Kurt B. Ferreira

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, Chairperson

Dorian Arnold

Michela Taufer

Jed Crandall

Keeping Checkpoint/Restart Viable for Exascale Systems

by

Kurt B. Ferreira

B.S., Computer Science, New Mexico Institute of Mining and Technology, 2000
B.S., Mathematics, New Mexico Institute of Mining and Technology, 2000
M.S., Computer Science, University of New Mexico, 2008

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

November, 2011

©2011, Kurt B. Ferreira

Dedication

To Summer, for always supporting and encouraging me.

Acknowledgments

I wish to express my sincere gratitude to the people who made my research possible. I was very fortunate to have had an excellent support system throughout this process which included advisors, collaborators, research group members, family and friends.

Firstly, I wish to thank my advisor, Patrick Bridges. With his help and guidance I have matured tremendously as a researcher. His quality advice and continual encouragement throughout my prolonged graduate career has proved invaluable. I feel privileged to have had the opportunity to be a part of his research group, it has given me a unique and unmatched opportunity.

I would also like to extend many thanks to my dissertation committee members, Dorian Arnold, Michela Taufer, and Jed Crandall. Dorian has always been generous with his time and advice and I am grateful to have had the opportunity to interact with him. I look forward to maintaining our collaboration. Both Michela and Jed have provided much helpful and valuable feedback that has greatly improved the quality of this work.

I am forever indebted to a number of past and present associates from Sandia National Laboratories. Particularly my manager, Ron Brightwell, whose patient mentorship, freedom, and advice has helped me greatly as a researcher. I also wish to thank Rolf Riesen [1]. Our work together was both fun and productive. Many of the results presented in this work are due to discussions we had. Thanks also to Mark Hoemenn, Kevin Pedretti, Ron Oldfield, Jon Stearley, and Jim Laros for your help.

Thanks to the current and former members of the Scalable System Software group at the University of New Mexico for their support and suggestions.

Finally, I would like to thank my wife Summer. Having gone through this dissertation process herself, she always knew how to help. She has been a source of strength, encouragement and motivation. I have always been able to count on your steady support. I also thank my family for their continual love and support with special thanks to my parents Dennis and Barbara for instilling the importance of hard work and higher education, and my siblings Aaron, Jodie, and Raúl.

[1] Rolf is now at IBM Research in Ireland

Keeping Checkpoint/Restart Viable for Exascale Systems

by

Kurt B. Ferreira

B.S., Computer Science, New Mexico Institute of Mining and Technology, 2000

B.S., Mathematics, New Mexico Institute of Mining and Technology, 2000

M.S., Computer Science, University of New Mexico, 2008

Ph.D., Computer Science, University of New Mexico, 2011

Abstract

Next-generation exascale systems, those capable of performing a quintillion (10^{18}) operations per second, are expected to be delivered in the next 8-10 years. These systems, which will be 1,000 times faster than current systems, will be of unprecedented scale. As these systems continue to grow in size, faults will become increasingly common, even over the course of small calculations. Therefore, issues such as fault tolerance and reliability will limit application scalability. Current techniques to ensure progress across faults like checkpoint/restart, the dominant fault tolerance mechanism for the last 25 years, are increasingly problematic at the scales of future systems due to their excessive overheads. In this work, we evaluate a number of techniques to decrease the overhead of checkpoint/restart and keep this method viable for future exascale systems. More specifically, this work evaluates state-machine replication to dramatically increase the checkpoint interval (the time between successive checkpoints) and hash-based, probabilistic incremental checkpointing using

graphics processing units to decrease the checkpoint commit time (the time to save one checkpoint). Using a combination of empirical analysis, modeling, and simulation, we study the costs and benefits of these approaches on a wide range of parameters. These results, which cover of number of high-performance computing capability workloads, different failure distributions, hardware mean time to failures, and I/O bandwidths, show the potential benefits of these techniques for meeting the reliability demands of future exascale platforms.

Contents

List of Figures **xv**

List of Tables **xxiv**

1 Introduction **1**

1.1 Overview 1

1.2 Reliability Challenges for Extreme-Scale Systems 2

1.3 Rollback Recovery 5

1.3.1 Overview 5

1.3.2 Scalability of Rollback Recovery 7

1.4 State-Machine Replication 8

1.5 Hash-Based Incremental Checkpointing 10

1.6 Contributions 11

1.6.1 Modeling Replication 11

1.6.2 *r*`MPI`: A Transparent MPI Replication Library 12

1.6.3 Hash-Based Incremental-Checkpointing using GPUs 12

1.6.4 Analysis of Checkpoint/Restart Viability for Extreme-Scale Systems 12

1.7 Document Organization 13

2 Background and Related Work 14

2.1 Failures, Faults, and Associated Models 14

2.2 Reliability Metrics 16

2.3 Traditional Rollback Recovery 17

2.3.1 Overview 17

2.3.2 Traditional Checkpoint/Restart 17

2.3.3 Optimizing Rollback/Recovery 20

2.3.4 High-speed Storage for Checkpoint/Restart 24

2.4 Other Checkpointing Systems 25

2.5 State Machine Replication 26

2.5.1 Overview 26

2.5.2 Passive versus Active Replication 27

2.5.3 Group Communication 27

2.6 Other Forward Recovery Methods 29

2.7 Fault-Tolerant Algorithms 30

2.7.1 Application-based Data Redundancy 31

2.7.2 Application-based Computational Redundancy 31

2.8 Proactive Migration 32

2.9 Summary 32

3 Replication in High-Performance Computing 34

3.1 Overview 34

3.2 Modeling Replication for HPC 36

3.2.1 Approximations for the Birthday Problem 38

3.2.2 Comparison of Approximations 40

3.3 Model-based Analysis 41

3.4 Simulation-Based Analysis 44

3.4.1 Simulator Details 44

3.4.2 Comparison of Simulation and Modeling 45

3.4.3 Non-Exponential Failure Distributions 46

3.5 Summary 47

4 *r*MPI: State-machine Replication in a Message Passing Environment 51

4.1 Overview 51

4.2 *r*MPI Design 52

4.2.1 Basic Consistency Protocols 52

4.2.2 MPI Consistency Requirements for Active Protocols 54

4.2.3 Failure Detection 58

4.3 *r*MPI Implementation 58

4.3.1 Basic Architecture 59

4.3.2 RAS Functionality 59

4.3.3 Usage 60

4.4 Summary 60

5 Evaluating State-machine Replication's Runtime Overheads 62

5.1 Methodology 63

5.1.1 Replica Placement 64

5.2 *r*MPI Runtime Results 64

5.2.1 Benchmark Details 64

5.2.2 Micro-benchmark Performance 66

5.2.3 Application Performance 70

5.3 Analysis of Run Time Overheads 75

5.4 Summary 76

6 Replication Analysis 79

6.1 Overview 79

6.2 Comparison Approach 80

6.2.1 Assumptions 80

6.3 Combined Hardware and Software Overheads 81

6.4 Scaling at Different Failure Rates 81

6.5 Scaling at Different Checkpoint I/O Rates 83

6.6 Triple Module Redundancy and Beyond 86

6.7 Simulation-Based Analysis 87

6.7.1 Overview 87

6.7.2 Non-Exponential Failure Distributions 88

6.8 Summary 91

7 Incremental Checkpointing 93

7.1 Introduction 93

7.2 A Model for the Viability of Hash-Based Incremental Checkpointing . 95

7.3 `Libhashckpt`: Hash-based Incremental Checkpointing 96

7.3.1 Overview 96

7.3.2 Implementation Details 97

7.3.3 Hash/Checksum Algorithms 98

7.4 State Compression Measurement 102

7.4.1 Applications and Platform 103

7.4.2 Hash-based Dirty Data Detection 103

7.4.3 Checkpoint File Size Comparison 106

7.5 Hashing Costs 108

7.5.1 Rotating XOR 109

7.5.2 CRC32 109

7.5.3 ADLER32 111

7.5.4 MD5 112

7.5.5 SHA256 113

7.6 Viability of Hash-Based Incremental Checkpointing 115

7.7 Summary 118

8 Conclusion **119**

8.1 Contributions 119

8.2 Future Directions 123

Appendices **125**

A *r*MPI Micro-benchmark Performance **126**

A.1 MPI_Allreduce() 127

A.2 MPI_Reduce() 127

A.3 MPI_Bcast() 132

A.4 MPI_Alltoall() 132

A.5 MPI_Barrier() 132

References **137**

List of Figures

1.1 Socket counts for the five fastest machines topping the Top500 [1] list for the past two decades. Also included are two proposed systems due to become operational in 2010 and 2011, ASC BlueWaters [2], Cielo [3], and Sequoia [4]. 4

1.2 Socket counts for the five fastest machines topping the Top500 [1] list from 2001 to 2010. Also included are upcoming systems and a proposed "Aggressive" exascale system [5]. The shaded region in the figure shows the range of socket counts using past data for a line fit. 5

1.3 System MTBF values expected from large-scale exascale systems. Current studies [6–11] place the node MTBF somewhere between 5 and 25 years, which is expected to continue in the future. Next-generation systems are expected to see multiple faults per hour. . . 6

3.1 Expected number of node failures before an application interrupt in a system with redundant nodes. Numbers are calculated using the birthday problem Equation 3.1. 38

3.2 Comparison of a number of methods for approximating the birthday problem. Q(m) refers to Equation 3.1, $R_2(X)$ refers to Equation 3.2, and $E_2(X)$ refers to the indicator variables method of Equation 3.7 with two replicas. 42

3.3 Modeled application Mean Time To Interrupt (MTTI) and efficiency with and without state machine replication for a 168-hour application, 5-year per-socket MTBF, and 15 minute checkpoint times. In the replication case we have two replicas per process rank. The shaded region corresponds to possible socket counts for an exascale class machine [12]. 43

3.4 Comparison of simulator and model for a dual-redundant, 336 hour work weak scaling problem with a 5 year MTBF and a 15 minute checkpoint write time. 46

3.5 Simulated application efficiency with and without state machine replication for a 168-hour application, 4-year per-socket MTBF (Θ), and 15 minute checkpoint commit times with failure rate drawn from exponential and Weibull distributions [11]. In the replication case we have two replicas per process rank. The shaded region corresponds to possible socket counts for an exascale class machine [12]. 48

3.6 Simulated application efficiency with and without state machine replication for a 168-hour application, 12-year per-socket MTBF (Θ), and 15 minute checkpoint commit times with failure rate drawn from exponential and Weibull distributions [11]. In the replication case we have two replicas per process rank. The shaded region corresponds to possible socket counts for an exascale class machine [12]. 49

4.1 Basic active replicated communication strategies for two different *r*MPI message consistency protocols. Additional protocol exchanges are needed in special cases such as MPI_ANY_SOURCE. 53

4.2 Basic passive replicated communication strategies for two different *r*MPI message consistency protocols. Additional protocol exchanges are needed in special cases such as MPI_ANY_SOURCE. 55

4.3 Original and redundant messages with the same tag must maintain the same order. 56

5.1 Bandwidth comparison. Native is benchmark without the *r*MPI library. Base is with *r*MPI for each protocol, but no redundant nodes. For this test the performance of forward, reverse, and shuffle fully redundant runs are equivalent. 67

5.2 Latency comparison. For this test the performance of forward, reverse, and shuffle is equivalent. 68

5.3 Latency comparison using MPI_ANY_SOURCE. For this test the performance of forward, reverse, and shuffle is equivalent. 69

5.4 *r*MPI message rate measurements. 70

5.5 Host CPU utilization for send and receive for the two protocols compared to native and baseline. Native is the benchmark without *r*MPI; baseline has *r*MPI linked in but does not use redundant nodes. . . . 71

5.6 LAMMPS *r*MPI performance comparison. For both mirror and parallel, baseline performance overhead is equivalent. For this application the performance of the forward, reverse, and shuffle fully redundant modes are equivalent. 72

5.7 SAGE *r*MPI performance comparison. For both mirror and parallel baseline performance overhead is equivalent. For this application the performance of forward, reverse, and shuffle fully redundant modes are equivalent. 73

5.8 CTH *r*MPI performance comparison. For both mirror and parallel baseline performance overhead is equivalent. For this application the performance of forward, reverse, and shuffle fully redundant modes are equivalent. 74

5.9 HPCCG *r*MPI performance comparison. Varying performance for native and baseline between mirror and parallel protocols is due to different node allocations. 75

5.10 Best-case (Equation 5.1) and worst-case (Equation 5.2) *r*MPI run time protocol overhead fit functions and corresponding data from CTH, SAGE, and LAMMPS. 77

6.1 Modeled application efficiency with and without replication including worst-case *r*MPI run time overheads. Shaded region corresponds to possible socket counts for an exascale class machine [12]. 82

6.2 Modeled replication break-even point assuming a constant checkpoint time (δ) of 15 minutes. Shaded region corresponds to possible socket counts and MTBFs for an exascale class machine [12]. Note that above the line within this region is where replication has significantly lower overheads compared to traditional checkpoint/restart. 84

6.3 "Break-even" points for replication for various checkpoint bandwidth rates. The shaded region corresponds to possible socket counts and socket MTBFs for exascale class machines [12]. Above the line within this region is where replication has significantly lower overheads compared to traditional checkpoint/restart. State machine replication is a viable approach for most checkpoint bandwidths, but with a checkpoint bandwidth greater than 30 TB/sec, replication is inappropriate for most of the exascale design space. 85

6.4 "Break even" points for replication for various numbers of replicas and a checkpoint time (δ) equal to 15 minutes. The shaded region corresponds to possible socket counts and socket MTBFs for exascale class machines [12]. Note that above the line within this region is where replication has significantly lower overheads compared to traditional checkpoint/restart. 87

6.5 Simulated application efficiency with and without state machine replication for a 168-hour application, 4-year per-socket MTBF (Θ), and 1TB/sec. checkpoint bandwidth with failure rate drawn from exponential and Weibull distributions [11]. In the replication case we have two replicas per process rank. The shaded region corresponds to possible socket counts for an exascale class machine [12]. 89

6.6 Simulated application efficiency with and without state machine replication for a 168-hour application, 12-year per-socket MTBF (Θ), and 1TB/sec. checkpoint bandwidth with failure rate drawn from exponential and Weibull distributions [11]. In the replication case we have two replicas per process rank. The shaded region corresponds to possible socket counts for an exascale class machine [12]. 90

7.1 Percent of application memory change detected using a hash-based incremental checkpointing mechanism for the CTH exploding pipe problem. The shaded region represents the average percent of memory written to using a page-protection based mechanism. 104

7.2 Percent of application memory change detected using a hash-based incremental checkpointing mechanism for the LAMMPS EAM problem. The shaded region represents the average percent of memory written to using a page-protection based mechanism. 105

7.3 Percent of application memory change detected using a hash-based incremental checkpointing mechanism for SAGE application. The shaded region represents the average percent of memory written to using a page-protection based mechanism. 106

7.4 Percent of application memory change detected using a hash-based incremental checkpointing mechanism for HPCCG. The shaded region represents the average percent of memory written to using a page-protection based mechanism. 107

7.5 A comparison of rotating `XOR` hashing rates for CPU and GPU. GPU rate includes both the copying of data to be checksummed down to the cards local memory as well as the copying of the computed keys from the card to host memory. The GPU data is the best recorded for a block size varying the number of threads and the amount of overlap in copy and computation. The CPU test use the `XOR` algorithm described previously 110

7.6 A comparison of `CRC32` hashing rates for CPU and GPU. GPU rate includes both the copying of data to be checksummed down to the cards local memory as well as the copying of the computed keys from the card to host memory. The GPU data is the best recorded for a block size varying the number of threads and the amount of overlap in copy and computation. The CPU numbers are using the Libgcrypt [13] `CRC32` hashing algorithm. 111

7.7 A comparison of `ADLER32` hashing rates for CPU and GPU. GPU rate includes both the copying of data to be checksummed down to the cards local memory as well as the copying of the computed keys from the card to host memory. The GPU data is the best recorded for a block size varying the number of threads and the amount of overlap in copy and computation. The CPU numbers are using the Libgcrypt [13] `ADLER32` hashing algorithm. 112

7.8 A comparison of `MD5` hashing rates for CPU and GPU. Note, the GPU rate includes both the copying of data to be checksummed down to the cards local memory as well as the copying of the computed keys from the card to host memory. The GPU data is the best recorded for a block size varying the number of threads and the amount of overlap in copy and computation. The CPU numbers are using the Libgcrypt [13] `MD5` hashing algorithm. 113

7.9 A comparison of SHA256 hashing rates for CPU and GPU. Note, the GPU rate includes both the copying of data to be checksummed down to the cards local memory as well as the copying of the computed keys from the card to host memory. The GPU data is the best recorded for a block size varying the number of threads and the amount of overlap in copy and computation. The CPU numbers are using the Libgcrypt [13] SHA256 hashing algorithm. 114

7.10 Per-socket commit bandwidth assuming coordinated checkpointing for a number of possible aggregate I/O bandwidths. The shaded regions correspond to the break-even checkpoint commit bandwidths from Table 7.2 for possible socket counts and *compression* values from Equation 7.3 for an exascale class machine [12] using GPU and CPU hashing. 117

A.1 MPI_Allreduce() micro-benchmark percent slowdown in comparison to the native MPI performance for the mirror protocol. 128

A.2 MPI_Allreduce() micro-benchmark percent slowdown in comparison to the native MPI performance for the parallel protocol. 129

A.3 MPI_Reduce() micro-benchmark percent slowdown in comparison to the native MPI performance for the mirror protocol. 130

A.4 MPI_Reduce() micro-benchmark percent slowdown in comparison to the native MPI performance for the parallel protocol. 131

A.5 MPI_Bcast() micro-benchmark percent slowdown in comparison to the native MPI performance for the mirror protocol. 133

A.6 MPI_Bcast() micro-benchmark percent slowdown in comparison to the native MPI performance for the parallel protocol. 134

A.7 `MPI_Alltoall()` micro-benchmark percent slowdown in comparison to the native MPI performance for the parallel protocol. 135

A.8 `MPI_Barrier()` micro-benchmark percent slowdown in comparison to the native MPI performance for the parallel and mirror protocols. . 136

List of Tables

1.1 Historical data of LINPACK and checkpoint performance for a number for machines in the top500 [14–16]. The RoadRunner and Red Storm checkpoint times are derived from memory size and parallel file I/O performance assuming 80% of memory must be written in the checkpoint. 8

7.1 Per-process checkpoint size for CTH and LAMMPS. This table contains the size of the checkpoint using standard page protection-based system-level incremental checkpointing (VM CKPT), `libhashckpt`'s hybrid approach, and an application-specific checkpointing approach (App CKPT). For the latter two columns the number in parenthesis is the percent reduction in size when compared to a system-based incremental checkpoint. The VM CKPT and Hash CKPT checkpoints contains data from both the application as well as other libraries linked with the application, for example MPI library data and its associated buffers. 107

7.2 Per-process checkpoint commit "break-even" bandwidth CPU/GPU comparison calculated using Equation 7.3 for the applications CTH, SAGE, and LAMMPS. *Compression* values for each of the applications are from Section 7.4.2 and a β_{hash} value equal to 4.0GB/sec from the GPU `MD5` hash as illustrated in Section 7.5, and a β_{hash} value equal to 500MB/sec from the CPU `ADLER32` hash. 115

Chapter 1

Introduction

1.1 Overview

Today's extreme-scale parallel supercomputers experience outages from a number of sources, including failed hardware components, software bugs, and power disruptions. Million-core machines for exascale computing will have so many parts that faults will be frequent. The system-wide Mean Time to Interrupt (MTTI) will become so small that more than 50% of an application's total execution time will be spent writing checkpoints and recovering from failures [6]. The more failures that occur during the execution of an application, the longer it will take to finish its work.

In this work, we propose a number of techniques to enhance traditional checkpoint/restart so that it remains a viable option on future extreme-scale systems. These techniques fall into two broad categories: those that reduce the frequency at which checkpoints are taken, and those that reduce the overhead of taking one checkpoint. Note that these methods are not mutually exclusive and can collectively be used to reduce the overhead of checkpoint/restart. To reduce the frequency of checkpoints, we investigate *redundant computing*, which uses a state-machine replication

scheme as well as a consistency protocol to ensure consistent state within replicas. To reduce the overhead of one checkpoint, we investigate *incremental checkpointing* and the use of computation accelerators such as graphics processing units (GPU) to speed checkpoint time. Our thesis is that state-machine replication and GPU-based incremental checkpointing can keep traditional rollback recovery as a viable option for exascale systems.

The remainder of this chapter is organized as follows. The following section provides a summary of the reliability challenges for exascale systems, offering a look at the increasing component counts for current and future systems and its implications for reliability. In Section 1.3 we briefly summarize the current practice for mainstream HPC fault-tolerance, rollback recovery, and the scalability challenges in this method for exascale systems. In Section 1.4 we offer a summary of state-machine replication, a common method used in distributed and mission critical systems to provide fault tolerance. Section 1.5 offers a solution to reduce the amount of state saved during a checkpoint: hash-based, incremental checkpointing. A summary of the main contributions of this work is provided in Section 1.6. We conclude the chapter with a roadmap for the remainder of this document in Section 1.7.

1.2 Reliability Challenges for Extreme-Scale Systems

Concern is rising in the High-Performance Computing (HPC) community on the reliability requirements of proposed and future large-scale systems. In the past, increasing numbers of processing elements have accounted for a significant portion of increased system capabilities. This trend of increased component count is expected to continue in proposed exascale systems. In this section, we examine the impact of

this on the reliability of these systems.

Increased system size for these future systems leads to systems with a very small mean time between failure (MTBF). The MTBF of a system is the mean time between two successive failures on the considered system. The MTBF of a system is equal to the sum of the mean time to interrupt (MTTI) and the mean time to repair (MTTR). For a system of identical components, the MTBF of the system (Θ_{system}) is inversely proportional to the number of sockets (N) and is defined as:

$$\Theta_{\text{system}} = \frac{\Theta_{\text{socket}}}{N} \tag{1.1}$$

where Θ_{socket} is the MTBF of a socket in the system. Reliability statistics from a number of top supercomputing centers [6–11, 17] also show that the MTBF of HPC systems shrinks proportionally with the number CPU sockets in the system.

Examining historical socket count data from the Top500 [1] supercomputer site, we see that socket counts have been steadily increasing. Figure 1.1 shows the socket count statistics for the five fastest machines in the world since 1993. From this figure, we see the progression of increasing socket count with time. Three future machines included in this figure and announced to be released in 2010 and 2011 continue this progression of increasing socket counts: Cielo [3] with 18,000 sockets, ACS BlueWaters [2] with nearly 40,000 sockets, and Sequoia [4] with 125,000 sockets. This steady increase of socket counts for the top capability-class machines, coupled with the fact that individual CPU reliability has stayed nearly constant in the past 10 years [18], suggests that we will soon reach a tipping point where the MTBF will be so low that the application will be unable to make forward progress.

