costs. Simulating the impact of this activity also requires that the simulator account for application communication patterns.

4.2.4 Impact of Failures

Meaningful evaluation of resilience mechanisms necessarily includes consideration of failures. To accurately simulate the impact of failures on application performance the simulator must consider: (a) failure characteristics; (b) restart time; and (c) the recovery model.

Failure Characteristics

To evaluate the impact of faults in the context of a resilience mechanism, a description of how failures occur in the simulated system is required. Initially, we consider only fail-stop failures. Although the occurrence of failures in the system could be expressed in many ways, the most common and succinct description of failure occurrences is in the form of a probability distribution.

Restart Time

When a failure occurs, some time elapses before any computation can be undertaken on the failed node. To accurately capture this behavior, the simulator must know the time between the occurrence of a failure and the moment when the failed node can resume computation. This includes time to restart failed nodes and processes and to read checkpoints from persistent storage, but does not include any time for recovery. For example, in the case of coordinated checkpointing, the end of the restart interval coincides with the beginning of rework (i.e., redoing work lost due to the failure).

Recovery Model

When the failed node has restarted and is able to resume computation, there is typically some amount of work that needs to be redone before the system can again make meaningful forward progress. For example, in coordinated checkpoint/restart, all of the computation between the last valid checkpoint and the occurrence of the failure needs to be redone. Typically, each resilience mechanisms presents a different method for recovering from a failure. Therefore, to accurately account for the cost of recovering from a failure, the simulator needs a model for each resilience mechanism that allows it to determine the amount of time that will elapse before the application

Required to Model	Parameter Name	Parameter Description
All Checkpointing	COORDINATION TIME	time for processes to coordinate the taking of a checkpoint
	CHECKPOINT COMPUTATION	time to compute a checkpoint
	CHECKPOINT COMMIT TIME	time to write a checkpoint to stable storage
	CHECKPOINT INTERVAL	time between consecutive checkpoints
	WORK TIME	time-to-solution without failures or resilience mechanisms
Uncoordinated Checkpointing	COMMUNICATION GRAPH	details of inter-process communication
	COMPUTATION EVENTS	failure-free computation pattern of the application
	DEPENDENCIES	partial ordering of communication and computation events
Failure Occurrences	FAILURE CHARACTERIZATION	rate and distribution of failures
	RESTART TIME	time to read a checkpoint from stable storage after a failure
	RECOVERY MODEL	a model of the time required before forward progress can resume

Table 4.1: Summary of the parameters needed for accurate simulation of HPC applications in a failure-prone system.

resumes forward progress.

4.3 LogGOPSim

This section describes `LogGOPSim` [81, 83], the simulator we extend to meet the requirements prescribed by the considerations in Section 4.2. We choose `LogGOPSim` because it has been shown to be accurate, is freely available, and is fast enough to support large-scale simulations while capturing many of the application and hard-

ware characteristics we require. As described in the next section, we simply needed to extend it to account for checkpoint/restart and failure recovery.

4.3.1 Simulating Application Characteristics

`LogGOPSim` [83] is an application simulator based on a variant of the LogP model of parallel computation [41]. The simulation framework consists of two major components: a trace collector (`liballprof`), and an optimized discrete-event simulator (`LogGOPSim`). The trace collector records the sequence of MPI communication operations executed by the target application. The discrete-event simulator uses the MPI traces to extract the required communication and computation characteristics of the application while preserving the happens-before relationship of events within the application.

This simulation framework was developed to simulate applications at scale, and has the ability to simulate large-scale systems by extrapolating traces that were collected on smaller scale systems. This allows for the simulation of platforms that are larger than those currently in existence while maintaining the same communication characteristics (equivalent to weak-scaling of the application). Although the extrapolated trace may not precisely represent the communication pattern on the larger system, the impact of this inaccuracy has been shown to be small [83] if extrapolation factors are bounded. This framework has been used to evaluate the performance of collective communication operations [84] and the impact of OS noise [82] on large-scale applications.

4.3.2 Simulating Hardware Characteristics

`LogGOPSim` is able to simulate systems with the hardware characteristics described

in Section 4.2. First, it provides the simulation scale necessary for evaluating checkpointing techniques. For a single collective operation, `LogGOPSim` can simulate up to 10,000,000 processes. For more general workloads, it is capable of simulating more than 64,000 processes.

Second, with some minor modifications, `LogGOPSim` is also capable of simulating the necessary temporal scale. The initial implementation of `LogGOPSim` was intended for comparatively short simulations. As a result, the temporal scope of the simulations that can be executed by the unmodified simulator was significantly limited by the size of the simulating system's memory. To achieve the necessary temporal scale with reasonable quantities of system memory, we made some simple modifications to the way that `LogGOPSim` handles trace data.

Third, `LogGOPSim` is able to model the impact of emerging interconnect technologies. Modifying the parameters of the underlying LogGOPS model enables `LogGOPSim` to simulate the impact of many changes in network behavior on resilience techniques. In addition, as discussed more fully below, this model of resilience mechanisms allows for the evaluation of how improvements to persistent storage systems (e.g., the widespread availability of node-local or rack-local SSDs) will affect the performance of resilience mechanisms.

4.4 Simulating Failures and Resilience with `LogGOPSim`

We use `LogGOPSim` to simulate activities associated with checkpoint/restart (e.g., writing checkpoints, restarting after a failure, redoing lost work) by modeling them as CPU *detours*. A CPU detour is a number of CPU cycles that are used for something other than the application, similar to OS noise [53, 82]. To do so, we modified

`LogGOPSim` to generate a sequence of CPU detours that represent checkpoint/restart activities. The duration and frequency of the CPU detours are determined by four user-specified parameters:

- *MTTI*, the mean of the exponential distribution that represents the time to the next interrupt;
- *restart time*, the amount of time that elapses between the occurrence of a failure and the moment when the application is able to resume computation;
- *checkpoint time* (δ), the total time required to generate a checkpoint (including interprocess coordination) and write it to persistent storage; and
- *checkpoint interval* (τ), the amount of time that elapses between checkpoints.

In the case of coordinated checkpoint/restart, the user can choose to have `LogGOPSim` compute the optimal checkpoint interval based on the other three parameters rather than explicitly specifying the checkpoint interval.

We have also added support to `LogGOPSim` for simulating rollback avoidance. The user can specify the desired rollback avoidance characteristics using the two parameters introduced in Chapter 3: *probability of rollback avoidance* (p_a), the stochastic description of when the application avoids rollback; and *overhead of rollback avoidance* (o_a), the degree to which the application's execution time is inflated by the activities of the rollback avoidance technique.

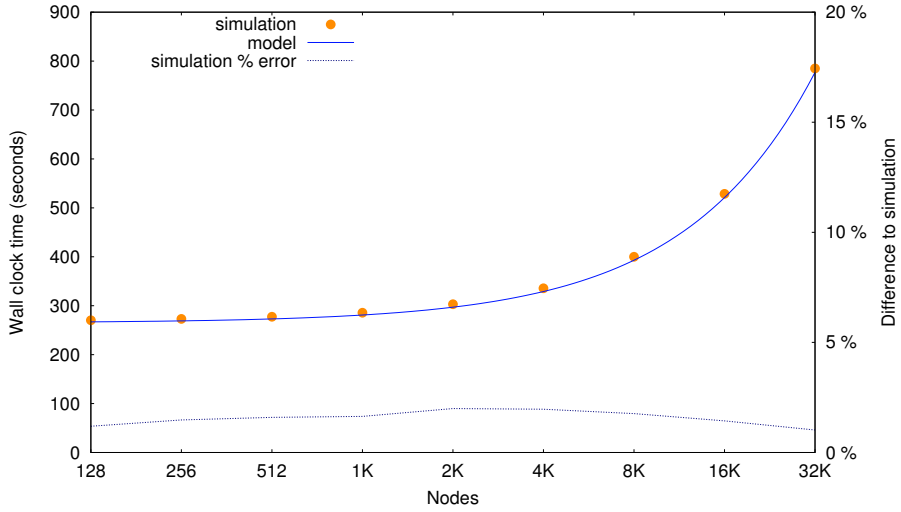
Every τ seconds, `LogGOPSim` simulates the end of a checkpoint interval and generates a detour of δ seconds to represent the time taken away from the application to take a checkpoint. When a failure occurs, as determined by the MTTI, `LogGOPSim` determines the amount of time that was lost due to the failure and generates a corresponding detour. This includes time for: restarting and resuming computation (including time lost to failed restarts), incomplete checkpoints, and lost work.

This chapter examines two checkpoint/restart-based fault tolerance protocols: coordinated checkpoint/restart and uncoordinated checkpoint/restart with optimistic message logging. For the case of uncoordinated checkpoint/restart with optimistic message logging, the application is assumed to have a very high bandwidth connection to persistent storage and that the time required to write to the message log is negligible. To consider the case where this assumption does not hold, `LogGOPSim` could be modified to generate message logging detours. Similarly, pessimistic message logging [46] can be accounted for by modifying the CPU overhead parameter (o in the LogGOPS model) for send operations (o_s) to account for the log write to stable storage.

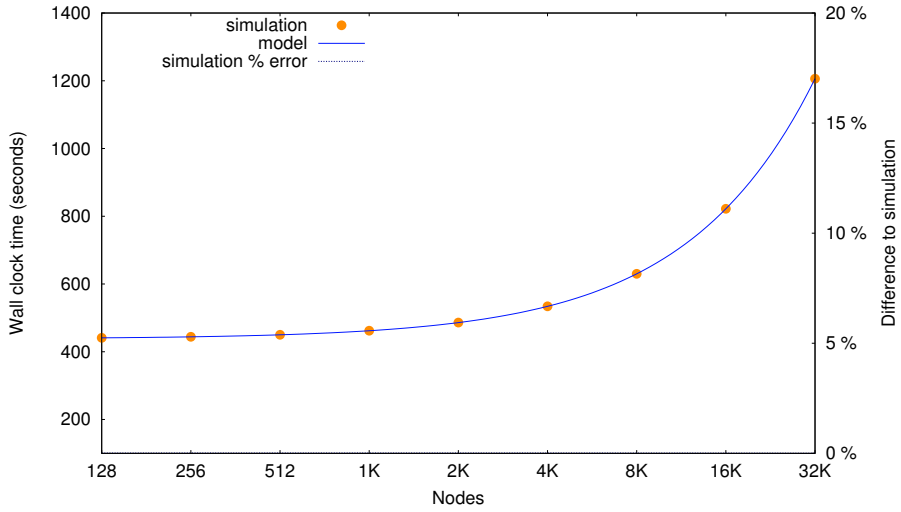
4.5 Validating `LogGOPSim`'s Simulation of Checkpoint/Restart

4.5.1 Validating Simulation of Error-Free Execution

This section presents the data collected to validate the simulator against error-free application performance. It validates the simulator against both analytic models and small-scale testing to ensure that the simulator accurately models the impact of resilience mechanisms in failure-free and failure-prone environments.



(a) CTH



(b) LAMMPS

Figure 4.1: Validation of the simulator against the simple analytic model described in Equation 4.1 for coordinated checkpointing to stable storage in a failure-free environment for CTH and LAMMPS. The model and the simulator use identical values for the T_s (for each application), τ, and δ. The simulation error is less than 3% for CTH and less than 1% for LAMMPS across the tested node count range.

Analytic Model of Coordinated Checkpointing in a Failure-Free Environment

Equation 4.1 models application performance in terms of its wall clock time-to-solution, T_w, in a failure-free environment.

$$T_w = T_s\left(1 + \frac{\delta}{\tau}\right) \tag{4.1}$$

where T_w is the wall clock time, T_s is the solve time of the application without any resilience mechanism, τ is the checkpoint interval [42], and δ is the checkpoint commit time (time to write one checkpoint). In the case of coordinated checkpoint/restart, all application processes are assumed to be writing to a shared persistent storage resource and contention for storage resources is assumed not to degrade the aggregate write bandwidth to the shared storage resource. Given these assumptions, the checkpoint commit time can be expressed as:

$$\delta = \frac{1}{\beta}\sum_{i=1}^{N} c_i \tag{4.2}$$

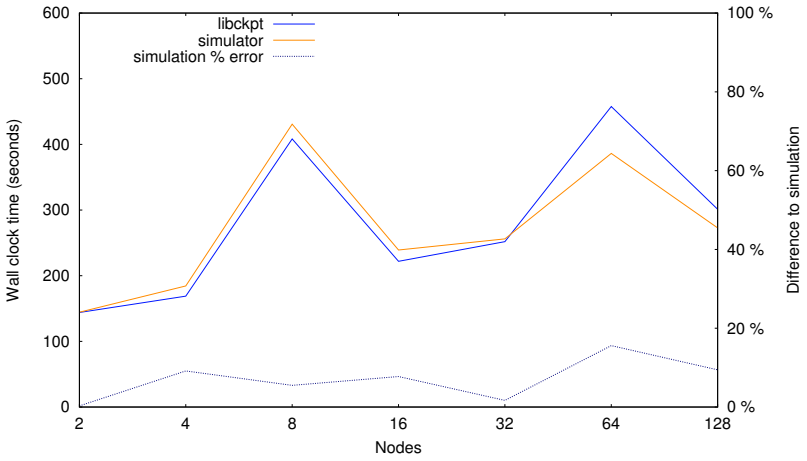
where N is the number of application processes, c_i is the size of the checkpoint for the i^{th} process, and β is the aggregate write bandwidth to stable storage. If checkpoints are roughly the same size for each process, then Equation 4.2 simplifies to:

$$\delta \approx \frac{Nc_0}{\beta} \tag{4.3}$$

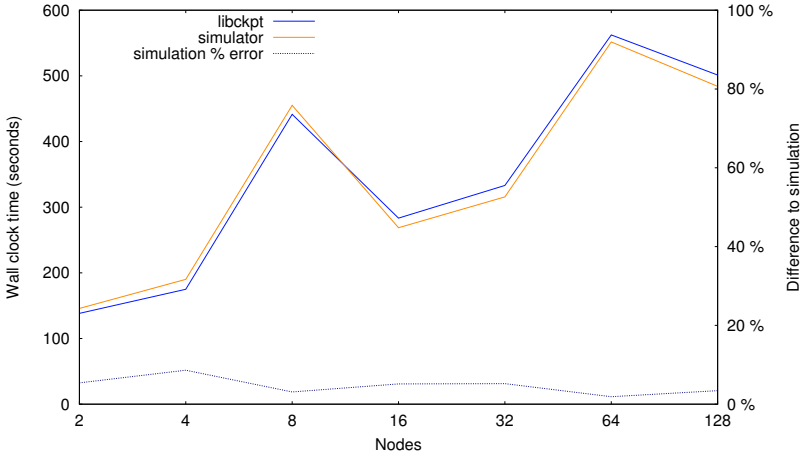
Equation 4.3 implies that the checkpoint commit time is roughly proportional to the total number of application processes. Each doubling of the number of processes means that the time to commit a checkpoint to persistent storage also roughly doubles.

Figures 4.1(a) and 4.1(b) compare the completion time predicted by this model to the completion time predicted by the simulator. The times-to-solution for CTH

predicted by the simulator are very accurate, about 3% greater than the model's predictions. More importantly, the simulator closely matches scaling trends predicted by the model. Moreover, the simulated times-to-solution for LAMMPS are within 1% of the analytic model. On the whole, these data suggest that the simulator is accurately modeling how the impact of checkpoint/restart scales with system size.

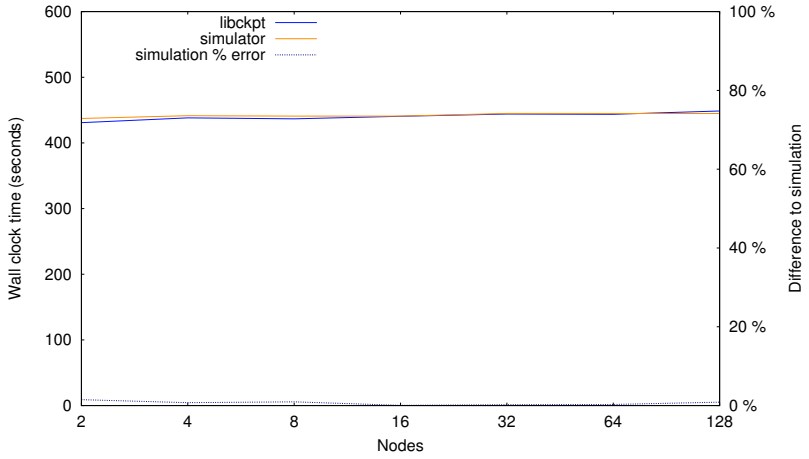


(a) Coordinated Checkpointing

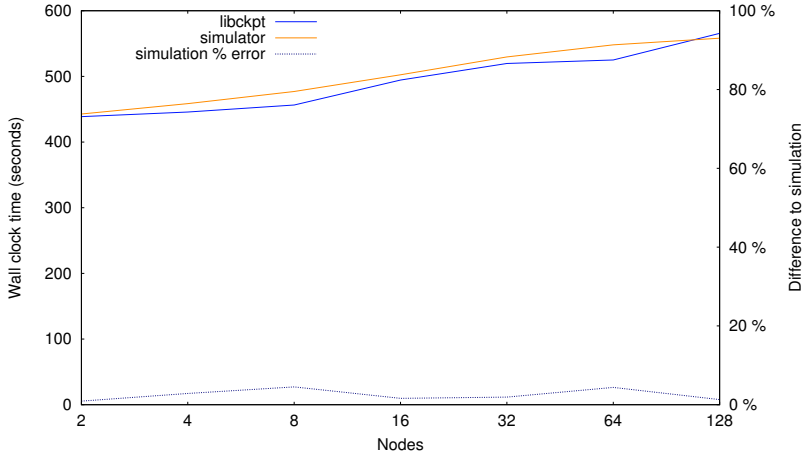


(b) Uncoordinated Checkpointing

Figure 4.2: Validation of `LogGOPSim` simulation against a coordinated and uncoordinated checkpointing library for CTH. The simulator and `libchkpt` use identical values for T_w (failure free performance), τ (checkpoint interval), and δ (checkpoint commit time). The simulation error in this figure is less than 20%, with this differences attributed to platform features not being simulated. For example, interference from the OS is not being generated in this case to simplify analysis. This OS interfere has been shown to greatly influence impact CTH performance [53].



(a) Coordinated Checkpointing



(b) Uncoordinated Checkpointing

Figure 4.3: Validation of `LogGOPSim` simulation against an coordinated and uncoordinated checkpointing library for LAMMPS. The simulator and `libchkpt` use identical values for T_w (failure free performance), τ (checkpoint interval), and δ (checkpoint commit time). The simulation error in this figure is shown to be less than 5% in the range tested.

Small-scale testing

To further validate the simulator, we compared its predictions against the results of small-scale tests on real hardware. The simulator provides fine-grained control over the checkpoint interval and duration. To mimic this degree of control on real hardware, we constructed an MPI profiling library, `libchkpt`. This library, based on the the `libhashckpt` incremental checkpointing library [56], includes support for full coordinated and uncoordinated checkpoints in addition to its support for incremental coordinated checkpoints. The full coordinated checkpointing functionality ensures all checkpoints are taken simultaneously on each node, while the uncoordinated approach takes checkpoints independently. While taking checkpoints, the CPU is taken from the application until the checkpoint commit time has completed.

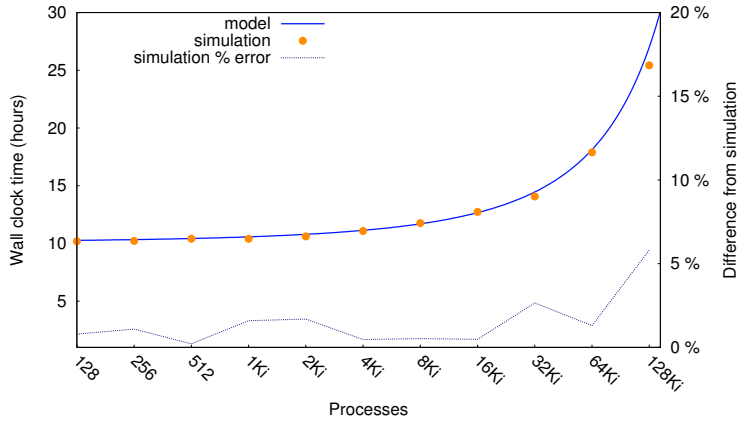
Figures 4.2 and 4.3 show the results of the validation experiments. These figures compare the total wall clock time simulated by `LogGOPSim` and measured with `libchkpt` running on a test platform. For reference, each figure also includes the total wall clock time in the absence of any failures. Each figure also shows the error in the simulator's predicted execution time. Note the performance of CTH in Figure 4.2 exhibits a distinct sawtooth pattern. This pattern is an artifact of how CTH scales the computation as nodes counts increase. The simulator accurately predicts this complex sawtooth pattern. This figure the error in the simulator's prediction. The predictive performance of the simulator is less accurate for CTH than for LAMMPS, but the error in the predicted time to solution is less than 20% in both cases. The size of the error in the predicted execution time for CTH is likely due to the fact that these experiments do not account for OS noise and they rely on a very simple network model that does not account for contention. CTH has been shown to be sensitive to this sort of perturbation [53]. Because CTH performs a significant amount of bulk data transfer, network contention may also negatively influence its performance.

Overall, these figures show that `LogGOPSim` closely tracks the results measured with `libchkpt`. For all the configurations examined, the absolute wall clock time simulated by `LogGOPSim` is within 20% of the measured values for CTH and within 5% of the measured values for LAMMPS. Although the error in `LogGOPSim`'s predictions of the execution time of CTH is, in some cases, relatively large, its results closely mimic the trends exhibited by CTH in the `libchkpt` results.

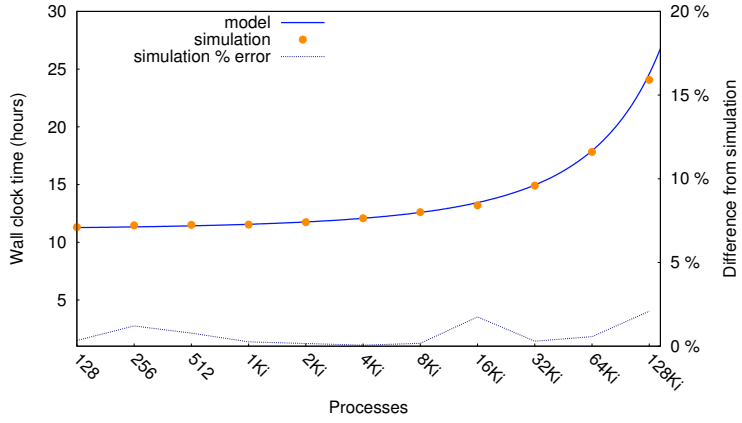
4.5.2 Validating Simulation of Failures and Rollback Avoidance

This subsection validates the simulator when errors may occur during an application's execution by examining the results of a sequence of experiments designed to measure the fidelity of the application execution times predicted by the simulator. It also considers the impact of using rollback avoidance to recover from errors without rolling back to an earlier checkpoint. The experiments discussed in this section use a trace collected from 128 MPI processes running LAMMPS with the SNAP potential. The trace was collected over 10.12 hours of execution. Using this trace as the baseline, I used the simulator's extrapolation feature to repeatedly double the size of the simulated system. The largest system considered consisted of 128 Ki MPI processes.