Figure 1.2 show this same data from 2001, along with the three aforementioned proposed systems and a proposed "Aggressive"proposed exascale architecture [5],

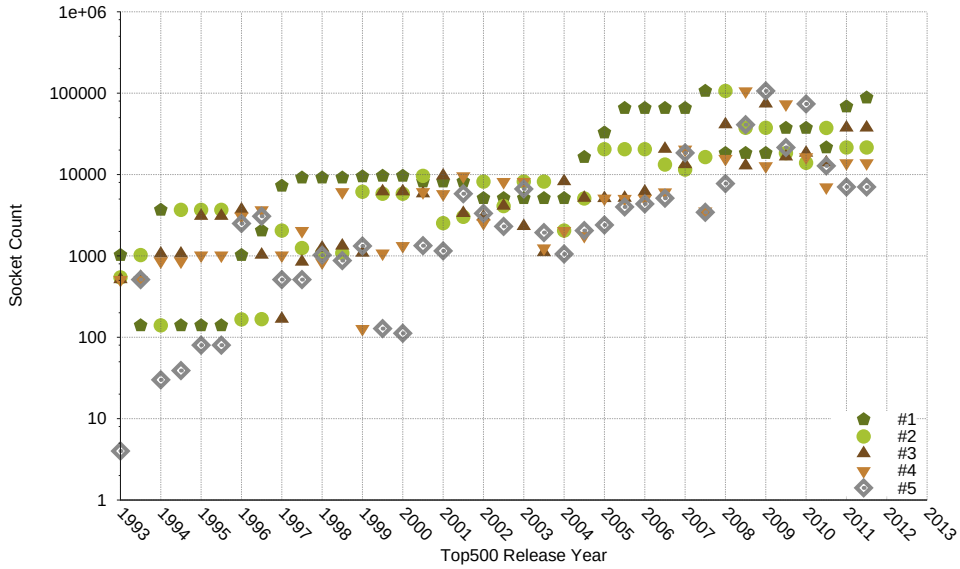


Figure 1.1: Socket counts for the five fastest machines topping the Top500 [1] list for the past two decades. Also included are two proposed systems due to become operational in 2010 and 2011, ASC BlueWaters [2], Cielo [3], and Sequoia [4].

an upper-bound "strawman" architecture for a 2018 exascale machine. The shaded region in this plot represents the range of possible socket counts fitting the past data to a line and extrapolating out to 2018. From this we see the possibility of a jump in socket count for future extreme-scale machines. The reason for this possible jump is related to power and memory bandwidth constraints on these systems [5]. All this suggests that socket counts will increase to an unprecedented level.

Finally, Figure 1.3 illustrates the impact of system MTBF for machines at scale using Equation 1.1. Recent studies [6–11] have placed the socket MTBF for current systems to be between 5 and 25 years. It is important to note that it is not clear these current socket MTBF's will increase; the CPU market is typically not driven by the HPC community and current MTBFs are adequate for the consumer and enterprise markets. In this figure, we see that at the scale of next-generation large scale machines, the system MTBF will decrease to an hour and in some cases minutes. This

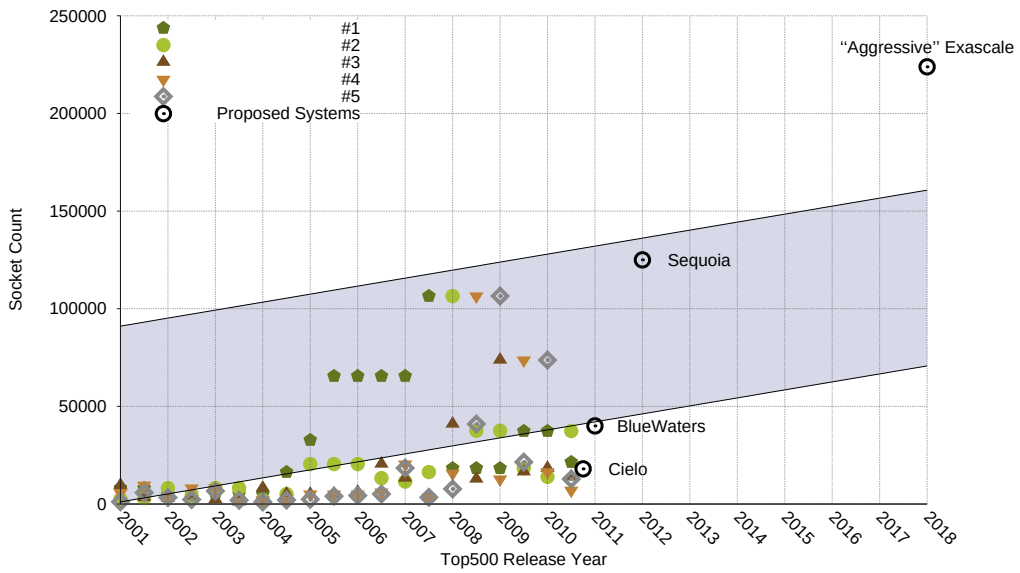


Figure 1.2: Socket counts for the five fastest machines topping the Top500 [1] list from 2001 to 2010. Also included are upcoming systems and a proposed "Aggressive" exascale system [5]. The shaded region in the figure shows the range of socket counts using past data for a line fit.

is important as the rapidly shrinking MTBFs typically impact application progress and scalability. In fact, if the MTBF is less than the mean time to repair (MTTR), no progress can be made at all.

1.3 Rollback Recovery

1.3.1 Overview

A common method to allow an application to continue in the presence of faults, checkpoint/restart saves application state at regular intervals and restarts the application from the most recent successful checkpoint after a fault occurs. In this section we summarize the scalability challenges for traditional rollback recovery for

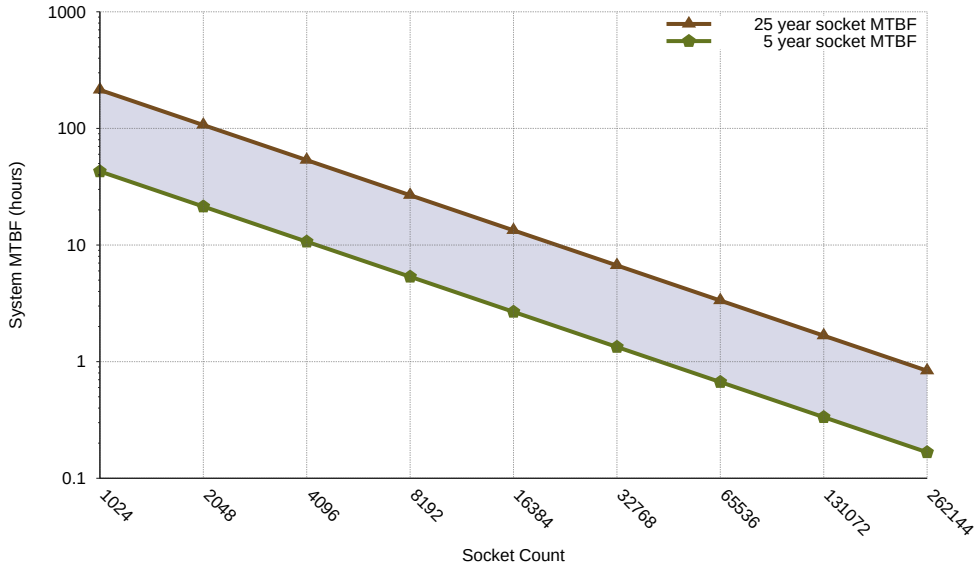


Figure 1.3: System MTBF values expected from large-scale exascale systems. Current studies [6–11] place the node MTBF somewhere between 5 and 25 years, which is expected to continue in the future. Next-generation systems are expected to see multiple faults per hour.

large-scale systems.

In rollback recovery, once a failure has been observed the current execution is stopped and execution is restarted from the last known good snapshot of the application's state. These last known good state snapshots are called checkpoints. To avoid rolling back execution of the application to the beginning, checkpoints are saved repeatedly throughout the lifetime of the application. Rollback recovery has a number of costs associated with it, most notably the time to save a checkpoint safely to stable storage. Performance is also affected by how frequently checkpoints are taken, but finding an optimal checkpoint frequency can sometimes be difficult.

Checkpointing often ensures that little work is lost at the expense of spending an increasing amount of time computing checkpoints and thereby halting application forward progress. Checkpointing rarely ensures high application efficiency and

forward progress at the expense of restarting from a point far in the past on failure, recomputing a large amount of work.

1.3.2 Scalability of Rollback Recovery

The use of coordinated checkpoint/restart as the primary HPC fault tolerance technique relies on two key assumptions that may not be true in future exascale systems:

1. **Application state can be saved and restored much more quickly than a system's mean time to interrupt (MTTI).** Upcoming systems are projected to have several orders of magnitude more components than current systems and suffer faults much more often (see Figure 1.2). In addition, the increasing disparity between available I/O bandwidth and memory density continues to increase the time it takes to write a checkpoint. Both of these are leading to exascale systems with MTTIs much less than the time it takes to write a checkpoint.

2. **Faults that result in Byzantine-type failures are very rare.** Current systems are already beginning to suffer from faults that lead to incorrect application results instead of crash failures [19], for example undetected DRAM errors in application memory. Because of the dramatic increase in component count in exascale systems, such faults will become increasingly common.

Table 1.1 shows the LINPACK R_{max} performance as well as estimated or measured checkpoint times for a number of machines in the Top500 [1] over the past 10 years. From this table we observe that, historically, the fastest machines in the Top500 take 20 to 30 minutes to perform a checkpoint. This stability over time of checkpoint time is due to a number of reasons: the balance between node memory density and I/O bandwidth to stable storage and the organization of the I/O system

Year	System	LINPACK Performance (TFLOPS)	Checkpoint (δ) (min.)
2010	Jaguar	1,759	26
2009	RoadRunner	1,105	∼20
2008	BlueGene/L	500	20
2007	Red Storm	100	∼20
2006	Zeus	11	26
1999	ASCI Red	2.3	∼20

Table 1.1: Historical data of LINPACK and checkpoint performance for a number for machines in the top500 [14–16]. The RoadRunner and Red Storm checkpoint times are derived from memory size and parallel file I/O performance assuming 80% of memory must be written in the checkpoint.

remains relatively identical in the systems above. This I/O subsystem organization is expected to remain in future large-scale systems.

Thus, from Table 1.1 we see that the time to take a checkpoint, around 20 to 30 minutes, plus the time to perform a restart, also 20 to 30 minutes, will take a total between 40 to 60 minutes. Referring back to Figure 1.3, we see that at the expected scale, the predicted reliability of an exascale machine [5] will be approximately an hour or less. Therefore, checkpoint/restart will be unable to make progress as most of the time will be spent writing checkpoints, recovering from faults reading restart files, and performing rework from the last checkpoint.

1.4 State-Machine Replication

The first technique we study for addressing these resiliency challenges for exascale is replication. Process-pair fault tolerance [20, 21], also referred to as *state machine replication* [22], is a well-known technique for tolerating faults in large-scale distributed and mission-critical systems. In this technique, a process's state and

computation are replicated across redundant hardware nodes. Replication allows systems to tolerate crash failures resulting from system faults; in particular, a failed process's replicas handle its responsibilities until it can be restarted from an existing replica on new or repaired hardware. Though costly in terms of hardware requirements, at least doubling the number of nodes required, it allows a computing system to continue to execute unimpeded in demanding environments with frequent failures. In addition, variants of this technique can also be used to handle faults that do not crash a node but instead cause it to yield incorrect results [23].

Replication has not traditionally been used in high performance computing systems, and only examined in a very limited sense, primarily due to its cost [24]. Instead, HPC systems have primarily relied on a combination of the rollback recovery techniques described in Section 1.3 and a number of fine-grained hardware error detection and correction techniques to allow large parallel applications to scale to over a petaflop of sustained performance.

We study the use of replication to supplement checkpoint/restart and extend the validity of its underlying assumptions to exascale systems. In particular, replication could be used to increase the *effective* system MTTI by allowing applications to continue executing as long as one replica of every process remains alive. This would allow applications to reduce the frequency with which they write checkpoints and increase the time available to write checkpoints to stable storage. Similarly, replication could be used to detect faults that silently corrupt application state by comparing replica state periodically (e.g. at checkpoint time). This would allow applications to detect the effects of such faults and recover to a previous checkpoint instead of wasting cycles computing a worthless results.

1.5 Hash-Based Incremental Checkpointing

The second technique we study to reduce the overhead of checkpoint/restart is hash-based incremental checkpointing. This variant of incremental checkpointing uses secure hashing to reduce checkpoint sizes by saving only the application state that has changed in the checkpoint interval.

As stated previously, the act of saving checkpoints, referred to as *committing* checkpoints, adds overheads to the application's running time. This overhead is due to the time it takes to copy the process state (which in some cases could be multi-gigabytes in size) to stable storage and possibly network bandwidth as thousands to hundreds of thousands or more processors would need to write their state over the network.

One known optimization to reduce the amount that needs to be saved is incremental checkpointing. Typically, this technique involves using the operating systems page protection mechanism to determine which pages have been written to, termed *dirty*, since the last checkpoint has been saved. Upon restart, the state restored is the original checkpoint with all the incremental differences applied in order.

One drawback to incremental checkpointing is that it operates at the granularity of the operating system page size. Therefore if just one bit on a page has changed, the entire page must be checkpointed. In fact, if the page is written with identical values it is still marked as dirty and must again be checkpointed. This problem is further compounded by the increasing maximum page sizes of modern processors and the increased performance for HPC applications on these larger page sizes.

To address this drawback this work investigates a hybrid incremental checkpointing solution that uses both the OS page protection mechanism and a hashing scheme to determine which locations within a page have changed. To address this we investi-

gate the use of GPUs to offload the hash computation. This approach has previously been proposed [25,26], but dismissed as being too computationally expensive [27,28].

1.6 Contributions

This work makes several important contributions in the field of fault-tolerance for exascale HPC systems. These include:

- A model to determine the expected number of failures absorbed, and corresponding MTTI increase, for an application using state-machine replication;
- An MPI-based implementation of application transparent state-machine replication outlining the consistency model required to meet MPI semantics;
- A GPU-assisted incremental checkpointing library that can determine the minimal state change in an application; and
- An evaluation of these techniques using a number of important HPC workloads, with guidance on the viability of checkpoint/restart for future extreme-scale systems.

A summary of these contributions is provided below with more details following in subsequent chapters.

1.6.1 Modeling Replication

We develop a model for state-machine replication using the the birthday problem. The birthday problem [29–32], sometimes referred to as the birthday paradox, is a common problem in probability theory. This model allows us to formulate the

expected number of faults a replicated system can sustain before the application sees an interrupt.

1.6.2 *r*MPI: A Transparent MPI Replication Library

We develop a portable, transparent replication library called *r*MPI. This library utilizes the MPI profiling layer [33] to enable redundant computation for MPI applications. In this work we detail the design of this library and a number of the protocols used to keep state consistent across the replicas. In addition, we directly quantify the cost of the *r*MPI library on a range of micro-benchmarks and real-world, large-scale applications.

1.6.3 Hash-Based Incremental-Checkpointing using GPUs

We develop a hybrid incremental checkpointing solution that uses both OS page protection mechanisms and a hash mechanism to determine which locations within a page have changed. This allows us, with no knowledge of the application, to determine the minimal amount of changed state in a checkpoint interval. In addition, we use graphics accelerators to perform the hashing and evaluate the advantages of using these GPU's over CPU's for a number of hashing algorithms.

1.6.4 Analysis of Checkpoint/Restart Viability for Extreme-Scale Systems

Lastly, in this work we examine the viability of using state-machine replication as the *primary* exascale fault tolerance mechanism, with hashed-based incremental checkpoint/restart providing *secondary* fault tolerance when necessary. Using the afore-

mentioned model, we show this fault-tolerance mechanisms *"break-even" point* (the point in which the nodes hours used for this method is less than a competitive method) is less then the projected sizes of next-generation exascale systems.

1.7 Document Organization

The remainder of this dissertation is organized as follows: In Chapter 2, we present background information and related research in reliability metrics, traditional roll-back recovery and other checkpointing methods, forward error recovery – most notably state machine replication, fault-tolerant algorithms, and proactive migration. Chapter 3 describes a model- and simulation-based approach to show the benefits of state-machine replication for exascale-class systems. In Chapter 4, we describe a library that implements state-machine replication in the HPC environment, *r*`MPI`. *r*`MPI` is a portable, transparent replication library implemented at the MPI profiling layer. We outline *r*`MPI`'s basic architecture, consistency requirements and protocols. In Chapter 5 we show the runtime overheads of the *r*`MPI` library on a set of standard micro-benchmarks and a number key capability HPC workloads. Combining the results from the previous chapters, we show the viability of our replication approach in Chapter 6 on a number of exascale system parameters, including CPU failure rates, aggregate checkpoint I/O bandwidths, and various degrees of redundancy. Chapter 7 looks at the viability of a hash-based incremental approach using GPUs to reduce checkpoint saved state and therefore commit times for extreme scale systems. We conclude with a summary of our research contributions and future directions in Chapter 8.

Chapter 2

Background and Related Work

This section describes previous research as it relates to fault tolerance for high-performance computing. First, we carefully define what is meant by faults and failures, as well as the the metrics used to describe the reliability of an HPC system. The remainder of the chapter outlines a number of popular methods used to ensure progress of applications across faults. This includes traditional rollback recovery and its various optimizations, i.e. incremental checkpointing and asynchronous checkpointing with message logging. The chapter concludes with a discussion of state machine replication and application algorithms that are resistant to faults.

2.1 Failures, Faults, and Associated Models

As this work deals with fault-tolerance for emerging exascale systems, it is important to carefully define what is meant by faults, errors, and failures. Generally a *fault* refers to the unexpected behavior or defect in the system at its lowest level [34, 35]. An example of a fault could be a memory cell which is stuck to the value "0". Faults are classified as reproducible if they always reoccur or non-reproducible otherwise.

Our previous stuck-to-zero memory cell example would be classified as reproducible assuming the defect which ties the cell at zero is not transient. In contrast to fault, a *failure* or *error* is the externally visible manifestation of the fault to the end user. This failure or error is a deviation of the system from its specification [34]. Returning back to the memory cell example, one possible error from this fault could be an incorrect arithmetic operation which uses a value previously stored at the location. This incorrect value can cause an error or failure of the system. In scenarios where there is no distinction between a fault and the corresponding failure, these terms are used interchangeably.

There are a number of different causes of failures in HPC systems. Design errors are those failures in which the system was not designed to correctly perform the expected behavior. Design errors include both software (bugs, race conditions, deadlocks, etc.) and hardware design errors. Additionally, failures can be attributed to system overloading (e.g. denial-of-service attack) and age and stress induced wear down of components. A distinction is typically made between *hard* and *soft* errors. A hard error is typically related to a component in the system that no longer functions properly and is therefore non-transient, while soft errors generally refers to those errors which are transient and can be resolved by resetting the system.

Traditionally, when talking about faults we describe the effect of a fault by describing the resulting behavior of the system when the fault has occurred. These behaviors are typically grouped in a hierarchic structure called *fault models* [34,36,37]. Correctness and cost of proposed fault-tolerant approaches are evaluated with respect to a specific fault model. The most popular of these models include:

Fail-Stop Processors stop executing but this failure can easily detected by its neighbors.

Crash Processors simply stop executing but neighbors may be unable to detect

when the error has occurred.

Byzantine Processors continue functioning but may behave in an arbitrary and sometimes malevolent manner [38–40].

In this hierarchy, Byzantine is the most general as it includes the behavior of all other models listed and fail-stop is the most limited as it requires malfunctioning units to no longer function and all other units atomically be made aware of the failure, which may not be realistic for the majority of failures seen on actual hardware.

2.2 Reliability Metrics

A number of metrics are used in addressing the resilience of large-scale HPC machines. The reliability of a component describes the probability that component will perform its intended function during a specified period of time. The metrics failure rate, Mean Time Between Failure (MTBF), Mean Time To Interrupt (MTTI), and Mean Time To Repair (MTTR) are fundamental terms used to describe the reliability aspects of a component or system.

The failure rate (λ) is the frequency with which a component experiences faults. The mean time between failures (MTBF) is the mean time between two failures of the considered system. This time is equal to the sum of the mean time to interrupt (MTTI) and mean time to repair (MTTR). Sometimes MTTI is replaced by the mean time to failure (MTTF).

Note that these reliability metrics do not include any information about the methods used to recover from failures or how efficient these methods are. We address recovery methods in the following sections.

2.3 Traditional Rollback Recovery

2.3.1 Overview

The presence of failures in the hardware or software of parallel computers has created a requirement for the use of fault tolerance mechanisms in order to make sure that the application finishes successfully. In case of failure, the application stops or terminates with incorrect results because it has not been designed to handle these error situations. To ensure progress across faults, a number of techniques have been created. The most common of which, described in this section, is rollback recovery.

Rollback recovery, or *backward-error*, protocols [41–45] have been the dominant fault tolerance mechanism in HPC for over 20 years. In rollback recovery, the process and/or communication state [46] is periodically saved to stable storage. Upon failure, the system rolls back computation to the last known good state saved to stable storage. In this protocol, the amount of work lost upon failure is the work computed since the last known good state was saved. The two main variants of rollback recovery are checkpoint and log-based protocols.

2.3.2 Traditional Checkpoint/Restart

Traditional checkpoint/restart has been the dominant fault tolerance mechanism in high performance computing systems for at least the last 30 years. In current systems this approach generally works by saving the state of the application periodically throughout the application's computation. When a failure has been reached, the last known good state is read from stable storage and computation is continued from that state.

Coordinated Checkpoint/Restart

Coordinated checkpoint/restart ensures the consistency of checkpoints by synchronizing all nodes before writing a checkpoint. This method generally works as follows:

1. Applications periodically quiesce all activity at a global synchronization point, for example a barrier;

2. After synchronization, all nodes send some fraction of application and system state, generally comprising most of system memory, over the network to dedicated I/O nodes;

3. These I/O nodes store received checkpoint information data to stable storage, currently hard disk-based storage;

4. In the event of an application failure, the stored checkpoint is used to restart the application at a prior known-good state.

This synchronization limits the domino effect [47] and ensures all checkpoints are globally consistent and only the last successfully saved checkpoint needs to be kept. This synchronization can be done by either quiesing the communication state before writing the checkpoint [41,48–50] using, for example, a barrier operation or by saving the communication state during the operation using a more complex non-blocking protocol [41,46,51,52]. In addition to synchronization protocols, using synchronized clocks to coordinate checkpoints has also been investigated [53–55]. This technique has rather limited applicability due to the difficultly of fine-grained synchronization on distributed memory, large-scale machines.

Modeling Coordinated Checkpoint/Restart Performance

In [56], Daly presents a validated model for coordinated checkpoint/restart assuming exponential failures. There are a number of similar first-order models in literature [56–60]. We choose Daly's model due to its accuracy above the others. In addition to the checkpoint model (presented here in Equation 2.1), Daly also derives an optimal checkpoint interval $\tilde{\tau}_{\text{opt}}$ (Equation 2.2).

$$T_w(\tau) = \Theta e^{\frac{R}{\Theta}}(e^{\frac{\tau+\delta}{\Theta}} - 1)\frac{T_s}{\tau} \quad \text{for} \quad \delta << T_s \tag{2.1}$$

$$\tilde{\tau}_{\text{opt}} = \begin{cases} \sqrt{2\delta\Theta}\left[1 + \frac{1}{3}\left(\frac{\delta}{2\Theta}\right)^{\frac{1}{2}}\right. & \\ \quad \left. + \frac{1}{9}\left(\frac{\delta}{2\Theta}\right)\right] - \delta & \text{for } \delta < 2\Theta \\ \Theta & \text{for } \delta \geq 2\Theta \end{cases} \tag{2.2}$$

Where:

δ Time to write one checkpoint

R Time to do a restart

Θ Mean time between failures for the system

τ Interval between successive checkpoints

T_s Application solve time with no overhead from checkpoint/restart

T_w Wallclock solve time including checkpoint/restart overheads

This model describes the wall clock time of given workload on a proposed exascale system which includes the overheads of writing checkpoints as well as performing restarts after faults. In this work, we use this validated model to evaluate the performance of checkpoint/restart at the scale of proposed exascale machines, a scale several orders of magnitude larger than what is currently available today.

2.3.3 Optimizing Rollback/Recovery

The overheads associated with traditional rollback recovery have been known and studied extensively in the past twenty years [6, 26, 41, 52, 61–66]. Due to this known overhead, a number of techniques have been proposed to decrease the costs while still delivering the fault tolerance benefits [41]. These techniques include incremental, copy-on-write (COW) or *forked* checkpointing; uncoordinated checkpointing; communication-induced checkpointing; and asynchronous checkpointing with message logging.

Incremental Checkpointing

Incremental checkpointing [25,26,41,67–73] decreases the overhead of taking a checkpoint by reducing the amount of application state or data saved to stable storage at each checkpoint [74]. Incremental checkpointing reduces the amount of state saved by only saving that state which has changed since that last checkpoint has been written. A variety of methods have been used to determine which state has changed, from compiler based to techniques [67] based on saving dirty virtual memory pages [68,69].

Hash-based Incremental Checkpointing

Hash-based Incremental Checkpointing, sometimes referred to as *probabilistic checkpointing* [25], is a system-based checkpoint method that attempts to minimize the state saved in a checkpoint and therefore optimize checkpoint commit times. This technique uses computational hash algorithms to determine the portions of a process' address space that has changed in a checkpoint interval, rather than the dirtied pages used in standard incremental checkpointing. Another key feature of this method is the ability to allow finer-grained detection of dirtied blocks than is currently possible

using mechanisms based solely on page protection mechanisms. This approach has previously been dismissed as being to computationally expensive [27, 28] to reap the meager benefits in state compression.

With a probabilistic hash-based approach *aliasing* is a concern. Aliasing, also referred to as collisions, comes about when modifications to a block are just such that the key values are identical. The danger with aliasing is the library will not save modified application data, thereby corrupting the application in the event of a restart. Previous studies have shown the likelihood of aliasing to be higher in practice then expected theoretically for a number of hash functions. Specifically, with the hash signature functions `CRC32` and `XOR`, the probability of collision has been shown to be too high to be considered safe [27]. Secure hash signatures like `MD5` and `SHA256`, however, have been shown to behave in practice as expected theoretically, and are therefore reliable enough to be used in a hash-based approach [28].

Recently, Agarwal et al. [26] investigated the performance characteristics of a hash-based adaptive incremental checkpointing library. The authors use an MD5 hash to determine the portions of an application address space that have changed in a checkpoint interval. This work failed to evaluate the merit of this hash-based technique on actual HPC capability workloads, instead using micro-benchmarks. In addition, the authors failed to evaluate the merit of this technique compared to application-specific checkpoint mechanisms that exist in many capability workloads.

Copy-on-write Checkpointing

Copy-on-write checkpointing, or forked checkpointing [43, 71, 73, 75–77] decreases the overhead of taking a checkpoint by having a background process save application state to stable storage while the original process continues computation. This allows both the checkpoint process and original process operate concurrently. If the

underlying system supports virtual memory page copy-on-write semantics, both the checkpoint process and the original process share the read only and unmodified writable pages of the process, thereby reducing the memory overheads to be only those memory pages that have been written to since the checkpoint was initiated.

While this method allows for the simultaneous execution of the checkpoint and application processes, there are associated costs. As outlined above, the memory footprint of this method is greater than traditional blocking checkpoint/restart. Even if copy-on-write is supported, this footprint difference can still be quite large in today's data-intensive applications. In addition, the page copy operation can be an expensive operation. Lastly, the concurrent checkpoint process typically uses valuable memory bandwidth, a known limiter of HPC application performance.

Remote Checkpointing

Remote checkpointing [78–80] saves checkpoints to remote resources, leveraging network resources on the nodes. This methods allows for performance advantages in environments where network bandwidth is greater than I/O bandwidth to local storage devices or environments where local storage is either not available or too small to save application state. As this checkpoint data is saved remotely, the data exists in the system even if the corresponding node has failed. This method seems to be losing favor with the advent of fast, inexpensive stable storage, for example solid-state disk (SSD) devices.

Uncoordinated Checkpoint/Restart

In contrast to coordinated CPR, in uncoordinated checkpoint/restart [51, 81–83] the processes checkpoint their state independently of each other. As these checkpoints are saved independently, ensuring a globally consistent checkpoint can be quite difficult.

Due to this difficulty, all local checkpoints must be saved. Upon failure, the runtime system must examine all checkpoints to compute a globally consistent checkpoint if it exists. If no such globally consistent checkpoint exists, computation can rollback execution to the beginning of execution, the so-called *domino effect* [47].

Communication Induced Checkpoint/Restart

A hybrid of both coordinated and uncoordinated checkpoint/restart, communication-induced checkpointing [84–89] attempts to avoid useless checkpoints. In communication-induced CPR, processes take independent checkpoints but must also take checkpoints based on the communication patterns of the application. This communication-induced checkpoint protocol is piggybacked on the application's messages. There are two main approaches for when these communication-induced checkpoints must be taken; model-based and index-based protocols. Model-based protocols [84–86] attempt to prevent saving useless checkpoints by protecting the patterns of checkpointing and communication that create them. Index-based protocols [88, 89] on the other hand, use ordering techniques such as logical clocks [87] to ensure no useless checkpoints are created.

Asynchronous Checkpointing with Message Logging

Asynchronous checkpointing with message logging, similar to uncoordinated checkpointing, [90–97] attempts to improve checkpoint performance by avoiding the synchronization that ensures a consistent checkpoint. In these systems, nodes generally checkpoint and restore from local storage without the synchronization used by coordinated checkpointing. To support a node restoring from a local asynchronous checkpoint, nodes in this approach keep a log of recent messages that they have sent. When a node restores from a previous checkpoint, it can then replay reception of

messages using a remote nodes log.

While this approach can increase checkpointing performance, it also generally assumes the availability of local storage. In addition, logging increases the latency of messaging operations and potentially takes significant amounts of space. Finally, asynchronous checkpointing approaches can result in cascading rollbacks; recent work attempts to bound the amount of rollback that may be necessary [98], but also places non-trivial limits on application behavior. We are unaware of any studies examining performance of message logging approaches at large scales (e.g. thousands of nodes or larger).

2.3.4 High-speed Storage for Checkpoint/Restart

High speed local storage, for example local disk and flash memory systems, has periodically been proposed to speed up checkpoint/restart systems by placing large amounts of high-speed storage near the data that must be checkpointed. The Exascale planning report [12] notes that placing spinning storage and a flash RAM in each system node would allow nodes to checkpoint in between four minutes and one second. This would in turn increase system utilization to from 59% to 97% ([12], Table 7.12, revised using Daly's second order model.)

However, deploying large amounts of local non-volatile storage in an exascale system is potentially very challenging. Local disk-based storage has traditionally been avoided because of the increased failures it may cause. Upcoming non-volatile phase change PCRAM and resistive RRAM devices provide high bandwidth and reliability, but are potentially very expensive. Unless their cost per bit rivals that of DRAM, using such technologies for checkpoint/restart purposes would result in checkpointing hardware that makes up a much larger portion of the system cost.

Modern NAND and NOR flash technologies are potentially the most promising

for buffering and storing local checkpoints because of their comparatively low cost, high density, and high reliability. NAND flash write bandwidths are currently in the low GB/s range, allowing them to checkpoint a node in a few minutes. Assuming that exascale MTTIs can be kept at or above one hour, this would result in system utilization of 80% or higher. However, their write durability would require periodically replacing all flash memory in the system.

2.4 Other Checkpointing Systems

Memory-based checkpointing [99–103] uses the the memory of a remote machine to checkpoint node state. Unless node memory is primarily read-only (in which case RAID 5-like techniques can be used), this approach doubles the memory demands of an application. Since memory is regarded as a key budget and power constraint in exascale systems, it is unclear if the benefits of replicating only memory are superior to the qualitative advantages of state machine replication described in this dissertation.

Multi-level checkpointing [80] is a library-based approach for controlling checkpointing to multiple storage targets, including memory-based checkpoints, checkpointing to local storage, and remote checkpoints, into a single system. Because of this, it shares some of the advantages and disadvantages of memory-based checkpointing and local storage techniques. Unlike these techniques, however, multi-level checkpointing has the flexibility to choose between multiple levels of storage based on system design parameters, making it a promising technique for exascale systems.

Lastly, hardware-based mechanisms utilize specialized hardware primitives to automatically perform checkpoint and logging of machine state [104, 105]. These memory-based hardware approaches share the speed advantages of memory-based checkpointing with only very modest hardware modifications. These modifications

typically include redundant storage in memory for the saved checkpoints and a directory-based cache controller. As these method requires hardware modifications and replication in memory, again it is unclear if this method is appropriate for large-scale systems.

2.5 State Machine Replication

2.5.1 Overview

Redundant computation, process replication, and state machine replication have long histories and have been used extensively in distributed [22, 97, 106–116], mission critical [20–22, 117], and storage systems [118–121] as a technique to improve fault tolerance. In state machine replication, one or more replicas of each process is maintained and every node computes deterministically in response to a given external input, for example a message being received. This technique then uses an ordering protocol to guarantee all replicas see the same inputs in the same order, and additional communication to detect and recover from failures. Where there are disagreements, output correctness may be performed using reliable quorum algorithms.

State machine replication offers a different set of trade-offs compared to rollback recovery techniques such as checkpoint/restart. In particular, it completely masks a large percentage of system faults, preventing them from causing application failures *without the need for rollback*. Some forms of state machine replication can also be used to detect and recover from a wider range of failures than checkpoint/restart, including Byzantine failures [23]. Unlike checkpoint/restart, however, state machine replication is not sufficient by itself to recover from all node crash failures; faults that crash all of a node's replicas will cause a computation to fail.

This approach has previously been dismissed in HPC as being too expensive for

the meager benefits that are seen at present machine scale [122–124]. For the reasons described in earlier this chapter regarding its scalability, however, several authors have recently suggested using this technique in HPC systems [7, 125, 126]. In later chapters, we examine the suitability of a specific type of state machine replication in HPC systems.

2.5.2 Passive versus Active Replication

As stated previously, in state machine replication the entity being replicated is a process. Two replication strategies have been used for replicating this process: active and passive replication. In passive replication [106,108] there is only one process that handles events. This process is called the leader or primary process. After processing a request, the leader updates the state on the other backup replica processes and sends back the response to the client. In active replication [97, 106, 108, 112] each request is processed by all replicas. To ensure all the replicas receive the same sequence of operations, an atomic broadcast protocol or *group communication* must be used. This group communication protocol [127–130] guarantees that either all the replica receive an event or none, and that they all receive events in the same order.

2.5.3 Group Communication

As described in the previous section, state machine replication involves communication and coordination among a set of replicated processes. Algorithms that coordinate the groups in the replica set are commonly referred to as *group communication* algorithms. These algorithms typically deal with reliable delivery and consensus issues such as consistency (all members of the group agree on a value) and liveness (all processes in the group eventually make progress).

The most prominent group communication mechanism in distributed systems is Lamport's Paxos consensus algorithm [130]. This distributed algorithm ensures consensus in a network of unreliable processes. Like most distributed consensus algorithms, this algorithm works on the idea that all members propose their output to the entire group, or a subset, and then coordinate to decide which output is correct. There is a great deal of previous work that focuses on the semantics, correctness, efficiency, and adaptability of group communication in financial and mission critical applications [127–129, 131–136].

One distributed consensus algorithm used in state machine replication is referred to as a total order broadcast. A total order broadcast algorithm [127, 128] provides reliable delivery of messages within a group in the same order for all processes. This ordering mechanism typically has an associated performance cost. For example, a message may not be delivered to a process until all other processes in the group have agreed upon its delivery. This cost is typically an increase in message latency.

Approaches widely used to implement total message ordering include sequencer, privilege-based, and communication history algorithms. In sequencer-based total ordering, one group member is responsible for the ordering and reliable delivery of messages within the group. A fail-over mechanism for the sequencer assures fault-tolerance for this mechanism. Example systems which utilize sequencer algorithms include the Amoeba distributed operating system [129] and the Isis communication system [137]. Privilege-based total ordering algorithms rely on the idea that group members can reliably broadcast only when granted to do so. For example, in the Totem protocol [138], a token is rotated among the replica group and only the holder can reliably broadcast. A token time-out ensures liveness in the system. Lastly, in communication history algorithms, messages are reliably broadcast by any member, at any time, without an a priori order. Total message order in the system is ensured by delaying message delivery until delivery information is gathered from other

members of the group [139].

These three ordering approaches have significant advantages and disadvantages. Sequencer and privilege approaches provide good performance when the system is relatively idle. When multiple group members are active and constantly broadcasting, however, the latency is limited by the time for the sequencer to produce the ordering or circulate the token. Communication history algorithms increase latency to detect the "happened before" [87] relationship between messages. This delay typically depends on the slowest group member.

Several studies have attempted to reduce the cost of these approaches. Early delivery algorithms [131, 132] reduce latency by reaching agreement with a subset of the process group; optimal delivery algorithms [133, 134] deliver messages before the total message ordering has been determined but notify applications if the final total order is different than that of the delivered order.

2.6 Other Forward Recovery Methods

In contrast to rollback recovery, forward recovery avoids restarting the application from a previously saved, known-good state by recovering on its own to a state corresponding to a normal, fault-free execution. Applications must typically be designed specifically with this forward recovery property. These algorithms eventually converge towards a correct state in the presence of perturbations or failures of any kind.

One well known class of forward-recovery algorithms is refered to as self-stabilizing codes [140]. An algorithm is self-stabilizing if, independent of each component's initial state, it arrives to a correct working state in a finite amount of time. To do so, this class of algorithms must assume that errors are transient and can occur in any part of the system. In these algorithms, there is typically a phase between the

receiving of an error and its stabilization where the algorithm may provide incorrect results. It is the responsibility of the user to handle this non-stabilized window.

While it is not clear if this forward recovery method is viable for HPC applications, it has been shown to have applications in collective communication libraries and runtime environments. As an example, Geist and Engelmann [141] proposed a forward recovery method for computing a global maximum in the presence of faults. While this algorithm correctly computes the maximum from the live nodes in a distributed system, its computation time is unbounded.

This same time-bound limitation applies to self-stabilizing algorithms: they can be used for collective operations but the time in the stabilization phase is unknown. In fact, most current self-stabilizing algorithms cannot tolerate failures during the stabilization phase. Due to the fact that these algorithms do not produce exploitable results during the stabilization phase, they cannot be used as they are in situations where the system suffers from very frequent failures (if the inter-failure period is shorter than the stabilization time). This is a very important limitation for numerical algorithms and HPC applications for exascale systems.

2.7 Fault-Tolerant Algorithms

The notion of application-based fault tolerance is to design computation algorithms that either ignore failures and still deliver a correct answer, or are able to recover using techniques such as redundant data or computation. An underlying requirement of these algorithm-based approaches is that the underlying system software is also capable of continuing in the presence of faults [142–146].

2.7.1 Application-based Data Redundancy

One recent mechanism for application fault tolerance is data redundancy. This method works by encoding redundant data into the problem such that data from failed nodes can be recomputed. In addition, the algorithm is modified to update the encoding as computation progresses. In general this method is adjustable by the encoding algorithm used such that a specified number of failures can be tolerated at a time. Recent results using this technique show this method can be used with a very low performance overhead [147–149].

2.7.2 Application-based Computational Redundancy

In contrast to the data redundancy method described in the last section, computation redundancy relies on the algorithm-specific relationship between the parallel application and its individual data chunks. If data is lost due to a failure, this impacts the result by possibly increasing the margin of error or by running the surviving nodes for longer until the problem has converged. Therefore, the number of nodes lost determines the application time-to-solution or margin of error.

Recently, Engelmann and Geist [150] used chaotic relaxation and meshless methods to ensure progress in the presence of faults. The authors showed that the convergence of a finite difference code is not significantly affected if the number of failed nodes is less than 1% of the total number of nodes. Though these algorithmic methods show promise, they have not been tested extensively and there is concern that they may not be applicable to all applications.

2.8 Proactive Migration

A recently proposed fault-tolerance method, proactive migration (or fault avoidance) [151, 152, 152–155] allows applications to survive faults by migrating when a fault is imminent. In contrast to the traditional reactive fault handling techniques, proactive migration utilizes reliability models based on historical events and current system health status information in order to avoid application faults. For example, a process may be temporarily migrated when it displays behavior that is similar to a component that is about to fail, such as increases in temperature or unusual communication errors.

This method is dependent on accurate fault predictor models. Therefore, much of the research in this area is in the development of these predictive models and algorithms and validating these models using limited reliability log data [151, 153, 155]. There is great concern on the accuracy of these predictors for next-generation systems. This is due to the fact that all available fault trace data is either not representative of a production system or extracted from system that are orders of magnitude smaller than proposed exascale systems.

Independent of the predictors' accuracy, this method must handle the scenario where a fault arrives before the application can be migrated. Therefore, this method must be combined with a method such as checkpoint/restart to handle uncaught error state. If failures can be predicted with great accuracy, these methods can increase the checkpoint interval and therefore lower its overhead.

2.9 Summary

In this chapter we evaluated previous research on tolerance to faults for long running distributed-memory applications, first providing definitions of what we mean

by a fault or failure as well as defining the metrics used to describe the reliability of large-scale machines. We then discussed a number of methods used by HPC applications to ensure progress in the presence of faults. This included the dominant checkpoint/restart and its variants, outlining the limitations of this methods for future extreme-scale systems. In addition, we described a number of emergent fault-tolerance methods for HPC including forward recovery, proactive process migration, and fault-tolerant algorithms. Lastly, we discussed a common method used in distributed and mission critical systems to mask faults, state machine replication. For each of these methods, the costs and benefits are still unclear for an exascale class systems. In this thesis we evaluate and analyze two of these methods, state machine replication and hash-based incremental checkpointing.

Chapter 3

Replication in High-Performance Computing

3.1 Overview

The first technique we study to keep checkpoint/restart viable for exascale systems is replication. In this work, we propose to use state machine replication to dramatically reduce the checkpoint frequency of the application. State machine replication is conceptually straightforward for message passing HPC applications. In this approach, each replica is created on independent hardware for every processor rank in the original application of which failure cannot easily be tolerated. Note that we do not require *all* ranks to be replicated—in master/slave-style computations where the master can recover from the loss of slaves, only the master might be replicated.

The replication system then guarantees that every replica receives the same messages in the same order and that a copy of each message from one rank is sent to each replica in the destination rank. In addition, the replication system must detect replica failures, repair failed nodes when possible, and restart failed nodes from

active replicas. The replication system may also periodically check that replicated ranks have the same state.

Checkpoint/restart recovery is still required when using replication, specifically when *all* replicas of a particular process rank have failed. Checkpointing is also needed to recover from situations where replica state becomes inconsistent, for example due to silent (undetected) failures.

Replication requires significantly increased computational resources – at least double the hardware for replicated ranks. In cases where only portions of an application must be replicated, these requirements are potentially modest. For many HPC applications (e.g. traditional stencil calculations), however, this approach *doubles* the required hardware—$2N$ nodes are required to fully replicate a job that would otherwise run (perhaps much more slowly due to failures) on N nodes. In addition, there are runtime overheads for maintaining replica consistency.

This cost in resources, however, comes with significant and important advantages:

- **Dramatically increased system MTTI.** This approach dramatically reduces the number of faults visible to applications. Specifically, the application only sees faults that crash (or otherwise fail) *all* replicas of a particular rank.
- **Significantly reduced I/O requirements.** Increased system MTTI reduces the speed at which checkpoints must be written to storage to allow applications to effectively utilize the system. A smaller fraction of the system cost and power budget must as a result be spent on the I/O system.
- **Detection of "silent errors."** By comparing the state of multiple replicas (e.g. using memory checksums) prior to writing a checkpoint, replication can detect if application state has been corrupted and trigger restart from a previous checkpoint.

- **Increased system flexibility.** The extra nodes used for redundant computation when running the largest jobs can be used for providing extra system *capacity* when running multiple smaller jobs for which fault tolerance is less of a concern. A system that uses N nodes and an expensive I/O system to reach exascale can only run 100 10PF jobs at a time, for example. A system that uses $2N$ nodes and a less expensive I/O system to reach exascale, however, can potentially run 200 10PF jobs at a time.

This chapter outlines our approach for evaluating replication for high-performance computing. First, in Section 3.2 we describe our method of modeling the quantitative cost and its benefits for extreme-scale systems, presenting an initial comparison of traditional checkpoint/restart to replication with checkpointing in Section 3.3. Then, in Section 3.4 we describe a simulation-based analysis of replication, along with a comparison to the model-based approach described previously. In later chapters, we examine the runtime overheads of replication on a number of capability HPC workloads and micro-benchmarks.

3.2 Modeling Replication for HPC

The potential benefits of redundant computing can be illustrated using a generalization of a common problem in probability theory called the *birthday problem* [156]. The birthday problem is concerned with the expected (or average) number of people needed to find two persons with the same month and day of birth. The birthday problem result is used in the analysis of many problems in computer science, including collisions and chaining in hashes [157].

For the purpose of this work, the results of the birthday problem are generalized to describe the impact of redundant computing on application fault tolerance and

the increase in MTTI of our redundant system. If we consider each of the processes of an application to be a bin with a capacity equal to the number of replicas, then asking how many faults this new system can handle without interruption is equivalent to asking what is the expected number of throws of random balls until one bin has been filled to capacity. In terms of the birthday problem, this is equivalent to asking, assuming birthdays are uniformly distributed throughout an N (the number of unique processes) day year, how many people on average are needed to ensure at least two share the same birthday. In the case of two replicas per process, the birthday problem tells us that the expected number of throws (or people) is $O(\sqrt{N})$ (again where N is the number of bins or unique processes). More generally, the average number of faults F our redundant system of N sockets can absorb, assuming double redundancy, is [157, 158]:

$$Q(N) = 1 + \sum_{k=1}^{N} \frac{N!}{(N-k)! \cdot N^k} \tag{3.1}$$

Figure 3.1 shows a plot of Equation 3.1 as a function of the number of sockets. From this figure we see the well known result for the birthday problem for $N = 365$ (around 24.16 people). We also see that adding replicated processes to our system dramatically increase its ability to absorb faults, thereby increasing the effective MTTI of the application. For example, for $N = 200,000$ nodes, on average, we can sustain 561 faults before our application will be interrupted. Therefore, with dual-redundancy, the MTTI will increase by a factor of 561 in the redundant case over the non-redundant case.