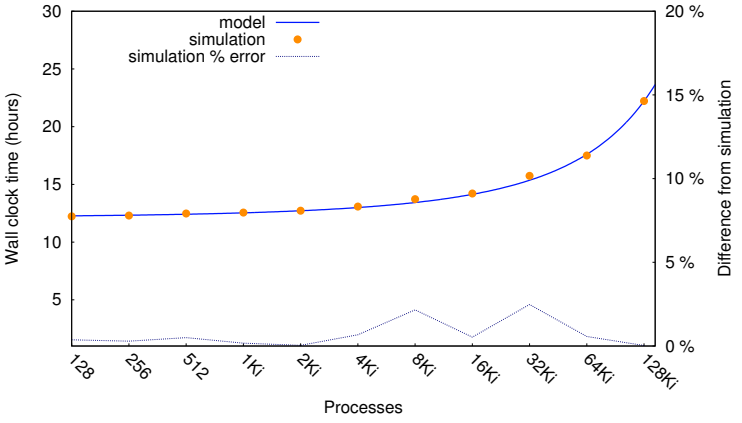
Figure 4.4 compares the results of this series of simulations against the execution times predicted by the model introduced in Chapter 3. This figure considers four different combinations of values for the probability of rollback avoidance (p_a) and runtime overhead (o_a). The primary y-axis shows the overall execution time predicted by our model and simulator. The secondary y-axis shows the percentage difference between the model and the simulator. Over this set of configurations, the execution times predicted by the simulator closely track those predicted by the model; in no case is the difference between the two larger than 6%.



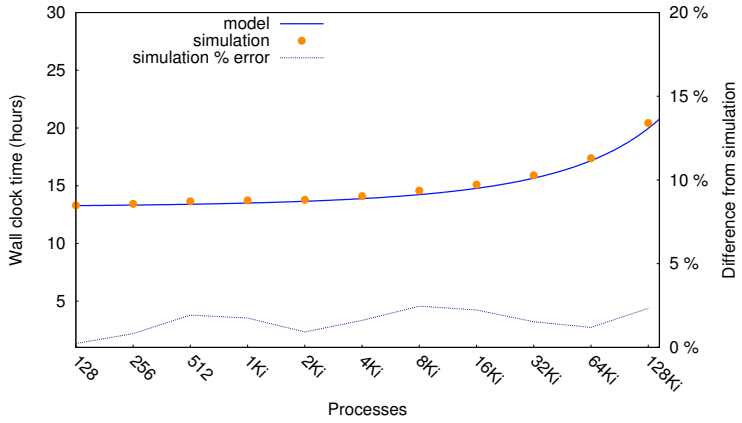
(a) $p_a = 0.00$, $o_a = 0.00$



(b) $p_a = 0.25$, $o_a = 0.10$



(c) $p_a = 0.50$, $o_a = 0.20$



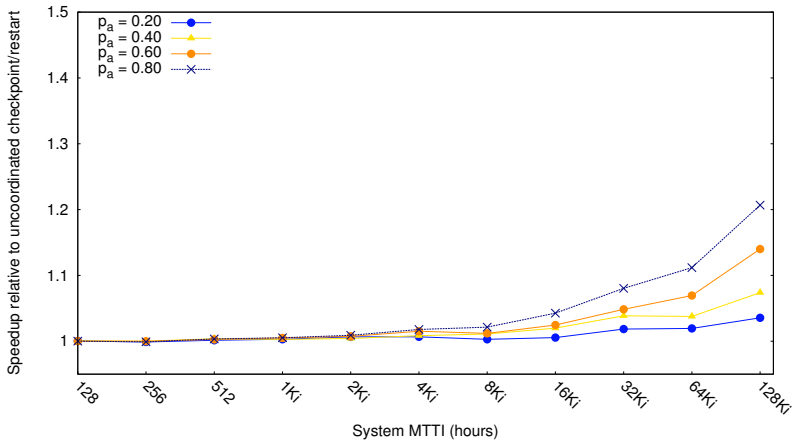
(d) $p_a = 0.75$, $o_a = 0.30$

Figure 4.4: *Validation of simulation framework against analytic model for coordinated checkpointing.* This figure compares the execution time predicted by the simulation framework to the model for coordinated checkpointing introduced in Chapter 3. Each simulation result represents the mean of 24 independent simulations. These data are based on the simulation of a 10.12 hour run of LAMMPS with the SNAP potential. The simulator and the model used identical values for the checkpoint commit time ($\delta = 5$ minutes) and node MTBF ($\Theta_n = 5$ years). Both use the optimal checkpoint interval (τ_{opt}) without considering the impact of rollback avoidance on the effective MTBF of the system. In all four experiments, the simulation data closely match the results predicted by the model.

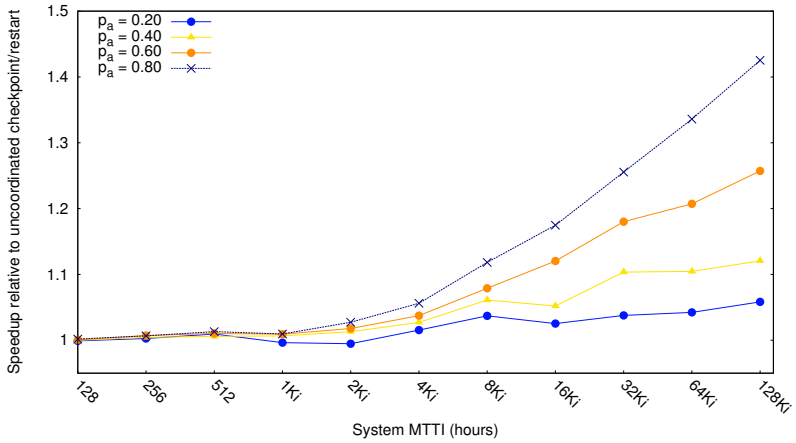
4.6 Simulating the Impact of Rollback Avoidance on Uncoordinated Checkpoint/Restart

The simulation framework described in this chapter facilitates the evaluation of the potential benefit of using rollback avoidance in conjunction with uncoordinated checkpointing. Figure 4.5 examines how increasing the fraction of errors for which rollback can be avoided impacts application execution time. The results in these figures were collected by simulating 65,536 nodes executing LAMMPS with the SNAP potential. The original trace was collected on 128 nodes and represents 10.12 hours of execution. Each simulation result is the arithmetic mean of 24 independent sim-

ulations. To understand the relationship between rollback avoidance and system reliability, experiments were conducted to examine two cases: (i) a moderately unreliable system ($\Theta_n = 5$ years) in Figures 4.5(a) and 4.6(a); and (ii) a very unreliable system ($\Theta_n = 1$ year) in Figures 4.5(b) and 4.6(b). The simulation results are presented from two perspectives. Figures 4.5(a) and 4.5(b) show how much faster the application is relative to uncoordinated checkpoint/restart when rollback avoidance is used. Figures 4.6(a) and 4.6(b) show the percentage of the application's overall execution time that is consumed by fault tolerance. To help isolate the impact of the probability of rollback avoidance (p_a), we assume that the overhead is zero (i.e., $o_a = 0.0$) in all cases. There is currently no widely-accepted method for calculating the checkpoint interval for uncoordinated checkpoint/restart. Because one of the explicit goals of uncoordinated checkpoint/restart is to reduce contention for write bandwidth to persistent storage, the checkpoint commit time (δ) used in these experiments is 2 seconds. The checkpoint interval (τ) is 2 minutes. To put this value in perspective, it is equal to the optimal checkpoint interval that would obtain for the more reliable configuration of this simulated system (i.e., $\Theta_n = 5$ years) if we used coordinated checkpoint/restart instead of uncoordinated checkpoint/restart.

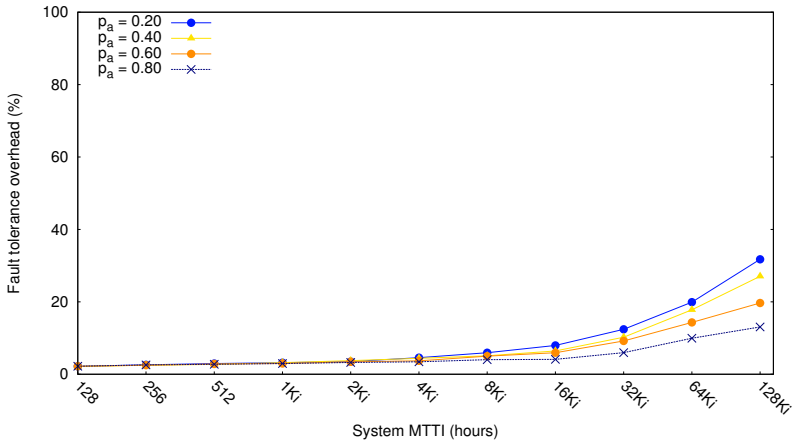


(a) $\Theta_n = 5$ year.

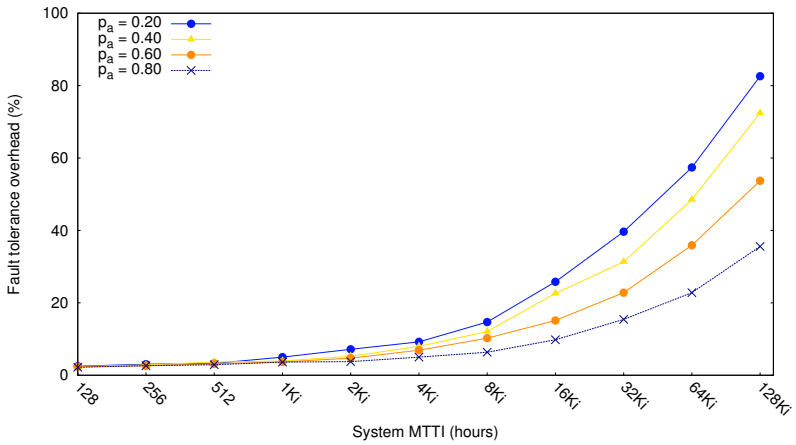


(b) $\Theta_n = 1$ year. Speedup relative to uncoordinated checkpoint/restart

Figure 4.5: *Effect of p_a on application execution time.* Evaluating the impact of rollback avoidance and uncoordinated checkpointing on application performance relative to uncoordinated checkpointing by itself for two different values of node MTBF (Θ_n). Each simulation result is average of 24 independent simulations. These data are based on the simulation of a 10.12 hour run of LAMMPS with the SNAP potential. The checkpoint commit time (δ) is 2 seconds, and the checkpoint interval (τ) is 2 minutes. To isolate the impact of the probability of rollback avoidance, p_a, all of these data assume zero overhead (i.e., $o_a = 0.0$).

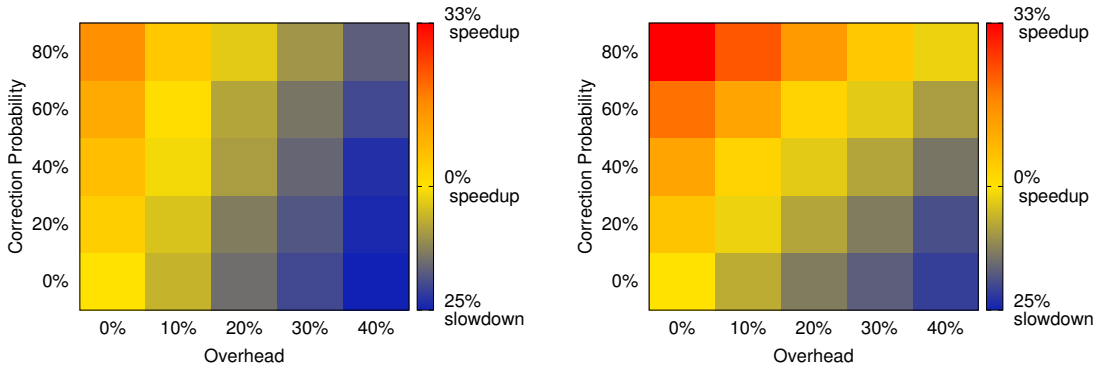


(a) $\Theta_n = 5$ years



(b) $\Theta_n = 1$ year

Figure 4.6: *Effect of p_a on fault tolerance overhead.* Evaluating the impact of rollback avoidance and uncoordinated checkpointing on the overall fault tolerance overhead for two different values of node MTBF (Θ_n). Each simulation result is average of 24 independent simulations. These data are based on the simulation of a 10.12 hour run of LAMMPS with the SNAP potential. The checkpoint commit time (δ) is 2 seconds, and the checkpoint interval (τ) is 2 minutes. To isolate the impact of the probability of rollback avoidance, p_a, all of these data assume zero overhead (i.e., $o_a = 0.0$).



(a) $\Theta_n = 5$ years (b) $\Theta_n = 1$ year

Figure 4.7: *Application speedup as a function of p_a and o_a.* Evaluating the relationship between the probability of rollback avoidance (p_a), the associated overhead (o_a) and application performance. These results were collected by simulating 65,536 nodes executing LAMMPS with the SNAP potential. This figure examines two different values for the MBTF (Θ_n), 1 year and 5 years, of the simulated nodes. The original trace was collected on 128 nodes and represents 10.12 hours of execution. Each simulation result is the arithmetic mean of 24 independent simulations. The checkpoint commit time (δ) is 2 seconds and the checkpoint interval (τ) is 2 minutes.

The data in this figure show that at least for this application (LAMMPS-snap), the benefits of rollback avoidance are somewhat modest even when the overhead is assumed to be zero. In all cases, the maximum observed speedup is less than 45%. However, Figures 4.6(b) and 4.6(a) show that the overall overhead of fault tolerance is small even on very unreliable systems. This is largely due to the relatively high efficiency of uncoordinated checkpoint/restart when paired with an application (LAMMPS-snap) that is relatively insensitive to the introduction of CPU detours (*cf.* [54]).

The largest system simulated in this section (128 Ki nodes) would have a system MTBF of approximately 40 minutes ($\Theta_n = 5$ years) or 8 minutes ($\Theta_n = 1$ year) if it used coordinated checkpoint/restart instead. By way of comparison, Chapter 3.6 examined the improvements in application performance that may result from using

rollback avoidance with coordinated checkpoint/restart. Figure 4.5 shows that the maximum speedup in application performance is approximately 1.4. In contrast, Figure 3.9(a) shows that in the case of coordinated checkpoint/restart on systems with very low MTBFs, rollback avoidance can potentially increase application performance by more than a factor of 6. However, Figure 3.9(b) shows that in the cases where rollback avoidance achieves this dramatic increase in performance, the overhead of fault tolerance is very large and likely prohibitive.

The next examination is of how the probability of rollback avoidance (p_a) and the overhead of avoiding rollback (o_a) interact and impact application execution time. Figures 4.7(a) and 4.7(b) contain heatmaps showing average speedup of LAMMPS-snap as a function of p_a and o_a. In these figures, speedup is calculated as the quotient of the average simulated execution time of the base case ($p_a = 0.0$, $o_a = 0.0$) divided by the average simulated execution time for each pair of values for p_a and o_a. In all cases, the average execution time is computed over 24 independent simulations. Figure 4.7(a) shows improvements in application execution time only when the overhead is very low. Increasing the overhead to 10% eliminates the benefits of rollback avoidance unless the probability of avoiding rollback exceeds 40%. As the overhead increases above 10%, rollback avoidance yields no execution time benefit. Figure 4.7(b) examines the impact on a less reliable system. In this case, more significant performance benefits are possible. However, these gains are possible only as failures become very prevalent in the system. Chapter 3 showed much larger benefits (on more reliable systems) when rollback avoidance is combined with coordinated checkpoint/restart.

The results presented in this section are consistent with existing research on the noise sensitivity of LAMMPS [54,162,163]. LAMMPS is relatively insensitive to noise events. As a result, when LAMMPS is combined with uncoordinated checkpointing its execution time is only modestly degraded. Therefore, the benefits of rollback

avoidance are also modest.

4.7 Chapter Summary

In this chapter, I presented a new and promising approach to simulating large-scale systems that use fault-tolerance mechanisms based on the checkpoint/restart model. We identified a set of platform, application, and resilience characteristics required for accurate and efficient simulation; described a prototype framework based on extensions to a validated and freely-available application simulator implementing the LogP model; showed how resilience processing overheads can be effectively modeled as CPU detours; and demonstrated empirically that the simulation approach described in this chapter accurately predicts the impact of resilience mechanisms.

I also used this simulation framework to evaluate the potential impact of using rollback avoidance with uncoordinated checkpoint/restart to reduce application execution time. As the data in this chapter show, the benefits of using rollback avoidance to improve the performance of noise-insensitive applications like LAMMPS-snap are limited. This result illustrates the potential limits to the benefits of rollback avoidance. However, combining rollback avoidance with uncoordinated checkpointing may yield greater benefits for applications that are more sensitive to noise.

Chapter 5

Similarity Engine: Exploiting Application Memory Redundancy to Improve Resilience

5.1 Introduction

In this chapter, I propose a novel rollback avoidance technique that leverages content similarity in the memory of HPC applications to improve resilience to uncorrectable memory errors. For example, when a memory error occurs on a page that is similar to one or more other pages in the address space of an application, information about the page's similarity can be used to reconstruct the contents of the damaged page without needing to terminate the affected application or restart it from a known good state (e.g., a checkpoint).

I begin by describing the design and implementation of an HPC-oriented memory similarity service: the *Similarity Engine*. The Similarity Engine provides a general, application-independent, lightweight service for detecting and tracking per-node

memory similarity in HPC applications. I then describe how this service can be leveraged to improve application performance by avoiding rollback when uncorrectable memory errors occur. I also describe two additional ways in which the service provided by the Similarity Engine can be used to improve application resilience characteristics.

5.2 Implementing the Similarity Engine

The goal of the Similarity Engine is to discover and exploit process-level memory similarity in HPC applications to improve their resilience and performance. This section presents the definition of memory similarity and describes the mechanisms that the Similarity Engine uses to identify and track this similarity.

5.2.1 Overview

The Similarity Engine categorizes all of the pages in an application's memory into four categories–zero, duplicate, similar, and unique–defined as follows:

- *Zero pages:* pages whose contents are entirely zero
- *Duplicate pages:* pages that (a) are not zero pages; and (b) exactly match the contents of one or more other pages
- *Similar pages:* pages that (a) are not duplicate or zero pages; and (b) can be paired with at least one other page in application memory such that the difference between the two is smaller than a tunable threshold: the *difference threshold*.
- *Unique pages:* pages that do not fall into any of the preceding three categories

Application	Description
HPCCG	One of the Mantevo mini-applications [144]. Designed to mimic finite element generation, assembly and solution for an unstructured grid problem.
LAMMPS	Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS). A classical molecular dynamics simulator from Sandia National Laboratories [143]. The data presented in this chapter are from experiments that use the Lennard-Jones (lammps-lj) and Embedded Atom Model (lammps-eam) potentials that are included with the LAMMPS distribution.
CTH	A multi-material, large deformation, strong shock wave, solid mechanics code [116] developed at Sandia National Laboratories. The data presented in this chapter are from experiments that use inputs that describe the simulation of the detonation of a conical explosive charge (CTH-st) and the simulation of an explosive detonated near a steel plate (CTH-blastplate).
LULESH	Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH). A proxy application from the Department of Energy Exascale Co-Design Center for Materials in Extreme Environments (ExMatEx). LULESH approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh [3, 108].
SAMRAI	Structured Adaptive Mesh Refinement Application Infrastructure (SAMRAI). A framework from the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory that is designed to enable the application of structured adaptive mesh refinement to large-scale multi-physics problems

Table 5.1: Descriptions of the set of workloads used for evaluating the performance characteristics of the Similarity Engine.

Although they are categorized separately here, when I discuss *similarity* in general in this chapter I mean the set of pages that are duplicate, zero, or similar.

Based on these definitions, the Similarity Engine works by first tracking application memory allocation and periodically scanning allocated memory. During this scan, it identifies zero and duplicate pages by computing hashes of page contents, and uses efficient heuristics to detect similarity. This similarity detection is based on using a difference algorithm that computes differences between pairs of pages, and heuristics that choose pairs of pages that are likely to be similar.

5.2.2 Tracking Application Memory

The Similarity Engine tracks an application's memory allocation by interposing code between the application and the standard C memory allocation functions (e.g., `malloc`, `calloc`, `free`) using features of the GNU linker. Each time the application allocates (or deallocates) memory, the Similarity Engine updates its view of the memory that is currently allocated by the application.

Managing Memory Modification

Exploiting memory contents requires an accurate picture of which pages have been modified since they were last categorized. To this end, the Similarity Engine uses `mprotect` to make every page of the application's allocated memory read-only. When the application writes to a page of read-only memory, a segmentation fault occurs. The Similarity Engine uses a `SIGSEGV` signal handler to receive notification of each segmentation fault. The Similarity Engine's signal handler updates its metadata to indicate that the page has been accessed and restores write privileges to the accessed page.

Periodically, the Similarity Engine needs to re-protect memory pages that the application has written to. Therefore, the Similarity Engine divides the application's execution time into tunable *protection intervals* to track memory accesses. By

default, each protection interval is 60 seconds long. The Similarity Engine configures a `SIGALRM` signal to notify it when each interval begins. At the beginning of each protection interval, the Similarity Engine makes every modified page of memory read-only and computes the MD5 hash of every page that has been accessed since the beginning of the preceding interval. We assume that the contents of application memory are not adversarial. As a result, two pages with same hash value have the same contents with very high probability. Moreover, if the hash value of a page has not changed, then the contents of the page have not been modified.

Handling System Calls

The Similarity Engine requires some pages of application memory to be read-only. However, when read-only memory is passed to the kernel in a system call, writes to this memory do not invoke Similarity Engine's user-level segmentation fault handler. As a result, system calls may fail. If return values are not carefully checked by the application, the consequences of the failure may be difficult to predict.

To determine the extent to which the applications examined in this chapter may attempt to pass references to read-only memory to the kernel, I created a version of the Similarity Engine that leverages Linux's `ptrace` mechanism to intercept and inspect the system calls made by the application. Specifically, this version of the Similarity Engine creates a child thread during its initialization and uses `ptrace` to attach to the application (its parent). Each time the application enters a system call, the child examines the contents of the registers that contain the arguments being passed to determine whether they contain a reference to memory that the Similarity Engine is actively tracking. For the set of applications and associated input decks considered in this chapter, no references to tracked memory in the arguments to were found in any system call.

There are likely applications for which references to user-allocated heap memory are passed to system calls. In these cases, efficiently identifying similarity without jeopardizing the correctness of the the simulation will likely require integrating the Similarity Engine into the kernel.

Passing memory references to MPI

Passing read-only message buffers to the MPI library can result in unpredictable behavior. The Similarity Engine uses the PMPI profiling layer to intercept MPI calls that reference one or more message buffers. For each message buffer, the Similarity Engine determines whether the buffer occupies memory that it is tracking (i.e., whether it is memory that is or may become read-only). If so, the Similarity Engine makes it writable and updates its metadata to reflect the fact that it is no longer managing this memory. As a result, the pages that comprise the message buffers passed to MPI are categorized as unique in all of the statistics presented in this chapter. The Similarity Engine currently lacks a mechanism for resuming the tracking of memory used for MPI message buffers.

5.3 Discovering Similarity

The performance of the Similarity Engine depends on how efficiently it is able to identify pairs of similar pages. This section describes these differencing and pair identification steps in detail. Because of the importance of heuristics to efficiently perform these steps, this section evaluates the tradeoffs for the differencing and pair identification steps.