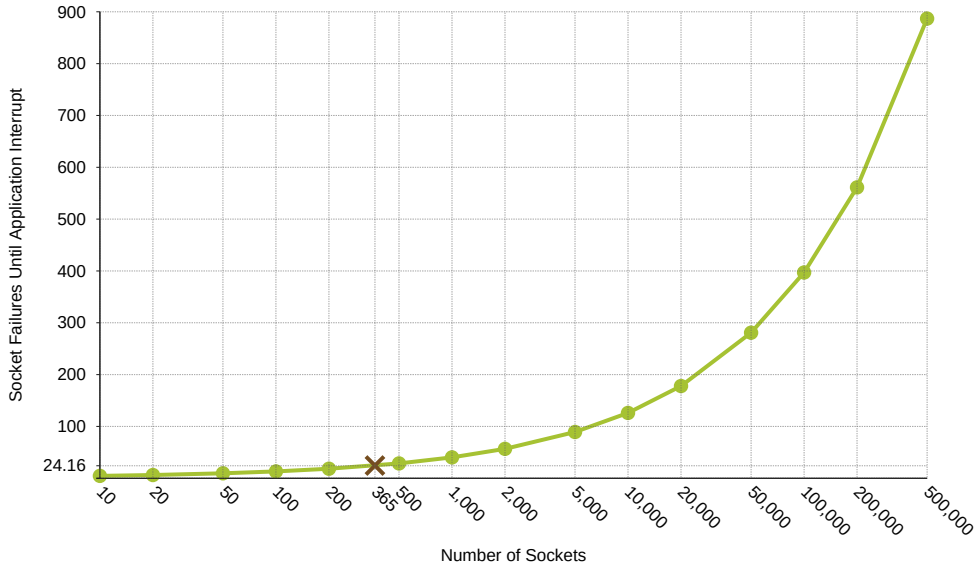


Figure 3.1: Expected number of node failures before an application interrupt in a system with redundant nodes. Numbers are calculated using the birthday problem Equation 3.1.

3.2.1 Approximations for the Birthday Problem

In the remainder of the section we investigate two approximations for solving the birthday problem. We use these approximations in following chapters to evaluate the benefits of replication by approximating the number of interrupts an application can absorb before needing to restart. The two approximations evaluated in this section include one attributed to Ramanujan [158] and one based on probabilistic indicator variables [159]. In describing each of these methods, we also outline its advantages and shortcomings.

Ramanujan's Approximation

One version of the birthday problem asks how many people on average need to be brought together until there are enough to have a 50% or better chance that two

of them share the same birth month and day. Equation 3.2, from [29, 30], shows how to calculate this version of the birthday problem. It is the so called Q-function described in [158] and examined by Knuth in [157] in the context of hashing. The answer for a $N = 365$ day year, and all days equally likely, is 24.6 people. Note that this is different from the 23 people needed in the classical birthday problem where it takes that many people to have a better than 50% chance that any person in the group has a matching birthday with any other person in the group.

$$Q(N) = 1 + \sum_{k=1}^{N} \frac{N!}{(N-k)! \cdot N^k} \approx \sqrt{\frac{\pi N}{2}} - \frac{1}{3} + \frac{1}{12}\sqrt{\frac{\pi}{2N}} - \frac{4}{135N} + \cdots \tag{3.2}$$

This Q function can be approximated for the case with two replicas as described in Equation 3.3 [158].

$$R_2(N) = \sqrt{\frac{\pi N}{2}} - \frac{1}{3} \tag{3.3}$$

Indicator Variables

Another method for solving the birthday problem is done using indicator variables and the linearity of expectation from probability theory [159]. While this method is a more coarse approximation then the ones described thus far, it has the advantage that it can be easily extended to scenarios of greater than two replicas.

In this section we outline this approximation for the birthday problem for two replicas to motivate how it can be modified for greater number of replicas. First, assume we have k individuals each with a birth date uniformly distributed from a year continuing N possible days. For each pair of individuals (i, j), we define a

random variable X_{ij} as follows:

$$X_{ij} = \begin{cases} 1 & \text{if } i \text{ and } j \text{ have some birth date} \\ 0 & \text{otherwise} \end{cases} \tag{3.4}$$

The probability that two individuals have the same birth date $\frac{1}{n}$. Therefore, from linearity of expectation, the expected value E of X_{ij} is:

$$E[X_{ij}] = 1 \cdot \frac{1}{n} + 0 \cdot (1 - \frac{1}{n}) = \frac{1}{n} \tag{3.5}$$

Now to get the expected number of pairs of all individuals having the same birthday, we sum over all pairs.

$$\sum_{i=2}^{k} \sum_{j=1}^{i-1} E[X_{ij}] = \binom{k}{2} \cdot \frac{1}{N} = \frac{k(k-1)}{2N} \tag{3.6}$$

It is straight-forward to show that if we extend Equation 3.6 for the expected number of groups of R people with the same birthday we get:

$$E_R(N) = \binom{k}{R} \frac{1}{N^{R-1}} \tag{3.7}$$

For this work, to get the expected number of faults that can be absorbed by our replicated system we input the number of replicas R, the number of application visible processes N, set the expected number of pairs equal to 1 and solve for k.

3.2.2 Comparison of Approximations

In this section we show two approximations that exist for solving the birthday problem, each with its own limitations. We look at a comparison of these functions to the value from Equation 3.1. Figure 3.2 shows a comparison of Equation 3.1,

Equation 3.3, and Equation 3.7. In the figure, the left-hand axis corresponds to the expected number of people that must be chosen at random in order to have at least one pair with the number of days in that year on the X-axis. The right-hand axis is the relative error of Equation 3.3 and Equation 3.7 in comparison to Equation 3.1.

From the figure we see that Equation 3.3 is an asymptotically accurate approximation for the Q-function. Equation 3.7, on the other hand, shows a nearly 14% overestimation for the birthday problem. The reason for this overestimation has to do with the replacement assumption inherent in the original birthday problem formulation. Our derivation of Equation 3.7 makes no such assumption and making such as assumption would greatly complicate the derivation. Again, while this indicator method formulation overestimates the birthday problem result, we can correct for outside of its derivation. Therefore, the Q function in Equation 3.1 is the most accurate, but only accounts for two replicas and is too expensive to compute. Equation 3.2 and Equation 3.3 asymptotically approximate Equation 3.1 and have the advantage of being much easier to compute, but still only model two replicas. Equation 3.7 is the least accurate approximation of Equation 3.1 but has the advantage of modeling any number of replicas.

3.3 Model-based Analysis

Using the model from Section 3.2, we examine the performance benefits of state machine replication compared to its fundamental redundant hardware costs. For this initial comparison, we assume every process is replicated, and make very simple assumptions about system characteristics. More specifically we assume that (1) There is no software overhead for maintaining replica consistency; (2) That the system can checkpoint in a fixed amount of time regardless of scale; and (3) That all failures follow a simple exponential distribution. We will relax these three assumptions in

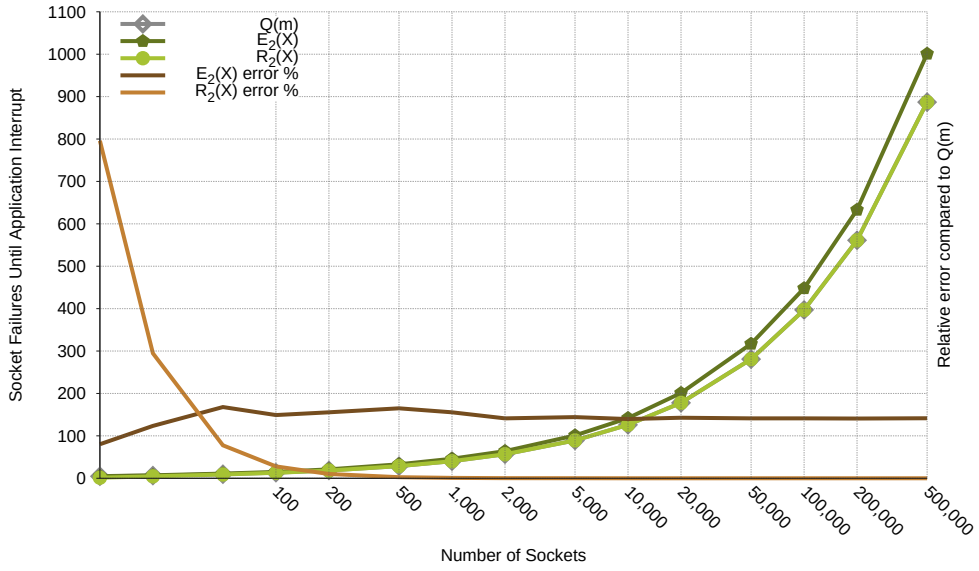
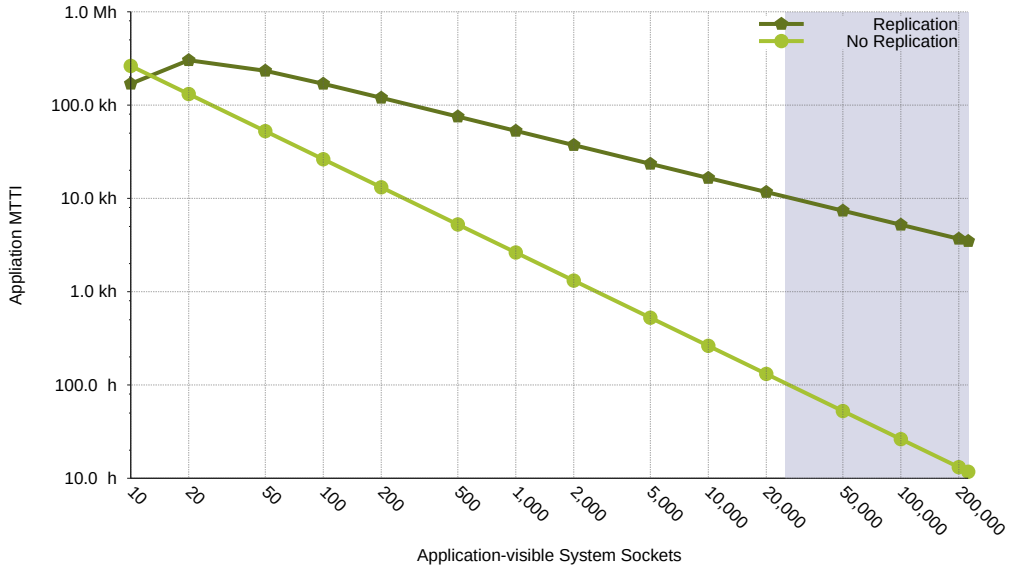


Figure 3.2: Comparison of a number of methods for approximating the birthday problem. Q(m) refers to Equation 3.1, $R_2(X)$ refers to Equation 3.2, and $E_2(X)$ refers to the indicator variables method of Equation 3.7 with two replicas.

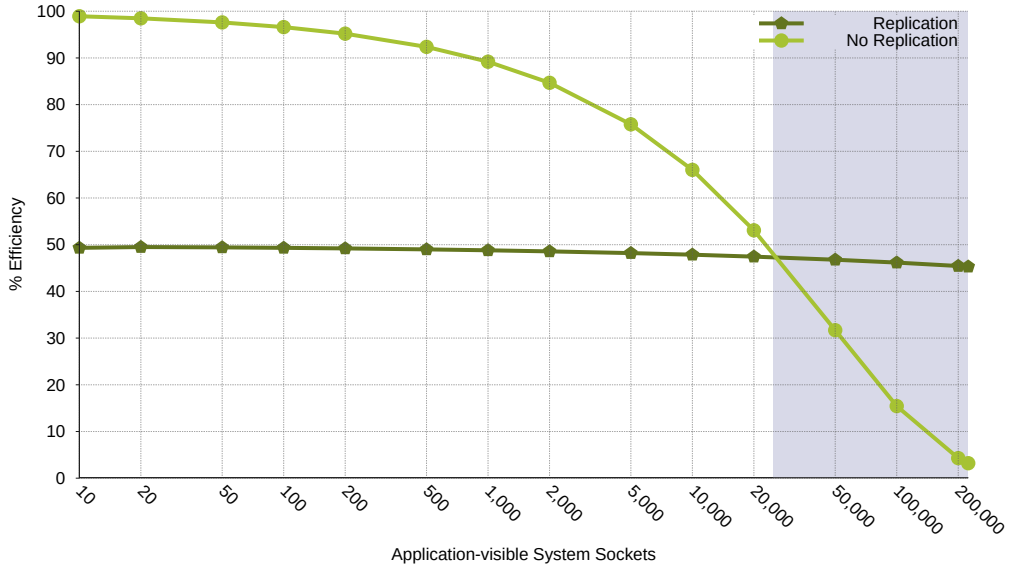
the following sections of this chapter, as well as in later chapters.

When two nodes are used to represent the same MPI rank, the failure of one node in a pair does not interrupt the application. Only when both nodes fail does the application need to restart. The frequency of that occurring is much lower than the occurrence of a single node fault and can be characterized using the birthday problem described in Section 3.2. In particular, we use Equation 3.3 to estimate the expected number of faults absorbed by the replication technique.

Figure 3.3 estimates the resulting application efficiency with optimal checkpoint intervals for both state machine replication and using only traditional checkpoint/restart. MTTI is computed directly from the birthday problem approximation in Equation 3.3, while the resulting efficiency is computed using Daly's higher-order checkpoint/restart model and optimal checkpoint interval [56]. These calculations assume a 43800 hour (5 year) per-socket MTBF based on past studies [11, 18], a



(a) MTTI



(b) Efficiency

Figure 3.3: Modeled application Mean Time To Interrupt (MTTI) and efficiency with and without state machine replication for a 168-hour application, 5-year per-socket MTBF, and 15 minute checkpoint times. In the replication case we have two replicas per process rank. The shaded region corresponds to possible socket counts for an exascale class machine [12].

constant 15 minute checkpoint time as shown in Table 1.1, and a 168 hour application solve time.

These results show the dramatic increase in system MTTI that state machine replication provides, allowing it to maintain efficiency close to 50% as system socket count increases dramatically towards the 200,000 heavyweight sockets suggested for exascale systems [12]. In contrast, the efficiency of a checkpointing-only approach drops precipitously as system scales approach those of upcoming exascale systems.

3.4 Simulation-Based Analysis

In addition to the model-based analysis present in the previous section, we also present a simulation-based approach. We use this simulator to verify, integrate, and expand the results from the previous sections into a more complete analysis of the costs and benefits of state machine replication for HPC systems. Using a simulator allows us to examine real failure distributions derived from studies of failures of real HPC systems in addition to the exponential distributions assumed by analytical models such as those of the Daly model or the birthday problem.

3.4.1 Simulator Details

This simulation tool written by Rolf Riesen and presented in [160] mimics application progress by assuming the application is always in one of four states:

Work Making progress towards a solution

Checkpoint Writing state information to stable storage

Recover Recovering from an interrupt

Rework Recomputing lost work

The simulator randomly generates node failures by determining which sockets fail and when they fail. The distribution and parameters of the generator are specified by the user. This simulator assumes a perfectly weak-scaling application; i.e., all nodes perform the same amount of work, specified as an input parameter to our tool.

Application interrupts can occur during any of the four phases, and the simulation continues until a specified amount of work has been completed. When an interrupt occurs, a restart from the last successful checkpoint is initiated. The work that was lost since the last checkpoint has to be redone in the rework phase. Following this rework stage, the regular cycle of work and checkpointing continues.

The transitions to the checkpoint state occur whenever the checkpoint interval timer expires, which is reset in the checkpoint state. The simulator uses Equation 2.2 from Daly [56] to calculate the optimal checkpoint interval.

3.4.2 Comparison of Simulation and Modeling

As stated previously, the simulator described in this section reproduces scenarios that cannot be done with the model, for example, non-exponential fault distributions. In this section we briefly compare the results of the simulator in situations that the model can accommodate.

Figure 3.4 shows a comparison of the model and simulator in one such proposed application run. This figure shows the time-to-solution for a 336 hour, dual redundant application. In this figure we assume a node MTBF of five years and a checkpoint time of 15 minutes independent of socket count. From this figure we see that the maximum percent difference between the model and simulator is 5% or less in this node count range. This difference is due to sampling errors at higher node counts as

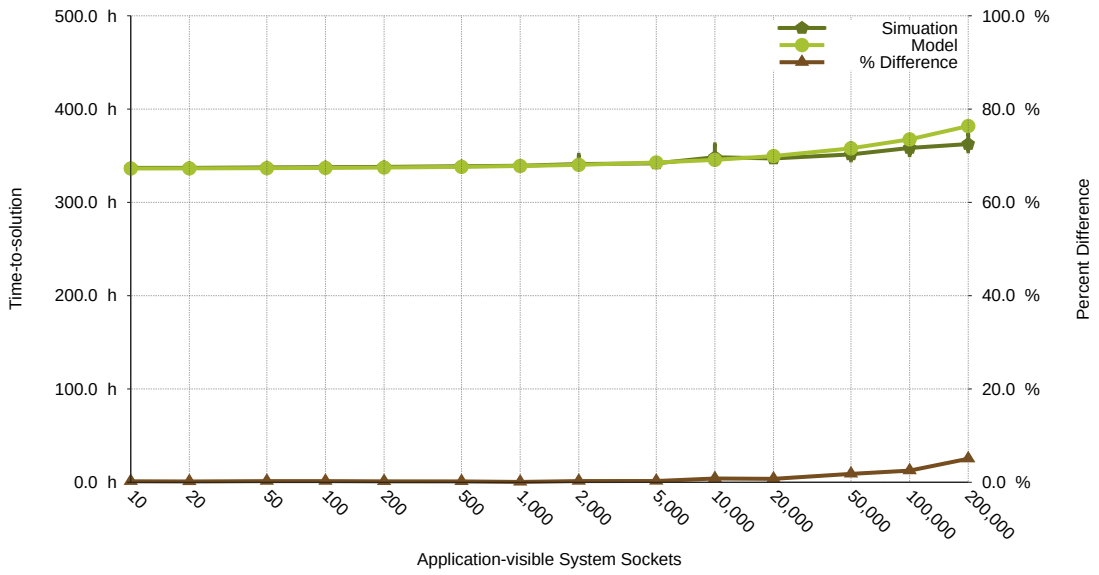


Figure 3.4: Comparison of simulator and model for a dual-redundant, 336 hour work weak scaling problem with a 5 year MTBF and a 15 minute checkpoint write time.

the simulator uses a probabilistic model to determine faulty nodes.

3.4.3 Non-Exponential Failure Distributions

Due to their more accurate modeling of system failures, it is important we examine the viability of replication with more realistic failure distributions. For failure information, we use numbers from a recent study of failures on two BlueGene supercomputer systems, a 16,384 node system at Rennesseler Polytechnic Institute (RPI) and a 4,096 node system at École Polytechnique Fédérale de Lausanne (EPFL) [11].

Results in [11] show that failures in these systems are best described by a Weibull distribution with MTBFs of 6.6 hours (11.7 years/socket) and 8.4 hours (3.9 years/-socket), and shape (β) values of 0.156 and 0.469, respectively. These β values ($\beta < 1.0$) describe distributions that decrease in probability over time; in HPC systems, this indicates that failures are more likely to happen at the start of a system's

lifetime or an application run and reduce in frequency as the system runs.

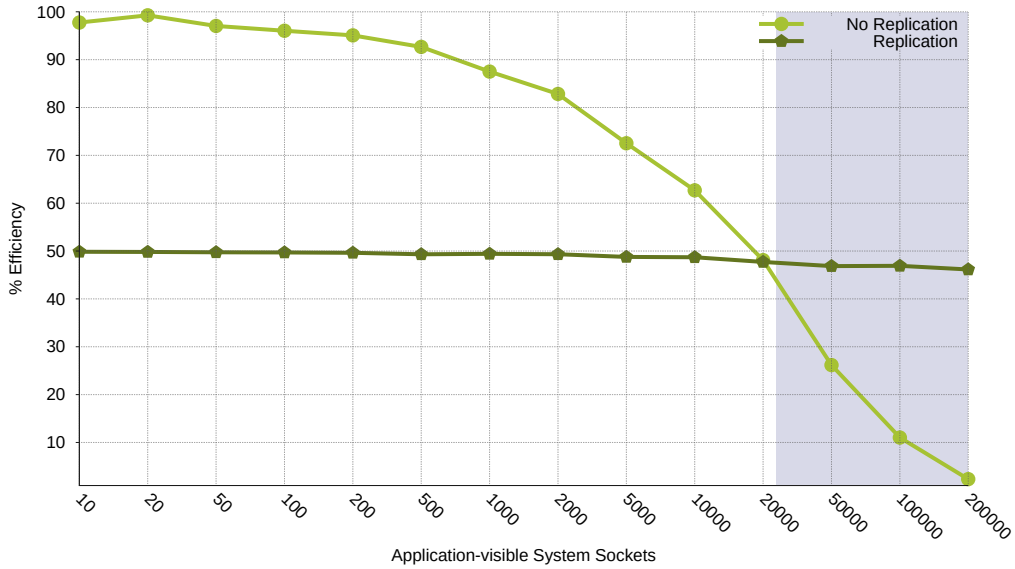
To examine the impact of these failure distributions, we build on the results of Section 3.3 and examine how the efficiency of replication and checkpoint/restart change under Weibull failures assuming again a fixed 15 minute checkpoint commit time. Note that the systems from which these distributions were measured experienced a significant number of I/O system failures, and it is unclear how these failures should be properly scaled up to larger systems. As a result, we focus on how Weibull distributions change the efficiency of replication and checkpoint/restart approach as opposed to the specific efficiency crossover point.

Figure 3.5 and Figure 3.6 present the impact of these failure distributions on both a replication-based approach and a purely checkpoint-based approach. In both these figures we note that for node counts greater than 100,000 sockets, the MTTI for the application is around the checkpoint time (δ); therefore little application progress is made in a checkpoint interval.

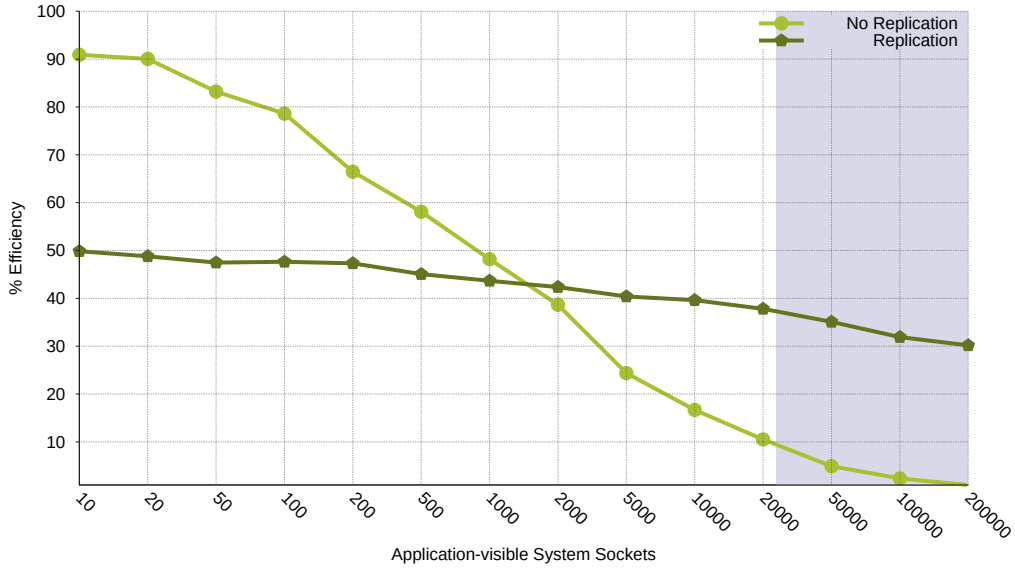
These results show that Weibull failures experienced by real-world systems result in a much more challenging fault tolerance environment, reducing the effectiveness of both replication and traditional checkpointing approaches. However, replication is less severely impacted than traditional checkpointing, again pointing to the potential more viability of a replication-based fault tolerance approach for exascale systems.

3.5 Summary

This chapter presented our initial evaluation of the costs and benefits of state machine replication for high-performance computing. We started by outlining the qualitative advantages of this approach over other fault-tolerance methods. We then described a number of methods for modeling and simulating the impact of replication on HPC.

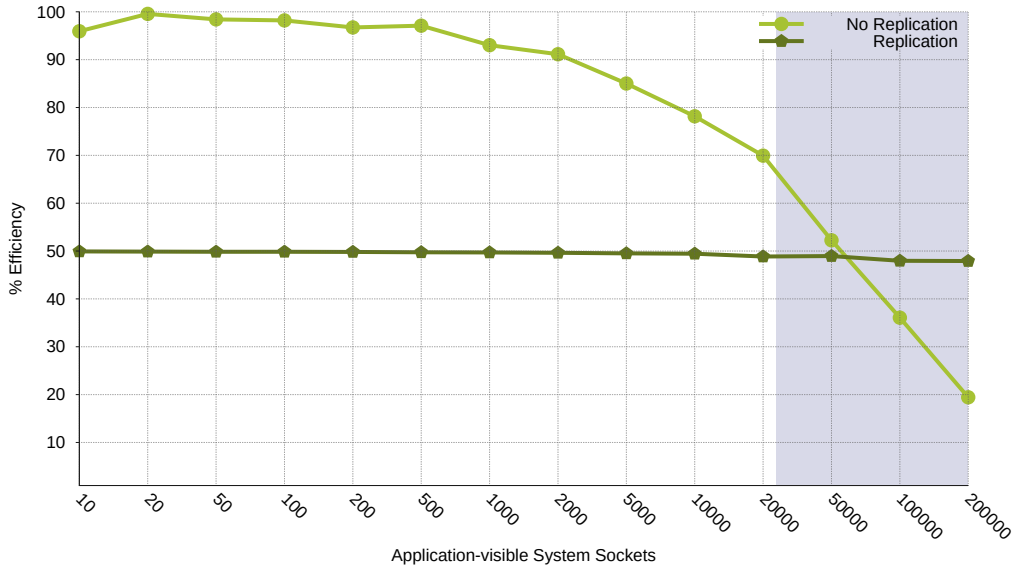


(a) Exponential, $\Theta = 4$ years

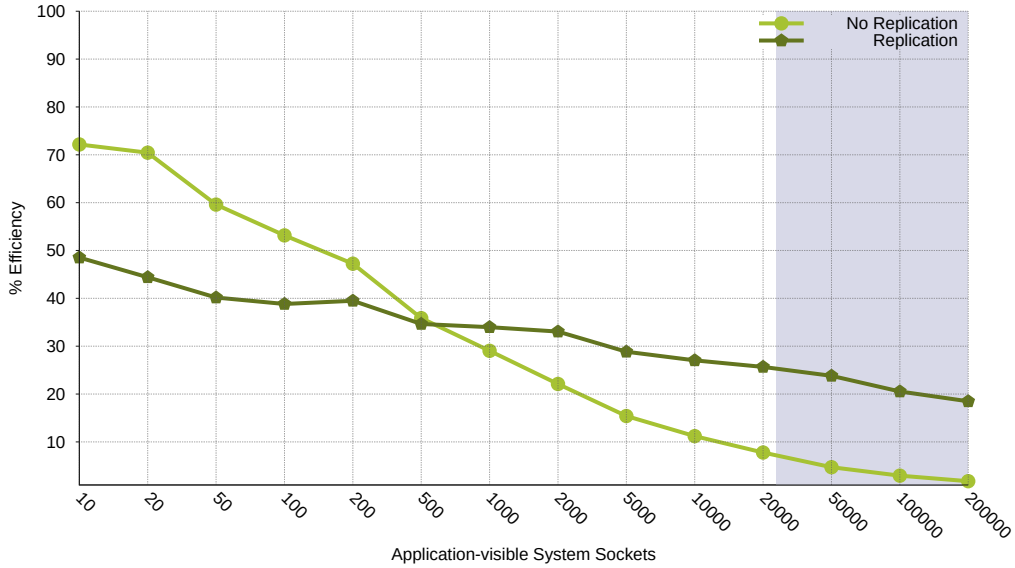


(b) Weibull, $\Theta = 4$ years, shape $= 0.469$

Figure 3.5: Simulated application efficiency with and without state machine replication for a 168-hour application, 4-year per-socket MTBF (Θ), and 15 minute checkpoint commit times with failure rate drawn from exponential and Weibull distributions [11]. In the replication case we have two replicas per process rank. The shaded region corresponds to possible socket counts for an exascale class machine [12].



(a) Exponential, $\Theta = 12$ years



(b) Weibull, $\Theta = 12$ years, shape $= 0.156$

Figure 3.6: Simulated application efficiency with and without state machine replication for a 168-hour application, 12-year per-socket MTBF (Θ), and 15 minute checkpoint commit times with failure rate drawn from exponential and Weibull distributions [11]. In the replication case we have two replicas per process rank. The shaded region corresponds to possible socket counts for an exascale class machine [12].

This included a discussion of the birthday problem and a number of approximations to this common problem from probability theory. Using this model, this work showed the significant impact replication has on application MTTI and efficiency. Lastly, we described a coordinated checkpoint simulator and compared the results of this simulator with the replication model. These results all showed that this replication technique has a higher efficiency in comparison to traditional checkpoint/restart at the socket counts expected in exascale systems, assuming no run time overheads. In addition, using the described simulator we show that for more realistic distributions the overheads of checkpoint/restart are more dramatic than seen with the currently accepted exponential model.

Chapter 4

*r*MPI: State-machine Replication in a Message Passing Environment

4.1 Overview

While the previous chapter demonstrates that state machine replication is viable at exascale in terms of the basic hardware costs, it does not evaluate the runtime overhead of the necessary consistency management protocols. Transparently supporting state machine replication for MPI applications on HPC systems requires maintaining sequential consistency between replicas. It also requires protocols for detecting and repairing failures. As mentioned in Chapter 2, these consistency protocols are potentially expensive in communication-intensive HPC systems as every replica must see messages arrive in the same order.

To study the associated overhead, we designed and implemented *r*MPI, a portable user-level MPI library that provides redundant computation transparently to deterministic MPI applications. *r*MPI is implemented on top of an existing MPI implementation using the MPI profiling hooks. In the remainder of this chapter, we outline

the basic design and implementation of *r*MPI. In Chapter 5 we measure the runtime overhead of this implementation for several micro-benchmarks and HPC applications on a large scale Cray XT-3/4 system.

This chapter is organized as follows. Section 4.2 outlines the design of *r*MPI, describing the consistency protocols and the MPI consistency requirements. In Section 4.3 we outline the design of the *r*MPI prototype architecture and its usage, and conclude the chapter in Section 4.4 with a summary.

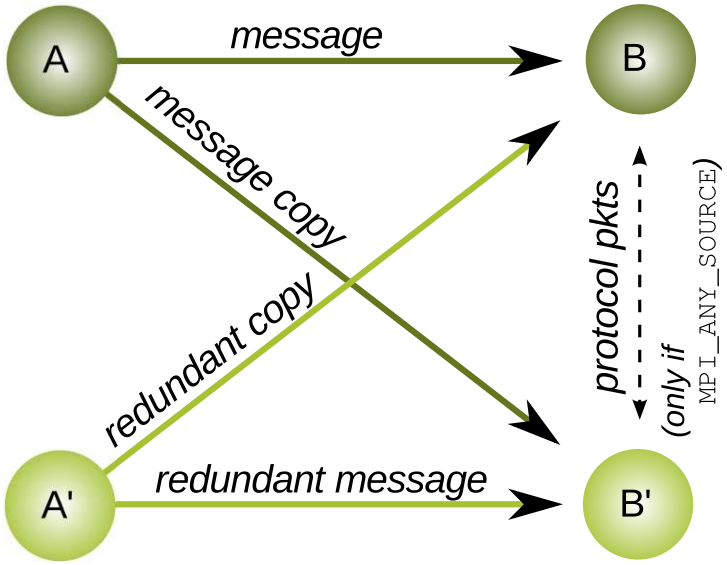
4.2 *r*MPI Design

The basic idea for the *r*MPI library is simple: replicate each MPI rank in an application and let the replicas continue when an original rank fails. To ensure consistent replica state, *r*MPI implements consistency protocols that assure identical message ordering between replicas. Unlike more general state machine replication protocols [22, 23], these protocols are specific to the needs of MPI in an attempt to reduce runtime overheads. In addition, *r*MPI uses the underlying Reliability, Availability, and Serviceability (RAS) system to detect node failures, and implements simple recovery protocols based on the consistency protocol used.

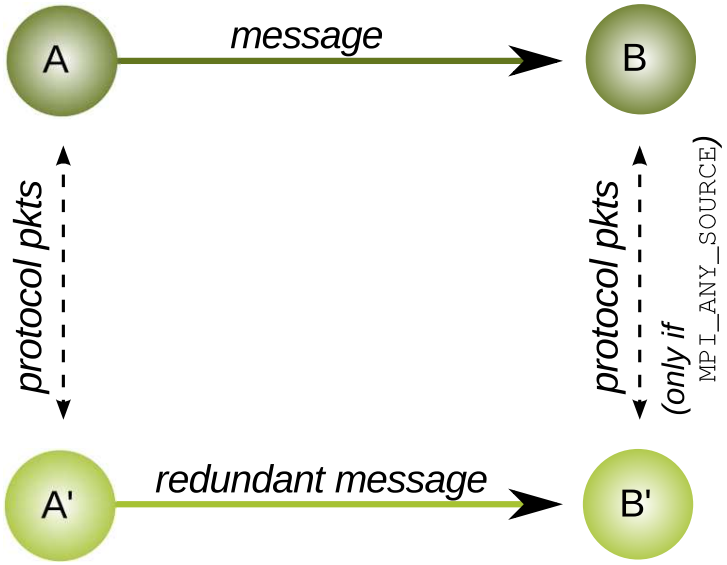
4.2.1 Basic Consistency Protocols

The *r*MPI design contains a number of different consistency protocols. These protocols vary in whether active or passive replication is used. The active replication protocols, named mirror and parallel, ensure that every replica receives a copy of every message and it orders message reception at the replica. Both active protocols take special care when dealing with MPI operations that could potentially result in different message orders or MPI results being seen at different replicas. Note that

collective operations in *r*MPI call the point-to-point operations internal to *r*MPI.



(a) Mirror Protocol



(b) Parallel Protocol

Figure 4.1: Basic active replicated communication strategies for two different *r*MPI message consistency protocols. Additional protocol exchanges are needed in special cases such as MPI_ANY_SOURCE.

Figure 4.1(a) shows the basic organization of the mirror protocol. The protocol

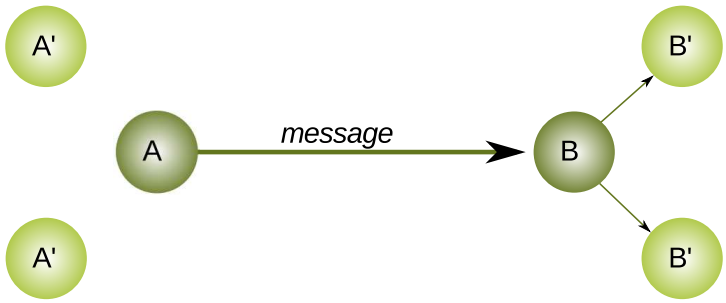
assures that all replicas see the same messages. In this figure, A and B represent distinct MPI ranks and A' and B' are A's and B's replicas respectively. In this protocol, each sender transmits duplicate messages to each of the destinations. Similarly, receivers must post multiple receives for the duplicate messages, but only require one of those messages to arrive in order for the application to progress. While this approach eases recovery after a failure, it effectively doubles network bandwidth requirements.

The parallel protocol is shown in Figure 4.1(b). For this protocol, each replica has a *single* corresponding replica for each other rank with which it communicates in non-failure scenarios. In the case of failure, one of the remaining replicas of a rank takes over sending and receiving for the failed node. This failure detection requires frequent message-based interaction with the reliability system on current systems. As a result, the parallel protocol initiates approximately double the number of messages for each send operation. These extra messages contain MPI envelope information and are small. Therefore, the parallel protocol reduces network bandwidth requirements while increasing the number of short messages, thereby decreasing an application's message rate.

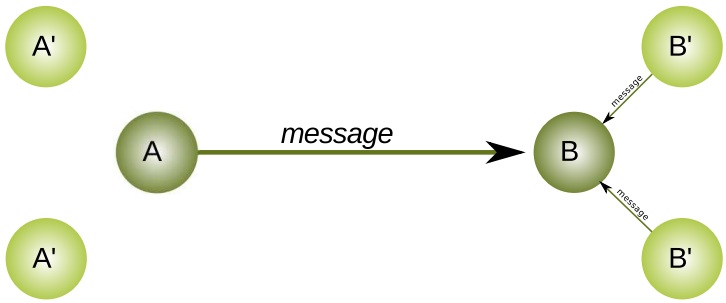
The passive protocols in Figure 4.2 vary from the active protocols described previously in that only the leader or primary ranks are involved in message sending and reception. The difference between these two protocols is, (1) whether the leaders push the messages to each of its replicas; or, (2) waits for the replicas to pull. As only one replica is involved with the reception of the message, an explicit ordering protocol is not needed. Ordering is done at the leader.

4.2.2 MPI Consistency Requirements for Active Protocols

*r*MPI assumes that only MPI operations can result in non-deterministic behavior, and there are a few specific MPI operations that can result in application-visible



(a) Leader's Push



(b) Replica's Pull

Figure 4.2: Basic passive replicated communication strategies for two different *r*MPI message consistency protocols. Additional protocol exchanges are needed in special cases such as MPI_ANY_SOURCE.

non-deterministic results. For example, *r*MPI must address non-blocking operations, wildcard (e.g. MPI_ANY_SOURCE and MPI_ANY_TAG) receives, and operations such as MPI_Wtime(). As a first step, both *r*MPI active protocols use the notion of a leader node for each replicated MPI rank, while non-leader nodes are referred to as replicas or redundant nodes. When a leader drops out of a computation, the protocol chooses a new replica from among those remaining for a rank to take over as leader.

For blocking non-wildcard receives, one of the the most common forms of MPI communication, the mirror protocol in *r*MPI posts a receive for *both* senders A and A' into the buffer provided by the user. Since the data in the two arriving messages is identical, there is no danger of corrupting the user buffer. If multiple messages from the replica set A arrive with the same tag, *r*MPI must make sure that the first

active and first redundant message arrive in the first buffer, and the second active and second redundant in the second buffer. *r*MPI achieves this by using one high-order tag bit, setting it on all outgoing redundant messages and setting the same bit for all receives of redundant messages.

This situation is illustrated in Figure 4.3. Node A sends messages *msg1* and *msg2* with the same tag to node B. MPI message ordering semantics demand that *msg1* arrives in *buf1* and *msg2* arrives in *buf2*. If the redundant messages *msg1'* and *msg2'* had the same tags as the original messages, then it would be possible for *msg1* and *msg2* to both arrive in *buf1* or *buf2*, since *r*MPI posts two receives for each buffer. Using an unused tag bit to mark redundant messages avoids the possible mix-up.

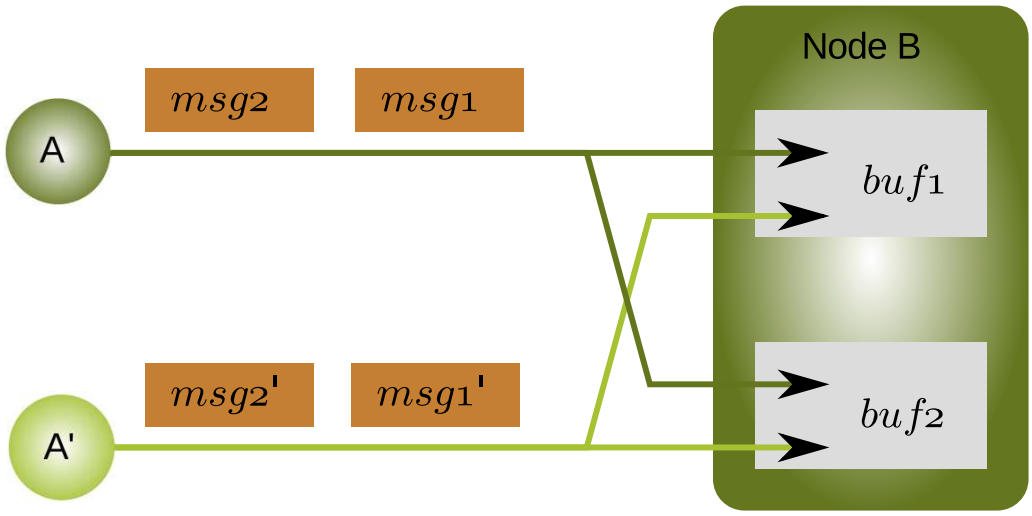


Figure 4.3: Original and redundant messages with the same tag must maintain the same order.

*r*MPI uses its own request handles to return to the user because many receives will not have been submitted to the MPI library at the time *r*MPI needs to return a request handle to the user. This means *r*MPI must maintain data structures that map its request handles to the ones used by the underlying MPI implementation.

Wildcard receives. Due to MPI message-passing semantics and the possibility of wildcard source receives, this basic consistency protocol is not completely sufficient. To handle `MPI_ANY_SOURCE` and `MPI_ANY_TAG`, *r*`MPI` relies on explicit communication between the leader of each rank and other replicas. Essentially, *r*`MPI` allows only one actual wildcard receive to be posted at any time on a node, and then only on the leader. When a wildcard receive is matched, the leader then sends the MPI envelope information to replica nodes which then post for the actual message needed. The situation is more complicated for non-blocking wildcard receives, test, and wait operations requiring a queue of outstanding wildcard receives, but the basic approach is similar. When the receive of a message is complete, the status information about the receive on node B and B' must be updated such that both nodes report the same message source and tag, without the extra bit set, to the user.

Groups and communicators. *r*`MPI` also needs to implement its own groups. Because *r*`MPI` re-maps ranks between the user level and the underlying MPI implementation, *r*`MPI` needs to carefully track which nodes and redundant nodes belong to which groups. This is necessary so that message transfer functions and function calls like `MPI_Group_rank()` work properly. The same is true for communicators and functions like `MPI_Comm_dup()`. Implementing collectives, request handles, groups, and communicators inside *r*`MPI` reduces the underlying MPI implementation to a simple transport mechanism and increases the complexity of *r*`MPI` greatly.

r`MPI` must carefully keep track of node rank information and always let replica nodes return to the user the rank of the leader node in a bundle. For example, `MPI_Comm_rank()` must return the same value on an leader node and its replica; similarly `MPI_Comm_size()` must return the number of the unique ranks in the application. Message destinations and sources must be treated similarly.

Finally, *r*`MPI` must guarantee that operations such as `MPI_Wtime()` return the same

value on active and redundant nodes, as some applications make decisions based on the amount of time elapsed. For these situations, the leader node sends its computed value to the redundant node. As an option, *r*MPI can synchronize the `MPI_Wtime()` clocks across the nodes [87].

4.2.3 Failure Detection

*r*MPI's failure detection requirements are relatively modest, and use the underlying supercomputer RAS system to provide much of this failure detection functionality. Both active and passive protocols require that messages from failed nodes will be consumed and do not deadlock the network or cause other resources, such as status in the underlying MPI implementation to be consumed. Furthermore, failing nodes must not corrupt the state of other nodes, i.e., corrupted or truncated messages in flight must be discarded. Most networking technologies already do this using CRC or other mechanisms to detect corrupt messages. The RAS system is also responsible for the machine stopping the retransmission of messages from and to failed nodes.

For the parallel and both passive protocols we also require a method to learn whether a given node is available or has failed. On the test systems, we typically emulate a RAS system at the user-level. This is a table which *r*MPI consults, and the RAS system updates, when a node's status changes. It could also be an event mechanism that informs *r*MPI whenever the RAS system detects a failed node.

4.3 *r*MPI Implementation

In this section we describe a prototype implementation of the *r*MPI design described in the previous section. In an effort to illustrate a worst-case overhead of replication in HPC, only the active protocols described in Section 4.2.1 are examined. These

active protocols are expected to have higher overheads as they require an ordering protocol, for example a total order broadcast.

4.3.1 Basic Architecture

The *r*MPI library is implemented as a library at the MPI profiling layer between an application and an MPI implementation. In this section we list some things that are specific to our current implementation.

The *r*MPI library is activated during MPI_Init(), at which time it partitions MPI_COMM_WORLD into a set of active and redundant nodes. We performed this work on a Cray XT4 Red Storm system which uses MPICH [161, 162] for message transport. Although the design described in the previous section is agnostic of the underlying MPI implementation, our current implementation of *r*MPI is tailored for a specific MPI implementation. To accelerate prototyping, we used several functions from MPICH, such as the MPI collective functions, which call our protocol aware point-to-point functions. While doing this, we left several low-level MPICH internal function calls in place. Examples include MPICH error handling and reporting functions, checking for thread-safety, and dealing with heterogeneous systems. This means *r*MPI will currently only work running on top of our specific MPICH version. Future work for this library includes removing this MPICH-specific dependency.

4.3.2 RAS Functionality

Since few machines actually provide a RAS system that gives us the minimal set of functions we need, we designed our own. *r*MPI maintains a table of all nodes in the application and their status. We use signals and messaging to update this table and can thus simulate the failure of nodes for testing purposes. However, since all nodes

still are part of a complete MPI application and due to the way MPICH interacts with the Red Storm RAS system, simulated failed nodes cannot simply exit. They enter `MPI_Finalize()` and wait for all other nodes to finish. This also means that if we failed a node during an *r*`MPI` operation that involves several MPI messages, MPICH may enter into an inconsistent state. Proper integration of *r*`MPI`, a RAS system, and MPI would solve this problem.

4.3.3 Usage

When users start an application linked with *r*`MPI` they specify how many redundant nodes to allocate and how to map them to the active nodes. An environment variable specifies this mapping. The *r*`MPI` implementation imposes some restrictions on these mappings. The redundant nodes must always be at the end of the `MPI_COMM_WORLD` rank list. Not every active node needs to be assigned a redundant partner. If nodes A, B, C, and D are active nodes, then ABCD|A'B'C'D', ABCD|A'B', ABCD|D'C'B'A', and ABCD|D'C' are some of the many valid mappings.

Lastly, to avoid using additional buffer space and to limit memory copies, *r*`MPI` receives both the original and the redundant message into the same buffer. We assume that two identical messages arriving in the same buffer will not "collide" and that, once both messages have been received, the buffer memory will be in the same state if only one message had been received. We are not aware of any system today which does not fulfill this requirement.