5.3.1 Experimental Setup

The remainder of this chapter presents the results of the experiments conducted to evaluate the potential costs and benefits of exploiting memory content similarity. The characteristics of seven workloads are considered. These workloads, described in Table 5.1, include two important DOE production applications (LAMMPS and CTH), a proxy application (LULESH) from the Department of Energy's Exascale Co-Design Center for Materials in Extreme Environments (ExMatEx) [61], a mini-application from Sandia's Mantevo suite (HPCCG) and an example application from an important library used in large-scale DOE production applications (SAMRAI). For each of LAMMPS and CTH, I consider two different input decks.

For all of the applications except for CTH, the experiments were performed using Compton, a Linux Infiniband cluster. Because CTH is export-controlled, the CTH experiments were performed on Chama, also a Linux Infiniband cluster. Details on the composition and configuration of these clusters are presented in Table 5.2. I use 4 KiB memory pages throughout. The remainder of this section evaluates the effectiveness and costs of computing differences and identifying potentially similar pages. It does so by presenting and examining the results of a series of small-scale experiments. These experiments were conducted by running each of the seven workloads on 8 MPI processes across 4 nodes of the clusters described above.

5.3.2 Computing Page Differences

At the beginning of each protection interval, the Similarity Engine identifies similarity by computing differences between pairs of pages. This section evaluates the performance characteristics of four algorithms for computing page differences. These algorithms include two well-known delta encoding algorithms, a novel lightweight

	Compton	Chama
Nodes	42	1,232
Sockets/Node	2	2
Processors	Intel Sandy Bridge (2.6 GHz / 8 cores)	Intel Sandy Bridge (2.6 GHz / 8 cores)
Operating System	Linux 2.6.32	Linux 2.6.32
Interconnect	Mellanox MT26428 QDR InfiniBand HCA	QLogic QLE7340 QDR InfiniBand HCA
MPI	OpenMPI 1.8.1	OpenMPI 1.8.4

Table 5.2: Configuration details of clusters used to gather experimental data.

differencing algorithm, and page compression.[1] A description of each of the four difference algorithms follows:

- *bsdiff* (and its mirror, *bspatch*) use suffix sorting and bzip2 to compute differences between pairs of binary files [39].
- *Xdelta* is an open-source delta encoder/decoder based on VCDIFF [94]. VCDIFF is both an algorithm and a format for encoding the differences between binary files [101].
- *xor+lz4* is a novel lightweight differencing algorithm that combine naive differences (bit-wise exclusive-or) with lightweight compression (lz4). lz4 is a lossless data compression algorithm that is based on LZ77 compression [2].
- *bzip2* is a lossless data compression algorithm that uses Burrows-Wheeler transforms and Huffman coding [1].

[1]In the case of page compression, the compressed page can be viewed as the difference between the target page and a null page.

These algorithms compute the difference between a *candidate page* and a *reference page*. This difference would allow the Similarity Engine to recreate the candidate page from the reference page in the event that the candidate page was corrupted. Because xor+lz4 generates symmetric differences, the differences it generates to can be used to reconstruct either the reference page or the candidate page from the other. The other three algorithms generate asymmetric differences (i.e., the difference between the candidate page and a reference page can only be used to reconstruct the candidate page). As a result, xor+lz4 requires the computation of half as many differences.

To efficiently identify similarity in application memory, the Similarity Engine must be able to quickly encode small differences between pairs of pages. It also must be able to decode these differences and quickly reconstruct a page from its reference page. The speed of difference encoding strongly influences the runtime overhead of the Similarity Engine. The size of the differences will dictate its memory overhead. To limit the memory overhead, similarity is defined relative to a tunable *difference threshold*. A memory page is similar only if the difference between it and a reference page falls below this threshold.

The suitability of these algorithms for identifying similarity was examined with two microbenchmarks. To run these microbenchmarks, I constructed a library that takes periodic snapshots of the allocated memory of each of our target workloads. I then randomly chose a maximum of 5,000 candidate pages from each snapshot of each application that were neither zero nor duplicate. For each candidate, I computed the difference between it and every other page in the same snapshot.

The first microbenchmark measures the speed of difference encoding and decoding. The results are shown in Figure 5.1. The fastest algorithm by a substantial margin is xor+lz4. It is more than eight times faster than Xdelta and nearly 200 times faster than bsdiff. Moreover, because Xdelta, bzip2 and bsdiff generate asymmetric differences, they must compute twice as many differences as xor+lz4 for the

same number of pages.

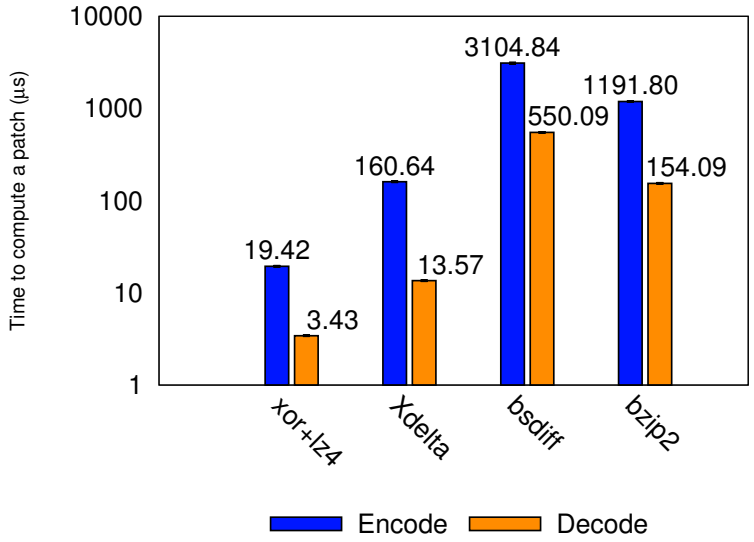


Figure 5.1: Difference speed microbenchmark. The average time required to compute the difference between two pages for each of four algorithms. *Encode* is the average time to compute the difference between two pages. *Decode* is the average time to reconstruct a page. The average is computed over 5,000 random pairs of pages taken from each of the snapshots taken for each application. This plot includes error bars showing the standard error. However, the error is too small to be easily seen.

The second microbenchmark measures the size of the differences generated by each algorithm. For each difference algorithm, the smallest difference computed for each candidate was recorded. The results are shown in Figure 5.2. In this figure, a point at (x, y) indicates that for a y fraction of the candidates were considered, the size of the smallest difference was less than or equal to x bytes.[2] As this figure demonstrates, the speed of xor+lz4 comes with a cost. It generates substantially fewer small differences than the other three algorithms. As a result, for a fixed difference threshold, xor+lz4 will tend to identify fewer similar pages.

[2]In principle, the difference between any pair of 4 KiB pages can be captured in 4 KiB (i.e., as the exclusive-or of the two). However, as this figures shows, none of these algorithms use this optimization. As a result, some differences produced by these algorithms are larger than 4 KiB.

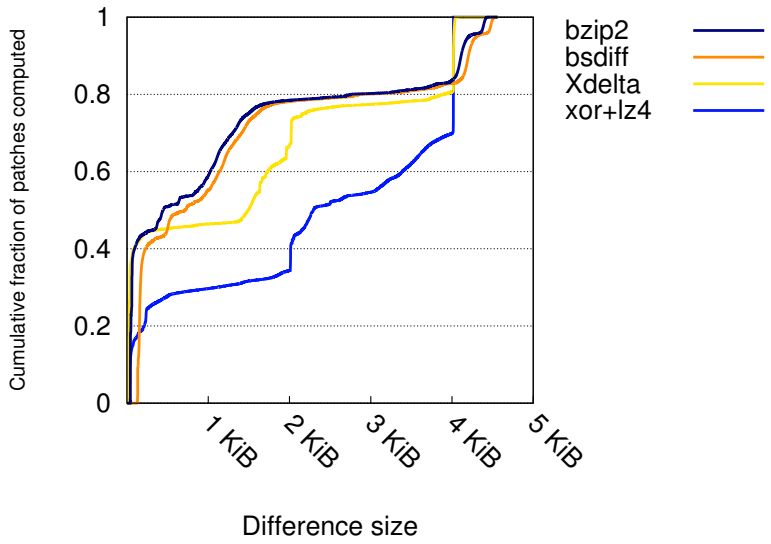


Figure 5.2: Difference size microbenchmark. The distribution of the size of the differences computed by the four difference algorithms considered. These data represent the result of computing the difference between 5,000 random pairs of pages from each snapshot of each application.

5.3.3 Finding Potentially Similar Pages

Due to the cost of exhaustively computing differences between the pages in an applications memory, an efficient method for identifying pairs of pages that are likely to be similar (i.e., pairs of pages for which the difference between them is small) is necessary. I examined four heuristics for identifying potentially similar pages:

- *Neighbor*: the pages within the same memory allocation that are immediately adjacent to the candidate page in the application's virtual address space;
- *Random*: two pages chosen randomly from the same memory allocation as the candidate page;
- *Same*: all of the pages (except for the neighbors) in the same memory allocation as the candidate page; and

- *Other*: all of the pages that are not in the same memory allocation as the candidate page.

I used a microbenchmark to determine the relative effectiveness of these approaches. Using the same set of snapshots described above, 5,000 candidate pages are randomly chosen that were neither duplicate nor zero. For each candidate, the difference between it and the pages in each of the four categories described above is computed. Within each of these categories, the size of the smallest observed difference is recorded. The results are presented in Figure 5.3. This figure shows that there are benefits to considering larger regions of memory. Considering pages within the same allocation as the candidate page is no worse than considering all of the application's other allocations. For all but CTH-st, this means computing fewer differences.

Table 5.3 shows that Same or Other requires the computation of hundreds or thousands of differences. Because Neighbor and Random represent the minimum of just two differences, these approaches are surprisingly effective. Nonetheless, developing more effective and efficient techniques for identifying potentially similar pairs of pages would increase the benefits of exploiting similarity.

Application	Neighbor	Random	Same	Other
CTH-blastplate	2	2	2858	4257
CTH-st	2	2	36279	639
HPCCG	2	2	9823	41622
LAMMPS-eam	2	2	3429	136988
LAMMPS-lj	2	2	4382	156924
LULESH	2	2	274	7492
SAMRAI	2	2	137	45157

Table 5.3: Difference size microbenchmark. Mean number of differences computed per application for each of four similarity heuristics: neighbor, random, same and other.

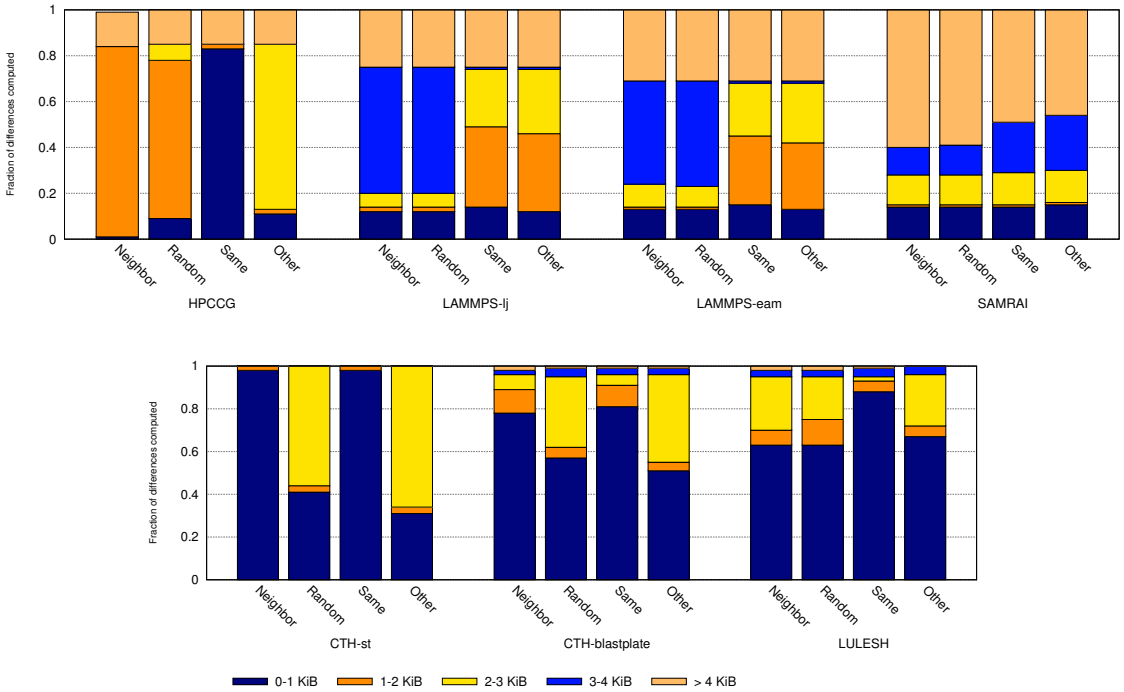


Figure 5.3: Similarity heuristic microbenchmark. For each target application a sequence of snapshots of allocated memory were collected. Up to 5,000 candidate pages are randomly selected from each snapshot of each application. For each candidate, differences are computed between it and all of the other pages in memory (differences between pairs of pages that are identical are not computed). The resulting set of differences is binned based on their size. This figure shows the size distribution for each of four categories of differences. *Neighbor* is the set of differences that are computed between two pages that are adjacent in the virtual address space and belong to the same allocation. *Random* is the set of differences the candidate page and each of two pages chosen randomly from the same memory allocation as the candidate page. *Same* is all of the pages (except for the neighbors) in the same memory allocation as the candidate page. *Other* are the pages that are not in the same memory allocation as the candidate page.

5.3.4 Analysis

Based on the results of these microbenchmarks, the Similarity Engine is configured to use the xor+lz4 algorithm for difference computation and to use the neighbor heuristic for choosing potentially similar pages. Although these choices reduce the

amount of similarity that the Similarity Engine is able to identify, they significantly reduce the cost of discovering the similarity found. For the applications examined in this section, the runtime overhead of other difference algorithms and similarity identification heuristics are much too large for the additional overhead to be amortized by the benefits of the additional similarity they discover.

5.4 Evaluating Similarity

This section evaluates the amount of similarity that Similarity Engine can discover and track in HPC applications, and measures the cost of tracking this similarity. The experiments presented in this section were conducted on the clusters described in Table 5.2. These experiments used the set of workloads described in Table 5.1. The input decks for each workload and the resultant application execution characteristics are described in Table 5.4. I ran each workload, except LULESH, with 128 processes on 16 nodes of these clusters. Because LULESH requires the number of processes be a perfect cube, it ran with 125 process. I repeated each experiment ten times. The data presented in this section required 900 experiments to be conducted in which the Similarity Engine was linked against one of the workloads. Approximately, 10% (91/900) of these experiments failed to complete. I discarded all of the data collected during these failed experiments and repeated the experiments.

5.4.1 Memory Overhead

Effectively exploiting similarity requires the Similarity Engine to maintain metadata about the memory that is currently allocated by the application. As pairs of similar pages are identified, the Similarity Engine needs to store the encoded difference and the address of the reference page. There is a tradeoff between the difference threshold

Application	Original median runtime (seconds)	Mean allocated memory (pages)	Input deck
CTH-st	324.53	37096	`st.128` stop cycle = 155
CTH-blastplate	429.04	5130	`blast-plate.in` 1200×1200 mesh
HPCCG	362.39	51457	`nx=128, ny=64, nz=64` max. number of iterations = 5000
LAMMPS-lj	492.24	161106	`in.lj x=32 y=16 z=16` 200 time steps
LAMMPS-eam	598.53	187673	`in.eam x=32 y=16 z=16`
LULESH	347.18	3167	`-s 32`
SAMRAI	440.03	9249	`octant_3blk.3d.input` $225 \times 225 \times 225$ grid geometry

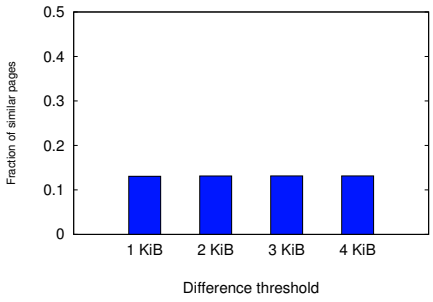
Table 5.4: Description of the execution parameters for the seven target workloads.

and the resulting overhead. Increasing the difference threshold identifies more similar pages, but results in the retention of more and larger differences.

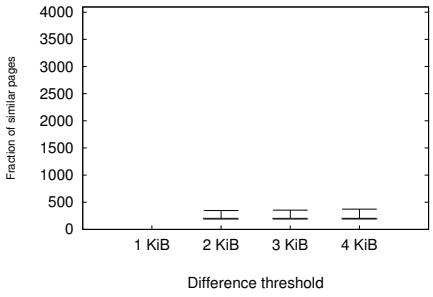
To examine this tradeoff, I ran each target application with four difference thresholds: 1 KiB, 2 KiB, 3 KiB and 4 KiB. Each experiment also measured the volume of metadata required to track the application's memory use. The metadata includes the data structures necessary to manage all of the application's memory allocations, the data structures for managing computed differences, and the corpus of the differences. The results are shown in Figure 5.4. With the exception of HPCCG, for difference thresholds that are 3 KiB or smaller, a few hundred bytes of metadata per 4 KiB

memory page is sufficient.

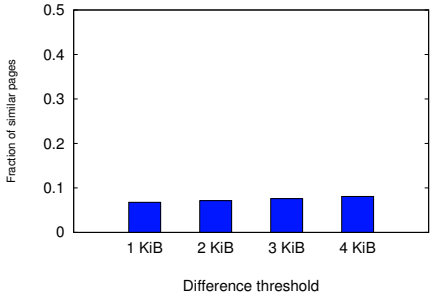
For workloads like CTH-blastplate and CTH-st, the fraction of similar pages that the Similarity Engine is able to identify changes relatively little as the difference threshold increases. For the remaining workloads there are benefits to be reaped by considering a larger difference threshold. These data suggest that a difference threshold of 3 KiB represents a good tradeoff between the number of similar pages and metadata volume.



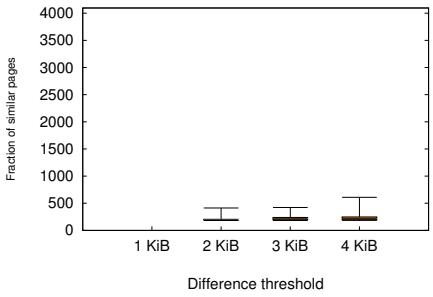
(a) CTH-st similarity



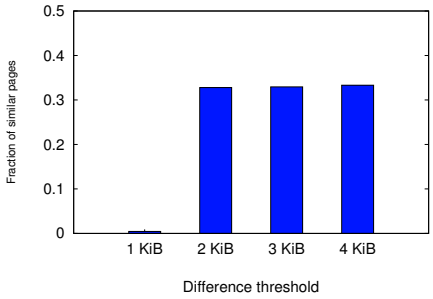
(b) CTH-st metadata



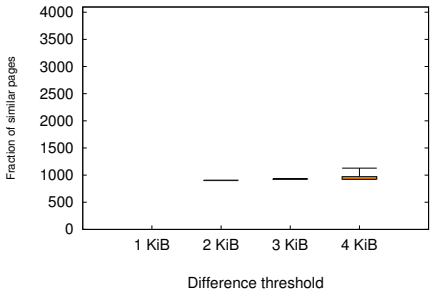
(c) CTH-blastplate similarity



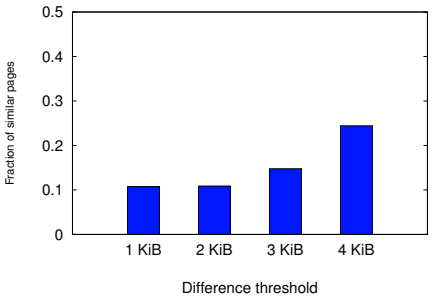
(d) CTH-blastplate metadata



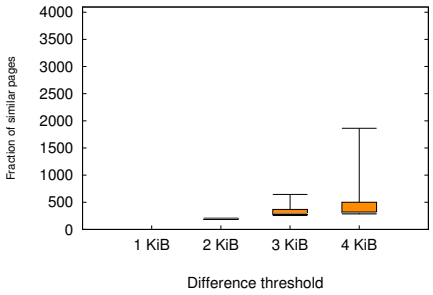
(e) HPCCG similarity



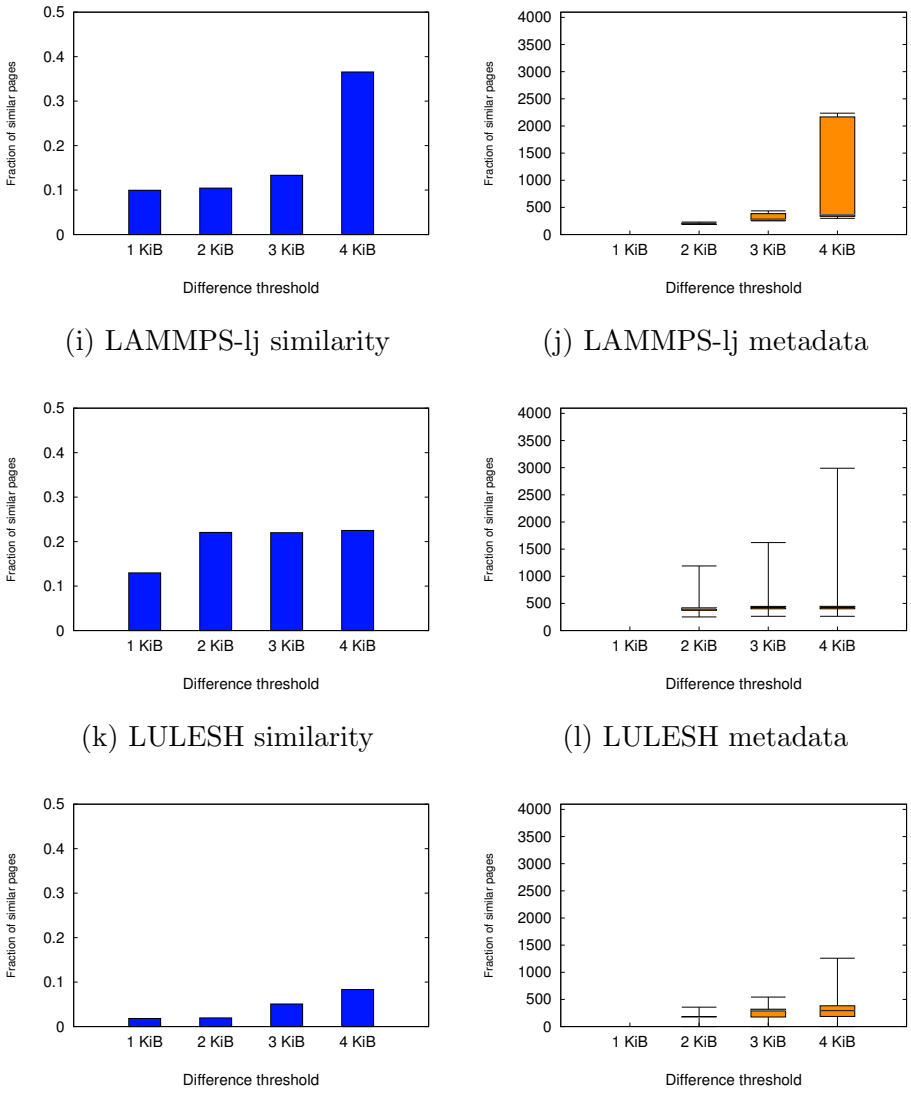
(f) HPCCG metadata



(g) LAMMPS-eam similarity



(h) LAMMPS-eam metadata



(i) LAMMPS-lj similarity

(j) LAMMPS-lj metadata

(k) LULESH similarity

(l) LULESH metadata

(m) SAMRAI similarity

(n) SAMRAI metadata

Figure 5.4: Difference threshold benchmark. For each target application, this figure consider the trade off between the number of similar pages and the metadata storage overhead. As the difference threshold increases, larger differences are allowed and the metadata overhead increases.