4.4 Summary

In this chapter we introduced the design and implementation of the *r*`MPI` library which inserts itself between an application and the MPI library. *r*`MPI` allows users

to allocate additional compute nodes for redundant computation. In the description of the design and implementation of *r*MPI, we detailed the techniques that are necessary to maintain MPI semantics, especially managing message ordering on the active replica protocols. In the next chapter we will use this replication library to quantify the runtime time overheads of the consistency protocols on a number of HPC workloads.

Chapter 5

Evaluating State-machine Replication's Runtime Overheads

The advantages described in Chapter 3 (i.e. significantly decreased checkpoint frequency and possible soft-error detection and correction) provide a compelling reason to examine the viability of state machine replication for extreme-scale HPC systems. Without quantifiable performance benefits compared to other approaches, however, state machine replication will not be viable for use in exascale systems. This chapter therefore examines the runtime performance costs of state machine replication. The remainder of this chapter is organized as follows. In Section 5.1, we describe the methodology used to evaluate the runtime overheads of replication, describing our test platform and replica placement options. Section 5.2 presents an evaluation of the runtime overheads of state-machine replication on a number of micro-benchmarks and HPC workloads. Section 5.3 characterizes these runtime overheads for incorporating into our previously described replication model. Finally, Section 5.4 summarizes the results of this chapter.

5.1 Methodology

From the discussion in the previous sections it should be clear that *r*`MPI` may add additional overhead and lengthen the execution time of an application. To empirically quantify this overhead we ran multiple tests with applications on the Cray Red Storm system at Sandia National Laboratories compiled with both *r*`MPI` and the original unmodified Cray MPI library. Red Storm is a XT-3/4 series machine consisting of over 13,000 nodes, with each compute node containing a 2.2 GHz quad-core AMD Opteron processor and 8 GB of main memory.

Additionally, each node contains a Cray SeaStar [163] network interface and high-speed router. The SeaStar is connected to the Opteron via a HyperTransport link. The current generation SeaStar is capable of sustaining a peak unidirectional injection bandwidth of more than 2 GB/s and a peak unidirectional link bandwidth of more than 3 GB/s.

To ensure leader and replica are on separate physical nodes, and to avoid memory and bandwidth bottlenecks on the nodes themselves, we only used one CPU on each node.

We expect that the *r*`MPI` library adds some overhead, even if no redundant nodes are used, due to the checks whether there are redundant nodes available and the way we implement the collective operations. We compare this *baseline* overhead to the *native* performance when the *r*`MPI` library is not linked in at all. To get a worst case bound on the cost of rank level replication, a fully redundant configuration is used for the forward, reverse, and shuffle mappings.

5.1.1 Replica Placement

To ensure leader and replica are on separate physical nodes, and to avoid memory and bandwidth bottlenecks on the nodes themselves, we only used one CPU on each node.

Redundant nodes should be physically as far away from their active node as possible. The goal is to share as few hardware resources between these nodes as possible. Co-locating an active and its redundant node on two cores of the same CPU makes sense from a performance perspective, but not for reliability. Ideally, no power-supplies, fans, communication channels to other nodes, boards, or chips are shared. However, that is difficult to achieve in today's machines. Furthermore, it is often impossible to assign MPI ranks to specific nodes in the system.

Because of this and because of the impact a given allocation may have on the performance of an application, we ran our tests in three different modes: forward, reverse, and shuffle. The first mode, *forward*, assigns rank $\frac{N}{2}$ as a redundant node to rank 0, rank $\frac{n}{2}+1$ to rank 1, and so on resulting in a mapping like this: ABCD|-A'B'C'D'. *Reverse* mode is ABCD|D'C'B'A', and *shuffle* mode is a random shuffle (Fisher/Yates) such as ABCD|C'B'D'A'.

5.2 *r*MPI Runtime Results

5.2.1 Benchmark Details

To evaluate the performance of the two active *r*MPI protocols we will present results of a number of micro-benchmarks and a number of applications. The MPI micro-benchmarks present in this chapter include: latency, bandwidth, message rate, and host CPU utilization. See Appendix A for `MPI_Allreduce()`, `MPI_Reduce()`, `MPI_-`

`Bcast()`, `MPI_Barrier()`, and `MPI_Alltoall()` micro-benchmark performance results.

Our four representative HPC application workloads are: CTH [164], SAGE [165], LAMMPS [166,167], and HPCCG [168]. These application represent a range of computational techniques, are frequently run at very large scales, and are key simulation workloads to both the US Department of Defense and Department of Energy. These four applications represent different communication characteristics and compute to communication ratios. Therefore, the overhead of *r*`MPI` affects them in different ways.

1. CTH [164] is a multi-material, large deformation, strong shock-wave, solid mechanics code developed by Sandia National Laboratories with models for multi-phase, elastic viscoplastic, porous, and explosive materials. CTH supports three-dimensional rectangular meshes; two-dimensional rectangular, and cylindrical meshes; and one-dimensional rectilinear, cylindrical, and spherical meshes, and uses second-order accurate numerical methods to reduce dispersion and dissipation and to produce accurate, efficient results. It is used for studying armor/anti-armor interactions, warhead design, high explosive initiation physics, and weapons safety issues.

2. SAGE, SAIC's Adaptive Grid Eulerian hydro-code, is a multi-dimensional, multi-material, Eulerian hydrodynamics code with adaptive mesh refinement that uses second-order accurate numerical techniques [165]. It represents a large class of production applications at Los Alamos National Laboratory. It is a large-scale parallel code written in Fortran 90 and uses MPI for inter-processor communications. It routinely runs on thousands of processors for months at a time.

3. LAMMPS [166] is a classical molecular dynamics code developed at Sandia National Laboratories. For our experiments we use the embedded atom method

(EAM) metallic solid input script which is used by the Sequoia benchmark suite. The LAMMPS code and input scripts are provided on the LAMMPS web site [167]. For this experiment we ran LAMMPS in weak-scaling mode.

4. The HPCCG mini-application, part of the Mantevo project [168], is a simple sparse conjugate gradient solver designed to capture an important component of Sandia's production workload. The majority of its runtime is spent performing sparse matrix-vector multiplies, where the sparse matrix is encoded in compressed row storage format. The interprocessor communication is minimal, requiring exchange of nearest neighbor boundary information, in addition to global `MPI_Allreduce()` operations required for the scalar computations in the conjugate gradient algorithm.

5.2.2 Micro-benchmark Performance

Because these benchmarks do nothing but transmit messages, we expect them to show greater overhead than full applications. For the MPI latency tests, we show the performance overhead for both specific as well as `MPI_ANY_SOURCE` receives as each scenario has different performance characteristics. Again, see Appendix A for more micro-benchmark performance numbers.

Our bandwidth experiment in Figure 5.1 shows that baseline (*r*`MPI` linked in, but no redundant nodes used) for both protocols does not lower bandwidth appreciably compared to native; especially at larger message sizes. The parallel protocol redundant runs, on the other hand, shows considerable overhead, especially at smaller message sizes, showing 60% to 70% slowdown in comparison to native. This slowdown is identical for each of the three tested mappings (forward, reverse, and shuffle). The overhead for the parallel runs is due to the overhead of the increased number of messages required for bundle synchronization. As message size increases the per-

formance of parallel approaches that of native. The mirror protocol redundant run performance is also identical among the three mappings, but its 60% slowdown over native remains nearly constant through the range tested. This halving of bandwidth is expected and consistent with the fact that we are sending twice as much data through a given network interface card (NIC).

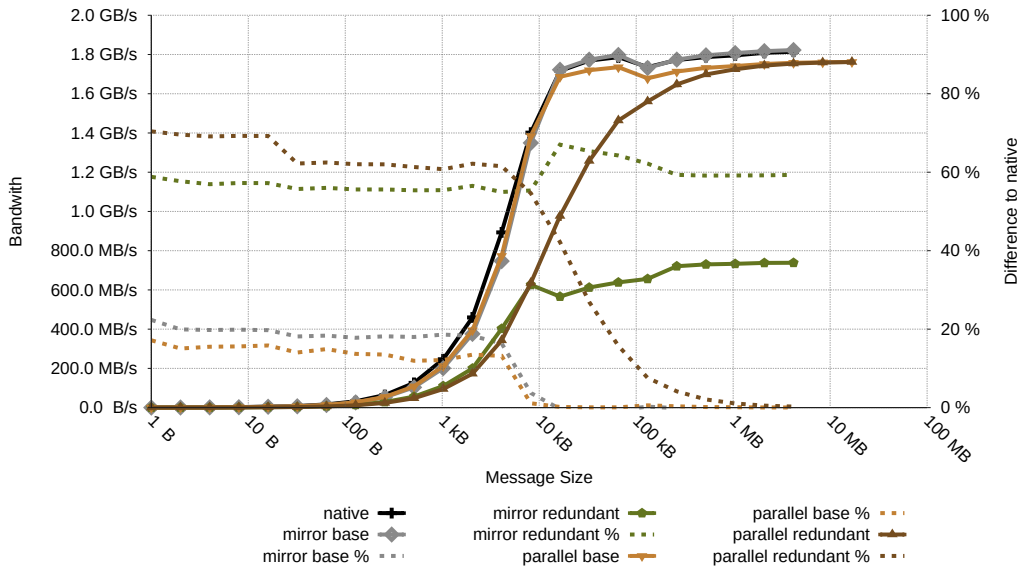


Figure 5.1: Bandwidth comparison. Native is benchmark without the *r*MPI library. Base is with *r*MPI for each protocol, but no redundant nodes. For this test the performance of forward, reverse, and shuffle fully redundant runs are equivalent.

Figure 5.2 illustrates the overhead of our MPI latency tests without MPI_ANY_SOURCE. The baseline mapping for the two protocols shows some overhead over native which is due to the accounting done in *r*MPI and becomes negligible as message size increases. The latency overhead for the redundant runs is a factor of 1.5 over native for smaller messages. For parallel this latency increase is one third that of native and is due to the extra messages used for synchronization on sends and decreases with message size. The reason the latency is less than N extra message latencies is that, assuming no nodes have failed, a sender node first performs the send operation and then performs the synchronization with replicas in its rank bundle to ensure it

does not need to fulfill another send. If a node has failed, the performance of parallel closely matches that of mirror. For mirror, the slowdown with full redundancy is $\frac{3}{4}$ that of native. The reason for this increased slowdown is as follows. A receive in mirror can return once at least one of the two possible receives has completed. Before the receive can return we must wait for the other receive or cancel it. `MPI_Cancel()` in our MPICH implementation is an expensive and non-local operation. The current implementation waits twice a measured round trip time for the other send to arrive. If it has not been received in that time, the library cancels the other receive and then returns. Similar to our bandwidth tests, the overhead of the redundant runs is identical for each of the three replica node mappings tested.

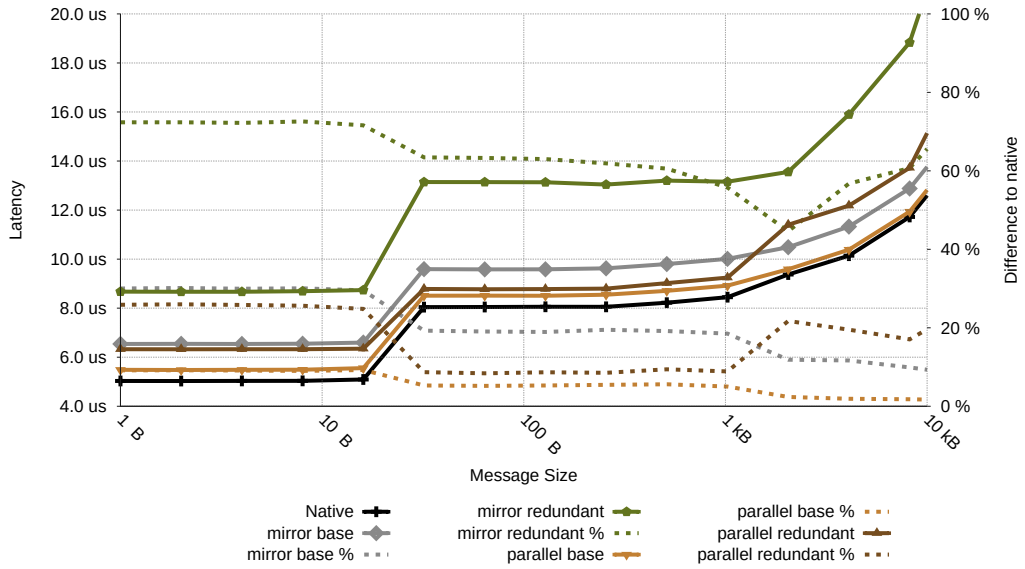


Figure 5.2: Latency comparison. For this test the performance of forward, reverse, and shuffle is equivalent.

The coordination overhead between leader and replica nodes becomes more severe when `MPI_ANY_SOURCE` is used. Recall from the discussion in Section 4.2, `MPI_ANY_SOURCE` causes replica nodes to delay the posting of receives until the leader node has received its message and informed the redundant node. In Figure 5.3 we see the

result of this. Latency increases by a factor of 1.5 across the board over native.

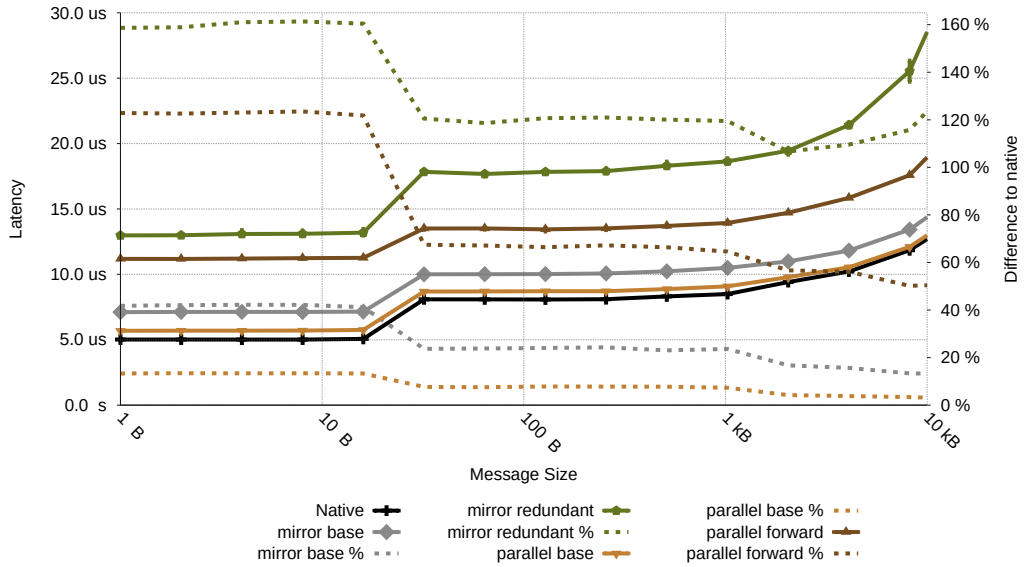


Figure 5.3: Latency comparison using `MPI_ANY_SOURCE`. For this test the performance of forward, reverse, and shuffle is equivalent.

Figure 5.4 illustrate the performance impact of the protocols on MPI message rate. From the figure we see that for smaller messages mirror is able to achieve a higher message rate than parallel (with mirror's rate around half of that of native), but as message size increases, parallel's rate approaches to within 10% of native.

Figure 5.5 illustrates the impact on CPU availability for `MPI_Send()` and `MPI_Recv()` operations. From the figure we see that the consistency protocols included in *r*`MPI` have little impact on CPU availability. The exception to this is the large message `MPI_Send()` operations in Figure 5.5(b). This impact is due to the packetization engine needed for larger messages.

Overall, we observe that the overheads due to replication is quite high for the tested micro-benchmarks. For example, MPI bandwidth tests show the mirror protocol decreases the available bandwidth by half while the parallel protocol decreases

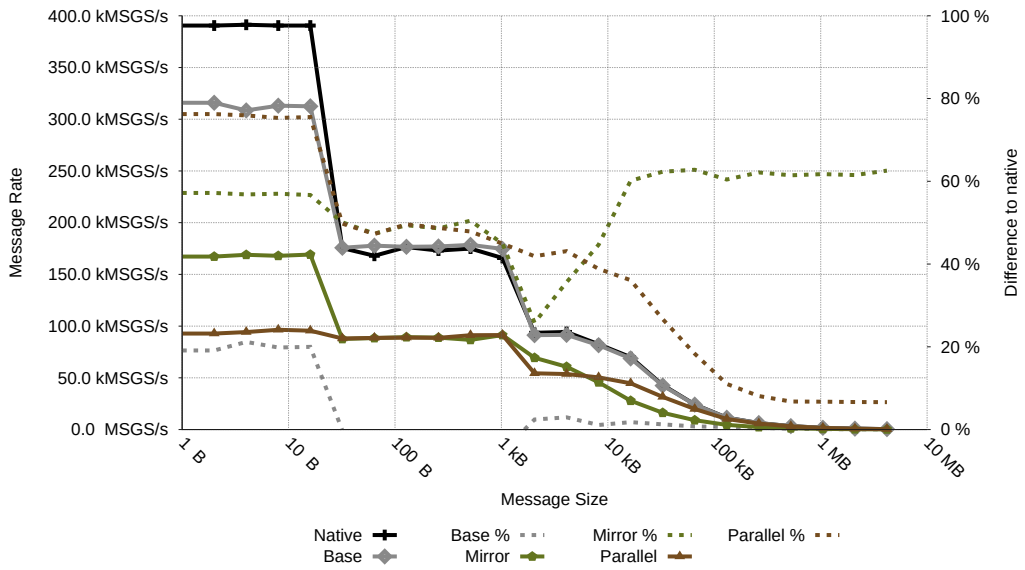


Figure 5.4: *r*MPI message rate measurements.

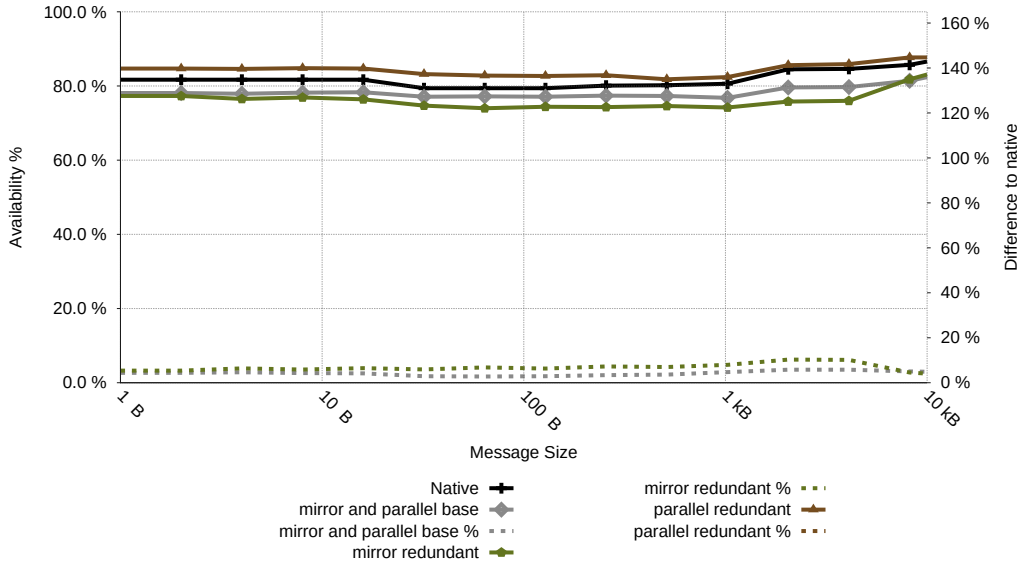
the observed message rate also by half.

5.2.3 Application Performance

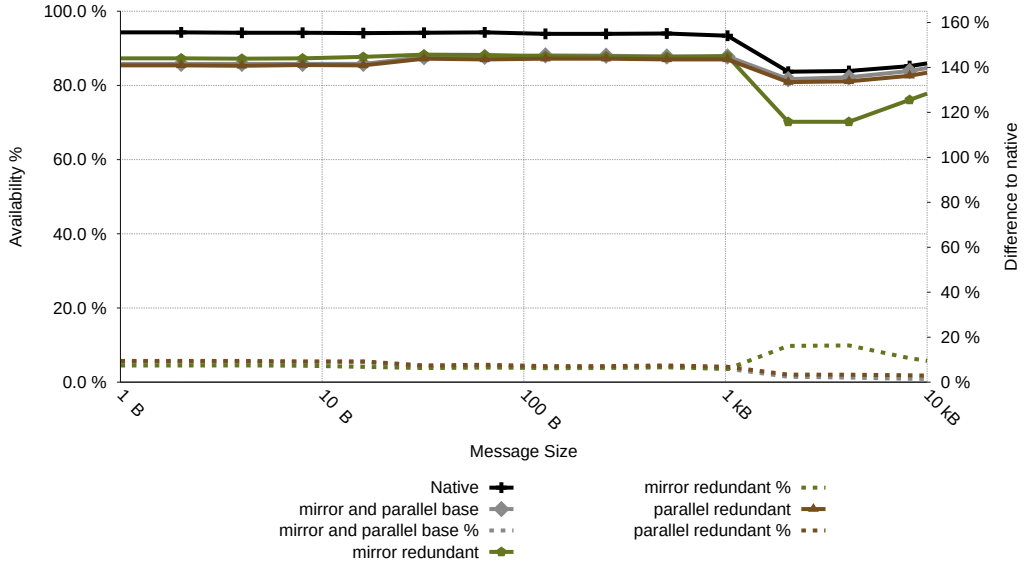
In this section we outline the performance impact of replication on real HPC workloads. In contrast to the micro-benchmark numbers of the last section, the runtime overhead of replication is much lower.

LAMMPS

Figure 5.6 shows the performance impact of *r*MPI with both the mirror and parallel protocol. The impact of each redundancy protocol is less than 5%, independent of the nodes used, while the baseline overhead for each is negligible.



(a) Receive



(b) Send

Figure 5.5: Host CPU utilization for send and receive for the two protocols compared to native and baseline. Native is the benchmark without *r*MPI; baseline has *r*MPI linked in but does not use redundant nodes.