5.4.2 Runtime Overhead

Identifying similarity using these techniques requires that the Similarity Engine occasionally interrupt the application. It interposes metadata maintenance operations between the application and the standard C memory allocators. It restores write privileges as accesses to read-only pages generate segmentation faults. As each protection interval begins, it must also change the access privileges for pages of allocated memory to be read-only and identify similar pairs of pages. For each of the target workloads, Table 5.5 shows the inflation of the application's execution time due to the Similarity Engine.

Application	Runtime Overhead (all similar)	Runtime Overhead (duplicate only)	Write Exceptions Per Second	Allocations Added Per Second
CTH-blastplate	0.60%	4.69%	23.99	1.04
CTH-st	2.76%	1.99%	485.46	0.06
HPCCG	1.26%	1.30%	54.31	0.06
LAMMPS-eam	14.60%	11.14%	2471.62	1.65
LAMMPS-lj	13.28%	9.92%	1984.47	1.96
LULESH	9.90%	8.89%	31.42	3319.75
SAMRAI	1.84%	1.58%	67.64	594.68

Table 5.5: Median runtime overheads (ratio of median execution times) using xor+lz4, a 3072-byte difference threshold, and a 60 second protection interval. The coefficient of variation for the execution time data used to generate this table is below 1.5% for CTH-st, HPCCG, LAMMPS-eam, LAMMPS-lj, and SAMRAI. The coefficient of variation for the "all similar" runtime overhead of LULESH is approximately 22%. The coefficients of variation for the other two LULESH experiments is less than 1%. The coefficients of variation for all of the CTH-blastplate experiments exceed 20%.

This table shows that the applications with the highest overheads have high rates of memory allocation or they write frequently to memory that the Similarity Engine has made read-only. LULESH frequently allocates and deallocates mem-

ory. Frequent changes to the application's memory allocation necessitates frequent metadata modification to maintain an accurate view of memory. Accessing pages of read-only memory can also increase runtime overhead. In addition to the signal handling overhead, at the end of each protection interval the Similarity Engine computes and encodes differences for every page that has been written to during the interval. LAMMPS-lj and LAMMPS-eam have relatively stable memory allocations, but they write to large numbers of read-only pages. The four workloads (CTH-st, CTH-blastplate, SAMRAI, and HPCCG) that write to relatively few read-only pages and have stable memory allocations exhibit the lowest overheads.

This table also considers the incremental cost of identifying similar pages in addition to duplicate and zero pages. The second column of Table 5.5 shows the overhead if the Similarity Engine only considers duplicate and zero pages. For all of the workloads save LAMMPS-lj and LAMMPS-eam the additional runtime overhead of identifying similar pages is quite modest.

Table 5.5 shows that for CTH-blastplate, the results for the "duplicates only" experiments indicate higher overhead than the results of the "all similar" experiments. These results are counter-intuitive given that the work of tracking duplicate and zero pages is a subset of the work of tracking similar, duplicate and zero pages. Figure 5.5 shows the execution times measured over twenty trials of the "duplicates only" and "all similar" experiments. These results are very noisy; even for the unmodified case, the fastest and slowest execution times differ by hundreds of seconds. As a result, it is difficult to make fine-grained distinctions between the runtimes of each of these three cases.

Another factor to consider in the runtime overtime analysis is the extent to which the Similarity Engine is doing useful work. At the beginning of each protection interval, the Similarity Engine computes differences between pairs of memory pages. A *useful difference* is a difference that is smaller than the difference threshold. On

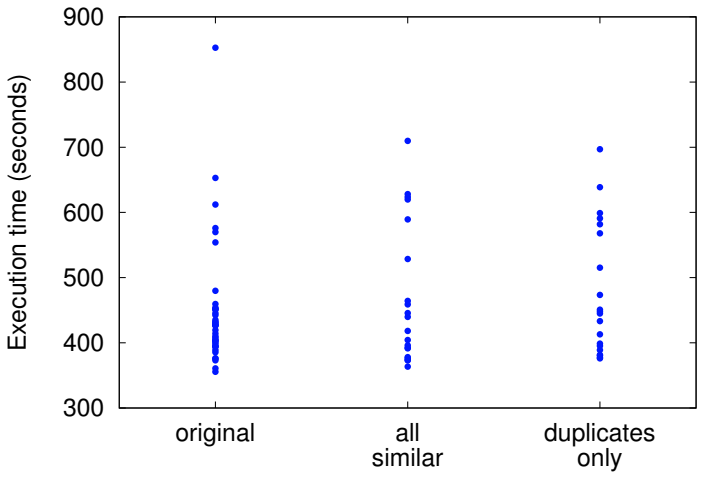


Figure 5.5: Raw execution time data for CTH-blastplate. The amount of time required to complete a fixed problem varies significantly across runs. As a result, it it difficult to make detailed comparisons between these three application configurations: *original* is the original CTH executable (i.e., not linked against the Similarity Engine), *all similar* is CTH linked against a version of Similarity Engine that tracks similar, duplicate, and zero pages, *duplicates only* is CTH linked against a version of Similarity Engine that only tracks duplicate and zero pages.

the other hand, if the difference exceeds the difference threshold, it is discarded and the time spent computing the difference is wasted. Table 5.6 examines the percentage of useful differences that are computed as a function of the difference threshold. As this table shows, few of the differences computed in the memory of LAMMPS-lj, LAMMPS-eam, and SAMRAI are useful. Given that LAMMPS-lj and LAMMPS-eam exhibit the highest runtime overheads of these seven workloads, a lightweight heuristic for identifying and excluding pairs of pages that are unlikely to be similar could improve the performance of the Similarity Engine with these workloads.

Application	Difference Threshold			
	1 KiB	2 KiB	3 KiB	4 KiB
LAMMPS-lj	3.16%	3.70%	10.16%	72.32%
CTH-blastplate	49.67%	57.43%	76.12%	83.31%
SAMRAI	14.13%	14.99%	16.56%	32.10%
LAMMPS-eam	2.21%	2.59%	13.51%	64.21%
HPCCG	43.36%	70.49%	72.98%	100.00%
CTH-st	98.01%	99.86%	99.92%	99.96%
LULESH	47.35%	61.33%	69.53%	76.79%

Table 5.6: Useful difference rate. For each combination of workload and difference threshold, this table contains the percentage of computed differences that are *useful differences*, i.e., smaller than the associated threshold. Differences that are larger than the threshold are discarded. The time spent computing differences that are ultimately discarded is therefore wasted.

5.4.3 Prevalence of Similarity

The ultimate objective is to identify similarities in memory. To evaluate how effectively the Similarity Engine can extract similarity, I configured it with a protection interval of 60 seconds and a sample interval of 20 seconds. Configured this way, the Similarity Engine looks for similarity and re-protects memory every 60 seconds. Every 20 seconds it examines the current state of memory to determine how much similarity remains since the beginning of the protection interval. Each time the application writes to a similar, duplicate or zero page, its state and its relationship to other pages in memory are no longer known. Because accessed pages must be classified as unique, the prevalence of similarity necessarily decreases between protection intervals.

The results of these experiments are shown in Figure 5.6. This figure shows the mean fraction of the pages in application memory that fall into the four page categories defined in this chapter. For each application, this figure shows the average fraction of memory in each category for samples taken at the beginning of a pro-

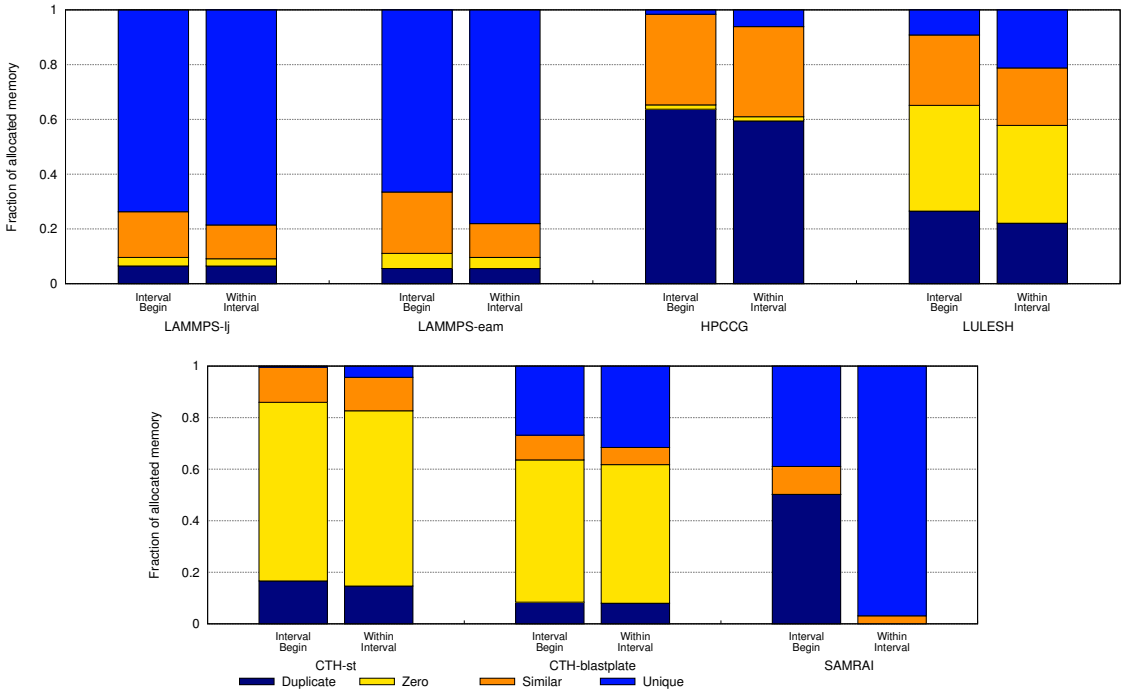


Figure 5.6: Page categorization. A comparison of the average page categorization at the beginning of a protection interval and the average within a protection interval. When a page is modified, the Similarity Engine no longer knows its relationship to other pages in memory. As a result, it must be categorized as a unique page. These data use xor+lz4 and a difference threshold of 3 KiB.

tection interval. It also shows the average fraction of memory in each category for samples taken within a protection interval. The mean is computed across all application processes, samples and trials. For each application, there is more similarity at the beginning of a protection interval than within the interval. The difference is particularly stark for SAMRAI. On average nearly 75% of memory is similar, duplicate or zero at the beginning of a protection interval; these categories comprise less than 6% of memory within an interval. The differences are more modest, but still significant, for the other four applications. For HPCCG and LULESH significant similarity exists even within a protection interval. More modest similarity is found in the memory of LAMMPS-lj and LAMMPS-eam, but it may still be possible to

effectively exploit it.

5.4.4 Variability of Similarity

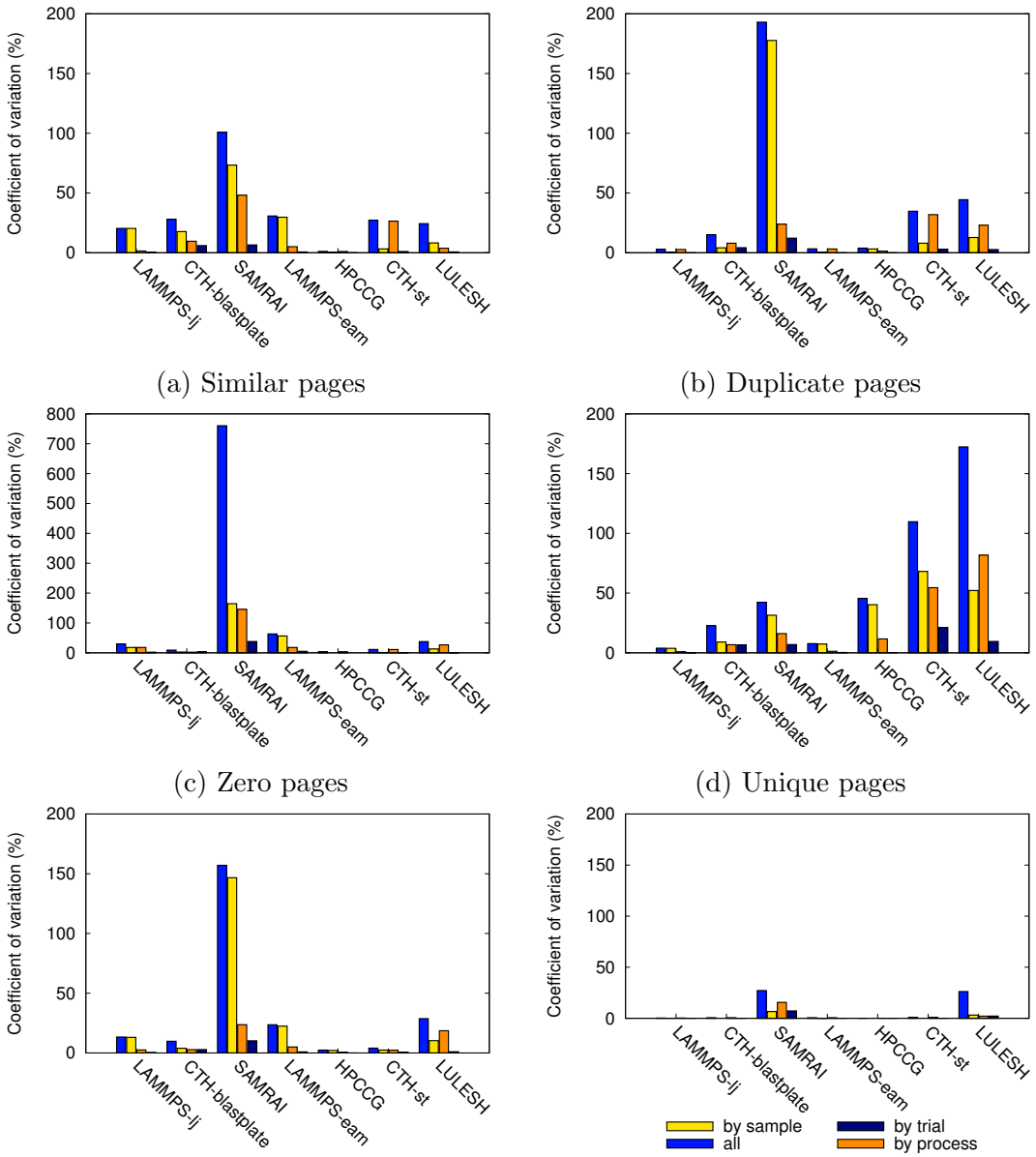
Figure 5.6 considers the mean fraction of pages in each category. Because the dataset is multi-dimensional, this subsection also examines how much variability is associated with each dimension of the dataset. To isolate the variability due to changes in a single dimension, I computed the mean over the data points that correspond to a particular value in a given dimension. For example, to evaluate the variability by process, I computed the mean fraction of pages in each category for all of the samples collected from the process that was assigned MPI rank 0. To examine the variability in the page categorization across processes, I computed the mean in this way for every process.

I measure the variability in the data by computing the coefficient of variation over each dimension.[3] The results are shown in Figure 5.7. First, there is little variability in these data across trials. Additionally, the total number of memory pages allocation by these workloads is relatively constant across all dimensions. SAMRAI exhibits signficant variation in the fraction of duplicate, zero, and similar that comprise its memory. This behavior is largely due to the fact that a significant fraction of SAMRAI's memory is composed of unique pages. As a result, small changes in the absolute number of duplicate, similar, and zero pages can result in large variations in their respective fractions. Similarly, there is a larger variation in unique pages for CTH-st and LULESH. This is due in part to the fact that the memory allocations for these applications contain a relatively small fraction of unique pages.

The fraction of memory that is comprised of non-unique (similar, duplicate and zero) pages is a good proxy for the variability of the potential benefits of exploiting

[3]The *coefficient of variation* is the standard deviation (σ) divided by the mean (μ).

memory content similarity. Non-unique pages are those pages that the Similarity Engine approach is able to exploit. Figure 5.7(e) demonstrates that there is very little variation in the fraction of non-unique pages in any of the dimensions of the dataset. The exception is again SAMRAI. There are two principal reasons for this. Figure 5.6 shows that for SAMRAI the fraction of duplicate pages at the beginning of a protection interval is much larger than the fraction within a protection interval. Moreover, because the fraction of non-unique pages in SAMRAI's memory allocations is quite small within a protection interval, small changes in the absolute number of non-unique pages can lead to significant variation in the overall fraction.



(a) Similar pages

(b) Duplicate pages

(c) Zero pages

(d) Unique pages

(e) Non-unique pages

(f) Total pages

Figure 5.7: Page category variability. An examination of the variability of the composition of each application's memory using a 3072-byte difference threshold. *All* is the value of the coefficient of variation when computed over the entire dataset. *By process* represents how much variation exists across application processes. *By sample* represents the variation over the lifetime of the application. *By trial* represents the variation across successive executions of the application.

5.5 Exploiting Memory Similarity

5.5.1 Uncorrectable Memory Errors

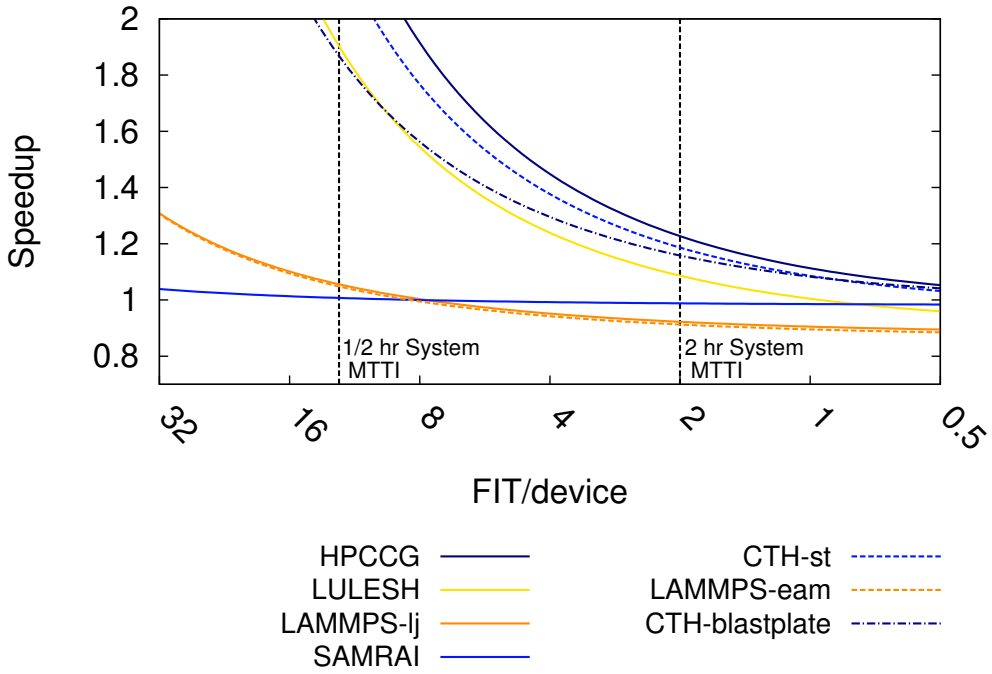


Figure 5.8: Modeled application speedup. This figure uses the average fraction of memory that is duplicate, zero or similar during a protection interval and an existing model of rollback avoidance [109] to predict application speedup for next-generation applications whose memory characteristics resemble one of the seven target workloads. This figure was generated using the runtime overhead from Table 5.5 and the probability of correcting a memory error from Figure 5.6. The system characteristics are drawn from the hypothetical extreme-scale system described in Table 5.7. The x-axis corresponds to the reliability of a single DRAM device in FIT, or failures per billion devices-hours of operation.

Uncorrectable DRAM errors have been shown to be a significant source of failure on current and future leadership-class HPC systems [155]. When an uncorrectable ECC error is detected on a modern x86 system, the memory controller raises a Machine Check Exception (MCE) in the processor. The consequences of raising an

MCE vary by operating system. Recent versions of Linux attempt to minimize the impact of an MCE by adopting simple recovery strategies. For example, if the fault occurred on a page whose contents are backed up by disk (e.g., a clean page in the page cache), the error can be handled by invalidating the appropriate cache or page table entry. In the event that none of its recovery strategies is successful, Linux poisons the hardware page and kills all of the processes that had the faulted page mapped into their address space [99]. In other operating systems (e.g., the Kitten lightweight kernel [145], older versions of Linux), an MCE simply crashes the node.

For each duplicate or similar page, the Similarity Engine maintains a description of its reference page(s) (i.e., the other pages in the system that are either duplicated by or similar to the page under consideration). In the case of similar pages, it also stores the appropriate encoded difference. When an uncorrectable memory error occurs on a similar, duplicate or zero page, the metadata maintained by the Similarity Engine can be used to reconstruct the contents of the damaged page. Reconstructing a duplicate page is straightforward. The contents of the damaged page are reconstructed from the contents of one of its reference pages.[4] For zero pages, the damaged page can be replaced with a page filled with zeros. For similar pages, the process is only slightly more complex. The contents of a damaged page can be reconstructed by applying the difference stored in the Similarity Engine's metadata to the associated reference page.

As shown in Figure 5.1, using xor+lz4 to reconstruct a damaged page by decoding the difference stored in the metadata is extremely fast: on average $3.23\mu s$ per 4 KiB page. The additional memory required for metadata does not significantly increase the vulnerability of the application to memory errors. An error in the metadata does not affect the continued operation application. Moreover, if an error occurs in the set of stored differences (which is the majority of the metadata), the Similarity

[4]In practice, it may be prudent to reconstruct the page in a different physical location in memory

Engine can, with high probability, invalidate the difference and regenerate it at the beginning of the next protection interval. In the worst case, the system can take a proactive checkpoint (to eliminate lost work that would need to be re-executed) and restart the application.

PARAMETER	VALUE
nodes	131,072
total system memory	32 PiB
memory devices/node	1152
FIT/node (excluding memory)	1100
checkpoint commit time	10 minutes

Table 5.7: Characteristics of the hypothetical next-generation, extreme-scale system used to generate Figure 5.8. A FIT value of 1100 corresponds to an MTBF of approximately 100 years The checkpoint commit time assumes that checkpoints are written to a parallel file system. Its value is based on existing studies of checkpoint performance [25, 52, 119].

Exploiting similarity in this way allows the application to continue execution when an otherwise uncorrectable memory error occurs rather than restarting and rolling back to the last checkpoint. This increases the mean time to interrupt (MTTI) of the system. The model of rollback avoidance presented in Chapter 3 can be used to examine how the increased MTTI would affect the performance of an application executing on a hypothetical next-generation system (*see* Table 5.7). Based on the characteristics of this system, the predicted performance of next-generation applications whose memory characteristics are represented by one of the seven target applications is shown in Figure 5.8. This figure examines the potential benefit of this approach as a function of memory reliability. Memory reliability is measured in terms of the failures in time (FIT) per DRAM device: the number of expected failures per billion hours of device operation. The strawman system is comprised of nodes whose MTBF is approximately 100 years when memory failures are excluded. This hypothetical system would have an MTBF of approximately 7 hours. Figure 5.8

examines a range of potential values for memory reliability, from very reliable (one or two memory failures in the system per day) to very unreliable (several memory failures in the system per hour). This range of memory reliability is consistent with existing projections of memory performance on future systems [50, 112, 155].

As this figure shows, exploiting similarity to correct memory errors is effective for some but not all memory-use profiles. Applications whose memory-use patterns resemble those of SAMRAI are unlikely to see much benefit from this approach. However, applications that use memory like HPCCG and LULESH do can potentially see substantial increases in application execution speed. If future memory devices fall on the unreliable end of this range, as some predict for future leadership-class systems, then applications with memory-use patterns that are similar to LAMMPS-lj may also see significant gains.

5.5.2 Checkpoint Compression

The Similarity Engine also enables efficient compression of system-level checkpoints by excluding redundant information. The contents of zero pages can be easily reconstructed when the checkpoint is restored. Similarly, only one copy of every set of duplicate pages needs to be included in the checkpoint. Appendix B proves that if the smallest difference is retained for each similar page then the checkpoint needs to include no more than half of the similar pages. At the end of a checkpoint interval, the Similarity Engine would execute the same code that is executed at the end of a protection interval to identify similarity.

Checkpoint compression does not always reduce the checkpoint commit time. The viability of compressing checkpoints depends on the: (i) compression ratio, (2) compression/decompression speed, and (3) checkpoint commit bandwidth. If the commit bandwidth is high enough, the benefit of writing a smaller checkpoint is outweighed

by the time required for compression. The compression ratio and the effective compression speed are computed based on the results presented Section 5.4. Figure 5.9 shows the compression performance metrics for this approach. Figure 5.9(a) shows the effective compression rate of the technique for each of the seven workloads. For all of these workloads, the effective compression rate exceeds 80 MiB/s. To put these results in perspective, I compare against the performance of standard compression algorithms presented by Ibtesham et al. [88]. Due to variations in hardware resources, workload selection, and workload configuration, a direct comparison may not be meaningful. However, their results provide context for the results presented in this section. Ibtesham and his co-authors observed compression rates that were, with one exception, less than 70 MiB/s. In subsequent experiments, they demonstrated compression rates above 150 Mib/s for LAMMPS, CTH, and MiniMD [89].

Figure 5.9(b) shows the decompression rate for each of the workloads. These results show decompression rates in excess of 5 GiB/s. In comparison, Ibtesham et al. saw decompression rates below 600 MiB/s [88, 89]. While the similarity-based technique exhibits very high compression and decompression rates, its compression factors tend to be comparatively modest. In this context, I borrow the compression factor from Ibtesham et al.: the *compression factor* is extent to which the size of the checkpoint is reduced. For example, a compression factor of 10% means that the compressed checkpoint is 10% smaller than the original. Figure 5.9(c) shows that the compression factors achieved by this technique are below 70% for all of the target applications except CTH-st. For the two LAMMPS problems, the compression factor is less than 10%. In contrast, Ibtesham and his coauthors were able to achieve compression factors in excess of 70% for all but one of their applications by using off-the-shelf compression libraries.

Given these individual metrics, I use an existing model to calculate the break-even commit bandwidth: the maximum aggregate commit bandwidth where using

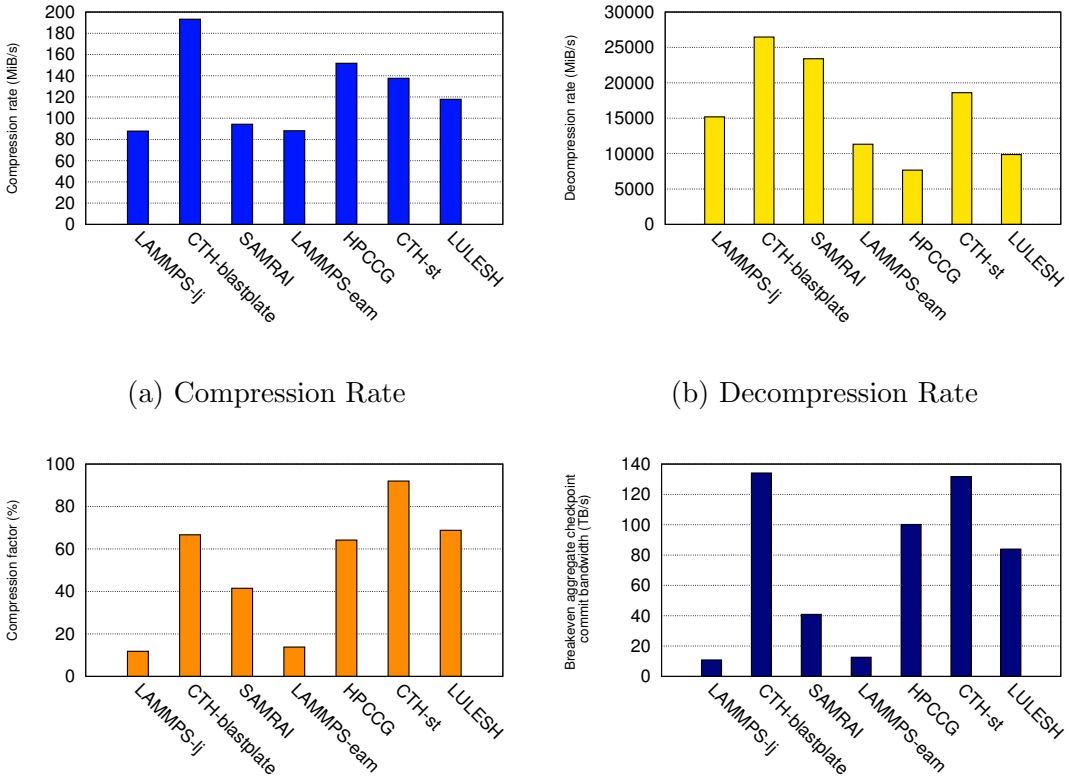
compression reduces the checkpoint commit time [88]. If the sustained bandwidth to persistent storage exceeds this breakeven point, then it is faster to store uncompressed checkpoints. If the sustained bandwidth is below this breakeven point, then it is faster to compress before writing them to persistent storage. The mathematical expression of this model is reproduced in Figure 5.1.

$$\frac{2\alpha \times r_{comp} \times r_{decomp}}{r_{comp} + r_{decomp}} = r_{breakeven} \tag{5.1}$$

where α is the *compression factor*, the percentage reduction in checkpoint volume due to compression, r_{comp} is *compression speed*, the rate of data compression, r_{decomp} is *decompression speed*, and $r_{breakeven}$ is the breakeven point, the value of the sustained bandwidth to persistent storage at which the speed of compressing checkpoints is exactly equal to storing uncompressed checkpoints.

The results of using this model are shown in Figure 5.9(d). This figure assumes a system comprised of 524,288 processes. By comparison, Ibtesham et al. observed system breakeven bandwidths equivalent to between 25 and 115 TiB/s [88, 89]. For current systems, aggregate checkpoint commit bandwidths of hundreds petabytes per second or a few terabytes per second are common. The K computer, the fourth fastest machine on the November 2014 Top500 list [4], has an aggregate file system bandwidth of 340 GB/s [141]. Mira, a BlueGene/Q machine at Argonne National Laboratory and the fifth fastest machine on the November 2014 Top500 list [4], has has an aggregate file system bandwidth of 240 GB/s [13]. Trinity, a next-generation machine that is being installed at Los Alamos National Lab is projected to have an aggregate bandwidth to the parallel file system of 1.45 TB/s [114]. Figure 5.9(d) shows that until aggregate commit bandwidths approach 10 terabytes per second, using similarity to compress checkpoints will reduce checkpoint commit time. Using this approach with applications whose memory usage patterns are similar to LULESH, SAMRAI or HPCCG is likely to yield the largest benefits. As checkpoint

commit bandwidths increase, the benefits of applications that behave like LAMMPS will quickly erode.



(a) Compression Rate

(b) Decompression Rate

(c) Compression Factor

(d) Compression Breakeven Parallel Filesystem Bandwidth

Figure 5.9: Checkpoint compression metrics. The performance characteristics of using similarity-based compression on system-level checkpoints.

5.5.3 Silent Data Corruption

Similarity in application memory can also be used to detect some errors that would be otherwise undetected. At the end of each protection interval, the Similarity Engine knows the set of pages for which there were no write accesses during the protection

interval. By recomputing the hash value of each of these un-accessed pages, the Similarity Engine can determine, with high probability, whether the contents of the page have changed. Similarly, it can also use the set of encoded differences to determine whether the difference between two pages has changed. If the contents of a page have changed without a write-access, then some form of silent data corruption has occurred.

Application	Runtime Overhead	Difference from Baseline
CTH-blastplate	0.81%	+0.20%
CTH-st	1.99%	-0.75%
HPCCG	2.48%	+1.20%
LAMMPS-eam	15.89%	+1.12%
LAMMPS-lj	14.76%	+1.31%
LULESH	9.45%	-0.41%
SAMRAI	1.82%	-0.02%

Table 5.8: Runtime overheads of silent data corruption detection. These data were collected using xor+lz4, a 3072-byte difference threshold, and a 60 second protection interval

The ability to detect some occurrences of SDC reduces the risk of the application producing an incorrect result. Although it is difficult to quantify the tradeoffs between increased execution time and decreased vulnerability to SDC, I examine the runtime overhead of exploiting similarity for this purpose in Table 5.8. Comparing these results to the values in Table 5.5 demonstrates that the additional cost of exploiting similarity to detect SDC is very small. For all of the workloads considered here, the runtime with the addition of page validation is within 1.5% of the runtime obtained without this feature enabled. This is due in large part to the fact that decompressing encoded differences with xor+lz4 is extremely fast (*cf.* Figure 5.1).

5.6 Chapter Summary

This chapter introduced a memory similarity service, Similarity Engine, and demonstrated that it can be used to identify significant similarity in application memory: 95% in CTH-st, 94% in HPCCG, 79% in LULESH, and 70% in CTH-blastplate. The Similarity Engine is able to extract more modest similarity in the LAMMPS problems, LAMMPS-eam and LAMMPS-lj. However, there are applications for which the Similarity Engine is unable to identify significant similarity using this approach (e.g., SAMRAI). I also showed how the similarity that the Similarity Engine identified can be used to improve performance by increasing application resilience to memory errors. Specifically, for extreme-scale systems where memory failures are projected to occur frequently the Similarity Engine can exploit similarity to:

- increase application performance by more than double for HPCCG and CTH-st, by nearly 80% for LULESH and CTH-blastplate, and by 10% for LAMMPS-lj and LAMMPS-eam;
- reduce checkpoint commit times by compressing checkpoints; and
- efficiently detect silent data corruption over a significant fraction of memory for LAMMPS-lj, LAMMPS-eam, CTH-st, CTH-blastplate, LULESH, and HPCCG

The Similarity Engine imposes low overhead due to very efficient methods for identifying pairs of potentially similar pages and for computing differences. The benefits of this approach could be significantly improved with the development of more efficient difference algorithms. The algorithm that the Similarity Engine uses, xor+lz4, is very fast but it identifies much less similarity than the other differencing algorithms. Similarly, although the simple neighbor heuristic is surprisingly effective, a more

sophisticated heuristic for identifying pairs of potentially similar pages would also improve the impact of this technique.

Chapter 6

Characterizing Memory Content Similarity in Kernel Memory

6.1 Introduction

The kernel comprises some of the most important software in an HPC system. It mediates access to hardware resources and ensures that processes are properly isolated from the misbehavior of their peers. A single kernel instance may be responsible for managing the execution of many application processes running on several physical cores. Because multiple processes may be dependent on the services provided by a single instance of the kernel, the consequences of a failure in the kernel are frequently more severe than failures that only affect a single application process. When a failure occurs in kernel memory, modern kernels will, with a few exceptions, simply panic [98]. All processes running within the affected operating system are killed as a consequence.

Currently, most current HPC operating systems provide no memory protection beyond that provided by the hardware (e.g., error-correcting codes (ECC)). However,

given the frequency of accesses to kernel memory, there is evidence that errors in these regions of memory are more common than errors in other regions of memory [87]. Using the technique described in Chapter 5 for exploiting memory content similarities to protect against memory faults may also be a viable approach for protecting kernel memory.

In this chapter, I use offline analysis to examine the potential benefits of using memory content similarity to protect against faults in the kernel memory of two HPC operating systems. I also examine the costs of detecting similarity and of maintaining the associated metadata.

6.2 Proposed Approach

In this chapter, I propose to use the memory content similarity techniques described in Chapter 5 to allow the kernel to recover from uncorrectable DRAM ECC errors that would otherwise lead to kernel panic and node failure. In this chapter, pages of kernel memory are divided into the four page categories defined in Chapter 5.2.1: duplicate, zero, similar and unique. Similar pages are identified by computing differences between pages using the `cx_bsdiff` [160] differencing algorithm.[1] The differences computed by `cx_bsdiff` are asymmetric. As a result, the relationship between similar pages may also be asymmetric (i.e., the fact that page A is similar to page B does not guarantee that the reverse is true). Finally, the difference threshold is set to 1 KiB.

When an uncorrectable memory fault occurs, knowledge of the similarities within kernel memory can potentially allow for the fault to be corrected. As discussed in Chapter 5.5.1, the contents of a similar (or duplicate) page can be used to recon-

[1] `cx_bsdiff` is a Python implementation of *bsdiff* [39].

struct the contents of the damaged page.

6.3 Evaluation

To evaluate the viability of this approach, this section considers the memory of two important HPC operating systems running six HPC workloads. The two operating systems examined were: (i) Linux 2.6.37 (a full-weight kernel) and; (ii) Kitten (a lightweight kernel) [145]. Although lightweight kernels have been shown to have superior performance characteristics [137], full-weight kernels dominate today's largest machines because of their generality, familiarity, and programmability [5, 50, 125]. The six workloads are briefly described in Table 6.1. They include a production workload from the U.S. Department of Energy (DOE) and Marquee Performance Codes used to evaluate candidate systems in the acquisition of the Sequoia supercomputer at Lawrence Livermore National Laboratory.

6.3.1 Running Workloads on Kitten

Kitten is a lightweight kernel designed for HPC systems. As such, its underlying design objective is to provide a familiar Linux-compatible interface where it was possible to do so without compromising scalability [103]. One of the common Linux features that is not included in Kitten is a full-featured file system. Instead, it provides storage in the form of a simple key-value store, where the key is the absolute path of the file and the value is the contents of the file. Kitten also does not support the specification of an initial file system image (e.g., initramfs).

Many of the workloads evaluated in this chapter expect to read input data from a file. As a result, I modified Kitten to create the necessary input files before launching the application code. Specifically, for each the target workloads, I embedded the

required input files in the ELF file containing the application's executable. As part of its SMARTMAP functionality [30], Kitten includes a loader that launches the application from an ELF file. I modified this loader to extract the input files from the ELF file and write their contents to the simple Kitten file system before loading and executing the application itself.

ASC Sequoia Marquee Performance Codes [107]	AMG	A parallel algebraic multigrid solver for linear systems arising from problems on unstructured grids [78].
	IRS	Implicit Radiation Solver. Solves the radiation transport equation by the flux-limited diffusion approximation using an implicit matrix solution [105].
DOE Production Application	LAMMPS	Large-scale Atomic/Molecular Massively Parallel Simulator. A classical molecular dynamics simulator [133, 143].
Mantevo Mini-Applications [80, 144]	HPCCG	Designed to mimic the finite element generation, assembly and solution for an unstructured grid problem.
	phdmesh	Parallel Heterogeneous Dynamic Mesh. An application designed to mimic the contact search applications in an explicit finite element application.
Miscellaneous Application	SAMRAI	Structured Adaptive Mesh Refinement Application Infrastructure. Designed to enable the application of structured adaptive mesh refinement to large-scale multi-physics problems [106].

Table 6.1: Summary of HPC applications used to evaluate similarity in the contents of the kernel memory of Linux and Kitten.

6.3.2 Identifying Kernel Memory

Memory snapshots of kernel memory were collected and examined offline to evaluate the prevalence of similarity. To examine the contents of kernel memory, I first needed to determine which memory was being used by the kernel. I accomplished this goal by instrumenting the two kernels under consideration to provide information that I could use to identify regions of active kernel memory.

Linux

Exhaustively tracking all of the pages used by the Linux kernel is not feasible. To capture as much of the kernel's memory as possible, I developed a kernel module that tracks all of the pages that belong to a slab allocator.[2] Although this approach does not capture all of kernel memory (and captures some memory that may not be in active use), it does capture all of the memory that the Linux kernel allocates with calls to `kmalloc`. Moreover, given the overall complexity of memory allocation in the Linux kernel, this is a relatively straightforward approach for approximating the similarity characteristics of kernel memory.

In Linux, every page of physical memory is represented by an instance of `struct page`. The `flags` field within each of these structures describes the characteristics of the associated memory page. In particular, pages managed by a slab allocator have the `PG_slab` bit set within the `flags` field. Based on this structure, I built a kernel module, `meminfo`, that allowed for the traversal of physical memory to determine which pages belonged to a slab cache.

The process of installing and initializing the `meminfo` module causes a virtual device, `/dev/meminfo`, to be created. By using that device as the target of an `ioctl`

[2]Although the slab allocator has been largely replaced by the more efficient slub allocator [40], the "slab" nomenclature still predominates in the literature.

command, I can direct the kernel module to write a summary of all of the pages that currently belong to a slab cache to the system log.

During initialization, the kernel module also registers two kernel probes, a `jprobe` and a `kretprobe`, to track when memory pages are added to or removed from a slab cache. Kernel probes are provided to allow for debugging the execution of kernel functions.[3] A `jprobe` allows the user to specify a handler that is called each time that a specified function is called. A `kretprobe` allows the user to specify a handler that is called each time that the kernel returns from a specified function. To track changes to slab cache membership, I register a `jprobe` on the return from `new_slab` and a kretrpobe on the entry to `__free_slab`. In each case, when the probe is triggered, an appropriate message is written to the system log.

Kitten

In the Kitten lightweight kernel, a region of low memory (by default, 64 MB) beginning at address 0 is reserved for kernel use. During the initial boot sequence, a very simple allocator (`bootmem`) is used to manage this memory. Near the end of the boot sequence, management of all unused kernel memory is transferred to a buddy allocator. The buddy allocator handles all of the allocation and deallocation of memory from the kernel memory region, with the exception of those pages allocated by the `bootmem` allocator during the boot sequence. The Kitten buddy allocator, like most buddy allocators, manages memory in blocks that are a power of two in size. The minimum block size is 32 bytes.

To determine which blocks of kernel memory have been allocated by the buddy allocator, I modified the buddy allocator source to track the blocks of memory that it allocates. Although this approach fails to capture the memory allocated by the

[3]Detailed information about kernel probes is contained in the `Documentation/kprobes.txt` file included with the Linux kernel source.

`bootmem` allocator, it does allow all of the memory allocated after the kernel memory subsystem has been initialized to be identified. I also created a new system call, `get_meminfo`. When this new system call is invoked, it writes a list of all of the memory blocks that were ever allocated by the buddy allocator to standard out. I modified each of the target workloads to invoke this new system call at the beginning and end of their execution.

6.3.3 Experimental Methodology

To minimize the perturbation of our experimental framework on the operation of the kernel, I used the checkpointing functionality of the Palacios Virtual Machine Monitor (VMM) [103] to collect snapshots of the memory of a virtual machine running each of the six workloads on both of our operating systems. I created two guest machines: one that runs Linux 2.6.37 and another that runs Kitten 1.3.0. For simplicity, all of our applications were run using a single MPI process.

The checkpointing facility of Palacios allowed me to periodically capture snapshots of the guest machine's memory (once every 60 seconds for the data presented in this chapter). As described above, these two operating systems write information about kernel memory usage to standard out (Kitten) or to the system log (Linux). The metadata provided by the modified kernels describes page frames in the guest machine's memory that are part of kernel memory. Combining this metadata with snapshots of the guest machine's memory allows the contents of kernel memory to be examined.

6.3.4 Data Analysis

I analyzed each of the collected kernel memory snapshots offline. For each snapshot, I walked through the address space from low addresses to high, categorizing each page of memory into one of the four categories described earlier: (a) duplicate; (b) similar; (c) zero; or (d) unique. As described in Chapter 5, duplicate pages are identified by computing the MD5 sum of each page. I assume that two pages with the same hash are duplicate.

For this offline inquiry, I use a different approach for identifying similar pages than what was used in Chapter 5. The approach used in this section was inspired by the Difference Engine [76]. Instead of computing patches between every pair of pages, I attempt to identify a tractably small set of pages for each candidate page that are likely to be similar to it.

For each page, I collect four 128-byte blocks of memory. These blocks are treated as a *signature* of the page contents. These signatures are evenly distributed within each 4 KiB page of memory (i.e., at offsets of 0, 1024, 2048 and 3072 bytes).

As each candidate page in the address space of an application is examined, pages that match one or more of the candidate page's signatures are identified. In the event that more than one page matches a single signature, I choose the page nearest to the candidate page. This approach identifies up to four pages that may be similar to the current candidate page. In addition to these pages, I also consider the page of kernel memory that occupies the next smallest page frame number. In all, this approach identifies as many as five pages that are likely to be similar to the candidate page.

I then compute a patch between the current candidate page and each member of the set of likely similar pages. If any patch is smaller than a threshold, in this case 1024 bytes, the current candidate page is marked as similar. Because `cx_bsdiff` does not generate symmetric patches, observing a single patch that falls below the

threshold is sufficient to categorize only a single page as similar. Therefore, I also compute the reciprocal patch of each of the pages in the set of likely similar pages to determine whether any of them should also be marked as similar.

This is a heuristic approach (*cf.* [76]) based on the idea that the contents of a page of memory can be meaningfully summarized as a set of signatures. Although there may be methods that would yield greater numbers of similar pages by generating smaller differences, the fraction of similar pages identified by this approach is a lower bound on the total number of similar pages in kernel memory.

6.4 Similarity in Kernel Memory

This section examines the potential costs and benefits of exploiting similarity in kernel memory. Although the case for resilient operating systems is still emerging [55], these results suggest that the proposed approach has promise.

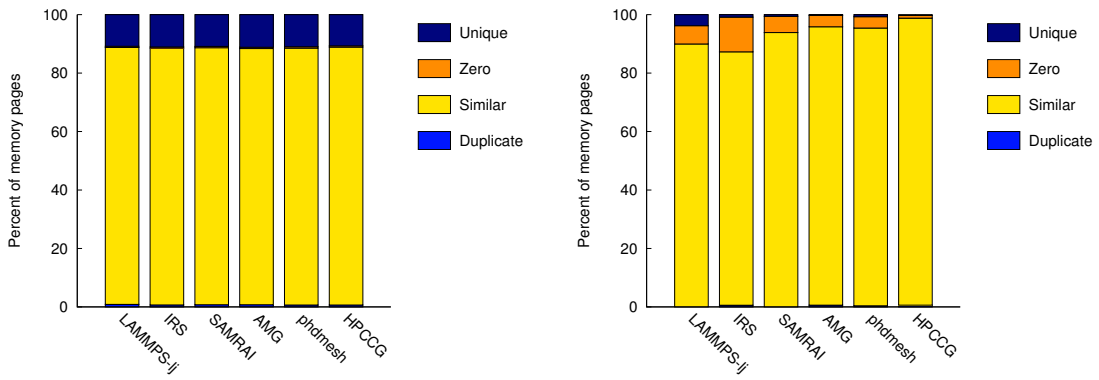
6.4.1 Similarity Overview

Figures 6.1(a) and 6.1(b) show the composition of kernel memory for Linux (a heavyweight OS) and Kitten (a lightweight OS). The data for each operating system represent the mean fraction of the kernel memory pages in each of four categories. Tables 6.2(a) and 6.2(b) provide detailed statistics about the number of pages in each category. These data were collected over ten trials of each combination of two operating systems and six workloads. For Kitten, all of the pages of kernel memory whose contents are ever managed by the buddy allocator are considered. For Linux, all of the kernel memory that managed by a slab allocator at any point during the application's execution are considered.