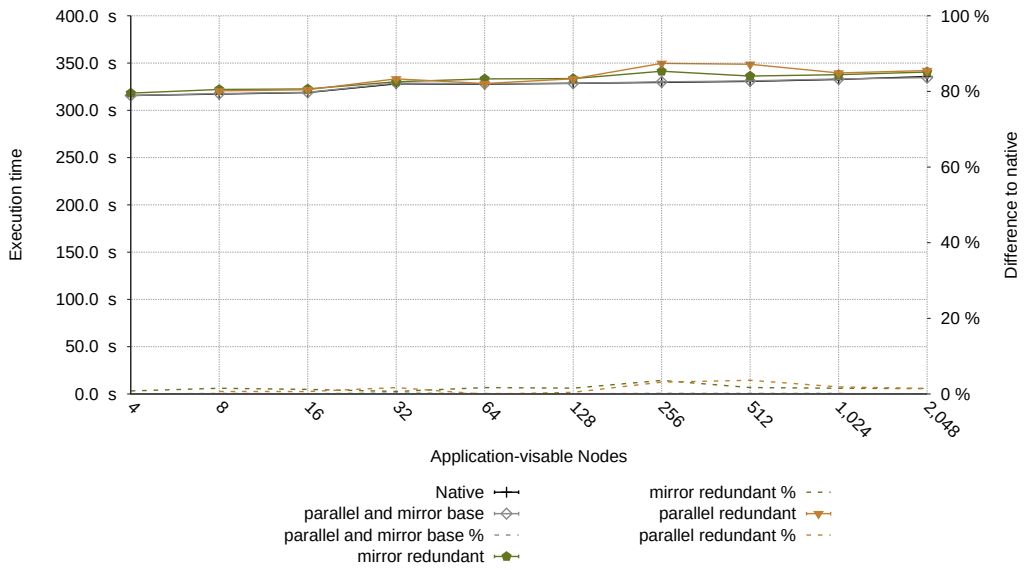


Figure 5.6: LAMMPS *r*MPI performance comparison. For both mirror and parallel, baseline performance overhead is equivalent. For this application the performance of the forward, reverse, and shuffle fully redundant modes are equivalent.

SAGE

Figure 5.7 shows the *r*MPI performance for SAGE. Similar to LAMMPS, the baseline performance degradation is negligible. Also similar to LAMMPS, the parallel protocol performance remains nearly constant and performance decrease is negligible in the tested node range; with performance overhead generally less than 5%. In contrast, full redundancy for the mirror protocol loses about 10% performance over native, with performance increasing with scale. We attribute the performance degradation for SAGE to the factor of two increase of large network messages sent by SAGE and the limited available network bandwidth.

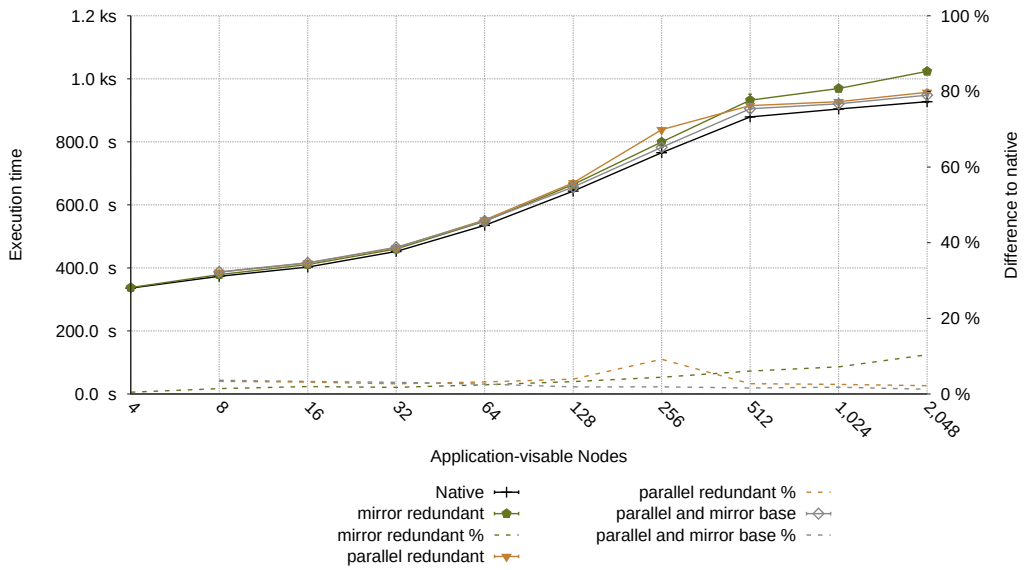


Figure 5.7: SAGE *r*MPI performance comparison. For both mirror and parallel baseline performance overhead is equivalent. For this application the performance of forward, reverse, and shuffle fully redundant modes are equivalent.

CTH

In Figure 5.8, we see the impact of our consistency protocols for CTH at scale. Again, baseline for both mirror and parallel shows little performance difference. For CTH, mirror has the greatest impact on performance with full redundancy. This impact, which is nearly 20% at the largest scale, is due to CTH's known sensitivity to network bandwidth [169] (the greatest of each of the applications tested) and the increased bandwidth requirements of the mirror protocol. Interestingly, the parallel protocol version of CTH runs slightly *faster* then the native versions (around 5-8%) for forward, reverse, and shuffle replica node mappings. Though further testing is needed, current performance analysis results suggest this decrease in application runtime is due to parallel reducing the number of unexpected messages received.

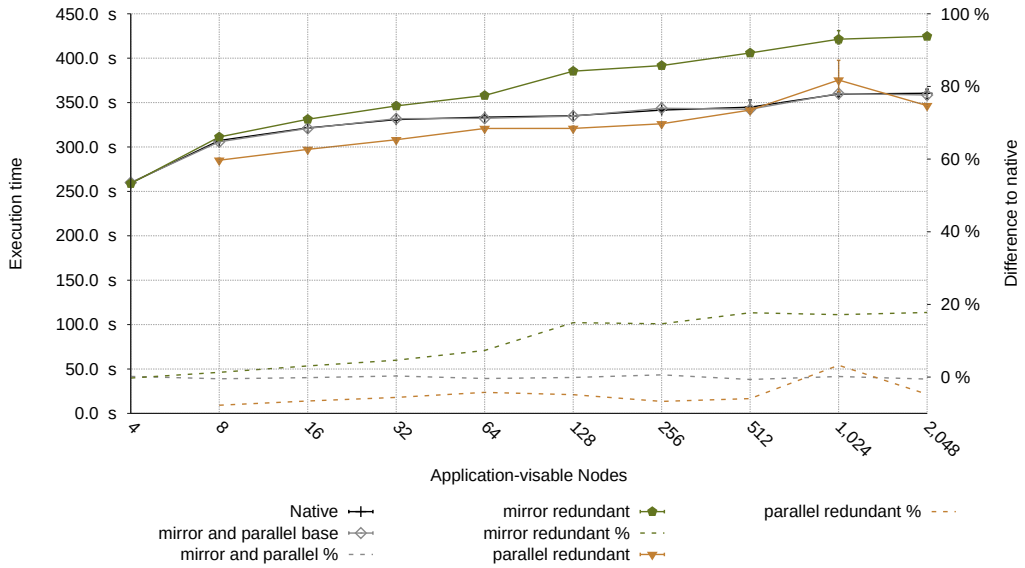
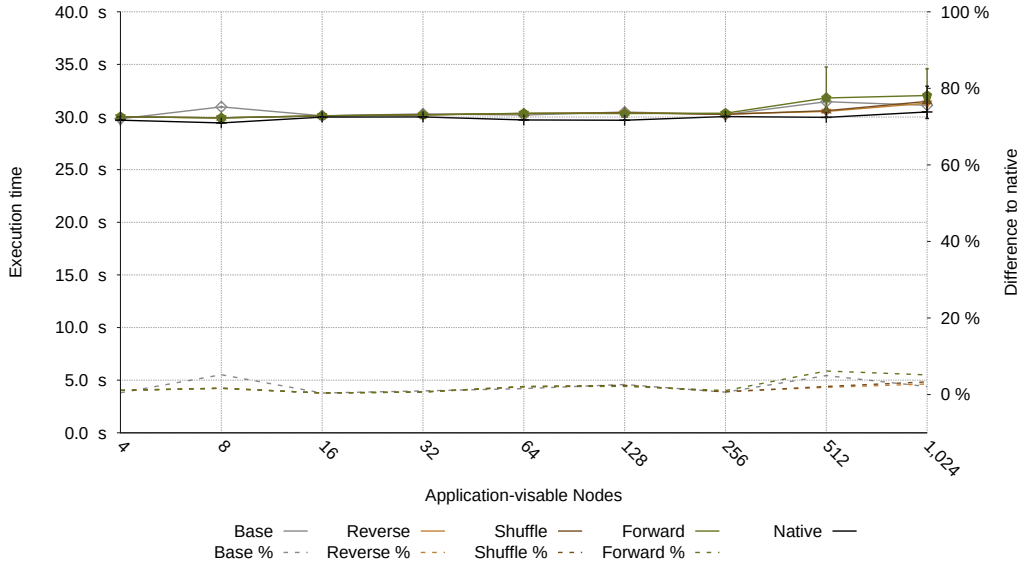


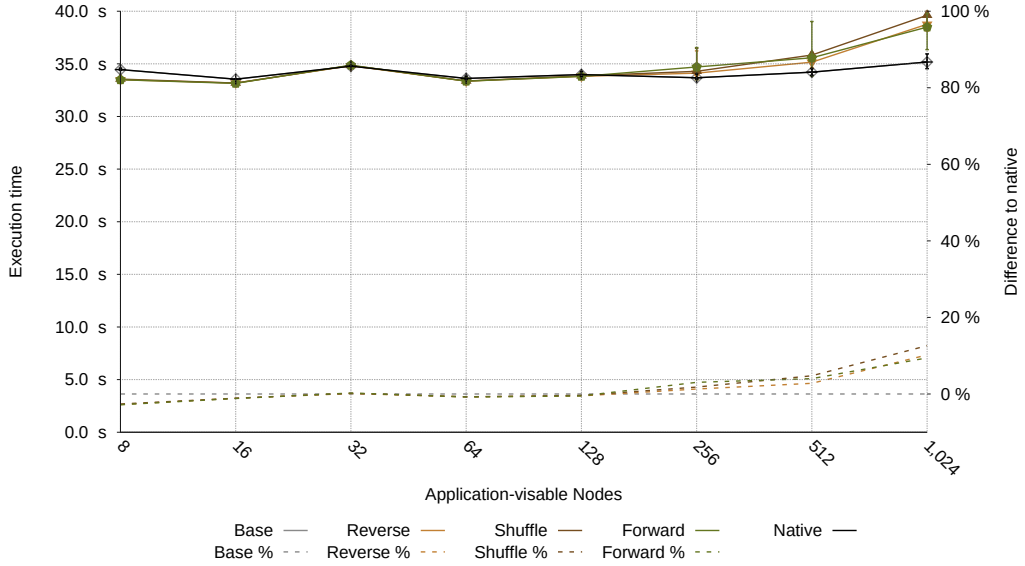
Figure 5.8: CTH *r*MPI performance comparison. For both mirror and parallel baseline performance overhead is equivalent. For this application the performance of forward, reverse, and shuffle fully redundant modes are equivalent.

HPCCG

Figure 5.9 shows the performance impact of *r*MPI on the HPCCG mini-application. In contrast to the other results presented in this section, we present the mirror and parallel results separately. Though the results presented in Figure 5.9(a) and Figure 5.9(b) represent the same computational problem, the native results of each vary due to different node allocations between the two plots. Allocation issues aside, we see that mirror has very little impact. Parallel on the other hand shows a significant impact at higher node counts, with slowdowns of around 10% at 1,024 nodes. Also, in contrast to all the other applications tested, impact from the parallel protocol is greater than that of mirror. This is because unlike other applications, HPCCG stresses the system's message rate and parallel's synchronization messages are causing it to reach the maximum messaging rate of a node.



(a) Mirror protocol



(b) Parallel protocol

Figure 5.9: HPCCG *r*`MPI` performance comparison. Varying performance for native and baseline between mirror and parallel protocols is due to different node allocations.

5.3 Analysis of Run Time Overheads

Our results evaluating the runtime overhead of state machine replication show that the runtime costs of implementing state machine replication for a wide range of pro-

duction HPC applications at significant scale is minimal. In particular, for each application either the parallel or mirror protocol provides almost negligible performance impact. Examining the best protocol for each application, SAGE has the highest net overhead, 2.2% at 2048 application-visible nodes. A logarithmic curve can be fit to the overhead for this worst-case, with the fit curve shown in Equation 5.1.

$$g(S) = \frac{1}{10} \log S + 3.67 \tag{5.1}$$

This curve would result in a 4.9% additional overhead on a projected exascale system with 200,000 sockets.

For comparison, the worst-case overhead over *all* protocols can be fit with a curve shown in Equation 5.2.

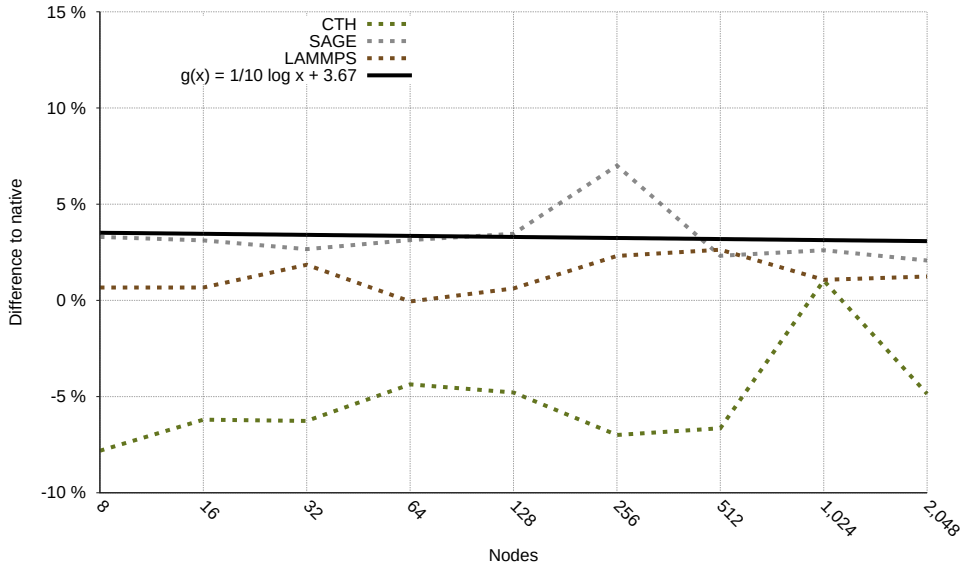
$$g(S) = 3.36 \log S - 5.31 \tag{5.2}$$

This worst-case curve would result in 35.7% additional overhead on a projected exascale system with 200,000 sockets.

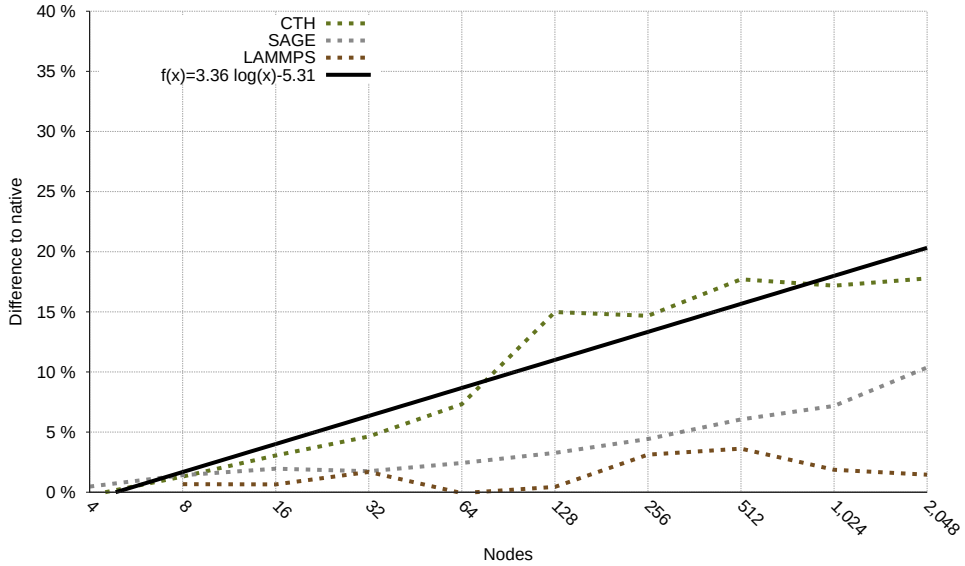
Figure 5.10 shows these overheads along with the corresponding application slowdown measurements. In Chapter 6 we incorporate these measured run-time overheads into our state machine replication model to examine the merit of replication for exascale systems.

5.4 Summary

In this chapter we presented the run time protocol overheads for *r*`MPI`, a MPI library that enables transparent, user-level rank level replication. Using this library



(a) Best Run Time Overhead



(b) Worst Run Time Overhead

Figure 5.10: Best-case (Equation 5.1) and worst-case (Equation 5.2) *r*`MPI` run time protocol overhead fit functions and corresponding data from CTH, SAGE, and LAMMPS.

we showed that while the protocol overheads are quite high for a number of communication micro-benchmarks, there is a relatively low overhead protocol choice for each of the tested applications. In the following chapter we incorporate this overhead in our replication model to more accurately examine the costs associated with state machine replication.

Chapter 6

Replication Analysis

6.1 Overview

In this chapter, we combine the results from the previous sections into a more complete analysis of the costs and benefits of state machine replication for HPC systems. By doing so, we examine additional machine parameters and their impact on the viability of state machine replication, particularly variations in available I/O system bandwidth and failure rates.

In the remainder of this chapter, all results assume software runtime overheads as shown in Equation 5.1 and Equation 5.2; efficiency results incorporate the factor of two reduction for state machine replication due to the required redundant hardware. Unless otherwise stated, we also continue to assume checkpoint and restart times of 15 minutes as in previous chapters.

We describe our comparison approach in Section 6.2. We then extend our model-based analysis, incorporating the measured runtime overheads, node failure rates, and I/O commit bandwidth rates in Section 6.3, Section 6.4, and Section 6.5. Sec-

tion 6.6 further extends our model to account for triple-modular redundancy and and higher replica count. In Section 6.7 we introduce a simulation-based analysis of state-machine replication which allows for more realistic failure distributions. We conclude the chapter in Section 6.8.

6.2 Comparison Approach

Our primary performance evaluation criteria is as follows: at what node counts, if any, does state machine replication provide quantitative performance *advantages* over past approaches particularly in terms of system utilization, *after* accounting for the overheads of state machine replication. If, for example, state machine replication achieves 46% utilization at a given system socket count and another technique only achieves 40% system utilization, we regard state machine replication as superior at that point.

We use traditional checkpoint/restart fault tolerance as the baseline technique against which to compare because its performance characteristics are well known and understood. We believe that comparing against a well-understood baseline will facilitate future comparisons against other proposed exascale fault tolerance techniques, as their costs and benefits at scale are more fully quantified. A brief qualitative comparison with several such techniques is provided in Chapter 2.

6.2.1 Assumptions

Because we are comparing a new technique on projected hardware systems, our comparisons make a number of assumptions that are important to make explicit.

We assume:

1. Fully replicated hardware redundancy for all applications, resulting in a maximum possible efficiency for state machine replication at 50%.

2. The MPI library is the only potential source of non-determinism in the application.

3. Machines suffer from only crash failures, and not from more general failures, from which checkpoint/restart may not be able to recover.

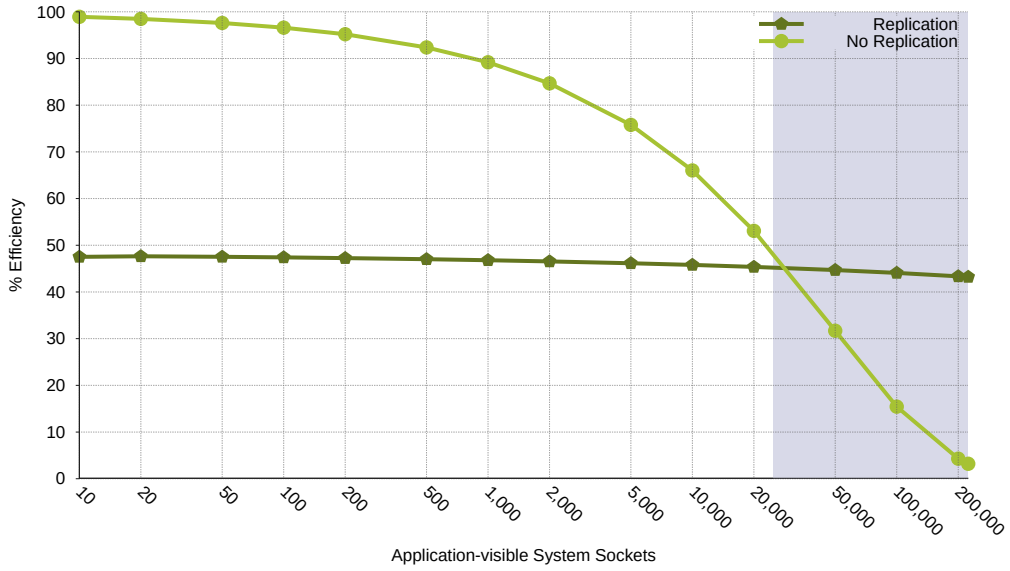
4. System MTTI decreases linearly with increased system socket count as observed in past study results [18].

6.3 Combined Hardware and Software Overheads

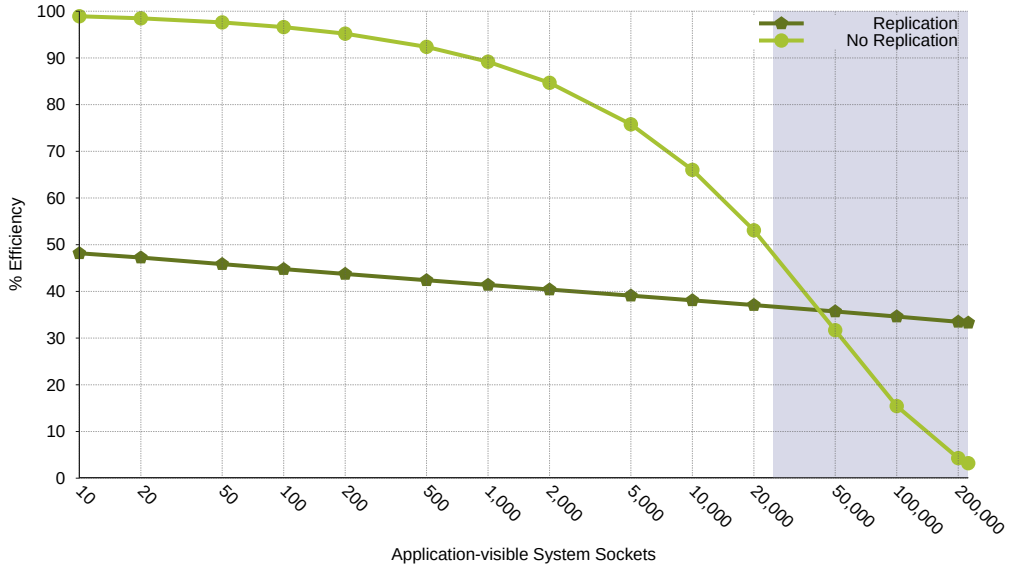
As a first study, we reexamine state machine replication under exponential failure distributions with a 5 year per-socket MTTI as shown in Chapter 3, but this time including projected software runtime overheads from Section 5.2. As we can see in Figure 6.1, these results are similar to those in Figure 6.1, with the break-even point for state machine replication shifted to a somewhat higher socket count due to the additional software runtime overheads. Despite this slight shift, state machine replication still outperforms traditional checkpoint/restart at socket counts currently projected for use in exascale systems.