Figures 6.1(b) and 6.1(a) show that both Linux and Kitten have a very large number of similar pages. Also, the kernel memory of both operating systems contains very few duplicate pages. This result is consistent with how the OS uses this memory; the majority of the state maintained by these OSs is comprised of table-based structures containing objects such as page table mappings. Given the nature of page tables, I would expect to find large numbers of similar pages in memory allocated for page table data structures. For x86 processors, each element in the page table hierarchy occupies a full 4 KiB page of memory even if only a handful of pages are mapped in the referenced region of virtual memory. As a result, page table data structures tend to be very sparse: they contain large numbers of null entries. The presence of few non-null entries in these structures allows the differences between pages containing them to be compactly represented.

To empirically validate these observations, I instrumented Kitten's buddy allocator to track the percentage of buddy-allocated memory that is used to store page table data structures. For each of the six applications considered in this chapter, the minimum observed percentage of memory that is allocated by the buddy allocator for page tables is shown in Table 6.2. These data show that page table data structures occupy the vast majority of buddy-allocated memory and thus must also be a significant source of the similarity observed in Figure 6.1(b).

The page categorization for kernel memory in Linux exhibits much less variation than kernel memory in Kitten. As shown in Table 6.2(a), there is little variation between trials or across applications in the page categorization of Linux kernel memory. In contrast, larger variations are visible in the composition of Kitten kernel memory both between trials and across applications. While both kernels exhibit significant similarity, it is likely that both the costs and the benefits in Kitten will exhibit larger variation than in Linux.



(a) Linux (b) Kitten

Figure 6.1: *Mean page categorization of kernel memory.* This figure shows the mean page categorization of kernel memory for each operating system using a 1024 byte patch threshold. Each bar represents the mean fraction of memory pages in each of the four page categories observed over a sequence of 10 trials. As discussed in Section 6.3.2, memory snapshots were collected every 60 seconds of application execution time, for a total of 5-6 snapshots per workload.

Application	Page Category	Mean ($\overline{x}$)	Standard Deviation (s)	99% Confidence Interval
LAMMPS-lj	DUPLICATE	26.60	2.75	(25.65 - 27.55)
	ZERO	13.10	0.30	(13.00 - 13.20)
	SIMILAR	2706.50	12.18	(2702.32 - 2710.68)
	UNIQUE	333.10	11.39	(329.19 - 337.01)
	TOTAL	3079.30	3.61	(3078.06 - 3080.54)
IRS	DUPLICATE	20.68	3.63	(19.44 - 21.93)
	ZERO	13.57	1.03	(13.21 - 13.92)
	SIMILAR	2695.30	16.31	(2689.69 - 2700.91)
	UNIQUE	338.75	16.24	(333.17 - 344.33)
	TOTAL	3068.30	3.25	(3067.18 - 3069.42)
SAMRAI	DUPLICATE	23.60	3.16	(22.51 - 24.69)
	ZERO	14.18	1.40	(13.70 - 14.66)
	SIMILAR	2698.93	18.05	(2692.73 - 2705.14)
	UNIQUE	333.38	18.02	(327.19 - 339.58)
	TOTAL	3070.10	4.41	(3068.59 - 3071.61)
AMG	DUPLICATE	23.20	3.09	(22.14 - 24.26)
	ZERO	13.58	1.01	(13.24 - 13.93)
	SIMILAR	2690.13	20.97	(2682.93 - 2697.34)
	UNIQUE	342.38	22.24	(334.74 - 350.02)
	TOTAL	3069.30	2.55	(3068.42 - 3070.18)
phdmesh	DUPLICATE	20.30	2.26	(19.52 - 21.08)
	ZERO	13.85	1.16	(13.45 - 14.25)
	SIMILAR	2696.27	16.94	(2690.45 - 2702.09)
	UNIQUE	336.78	17.79	(330.67 - 342.90)
	TOTAL	3067.20	3.28	(3066.07 - 3068.33)
HPCCG	DUPLICATE	19.70	3.77	(18.40 - 21.00)
	ZERO	14.00	1.10	(13.62 - 14.38)
	SIMILAR	2700.02	16.40	(2694.38 - 2705.65)
	UNIQUE	326.18	14.38	(321.24 - 331.12)
	TOTAL	3059.90	1.65	(3059.33 - 3060.47)

(a) Linux

Application	Page Category	Mean ($\overline{x}$)	Standard Deviation (s)	99% Confidence Interval
LAMMPS-lj	DUPLICATE	0.00	0.00	(0.00 - 0.00)
	ZERO	63.45	6.81	(61.11 - 65.79)
	SIMILAR	908.50	145.64	(858.45 - 958.55)
	UNIQUE	38.05	152.45	(0.00 - 90.44)
	TOTAL	1010.00	0.00	(1010.00 - 1010.00)
IRS	DUPLICATE	6.17	33.41	(0.00 - 17.65)
	ZERO	127.13	26.34	(118.08 - 136.18)
	SIMILAR	929.35	97.99	(895.68 - 963.02)
	UNIQUE	9.35	39.08	(0.00 - 22.78)
	TOTAL	1072.00	0.00	(1072.00 - 1072.00)
SAMRAI	DUPLICATE	0.00	0.00	(0.00 - 0.00)
	ZERO	56.82	1.98	(56.14 - 57.50)
	SIMILAR	963.25	1.75	(962.65 - 963.85)
	UNIQUE	5.93	0.36	(5.81 - 6.06)
	TOTAL	1026.00	0.00	(1026.00 - 1026.00)
AMG	DUPLICATE	5.87	45.44	(0.00 - 21.48)
	ZERO	37.90	45.70	(22.20 - 53.60)
	SIMILAR	927.67	98.69	(893.76 - 961.58)
	UNIQUE	2.57	7.56	(0.00 - 5.16)
	TOTAL	974.00	0.00	(974.00 - 974.00)
phdmesh	DUPLICATE	3.83	29.69	(0.00 - 14.04)
	ZERO	37.88	30.08	(27.55 - 48.22)
	SIMILAR	926.53	96.57	(893.35 - 959.72)
	UNIQUE	6.75	36.79	(0.00 - 19.39)
	TOTAL	975.00	0.00	(975.00 - 975.00)
HPCCG	DUPLICATE	5.52	42.73	(0.00 - 20.20)
	ZERO	8.55	42.99	(0.00 - 23.32)
	SIMILAR	924.18	98.36	(890.38 - 957.98)
	UNIQUE	2.75	12.64	(0.00 - 7.09)
	TOTAL	941.00	0.00	(941.00 - 941.00)

(b) Kitten

Figure 6.2: *Detailed page categorization statistics.* For each combination of six workloads and two operating systems, this table provides statistics on the number of similar, duplicate, zero and unique pages observed in kernel memory. These statistics were generated from data collected over ten trials of each application/operating system pair.

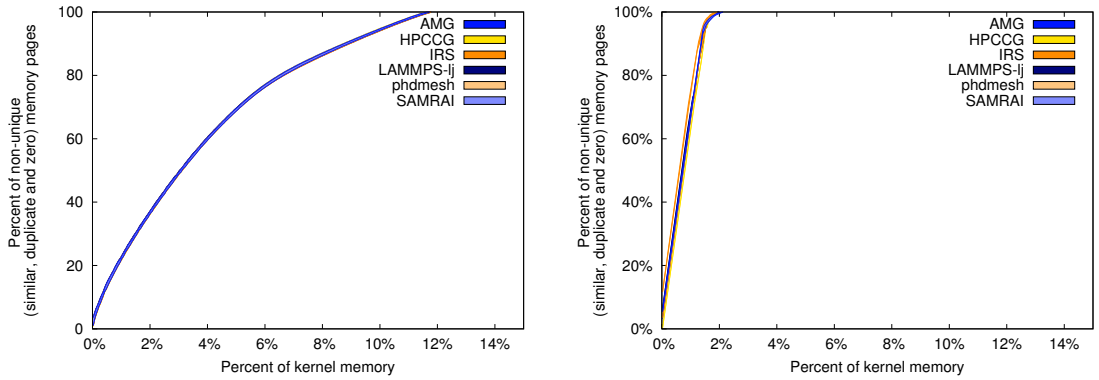
Application	Minimum Percentage of Buddy-Allocated Memory Used for Page Tables
AMG2006	93.7%
IRS	85.3%
LAMMPS-lj	90.8%
SAMRAI	90.9%
HPCCG	97.0%
phdmesh	93.7%

Table 6.2: *Fraction of Kitten kernel memory used to store page tables.* This table shows the minimum percentage of memory allocated by Kitten's buddy allocator that is used for page table data structures over the ten trials of each of six workloads.

Patch Size Threshold

Figure 6.3 facilitates an examination of the tradeoff between the size of the difference and memory overhead in these operating systems. It shows the fraction of similar and duplicate pages as a function of metadata size. The data in this figure were collected over ten trials of each combination of the two kernels and six applications considered in this chapter. For each application, this figure shows a very narrow shaded a region that contains all of the observations collected over the series of trials. Consistent with observations in the preceding section, there is more variation across applications in the Kitten data. However, taken as a whole, the tradeoff between memory to store differences and the number of similar pages is consistent from trial to trial and largely independent of the application.

The slope of these curves represents the ratio of cost to benefit. For Kitten, increasing the difference threshold results in a dramatic increase in the fraction of similar pages yet it requires only a very small increase in the size of the metadata. For Linux, increasing the patch size comes at a greater (but still modest) cost. For both OSs, only a modest amount of metadata (less than 12%) is required to protect all of kernel memory using the proposed approach.



(a) Linux (b) Kitten

Figure 6.3: *Non-unique memory pages as a function of metadata volume.* This figures shows the fraction of non-unique (i.e., duplicate, zero and similar) pages as a function of metadata size for Kitten and Linux running six workloads. Ten trials were conducted for each application and six memory snapshots were collected during each run. Due to variations in the timing of the first snapshot, only the results from the last five snapshots for each trial are included here. The shaded region for each application shows the range of observed results. For Linux, there is very little variation between applications and the last sequence rendered is the only one that is visible.

Modification Behavior

The cost of maintaining the metadata necessary to correct memory errors will depend, in part, on the rate at which similar and duplicate pages are modified. To examine the frequency of kernel memory modification, Table 6.3 compares the contents of kernel memory pages across the sequence of collected snapshots. The data in this table are aggregated over the series of trials that were performed. As these data show, a significant majority of similar and duplicate pages are not modified during the application's execution. These results are consistent with how these operating systems use memory; they construct tables that are written once and read many times. The infrequent modification of similar and duplicate pages in kernel memory

Application	Similar/Duplicate Pages	Changed 1+ Times	Changed 1 Time	Changed 2 Times	Changed 3 Times	Changed 4+ Times
AMG2006	27211	351 (1.29%)	240 (0.88%)	10 (0.04%)	6 (0.02%)	95 (0.35%)
IRS	27252	411 (1.51%)	196 (0.72%)	11 (0.04%)	0 (0.00%)	204 (0.75%)
LAMMPS-lj	27395	359 (1.31%)	259 (0.95%)	28 (0.10%)	0 (0.00%)	72 (0.26%)
SAMRAI	27288	488 (1.79%)	274 (1.00%)	19 (0.07%)	10 (0.04%)	185 (0.68%)
HPCCG	27288	415 (1.52%)	233 (0.85%)	17 (0.06%)	1 (0.00%)	164 (0.60%)
phdmesh	27257	441 (1.62%)	253 (0.93%)	15 (0.06%)	0 (0.00%)	173 (0.63%)

(a) Linux

Application	Similar/Duplicate Pages	Changed 1+ Times	Changed 1 Time	Changed 2 Times	Changed 3 Times	Changed 4+ Times
AMG2006	9730	20 (0.21%)	10 (0.10%)	0 (0.00%)	0 (0.00%)	10 (0.10%)
IRS	10688	138 (1.29%)	96 (0.90%)	4 (0.04%)	0 (0.00%)	38 (0.36%)
LAMMPS-lj	10072	29 (0.29%)	20 (0.20%)	0 (0.00%)	0 (0.00%)	9 (0.09%)
SAMRAI	10204	152 (1.49%)	81 (0.79%)	27 (0.26%)	32 (0.31%)	12 (0.12%)
HPCCG	9401	10 (0.11%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	10 (0.11%)
phdmesh	9731	1 (0.01%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	1 (0.01%)

(b) Kitten

Table 6.3: *Modification behavior of similar and duplicate kernel memory pages.* This table shows the frequency with which pages in kernel memory that are ever categorized as similar or duplicate. Ten trials were conducted for each application/operating system pair. Six memory snapshots were collected during each run. Due to variations in the timing of the first snapshot, only the results from the last five snapshots for each trial are included here. For each application, these data represent the total number of pages captured over all ten trials. These results demonstrate that similar and duplicate pages in kernel memory change very little over the lifetime of an application.

suggests that the cost of metadata maintenance will be low.

6.5 Chapter Summary

This chapter examined the feasibility of exploiting memory content similarity in kernel memory. The results in this chapter point to the potential of this novel technique to efficiently protect against uncorrectable memory errors in kernel memory. For both Linux and Kitten, significant similarity exists in regions of kernel memory. Additionally, similar and duplicate pages in kernel memory are infrequently modified

and can be protected with small volumes of metadata.

Chapter 7

Conclusion and Future Work

The objective of the work presented in this dissertation was to demonstrate that rollback avoidance techniques can be used effectively to mitigate the performance impact of failures on next-generation extreme-scale systems. In this chapter, I summarize my contributions and explore potential next steps.

7.1 Summary

In this dissertation, I examined the benefits and costs of using rollback avoidance to augment checkpoint/restart-based fault tolerance mechanisms. Using a combination of numerical models, simulation, and implementation, I demonstrated that rollback avoidance techniques have the potential to address the deleterious performance impact of the increase in failure frequency that is projected for next-generation systems. We discuss each of the major contributions of the work presented in this dissertation below.

1. **An Analytic Model of Rollback Avoidance and Coordinated Checkpoint/Restart**

   In Chapter 3, I developed and validated an analytic model of the impact of rollback avoidance on application performance. I used this model to demonstrate that when coordinated checkpoint/restart is used on systems that experience frequent failure, rollback avoidance can yield significant performance benefits. I also showed that checkpoint/restart will likely still be an important part of fault tolerance; it is unlikely that rollback avoidance by itself will be sufficient on next-generation systems. This model also allowed us to demonstrate that exploiting memory content similarity as described in Chapter 5 has the potential to yield significant improvements in application performance on next-generation systems.

2. **Simulation of Rollback Avoidance and Coordinated Checkpoint/Restart**

   In Chapter 4, I presented and validated a simulation framework for simulating the performance impact of fault tolerance activities on next-generation systems. Using this framework, I showed the limitations of rollback avoidance. For some applications, namely those that are insensitive to noise and noise-like interruptions, I demonstrated that rollback avoidance may only modestly improve application performance when asynchronous checkpoint/restart and message logging are employed.

3. **Exploiting Memory Content Similarity to Protect Against Faults in Application Memory**

   In Chapter 5, I presented and evaluated a software library for extracting memory content similarity: the *Similarity Engine*. Using this library, I characterized the prevalence of similarities in application memory for several important applications, proxies, and mini-applications. We then examined and evaluated three

techniques for exploiting this information to reduce the performance impact of rollbacks caused by faults in application memory.

4. **Examining the Viability of Using Memory Content Similarity to Protect Against Faults in Kernel Memory**
   Finally, in Chapter 6, I examined the feasibility of using my memory content similarity technique to protect kernel memory against memory faults. I demonstrated that in Linux and Kitten kernel memory contains significant similarity. Moreover, I showed that the overhead of this approach is likely to be modest because kernel memory changes infrequently and it can be protected by a small volume of metadata.

7.2 Future Work

In this dissertation, I have thoroughly examined the potential costs and benefits of using rollback avoidance on next-generation systems. However, a number potential research inquiries remain.

My model of rollback avoidance in Chapter 3 is an extension of Daly's model of application execution. As a result, it is limited to modeling exponentially-distributed failures and coordinated checkpoint/restart. While overcoming these limitations pose significant challenges, extending the model to account for other failure distributions and to account for the performance impact of uncoordinated checkpoint/restart would increase the power of this model.

The simulation framework introduced in Chapter 4 is a powerful tool. However, like my model, it also only supports exponentially distributed failures. Extending it to simulate other failure distributions would allow us to examine the impact of the failures distribution on rollback avoidance. From a software engineering perspective,

there are a number of enhancements that would increase the power of the simulator as a research tool. These include parallelization and adding support for MPI subcommunicators. These features would allow us to simulate a wider variety of applications running on larger systems.

In Chapter 5, I demonstrated that the memory of several applications contain significant similarity. Understanding how similarity arises in application memory might allow us to structure the application in a way that increases this similarity. One of the biggest costs of my approach is the cost of computing differences. In Chapter 5.3.3, I demonstrated that the neighbor heuristic is efficient and reasonably effective. However, for a given difference threshold, it identifies only a fraction of the total similar pages. Identifying more effective methods for identifying pairs of likely similar pages would improve the overall impact of our approach.

Finally, in Chapter 6 I examined the feasibility of exploiting content similarity in pages of kernel memory. The next step is to implement a runtime system in the kernel that can identify and exploit similarities.

Appendices

Appendix A

Derivation of Rollback Avoidance Models

A.1 Modeling the Probability of Rollback Avoidance

This section examines expected time between errors that result in the application rolling back to an earlier checkpoint. X_i is an indicator random variable to indicate when rollback due to an error is avoided.

$$X_i = \begin{cases} 1 & \text{if } i \text{ or more consecutive failures can be avoided} \\ 0 & \text{otherwise} \end{cases}$$

Using X_i, the interarrival time of unavoidable rollbacks can be captured in the following way.

Y_i = the interarrival time of the i^{th} failure

p = probability of avoiding an error

Y = the interarrival time of the next unavoidable rollback

Therefore:

$$
\begin{aligned}
Y &= \sum_{i=0}^{\infty} X_i Y_i \\
E(Y) &= \sum_{i=0}^{\infty} E(X_i Y_i) \\
&= \sum_{i=0}^{\infty} E(X_i) E(Y_i) \\
&= M \sum_{i=0}^{\infty} E(X_i) \\
&= M \sum_{i=0}^{\infty} p^i \\
&= \frac{M}{1-p}
\end{aligned}
$$

Because rollback avoidance and interarrival times are independent in this model, the expectation of the product is equal to the product of the expectation. Finally, the expected value of Y (i.e., $E(Y)$) is the effective MTBF after the effect of rollback avoidance is accounted for. Further, because the interarrival times of errors are exponentially distributed and the number of consecutive errors for which rollback is avoided rollback is geometrically distributed, the resulting times between unavoidable rollbacks are also exponentially distributed [156, p. 320].

A.2 Modeling Rollback Avoidance + Coordinated Checkpoint/Restart

A.2.1 Showing the Limit with Optimal Checkpoint Interval

When $p_a = 1.0$ this model is undefined. However, this subsection shows that the model converges to the correct result as $p_a \to 1.0$. The limit of $T_w(p_a, o_a)$ depends on how the checkpointing interval is determined. The analysis in this section considers two cases: (i) Daly's optimal checkpoint interval [42], and (ii) a fixed interval. The limits are different in these two cases because the optimal checkpoint interval depends on p_a; as p_a increases so does the optimal checkpoint interval. The analysis begins by considering the optimal checkpoint interval. In this case, the result is that:

$$\lim_{p_a \to 1.0} T_w(p_a, o_a) = T_s' \tag{A.1}$$

In other words, if all rollbacks could be avoided the total work time would be equal to the application's solve time, including the overhead of avoiding rollbacks. Considering the impact of rollback avoidance, the value of the optimal checkpoint interval (τ'_{opt}) when $\delta < 2M$ is:

$$\tau'_{opt} = \sqrt{\frac{2\delta M}{(1-p_a)}}\left[1 + \frac{1}{3}\left(\frac{\delta(1-p_a)}{2M}\right)^{1/2} + \frac{1}{9}\left(\frac{\delta(1-p_a)}{2M}\right)\right] - \delta \tag{A.2}$$

Daly also provides an optimal checkpoint interval for the case where $\delta \geq 2M$. However, as $p_a \to 1.0$, the system MTBF effectively becomes infinite. As a result, for any finite value of δ, the optimal checkpoint interval will be defined by Equation A.2 when p_a is close to 1.0. T_w can be expressed in terms of p_a by combining Equations 3.1 and 3.3.

$$T_w(p_a) = \left(\frac{M}{1-p_a}\right) e^{R(1-p_a)/M}\left(e^{(\tau'_{opt}+\delta)(1-p_a)/M} - 1\right)\frac{T_s'}{\tau'_{opt}}$$

To allow for a more compact representation let:

$$\begin{aligned}\hat{\tau} &= \tau'_{opt}(1-p_a)/M \\ &= \frac{1-p_a}{M}\left(\sqrt{\frac{2\delta M}{1-p_a}}\left[1+\ \frac{1}{3}\left(\frac{\delta(1-p_a)}{2M}\right)^{1/2}+\ \frac{1}{9}\left(\frac{\delta(1-p_a)}{2M}\right)\right]-\delta\right) \\ &= \sqrt{2\delta(1-p_a)}M^{-1/2}-\ \frac{2\delta(1-p_a)}{3}M^{-1}+\ \frac{\delta\sqrt{\delta}(1-p_a)^{3/2}}{9}M^{-3/2}\end{aligned}$$

The result is:

$$\begin{aligned}T_w(p_a) &= \frac{M}{1-p_a}e^{R(1-p_a)/M}\left(e^{(\tau'_{opt}+\delta)(1-p_a)/M}-1\right)\frac{T'_s}{\tau'_{opt}} \\ &= e^{R(1-p_a)/M}\left(e^{\hat{\tau}+\delta(1-p_a)/M}-1\right)\frac{T'_s}{\hat{\tau}} \\ &= \frac{\left(e^{\hat{\tau}+\delta(1-p_a)/M}-1\right)T'_s}{\hat{\tau}e^{-R(1-p_a)/M}}\end{aligned}$$

Take the limit of T_w as p_a approaches 1.0.

$$\lim_{p\to 1.0}T_w(p_a)=\lim_{p\to 1.0}T_w(p_a)\ \frac{\left(e^{\hat{\tau}+\delta(1-p_a)/M}-1\right)T'_s}{\hat{\tau}e^{-R(1-p_a)/M}}$$

To simplify the algebra, let $M' = M/(1-p_a)$.

$$\begin{aligned}\lim_{p\to 1.0}T_w(p_a) &= \lim_{M'\to\infty}T_w(M') \\ &= \lim_{M'\to\infty}\ \frac{\left(e^{\hat{\tau}+\delta/M'}-1\right)T'_s}{\hat{\tau}e^{-R/M'}}\end{aligned}$$

Because the limit of the numerator and the denominator are both zero, L'Hôpital's Rule can be applied. This yields:

$$\begin{aligned}\lim_{M'\to\infty}T_w(p_a) &= \lim_{M'\to\infty}\ \frac{T'_s e^{\hat{\tau}+\delta/M'}\left(\frac{d\hat{\tau}}{dM'}-\frac{\delta}{M'^2}\right)}{\frac{R}{M'^2}e^{-R/M'}\hat{\tau}+e^{-R/M'}\frac{d\hat{\tau}}{dM'}} \\ &= \lim_{M'\to\infty}\ \frac{T'_s e^{\hat{\tau}+(\delta+R)/M'}\left(\frac{d\hat{\tau}}{dM'}-\frac{\delta}{M'^2}\right)}{\frac{R}{M'^2}\hat{\tau}+\frac{d\hat{\tau}}{dM'}}\end{aligned} \tag{A.3}$$

where:

$$\frac{d\hat{\tau}}{dM'}=-\frac{\sqrt{2\delta}}{2}M'^{-3/2}+\ \frac{2\delta}{3}M'^{-2}-\ \frac{\delta\sqrt{\delta}}{6}M'^{-5/2}$$

In Equation A.3, the limit of both the numerator and denominator are zero. However, a factor of $M^{'3/2}$ can be eliminated from each. This yields:

$$\begin{aligned}
\lim_{M'\to\infty} T_w(p_a) &= \lim_{M'\to\infty} \frac{T_s' e^{(\hat{\tau}+(\delta+R)/M')}\big(-\frac{\sqrt{2\delta}}{2}M^{'-3/2} - \frac{\delta}{3}M^{'-2} - \frac{\delta\sqrt{\delta}}{6}M^{'-5/2}\big)}{\frac{R}{M'^2}\hat{\tau} + \big(-\frac{\sqrt{2\delta}}{2}M^{'-3/2} - \frac{2\delta}{3}M^{'-2} - \frac{\delta\sqrt{\delta}}{6}M^{'-5/2}\big)} \\
&= \lim_{M'\to\infty} \frac{T_s' e^{(\hat{\tau}+(\delta+R)/M')}\big(-\frac{\sqrt{2\delta}}{2} - \frac{\delta}{3}M^{'-1/2} - \frac{\delta\sqrt{\delta}}{6}M^{'-1}\big)}{RM'^{-1/2}\hat{\tau} + \big(-\frac{\sqrt{2\delta}}{2} + \frac{2\delta}{3}M^{'-1/2} - \frac{\delta\sqrt{\delta}}{6}M^{'-1}\big)} \\
&= \frac{T_s' e^{(0+0)}\big(-\frac{\sqrt{2\delta}}{2} - \frac{\delta}{3}0 - \frac{\delta\sqrt{\delta}}{6}0\big)}{R\cdot 0\cdot 0 + \big(-\frac{\sqrt{2\delta}}{2} + \frac{2\delta}{3}0 - \frac{\delta\sqrt{\delta}}{6}0\big)} \\
&= \frac{T_s'\big(-\frac{\sqrt{2\delta}}{2}\big)}{-\frac{\sqrt{2\delta}}{2}} \\
&= T_s'
\end{aligned}$$

As the probability of avoiding rollback approaches 1.0, the cost of checkpointing and failure recovery approach zero. As a result, the total work time converges to the application's native time-to-solution ($T_s^{'}$) expanded by the overhead of avoiding rollback ($1 + o_a$).

A.2.2 Showing the Limit with Fixed Checkpoint Interval

In this case, the result is that:

$$\lim_{p_a\to 1.0} T_w(p_a, o_a) = T_s' \tag{A.4}$$

$$\lim_{p_a\to 1.0} T_w(p_a) = \lim_{p_a\to 1.0} \left(\frac{M}{1-p_a}\right) e^{R(1-p_a)/M}\left(e^{(\tau+\delta)(1-p_a)/M} - 1\right)\frac{T_s^{'}}{\tau}$$

To simplify the algebra, let $M^{'} = M/(1-p_a)$.

$$\begin{aligned}
\lim_{p_a\to 1.0} T_w(p_a) &= \lim_{M'\to\infty} M^{'} e^{R/M'}\left(e^{(\tau+\delta)/M'} - 1\right)\frac{T_s^{'}}{\tau} \\
&= \frac{T_s^{'}}{\tau}\lim_{M'\to\infty} \frac{e^{(\tau+\delta)/M'} - 1}{\frac{1}{M'}e^{-R/M'}}
\end{aligned}$$

Because the limit of the numerator and the denominator are both zero L'Hôpital's Rule can be applied.

$$\begin{aligned}
\lim_{p_a \to 1.0} T_w(p_a) &= \frac{T_s'}{\tau} \lim_{M' \to \infty} \frac{-\frac{(\tau+\delta)}{M'^2} e^{(\tau+\delta)/M'}}{-\frac{1}{M'^2} e^{-R/M'} + \frac{R}{M'^3} e^{R/M'}} \\
&= \frac{T_s'}{\tau} \lim_{M' \to \infty} \frac{-(\tau+\delta) e^{(\tau+\delta)/M'}}{-e^{-R/M'} + \frac{R}{M'} e^{R/M'}} \\
&= \frac{T_s'}{\tau} \frac{-(\tau+\delta)}{-1+0} \\
&= T_s' \left(1 + \frac{\delta}{\tau}\right)
\end{aligned}$$

A.3 Modeling Rollback Avoidance Without Checkpoint/Restart

A.3.1 Extending Daly's Model

Daly began with a high-level model of the time required to execute an application [42]. In this model, he expresses the total execution time, T_w, as:

$$T_w(\tau) = \text{solve time} + \text{dump time} + \text{rework time} + \text{restart time}$$

Because no checkpoints are taken in the case of model rollback avoidance without checkpointing, two changes to this high-level model are required: (i) set the dump time to zero; and (ii) express T_w as a function of p_a and o_a. This yields:

$$T_w(p_a, o_a) = \text{solve time} + \text{rework time} + \text{restart time}$$

Given this high-level model, the solve time is T_s': the native solve time of the application plus the overhead of avoiding rollback. Daly states that the cost of restarting

and redoing lost work can be expressed as:

$$\text{rework time} + \text{restart time} = \big[E(\Delta t) + R)P(\Delta t)n(\Delta t)\big] + \big[E(\Delta t + R)(1 - P(\Delta t))n(\Delta t)\big]$$

where:

$$\begin{aligned} E(\Delta t) &= \text{the expected point of failure in an interval of size } \Delta t \\ &= M + \frac{\Delta t}{1 - e^{\Delta/M}} \\ P(\Delta t) &= \text{the probability of completing an interval of size } \Delta t \\ &\quad \text{without interrupt} \\ &= e^{-\Delta t/M} \\ n(\Delta) &= \text{the expected number of interrupts in an interval of size } \Delta t \\ &= T_w(p_a, o_a)/M \end{aligned}$$

When rollback occurs without checkpointing, the application must start over from the beginning. As a result, the application will complete its execution only when an interval of size T_s completes without requiring rollback. This yields:

$$\begin{aligned} T_w(p_a, o_a) &= \text{solve time} + \text{rework time} + \text{restart time} \\ &= T_s^{'} + \big[(E(T_s^{'}) + R)P(T_s^{'})n(T_s^{'})\big] + \\ &\quad \big[E(T_s^{'} + R)(1 - P(T_s^{'}))n(T_s^{'})\big] \\ &= \frac{M^{'}T_s^{'}}{M^{'} - (E(T_s^{'}) + R)P(T_s^{'}) - E(T_s^{'} + R)(1 - P(T_s^{'}))} \\ &= \frac{M^{'}T_s^{'}}{-(E(T_s^{'}) + R)P(T_s^{'}) - \frac{T_s^{'}+R}{1-e^{(T_s^{'}+R)/M^{'}}} + E(T_s^{'} + R)P(T_s^{'})} \\ &= \frac{M^{'}T_s^{'}e^{(T_s^{'}+R)/M^{'}}}{-\frac{T_s^{'}}{1-e^{T_s^{'}/M^{'}}} - R - \frac{(T_s^{'}+R)e^{(T_s^{'}+R)/M^{'}}}{1-e^{(T_s^{'}+R)/M^{'}}} + \frac{T_s^{'}+R}{1-e^{(T_s^{'}+R)/M^{'}}}} \\ &= \frac{M^{'}T_s^{'}e^{(T_s^{'}+R)/M^{'}}}{-\frac{T_s^{'}}{1-e^{T_s^{'}/M^{'}}} - R + T_s^{'} + R} \\ &= M^{'}e^{R/M^{'}}(e^{T_s^{'}/M^{'}} - 1) \end{aligned}$$

A.3.2 Showing the Limit

When $p_a = 1.0$ this model is undefined. However, this subsection shows that the model produces the converges to the correct result as $p_a \to 1.0$. In particular, it shows that:

$$\lim_{p_a \to 1.0} T_w(p_a) = T_s^{'}$$

Because $M^{'} = \Theta/(1 - p_a)$:

$$\begin{aligned}
\lim_{p_a \to 1.0} T_w(p_a) &= \lim_{M^{'} \to \infty} T_w(M^{'}) \\
&= \lim_{M^{'} \to \infty} M^{'} e^{R/M^{'}} (e^{T_s^{'}/M^{'}} - 1) \\
&= \lim_{M^{'} \to \infty} \frac{(e^{T_s^{'}/M^{'}} - 1)}{M^{'-1} e^{-R/M^{'}}}
\end{aligned}$$

The limit of the numerator and the denominator are both zero. Therefore, L'Hôpital's Rule can be applied. This yields:

$$\begin{aligned}
\lim_{M^{'} \to \infty} T_w(M^{'}) &= \lim_{M^{'} \to \infty} \frac{-T_s^{'} M^{'-2} e^{T_s^{'}/M^{'}}}{-M^{'-2} e^{-R/M^{'}} + M^{'-1}(-RM^{'-2}) e^{-R/M^{'}}} \\
&= \lim_{M^{'} \to \infty} \frac{-T_s^{'} e^{T_s^{'}/M^{'}}}{-e^{-R/M^{'}} - M^{'-1} R e^{-R/M^{'}}} \\
&= T_s^{'}
\end{aligned}$$

Appendix B

Proof of Maximum Number of Similar Pages in a Checkpoint

This chapter establishes an upper bound on the number of similar pages that must be included in a checkpoint. In Chapter 5, the examination focuses on a symmetric difference algorithm (xor+lz4) and a simple similarity heuristic (neighbors). Showing an upper-bound on the number of pages that must be retained in this instance is straightforward. Here a more general problem is considered: asymmetric differences and arbitrary associations between similar pages and the other pages in application memory allocations.

The analysis begins with the construction of a graph of similar pages and their relationships.

$S =$ the set of similar pages in application allocated memory

$R =$ the set of non-similar reference pages associated with the similar pages

As discussed in Chapter 5, every similar page, $s \in S$, is defined by a set of tuples, $D = \{(d, p)\}$, where d is the value of the difference and $p \in (S \cup R)$ is the associated

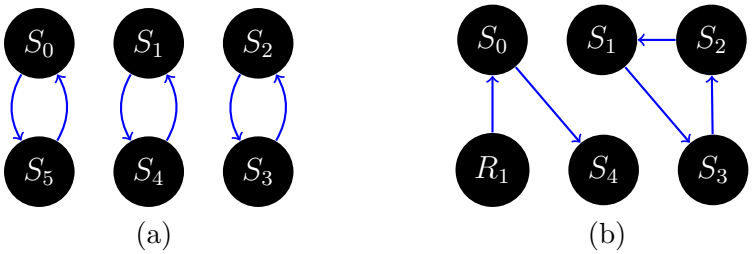


Figure B.1: Two hypothetical examples of the relationship between similar pages (S_i) and non-similar reference pages (R_i) in the memory of an application. The directed edges represent computed differences. An edge from S_i to S_j (or R_i to S_j) indicates a difference that would allow S_j to be recreated from the reference page. In B.1a, we would need to retain three similar pages, one from each cycle. In B.1b, we would need to retain one similar page from cycle on the right hand side of the figure.

reference page. Note that the reference page may be a similar page or it may be a non-similar page. There is a vertex in the graph for each $p \in (S \cup R)$. For each $s \in S$, I identify a tuple (d, p) such that for all $(d_i, p_i) \in D$, $d \leq d_i$. In other words, the smallest difference for each similar page is chosen. A directed edge from p to s is then created in the graph.

Given this graph, $G = (V, E)$ of similarity relationships, the analysis in this chapter will demonstrate that no more than half of the similar pages must be included in a checkpoint. The algorithm for identifying the set of similar pages that must be retained in a checkpoint is as follows.

1. Remove all of the vertices $s \in S$ that are reachable from $r \in R$.

2. Remove all of the vertices $r \in R$.

3. While there is at least one vertex v such that out-degree$(v) = 0$, remove v.

4. For each cycle in G, remove all but one vertex.

Examining the implications of each step in this algorithm proves that after the applying this algorithm, $|V| \leq \frac{1}{2}|S|$.

1. All of the similar pages that can be reconstructed from zero, duplicate or unique pages are removed from the graph. Chapter 5 describes how pages in each category are captured in the compressed checkpoint. In the worst case, this step removes no similar pages.

2. All non-similar pages can be independently reconstructed from the compressed checkpoint. This step will never remove any similar pages from G.

3. As this graph is constructed, for all $s \in S, \text{in-degree}(s) = 1$. At this point, for all $v \in V, v \in S$. In other words, the remaining vertices only represent similar pages. Therefore:

   $$\sum_{v \in V} \text{out-degree}(v) = |S|$$

   This property is invariant throughout this step of the algorithm. For each vertex, v, that is removed in this step $\text{out-degree}(v) = 0$ and $\text{in-degree}(v) = 1$. As a result, the total out-degree of G is reduced by 1 because removing v also removes exactly one edge which reduces the out-degree of one of the remaining vertices by 1. Removing v also reduces $|S|$ by 1.

4. At the conclusion of the previous step, the following holds for every remaining vertex, v:

   $$\text{out-degree}(v) > 0 \text{ and}$$
   $$\sum_{v \in V} \text{out-degree}(v) = |S|$$

   Therefore, $\text{out-degree}(v) = 1$. By construction:

   $$\text{in-degree}(v) = 1$$

Because every vertex has an in-degree and an out-degree of 1, each vertex must be part of a cycle. If one similar page is retained from each cycle, all of the pages in the cycle can be reconstructed. And because the smallest possible cycle consists of two vertices, no more than half of the similar pages in application-allocated memory must be included in the checkpoint.

References

[1] bzip2. http://bzip.org.

[2] Extremely fast compression algorithm. https://github.com/Cyan4973/lz4.

[3] Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory. Technical Report LLNL-TR-490254.

[4] November 2014 — TOP500 Supercomputer Sites. http://www.top500.org/ (visited March 2012).

[5] Top 500 Supercomputer Sites. http://www.top500.org/ (visited March 2012).

[6] SST: The structural simulation toolkit. http://sst.sandia.gov/about_sstmacro.html, 2011.

[7] Saurabh Agarwal, Rahul Garg, Meeta S. Gupta, and Jose E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th Annual International Conference on Supercomputing*, pages 277–286. ACM, 2004.

[8] Sean Ahern, Arie Shoshani, Kwan-Liu Ma, Alok Choudhary, Terence Critchlow, Scott Klasky, Valerio Pascucci, Jim Ahrens, Wes Bethel, Hank Childs, et al. Scientific discovery at the exascale: report from the DOE ASCR 2011 workshop on exascale data management, analysis, and visualization, 2011.

[9] Lorenzo Alvisi, Elmootazbellah N. Elnozahy, Sriram Rao, Syed Amir Husain, and Asanka de Mel. An analysis of communication induced checkpointing. In *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*, pages 242–249, 1999.

[10] Lorenzo Alvisi and Keith Marzullo. Message logging: Pessimistic, optimistic, causal, and optimal. *Software Engineering, IEEE Transactions on*, 24(2):149–159, 1998.

[11] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[12] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. In *Proceedings of the Linux Symposium, 2009, Montreal, Quebec*, pages 19–28, 2009.

[13] Argonne National Laboratory. Argonne Leadership Computing Facility. `https://www.alcf.anl.gov/user-guides/mira-cetus-vesta`, 2015.

[14] Steve Ashby, Pete Beckman, Jackie Chen, Phil Colella, Bill Collins, Dona Crawford, Jack Dongarra, Doug Kothe, Rusty Lusk, Paul Messina, et al. The opportunities and challenges of exascale computing. *Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee*, pages 1–77, 2010.

[15] Guillaume Aupy, Yves Robert, Frédéric Vivien, and Dounia Zaidouni. Checkpointing algorithms and fault prediction. *Journal of Parallel and Distributed Computing*, 74(2):2048–2064, 2014.

[16] Algirdas Avizienis and Jean-Claude Laprie. Dependable computing: From concepts to design diversity. *Proceedings of the IEEE*, 74(5):629–638, 1986.

[17] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: a checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 21. ACM, 2009.

[18] John Bent, Gary Grider, Brett Kettering, Adam Manzanares, Meghan McClelland, Aaron Torres, and Alfred Torrez. Storage challenges at Los Alamos National Lab. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–5. IEEE, 2012.

[19] Eduardo Berrocal, Leonardo Bautista-Gomez, Sheng Di, Zhiling Lan, and Franck Cappello. Lightweight silent data corruption detection based on runtime data analysis for HPC applications. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 275–278, 2015.

[20] Susmit Biswas, Bronis R. de Supinski, Martin Schulz, Diana Franklin, Timothy Sherwood, and Frederic T. Chong. Exploiting data similarity to reduce memory footprints. In *Proceedings of the 2011 IEEE International Parallel*

& Distributed Processing Symposium, IPDPS '11, pages 152–163, Washington, DC, USA, 2011. IEEE Computer Society.

[21] Swen Böhm and Christian Engelmann. xSim: The extreme-scale simulator. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 280–286. IEEE, 2011.

[22] Shekhar Borkar. The exascale challenge. In *VLSI Design Automation and Test (VLSI-DAT), 2010 International Symposium on*, pages 2–3. IEEE, 2010.

[23] George Bosilca, Aurélien Bouteiller, Elisabeth Brunet, Franck Cappello, Jack Dongarra, Amina Guermouche, Thomas Herault, Yves Robert, Frédéric Vivien, and Dounia Zaidouni. Unified model for assessing checkpointing protocols at extreme-scale. *Concurrency and Computation: Practice and Experience*, 26(17):2772–2791, 2014.

[24] Adrian Boteanu, Ciprian Dobre, Florin Pop, and Valentin Cristea. Simulator for fault tolerance in large scale distributed systems. In *Intelligent Computer Communication and Processing (ICCP), 2010 IEEE International Conference on*, pages 443–450. IEEE, 2010.

[25] Mohamed-Slim Bouguerra, Ana Gainaru, Leonardo Bautista Gomez, Franck Cappello, Satoshi Matsuoka, and Naoya Maruyama. Improving the computing efficiency of HPC systems using a combination of proactive and preventive checkpointing. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 501–512. IEEE, 2013.

[26] Mohamed-Slim Bouguerra, Thierry Gautier, Denis Trystram, and Jean-Marc Vincent. A flexible checkpoint/restart model in distributed systems. In *Parallel Processing and Applied Mathematics*, pages 206–215. Springer, 2010.

[27] Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack Dongarra. Correlated set coordination in fault tolerant message logging protocols. In *Euro-Par 2011 Parallel Processing*, pages 51–64. Springer, 2011.

[28] Daniele Briatico, Augusto Ciuffoletti, and Luca Simoncini. A distributed domino-effect free recovery algorithm. In *Symposium on Reliability in Distributed Software and Database Systems*, volume 84, pages 207–215, 1984.

[29] Patrick G. Bridges, Mark Hoemmen, Kurt B. Ferreira, Michael A. Heroux, Philip Soltero, and Ron Brightwell. Cooperative application/OS DRAM fault recovery. In *Euro-Par 2011: Parallel Processing Workshops*, pages 241–250. Springer, 2012.

[30] Ron Brightwell, Kevin Pedretti, and Trammell Hudson. SMARTMAP: operating system support for efficient data sharing among processes on a multi-core processor. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 25. IEEE Press, 2008.

[31] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM Trans. Comput. Syst.*, 15(4):412–447, November 1997.

[32] Franck Cappello. Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities. *International Journal of High Performance Computing Applications*, 23(3):212–226, 2009.

[33] Franck Cappello, Henri Casanova, and Yves Robert. Checkpointing vs. migration for post-petascale supercomputers. In *Parallel Processing (ICPP), 2010 39th International Conference on*, pages 168–177. IEEE, 2010.

[34] Franck Cappello, Amina Guermouche, and Marc Snir. On communication determinism in parallel HPC applications. In *Computer Communications and Networks (ICCCN), 2010 Proceedings of 19th International Conference on*, pages 1–8. IEEE, 2010.

[35] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

[36] Jacqueline H. Chen, Alok Choudhary, Bronis De Supinski, Matthew DeVries, Evatt R. Hawkes, Scott Klasky, Wei-Keng Liao, Kwan-Liu Ma, John Mellor-Crummey, Norbert Podhorszki, et al. Terascale direct numerical simulations of turbulent combustion using S3D. *Computational Science & Discovery*, 2(1):015001, 2009.

[37] Zizhong Chen. Extending algorithm-based fault tolerance to tolerate fail-stop failures in high performance distributed environments. In *Parallel and Distributed Processing Symposium, 2008. IPDPS 2008. 22nd International*, pages 1–8. IEEE, 2008.

[38] Zizhong Chen and Jack Dongarra. Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, april 2006.

[39] Colin Percival. Naive differences of executable code. `http://www.daemonology.net/bsdiff/`, 2010.

[40] Corbet. The SLUB allocator. `http://lwn.net/Articles/229984/`, April 2007.

[41] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, pages 1–12, New York, NY, USA, 1993. ACM.

[42] John T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computing Systems*, 22(3):303–312, 2006.

[43] Jack Dongarra, Thomas Herault, and Yves Robert. Revisiting the double checkpointing algorithm. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 706–715. IEEE, 2013.

[44] Jack Dongarra et al. The international exascale software project roadmap. *International Journal of High Performance Computing Applications*, 2011.

[45] James Elliott, Kishor Kharbas, David Fiala, Frank Mueller, Kurt B. Ferreira, and Christian Engelmann. Combining partial redundancy and checkpointing for HPC. In *Proceedings of the* 32^{nd} *International Conference on Distributed Computing Systems (ICDCS) 2012*, pages 615–626. IEEE Computer Society, Los Alamitos, CA, USA, June 18-21, 2012.

[46] Elmootazbellah N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[47] Elmootazbellah N. Elnozahy and James S. Plank. Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery. *Dependable and Secure Computing, IEEE Transactions on*, 1(2):97–108, 2004.

[48] Elmootazbellah N. Elnozahy and Willy Zwaenepoel. On the use and implementation of message logging. In *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on*, pages 298–307. IEEE, 1994.

[49] Christian Engelmann and Swen Böhm. Redundant execution of HPC applications with MR-MPI. In *Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN)*, pages 15–17, 2011.

[50] Keren Bergman et al. Exascale computing study: Technology challenges in achieving exascale systems, September 2008.

[51] Stuart I. Feldman and Channing B. Brown. Igor: A system for program debugging via reversible execution. In *ACM SIGPLAN Notices*, volume 24, pages 112–123. ACM, 1988.

[52] Kurt Ferreira, Rolf Riesen, Jon Stearley, James H. Laros III, Ron Oldfield, Kevin Pedretti, Patrick Bridges, Dorian Arnold, and Ron Brightwell. Evaluating the viability of process replication reliability for exascale systems. In *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage, and Analysis, (SC11)*, Nov 2011.

[53] Kurt B. Ferreira, Patrick Bridges, and Ron Brightwell. Characterizing application sensitivity to os interference using kernel-level noise injection. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 19. IEEE Press, 2008.

[54] Kurt B. Ferreira, Scott Levy, Patrick Widener, Dorian Arnold, and Torsten Hoefler. Understanding the effects of communication and coordination on checkpointing at scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC14)*, pages 883–894. IEEE Press, 2014.

[55] Kurt B. Ferreira, Kevin Pedretti, Ron Brightwell, Patrick G. Bridges, David Fiala, and Frank Mueller. Evaluating operating system vulnerability to memory errors. In *Proceedings of the 2nd International Workshop on Runtime and Operating Systems for Supercomputers*, page 11. ACM, 2012.

[56] Kurt B. Ferreira, Rolf Riesen, Rolf Brighwell, Patrick G. Bridges, and Dorian Arnold. libhashckpt: hash-based incremental checkpointing using GPUs. *Recent Advances in the Message Passing Interface*, pages 272–281, 2011.

[57] David Fiala, Kurt B. Ferreira, and Frank Mueller. FlipSphere: A software-based DRAM error detection and correction library for HPC. Technical Report SAND2014-0438C, Sandia National Laboratories, Feb. 2014.

[58] David Fiala, Kurt B. Ferreira, Frank Mueller, and Christian Engelmann. A tunable, software-based DRAM error detection and correction library for HPC. In *Lecture Notes in Computer Science: Proceedings of the European Conference on Parallel and Distributed Computing (Euro-Par) 2011: Workshop on Resiliency in High Performance Computing (Resilience) in Clusters, Clouds, and Grids*, Bordeaux, France, Aug 2011. Springer Verlag, Berlin, Germany.

[59] David Fiala, Frank Mueller, Christian Engelmann, Kurt B. Ferreira, Ron Brightwell, and Rolf Riesen. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the* 25^{th} *IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC12)*, pages 78:1–78:12, Salt Lake City, UT, USA, November 10-16, 2012. ACM Press, New York, NY, USA.

[60] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC12)*. IEEE Computer Society Press, 2012.

[61] DoE Exascale Co-Design Center for Materials in Extreme Environments. ExMatEx. `http://www.exmatex.org/`, 2012.

[62] ASCAC Subcommittee for the Top Ten Exascale Research Challenges. Top ten exascale research challenges. Technical report, United States Department of Energy, February 2014.

[63] Song Fu and Cheng-Zhong Xu. Exploring event correlation for failure prediction in coalitions of clusters. In *Supercomputing, 2007. SC'07. Proceedings of the 2007 ACM/IEEE Conference on*, pages 1–12. IEEE, 2007.

[64] Errin W. Fulp, Glenn A. Fink, and Jereme N. Haack. Predicting computer system failures using support vector machines. In *Proceedings of the First USENIX conference on Analysis of system logs*, WASL'08, pages 5–5, Berkeley, CA, USA, 2008. USENIX Association.

[65] Ana Gainaru, Franck Cappello, and William Kramer. Taming of the shrew: Modeling the normal and faulty behaviour of large-scale HPC systems. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 1168–1179. IEEE, 2012.

[66] Ana Gainaru, Franck Cappello, Marc Snir, and William 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 (SC12)*, page 77, 2012.

[67] Ana Gainaru, Franck Cappello, Marc Snir, and William Kramer. Failure prediction for HPC systems and applications: Current situation and open issues. *International Journal of High Performance Computing Applications*, 27(3):273–282, 2013.

[68] Marc Gamell, Daniel S. Katz, Hemanth Kolla, Jacqueline Chen, Scott Klasky, and Manish Parashar. Exploring automatic, online failure recovery for scientific applications at extreme scales. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC14)*, pages 895–906. IEEE Press, 2014.

[69] Marc Gamell, Keita Teranishi, Michael A. Heroux, Jackson Mayo, Hemanth Kolla, Jacqueline Chen, and Manish Parashar. Failure masking and local recovery for stencil-based applications at extreme scales. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC15)*. IEEE Press, 2015.

[70] Felix C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys (CSUR)*, 31(1):1–26, 1999.

[71] Roberto Gioiosa, Jose Carlos Sancho, Song Jiang, Fabrizio Petrini, and Kei Davis. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, page 9. IEEE Computer Society, 2005.

[72] Bharan Giridhar, Michael Cieslak, Deepankar Duggal, Ronald Dreslinski, Hsing Min Chen, Robert Patti, Betina Hold, Chaitali Chakrabarti, Trevor Mudge, and David Blaauw. Exploring DRAM organizations for energy-efficient and resilient exascale memories. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 23. ACM, 2013.

[73] James N. Glosli, David F. Richards, K.J. Caspersen, Robert E. Rudd, John A. Gunnels, and Frederick H. Streitz. Extending stability beyond CPU millennium: a micron-scale atomistic simulation of Kelvin-Helmholtz instability. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, page 58. ACM, 2007.

[74] Amina Guermouche, Thomas Ropars, Elisabeth Brunet, Marc Snir, and Franck Cappello. Uncoordinated checkpointing without domino effect for send-deterministic message passing applications. In *Proceedings of the 2011 IEEE International Parallel and Distributed Processing Symposium*, May 2011.

[75] Amina Guermouche, Thomas Ropars, Marc Snir, and Franck Cappello. HydEE: Failure containment without event logging for large scale send-deterministic MPI applications. In *IPDPS*, pages 1216–1227, 2012.

[76] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. Difference Engine: Harnessing memory redundancy in virtual machines. *Commun. ACM*, 53(10):85–93, October 2010.

[77] Evatt R. Hawkes, Ramanan Sankaran, James C. Sutherland, and Jacqueline H. Chen. Direct numerical simulation of turbulent combustion: fundamental insights towards predictive models. In *Journal of Physics: Conference Series*, volume 16, page 65. IOP Publishing, 2005.

[78] Van Emden Henson and Ulrike Meier Yang. BoomerAMG: A parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics*, 41(1):155–177, 2002.

[79] Thomas Herault and Yves Robert. *Fault-Tolerance Techniques for High-Performance Computing*. Springer, 2015.

[80] Michael A. Heroux, Douglas W. Doerfler, Paul S. Crozier, James M. Willenbring, H. Carter Edwards, Alan Williams, Mahesh Rajan, Eric R. Keiter, Heidi K. Thornquist, and Robert W. Numrich. Improving performance via mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, 2009.

[81] Torsten Hoefler. LogGOPSim - a LogGOPS (LogP, LogGP, LogGPS) simulator and simulation framework. `http://www.unixer.de/research/LogGOPSim/`, Apr. 10 2013.

[82] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. Characterizing the influence of system noise on large-scale applications by simulation. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, Nov. 2010.

[83] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. LogGOPSim - simulating large-scale applications in the LogGOPS model. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 597–604. ACM, Jun. 2010.

[84] Torsten Hoefler, Christian Siebert, and Andrew Lumsdaine. Group operation assembly language-a flexible way to express collective communication. In *Parallel Processing, 2009. ICPP'09. International Conference on*, pages 574–581. IEEE, 2009.

[85] Jeremy Hsu. When will we have an exascale supercomputer?[news]. *Spectrum, IEEE*, 52(1):13–16, 2015.

[86] Kuang-Hua Huang and Jacob A. Abraham. Algorithm-based fault tolerance for matrix operations. *Computers, IEEE Transactions on*, 100(6):518–528, 1984.

[87] Andy A. Hwang, Ioan A. Stefanovici, and Bianca Schroeder. Cosmic rays don't strike twice: understanding the nature of DRAM errors and the implications for system design. *ACM SIGPLAN Notices*, 47(4):111–122, 2012.

[88] Dewan Ibtesham, Dorian Arnold, Patrick G. Bridges, Kurt B. Ferreira, and Ron Brightwell. On the viability of compression for reducing the overheads of checkpoint/restart-based fault tolerance. In *Parallel Processing (ICPP), 2012 41st International Conference on*, pages 148–157. IEEE, 2012.

[89] Dewan Ibtesham, Kurt B. Ferreira, and Dorian Arnold. A checkpoint compression study for high-performance computing systems. *International Journal of High Performance Computing Applications*, page 1094342015570921, 2015.

[90] Tanzima Zerin Islam, Kathryn Mohror, Saurabh Bagchi, Adam Moody, Bronis R. De Supinski, and Rudi Eigenmann. McrEngine: A scalable checkpointing system using data-aware aggregation and compression. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11. IEEE, 2012.

[91] Curtis L. Janssen, Helgi Adalsteinsson, Scott Cranford, Joseph P. Kenny, Ali Pinar, David A. Evensky, and Jackson Mayo. A simulator for large-scale parallel computer architectures. *Technology Integration Advancements in Distributed Systems and Computing*, 179, 2012.

[92] Qiangfeng Jiang and D. Manivannan. An optimistic checkpointing and selective message logging approach for consistent global checkpoint collection in distributed systems. In *Proceedings of the 2007 IEEE International Parallel and Distributed Processing Symposium*, March 2007.

[93] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using asynchronous message logging and checkpointing. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 171–181, 1988.

[94] Josh Macdonald. Xdelta. `http://xdelta.org`, January 2013.

[95] Jungrae Kim, Michael Sullivan, and Mattan Erez. Bamboo ECC: Strong, safe, and flexible codes for reliable computer memory. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 2015.

[96] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *Proceeding of the 41st annual international symposium on Computer architecuture*, pages 361–372. IEEE Press, 2014.

[97] Youngbae Kim, James S. Plank, and Jack J. Dongarra. Fault tolerant matrix operations for networks of workstations using multiple checkpointing. In *High Performance Computing on the Information Superhighway. HPC Asia '97*, pages 460–465, Seoul, South Korea, April 1997. Los Alamitos, CA, USA : IEEE Comput. Soc. Press, 1997.

[98] Andi Kleen. Machine check handling on Linux. *SUSE Labs*, 2004.

[99] Andi Kleen. mcelog: memory error handling in user space. In *Proceedings of Linux Kongress 2010*, Nuremburg, Germany, September 2010.

[100] George Kola, Tevfik Kosar, and Miron Livny. Faults in large distributed systems and what we can do about them. In *Euro-Par 2005 Parallel Processing*, pages 442–453. Springer, 2005.

[101] David G. Korn, Joshua P. MacDonald, Jeffrey C. Mogul, and Kiem-Phong Vo. The VCDIFF generic differencing and compression data format. Technical Report 3284, RFC, June 2002.

[102] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[103] John R. Lange, Kevin T. Pedretti, Trammell Hudson, Peter A. Dinda, Zheng Cui, Lei Xia, Patrick G. Bridges, Andy Gocke, Steven Jaconette, Michael Levenhagen, and Ron Brightwell. Palacios and kitten: New high performance operating systems for scalable virtualized and native supercomputing. In *Proceedings of the 2010 IEEE International Parallel and Distributed Processing Symposium*, IPDPS'10, pages 1–12, 2010.

[104] Jean-Claude Laprie. Dependable computing and fault-tolerance. *Digest of Papers FTCS-15*, pages 2–11, 1985.

[105] Lawrence Livermore National Laboratories. IRS: Implicit Radiation Solver 1.4 Build Notes. `https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/irs/irs.readme.html`.

[106] Lawrence Livermore National Laboratories. SAMRAI. `https://computation.llnl.gov/casc/SAMRAI/index.html`.

[107] Lawrence Livermore National Laboratories. ASC Sequoia Benchmark Codes. `https://asc.llnl.gov/sequoia/benchmarks`, August 2009.

[108] Lawrence Livermore National Laboratory. Co-design at lawrence livermore national lab : Livermore unstructured lagrangian explicit shock hydrodynamics (lulesh). `http://codesign.llnl.gov/lulesh.php`.

[109] Scott Levy, Kurt B. Ferreira, and Patrick G. Bridges. Characterizing the impact of rollback avoidance at extreme-scale: A modeling approach. In *Parallel Processing (ICPP), 2014 43rd International Conference on*, pages 401–410. IEEE, 2014.

[110] Scott Levy, Kurt B. Ferreira, Patrick G. Bridges, Aidan P. Thompson, and Christian Trott. A study of the viability of exploiting memory content similarity to improve resilience to memory errors. *International Journal of High Performance Computing Applications*, 29(1):5–20, February 2015.

[111] Chung-Chi Jim Li and W. Kent Fuchs. CATCH-compiler-assisted techniques for checkpointing. In *Fault-Tolerant Computing, 1990. FTCS-20. Digest of Papers., 20th International Symposium*, pages 74–81, jun 1990.

[112] Sheng Li, Ke Chen, Ming-Yu Hsieh, Naveen Muralimanohar, Chad D. Kersey, Jay B. Brockman, Arun F. Rodrigues, and Norman P. Jouppi. System implications of memory reliability in exascale computing. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC11)*, page 46. ACM, 2011.

[113] Antonina Litvinova, Christian Engelmann, and Stephen L. Scott. A proactive fault tolerance framework for high-performance computing. In *Proceedings of the 9th IASTED International Conference*, volume 676, page 105, 2009.

[114] Los Alamos National Laboratory. Trinity. `http://www.lanl.gov/projects/trinity/`, 2015.

[115] Sandia National Laboratories. `App_model` - application simulator (gpl). `http://www.cs.sandia.gov/web1400/1400_download.html`, Feb. 10 2014.

[116] J.M. McGlaun, S.L. Thompson, and M.G. Elrick. CTH: A three-dimensional shock wave physics code. *International Journal of Impact Engineering*, 10(1):351–360, 1990.

[117] Sébastien Monnet, Christine Morin, and Ramamurthy Badrinath. A hierarchical checkpointing protocol for parallel applications in cluster federations. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 211. IEEE, 2004.

[118] Sébastien Monnet, Christine Morin, and Ramamurthy Badrinath. Hybrid checkpointing for parallel applications in cluster federations. In *Cluster Computing and the Grid, 2004. CCGrid 2004. IEEE International Symposium on*, pages 773–782. IEEE, 2004.

[119] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R. de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC10)*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[120] Andreas Moshovos and Alexandros Kostopoulos. Cost-effective, high-performance giga-scale checkpoint/restore. Technical report, University of Toronto, November 2004.

[121] Onur Mutlu and Lavanya Subramanian. Research problems and opportunities in memory systems. *Supercomputing frontiers and innovations*, 1(3), 2015.

[122] Prashant J. Nair, Dae-Hyun Kim, and Moinuddin K. Qureshi. Archshield: Architectural framework for assisting DRAM scaling by tolerating high error rates. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 72–83. ACM, 2013.

[123] Victor P. Nelson. Fault-tolerant computing: Fundamental concepts. *Computer*, 23(7):19–25, 1990.

[124] Bogdan Nicolae. Towards scalable checkpoint restart: A collective inline memory contents deduplication proposal. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 19–28. IEEE, 2013.

[125] Katherine Noyes. 94 percent of the world's Top 500 supercomputers run Linux. `http://www.linux.com/news/enterprise/high-performance/147-high-performance/666669-94-percent-of-the-worlds-top-500-supercomputers-run-linux-`, November 2012.

[126] Ron A. Oldfield, Sarala Arunagiri, Patricia J. Teller, Seetharami Seelam, Maria Ruiz Varela, Rolf Riesen, and Philip C. Roth. Modeling the impact of checkpoints on next-generation systems. In *24th IEEE Conference on Mass Storage Systems and Technologies*, pages 30–46, September 2007.

[127] James S. Plank. An overview of checkpointing in uniprocessor and distributed systems, focusing on implementation and performance. Technical Report UT-CS-97-372, Department of Computer Science, July 1997.

[128] James S. Plank, Youngbae Kim, and Jack J. Dongarra. Algorithm-based diskless checkpointing for fault tolerant matrix operations. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers*, pages 351–360, Pasadena, CA, USA, June 1995. Los Alamitos, CA, USA : IEEE Comput. Soc. Press, 1995.

[129] James S. Plank and Kai Li. Faster checkpointing with N+1 parity. In *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on*, pages 288–297. IEEE, 1994.

[130] James S. Plank and Kai Li. ickp: A consistent checkpointer for multicomputers. *Parallel & Distributed Technology: Systems & Applications, IEEE*, 2(2):62–67, 1994.

[131] James S. Plank, Kai Li, and Michael A. Puening. Diskless checkpointing. *Parallel and Distributed Systems, IEEE Transactions on*, 9(10):972–986, 1998.

[132] James S. Plank, Jian Xu, and Robert H. B. Netzer. Compressed differences: An algorithm for fast incremental checkpointing. Technical Report CS-95-302, University of Tennessee, August 1995.

[133] Steve Plimpton. Fast parallel algorithms for short-range molecular-dynamics. *Journal of Computational Physics*, 117(1):1–19, 1995.

[134] Sean Quinlan and Sean Dorward. Venti: a new approach to archival storage. In *Proceedings of the FAST 2002 Conference on File and Storage Technologies*, volume 4, 2002.

[135] Brian Randell. System structure for software fault tolerance. In *ACM SIGPLAN Notices*, volume 10, pages 437–449. ACM, 1975.

[136] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. SWIFT: Software implemented fault tolerance. In *Proceedings of the international symposium on Code generation and optimization*, pages 243–254. IEEE Computer Society, 2005.

[137] Rolf Riesen, Ron Brightwell, Patrick G. Bridges, Trammell Hudson, Arthur B. Maccabe, Patrick M. Widener, and Kurt Ferreira. Designing and implementing lightweight kernels for capability computing. *Concurrency and Computation: Practice and Experience*, 21(6):793–817, April 2009.

[138] Rolf Riesen, Kurt Ferreira, Dilma Da Silva, Pierre Lemarinier, Dorian Arnold, and Patrick G. Bridges. Alleviating scalability issues of checkpointing protocols. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11. IEEE, 2012.

[139] Rolf Riesen, Kurt Ferreira, Jon Stearley, Ron Oldfield, James H. Laros III, Kevin Pedretti, Ron Brightwell, et al. Redundant computing for exascale systems. Technical report, Technical report SAND2010-8709, Sandia National Laboratories, 2010.

[140] Thomas Ropars, Amina Guermouche, Bora Uçar, Esteban Meneses, Laxmikant V Kalé, and Franck Cappello. On the use of cluster-based partial message logging to improve fault tolerance for MPI HPC applications. In *Euro-Par 2011 Parallel Processing*, pages 567–578. Springer, 2011.

[141] Kenichiro Sakai, Shinji Sumimoto, and Motoyoshi Kurokawa. High-performance and highly reliable file system for the K computer. *FUJITSU Science Technology*, 48(3):302–209, 2012.

[142] Felix Salfner, Maren Lenk, and Miroslaw Malek. A survey of online failure prediction methods. *ACM Comput. Surv.*, 42(3):10:1–10:42, March 2010.

[143] Sandia National Laboratories. LAMMPS molecular dynamics simulator. `http://lammps.sandia.gov`, Apr. 10 2013.

[144] Sandia National Laboratory. Mantevo project home page. `https://software.sandia.gov/mantevo`, Apr. 10 2010.

[145] Sandia National Laboratory. Kitten lightweight kernel. `https://software.sandia.gov/trac/kitten`, March 2012.

[146] Nirmal R. Saxena and Edward J. McCluskey. Dependable adaptive computing systems-the ROAR project. In *Systems, Man, and Cybernetics, 1998. 1998 IEEE International Conference on*, volume 3, pages 2172–2177. IEEE, 1998.

[147] Tamara Schmitz. The rise of serial memory and the future of DDR. Technical Report WP456, Xilinx, March 2015.

[148] Bianca Schroeder and Garth A. Gibson. Understanding failures in petascale computers. In *Journal of Physics: Conference Series*, volume 78, page 012022. IOP Publishing, 2007.

[149] Bianca Schroeder and Garth A. Gibson. A large-scale study of failures in high-performance computing systems. *Dependable and Secure Computing, IEEE Transactions on*, 7(4):337–350, 2010.

[150] John Shalf, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. In *High Performance Computing for Computational Science–VECPAR 2010*, pages 1–25. Springer, 2011.

[151] Philip P. Shirvani, Nirmal R. Saxena, and Edward J. McCluskey. Software-implemented EDAC protection against SEUs. *Reliability, IEEE Transactions on*, 49(3):273–284, 2000.

[152] Luis M. Silva and João G. Silva. An experimental study about diskless checkpointing. In *24th EUROMICRO Conference*, pages 395 – 402. IEEE Computer Society Press, August 1998.

[153] Luis M. Silva, Bart Veer, and João G. Silva. Checkpointing SPMD applications on transputer networks. In *Scalable High-Performance Computing Conference, 1994., Proceedings of the*, pages 694–701. IEEE, 1994.

[154] Horst D. Simon. Barriers to exascale computing. In *High Performance Computing for Computational Science-VECPAR 2012*, pages 1–3. Springer, 2013.

[155] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. Memory errors in modern systems: The good, the bad, and the ugly. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 297–310, New York, NY, USA, 2015. ACM.

[156] David Stirzaker. *Probability and Random Variables: A Beginner's Guide*. Cambridge University Press, 1999.

[157] Rob Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(3):204–226, 1985.

[158] Anand Tikotekar, Geoffroy Vallee, Thomas Naughton, Stephen L. Scott, and Chokchai Leangsuksun. Evaluation of fault-tolerant policies using simulation. In *Cluster Computing, 2007 IEEE International Conference on*, pages 303–311. IEEE, 2007.

[159] Devesh Tiwari, Swastik Gupta, and Sudharshan S. Vazhkudai. Lazy checkpointing: Exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 25–36. IEEE, 2014.

[160] Anthony Tuininga. Cx_bsdiff. `http://starship.python.net/crew/atuining/cx_bsdiff/index.html`, February 2006.

[161] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive process-level live migration in HPC environments. In *Proceedings*

of the 2008 ACM/IEEE conference on Supercomputing, page 43. IEEE Press, 2008.

[162] Patrick Widener, Kurt B. Ferreira, Scott Levy, and Torsten Hoefler. Exploring the effect of noise on the performance benefit of nonblocking allreduce. In *Proceedings of the 21st European MPI Users' Group Meeting*, page 77. ACM, 2014.

[163] Patrick M. Widener, Kurt B. Ferreira, Scott Levy, Patrick G. Bridges, Dorian Arnold, and Ron Brightwell. Asking the right questions: benchmarking fault-tolerant extreme-scale systems. In *Euro-Par 2013: Parallel Processing Workshops*, pages 717–726. Springer, 2014.

[164] Joshua Wingstrom. *Overcoming The Difficulties Created By The Volatile Nature Of Desktop Grids Through Understanding, Prediction And Redundancy.* PhD thesis, Ph. D. thesis, University of Hawai i at Manoa, 2009.

[165] Lei Xia, Kyle Hale, and Peter Dinda. ConCORD: easily exploiting memory content redundancy through the content-aware service command. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing (HPDC)*, pages 25–36. ACM, 2014.

[166] Tianming Yang, Hong Jiang, Dan Feng, Zhongying Niu, Ke Zhou, and Yaping Wan. DEBAR: A scalable high-performance de-duplication storage system for backup and archiving. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[167] Doe Hyun Yoon and Mattan Erez. Virtualized and flexible ECC for main memory. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 397–408, New York, NY, USA, 2010. ACM.

[168] John W. Young. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530–531, 1974.

[169] Gengbin Zheng, Lixia Shi, and Laxmikant V. Kalé. FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. In *Cluster Computing, 2004 IEEE International Conference on*, pages 93–103. IEEE, 2004.

[170] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 18:1–18:14, Berkeley, CA, USA, 2008. USENIX Association.

[171] Willy Zwaenepoel and David B. Johnson. Sender-based message logging. In *Proceedings of the Seventeenth International Symposium on Fault-Tolerant Computing*, number LABOS-CONF-2005-064, 1987.