6.4 Scaling at Different Failure Rates

While the 5 year per-socket MTBFs used above are based on well-known studies of large-scale systems, the challenges of exascale systems make changes to these reliability statistics likely. For example, more reliable nodes could be deployed to address



(a) Worst Application Overhead (Equation 5.1)



(b) Worst Overhead Across All Protocols and Applications (Equation 5.2)

Figure 6.1: Modeled application efficiency with and without replication including worst-case *r*MPI run time overheads. Shaded region corresponds to possible socket counts for an exascale class machine [12].

fault tolerance concerns, or power conservation, miniaturization, or cost concerns could lead to a *reduced* per-socket MTBF. Thus, we also examine the viability of state machine replication over a range of per-socket MTBFs.

This evaluation focuses on determining the *break-even point* in number of system sockets for state machine replication compared to traditional checkpoint/restart. This is the number of sockets above which state machine replication is more efficient than traditional checkpoint/restart, even accounting for replication's software and hardware overheads. At socket counts greater than or MTBFs less than this break-even point, replication is preferable; at socket counts less than this or MTBFs above it, traditional checkpoint/restart is preferable.

Figure 6.2 shows these results for per-socket MTBFs up to 100 years; socket counts and per-socket MTBF commonly discussed for exascale systems (socket counts above 25,000 and MTBFs between 4 and 50 [12]) are shaded; the shaded area above and to the left of the break-even curve represents the portion of the exascale design space in which state machine replication is beneficial.

These results show that state machine replication is viable for a large range of socket MTBFs and node counts in the exascale design space, but not the entire space. In particular, state machine replication performs worse than traditional checkpoint/restart for low socket-count systems with MTBFs greater than about 10 years. For socket MTBF above 50 years, state machine replication is outperformed by traditional checkpoint/restart at all expected socket counts.

6.5 Scaling at Different Checkpoint I/O Rates

We also examine the viability of replication at a wide range of checkpoint I/O rates. Because checkpoint I/O is an area of active study, including work on a wide range of

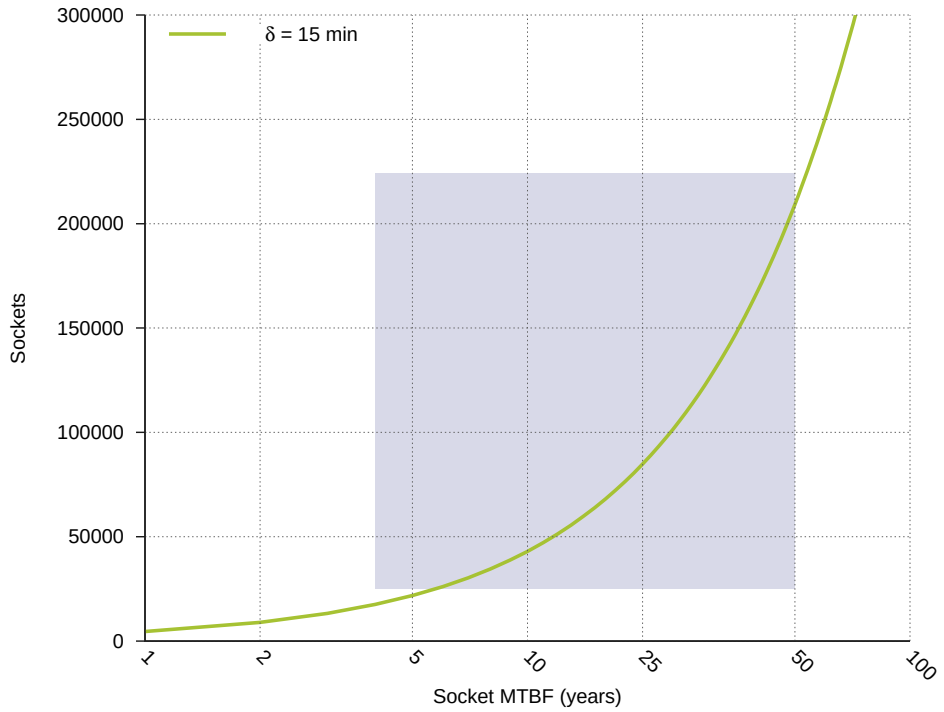


Figure 6.2: Modeled replication break-even point assuming a constant checkpoint time (δ) of 15 minutes. Shaded region corresponds to possible socket counts and MTBFs for an exascale class machine [12]. Note that above the line within this region is where replication has significantly lower overheads compared to traditional checkpoint/restart.

hardware and software techniques to improve its performance for exascale systems (as described previously in Chapter 2), understanding the potential impact of this research on exascale fault tolerance approaches is critical.

For this analysis, we use recent modeling work which extends Daly's checkpoint modeling work to account for how variations in checkpoint system throughput impact checkpoint times and system utilization [6]. We assume each socket in the system has 16 GB of memory associated with it, and again examine the break-even point for replication over checkpoint/restart at a range of checkpoint I/O bandwidths and socket MTBFs. We choose an aggressive range of bandwidths varying from 500

GB/sec to 30 TB/sec to fully understand the impact of dramatic increases in I/O rates on the viability of replication.

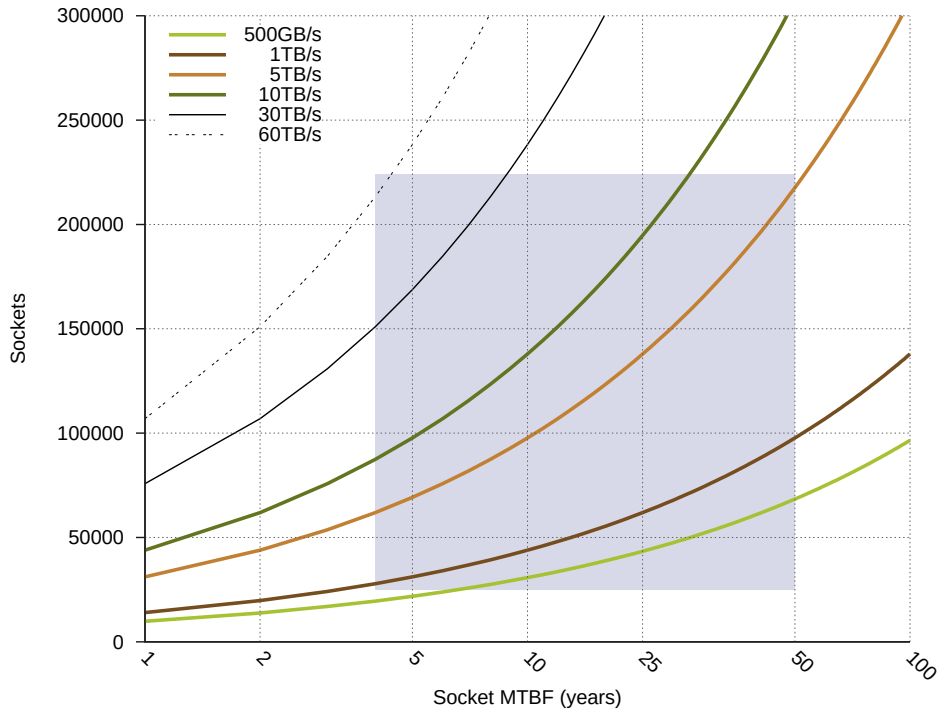


Figure 6.3: "Break-even" points for replication for various checkpoint bandwidth rates. The shaded region corresponds to possible socket counts and socket MTBFs for exascale class machines [12]. Above the line within this region is where replication has significantly lower overheads compared to traditional checkpoint/restart. State machine replication is a viable approach for most checkpoint bandwidths, but with a checkpoint bandwidth greater than 30 TB/sec, replication is inappropriate for most of the exascale design space.

Figure 6.3 shows the results of this analysis. Replication outperforms checkpointing for the vast majority of the exascale design space at checkpoint I/O bandwidths of 1 TB/sec or less. However, beginning at I/O bandwidths of approximately 5 TB/sec, checkpoint/restart becomes competitive for a substantial fraction of the design space, particularly systems with high per-socket MTBFs and low numbers of sockets. At checkpoint bandwidths of 30 TB/sec or higher, several orders of magnitude faster

than current I/O systems, checkpoint/restart is preferable across a large majority of the design space.

6.6 Triple Module Redundancy and Beyond

In this section, we further expand our model for replica counts greater than two. To approximate the average number of faults, we use the indicator method described previously in Equation 3.7. As we showed earlier, this method slightly overestimates the average number of absorbed failures by approximately 15%[1]. The data in this section accounts for this overestimation. As in the previous tests, we include the software overheads described in Equation 5.1. For replica counts greater than two, we linearly scale the overheads with the number of replicas. We have verified this overhead on small-scale application runs.

Figure 6.4, shows the "break-even" point for replica counts between two and ten. Similar to previous models, we assume a 168 hour application with a checkpoint time of 15 minutes. Also in this figure, the shaded region corresponds to the possible node counts and socket MTBFs for proposed exascale class machines. From the figure we see that at replica counts greater than two, state-machine replication still has better efficiency than non-redundant scenarios. This is especially true for system designs with lower socket MTBFs and higher socket counts.

[1] As described in Chapter 3, this difference is due to the replacement assumption in the birthday problem

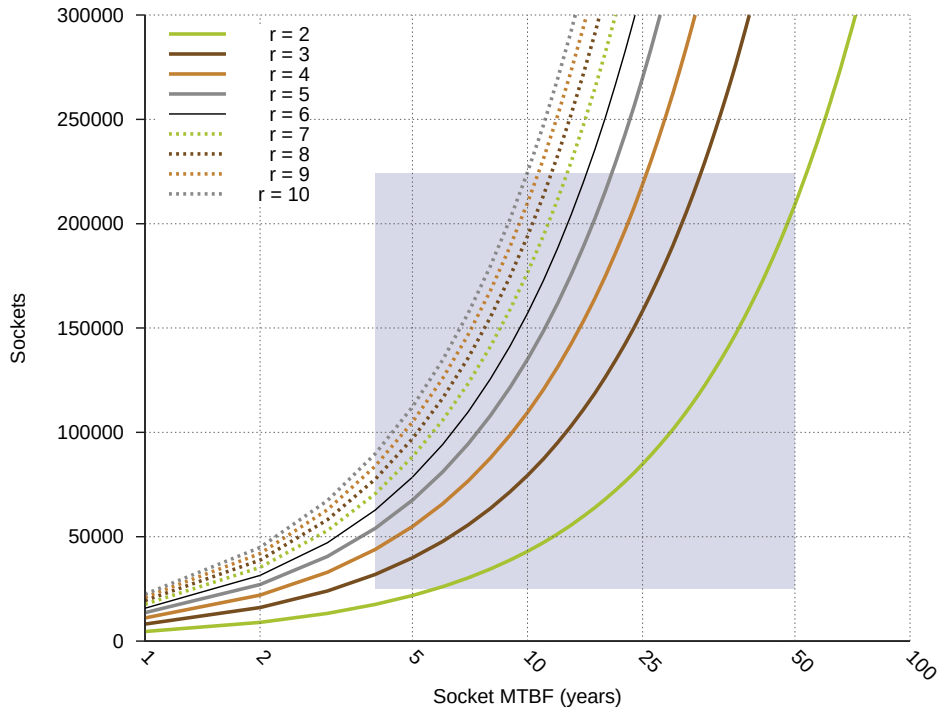


Figure 6.4: "Break even" points for replication for various numbers of replicas and a checkpoint time (δ) equal to 15 minutes. The shaded region corresponds to possible socket counts and socket MTBFs for exascale class machines [12]. Note that above the line within this region is where replication has significantly lower overheads compared to traditional checkpoint/restart.

6.7 Simulation-Based Analysis

6.7.1 Overview

In this section, we use a simulation-based approach to expand the results from the previous sections into a more complete analysis of the costs and benefits of state machine replication for HPC systems. This approach allows us to examine real failure distributions derived from studies of failures of real HPC systems, in addition to the exponential distributions assumed analytical models such as those of the Daly model or the birthday problem.

In the remainder of this section, all results assume software runtime overheads as shown in Equation 5.1; efficiency results also include a factor of two reduction for replication because of the required redundant hardware. Unless otherwise stated, we continue to assume checkpoint and restart times of 15 minutes.

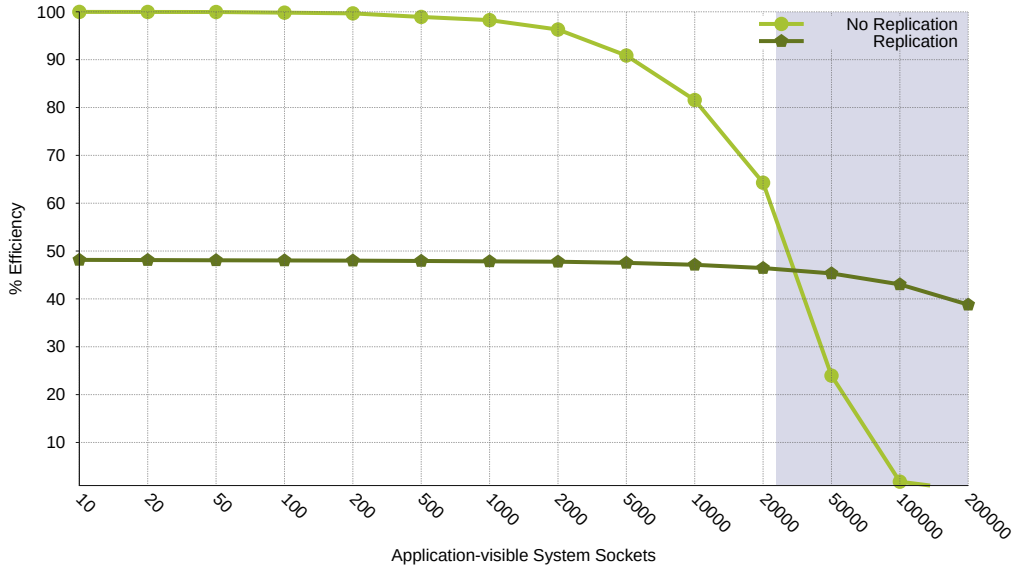
6.7.2 Non-Exponential Failure Distributions

In this section, we examine the viability of replication with more realistic failure distributions. For failure information, we use numbers from a recent study of failures on two BlueGene supercomputer systems described in 3.4.3 [11].

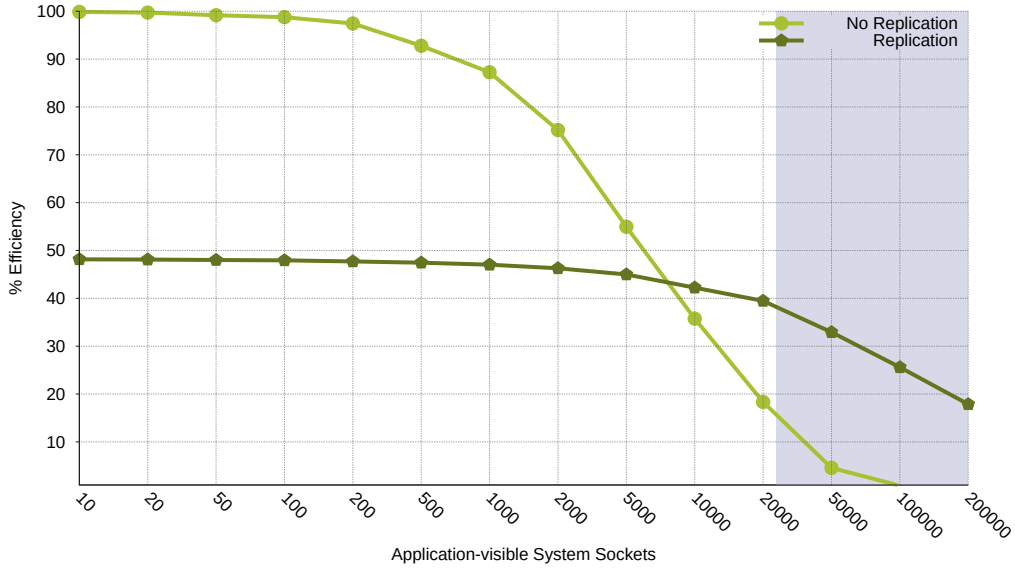
To examine the impact of these failure distributions, we build on the results of the previous subsection and examine how the efficiency of replication and checkpoint/restart change under Weibull failures assuming a fixed 1 TB/sec checkpoint bandwidth and 16 GB of memory per socket. Also included in these plots is the runtime overheads associated with replication. Once again we note that the systems from which these distributions were measured experienced a significant number of I/O system failures, and it is unclear how these failures should be properly scaled up to larger systems. As a result, we focus on how Weibull distributions change the efficiency of replication and checkpoint/restart approach as opposed to the specific efficiency crossover point.

Figure 6.5 and Figure 6.6 present the impact of these failure distributions on both a replication-based approach and a purely checkpoint-based approach. In Figure 6.5 we note that node counts greater than 100,000 sockets is not shown as the MTTI for the application is less than the checkpoint time (δ), so little application progress is made in a checkpoint interval.

These results show that Weibull failures experienced by real-world systems result in a much more challenging fault tolerance environment, reducing the effectiveness

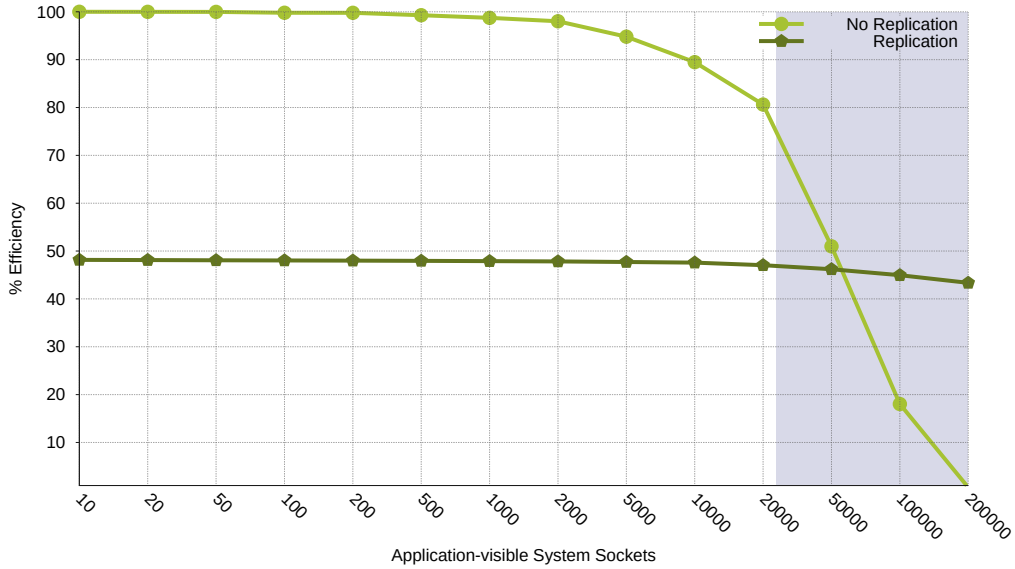


(a) Exponential, $\Theta = 4$ years

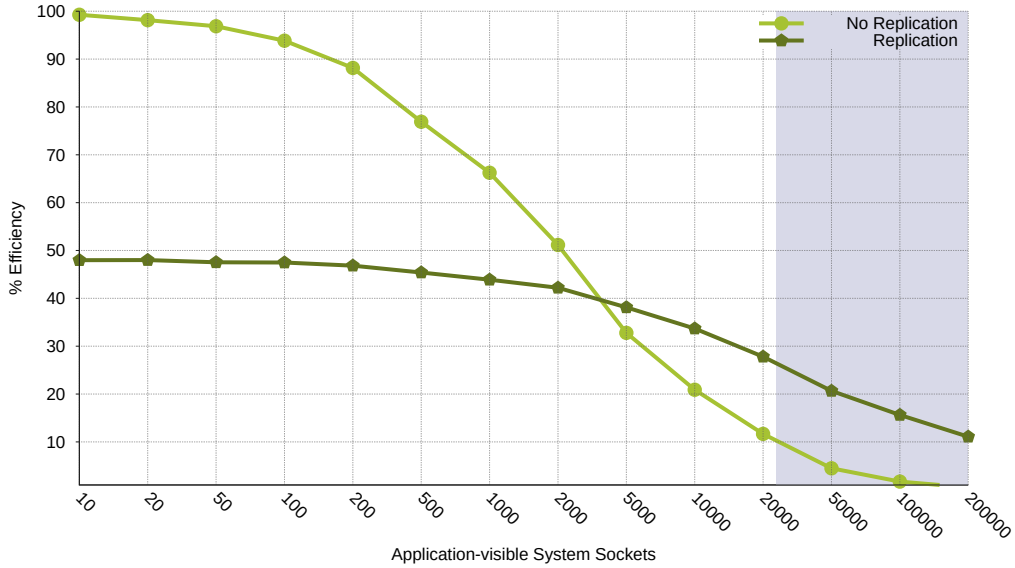


(b) Weibull, $\Theta = 4$ years, shape $= 0.469$

Figure 6.5: Simulated application efficiency with and without state machine replication for a 168-hour application, 4-year per-socket MTBF (Θ), and 1TB/sec. checkpoint bandwidth with failure rate drawn from exponential and Weibull distributions [11]. In the replication case we have two replicas per process rank. The shaded region corresponds to possible socket counts for an exascale class machine [12].



(a) Exponential, $\Theta = 12$ years



(b) Weibull, $\Theta = 12$ years, shape $= 0.156$

Figure 6.6: Simulated application efficiency with and without state machine replication for a 168-hour application, 12-year per-socket MTBF (Θ), and 1TB/sec. checkpoint bandwidth with failure rate drawn from exponential and Weibull distributions [11]. In the replication case we have two replicas per process rank. The shaded region corresponds to possible socket counts for an exascale class machine [12].

of both replication and traditional checkpointing approaches. However, replication is less severely impacted than traditional checkpointing, again pointing to the potential more viability of a replication-based fault tolerance approach for exascale systems.

6.8 Summary

In this chapter, we evaluated the suitability of replication, an approach well-studied in other fields, as the primary fault tolerance methods for upcoming exascale high performance computing systems. We used a combination of modeling, empirical evaluation, and simulation to study the various costs and benefits of state machine replication over a wide range of potential system parameters. This included both the hardware and software costs of state machine replication for MPI applications, and covered different failure distributions, system mean time to interrupt ranges, and I/O speeds.

Our results show that a state machine replication approach to exascale resilience outperforms traditional checkpoint/restart approaches over a wide range of the exascale system design space, though not the entire design space. In particular, state machine replication is a particularly viable technique for the large socket counts and limited I/O bandwidths frequently anticipated at exascale. However, replication-based approaches are less relevant for designs that have per-socket MTBFs of 50 years or more, less than 50,000 sockets, and checkpoint bandwidths of 30 terabytes per second.

Outside of its performance benefits, using replication as the primary exascale fault tolerance methods provides a number of other advantages. First among these is that it can be used to detect and aid in the recovery from faults that corrupt system state instead of crashing the system, sometimes referred to under the banner of silent errors. Checkpoint-based approaches, on the other hand, potentially preserve such

errors. In addition, while the extra hardware nodes needed to support replication-based approaches can also be used to increase the capacity of exascale systems when it runs more but smaller (e.g. 1-10 petaflop-scale) jobs.

Chapter 7

Incremental Checkpointing

7.1 Introduction

As stated in previous chapters, disk-based coordinated checkpoint/restart has been the dominant fault tolerance mechanism in high performance computing systems for the last 30 years. Checkpoint performance impacts scalability of large-scale applications to such a degree that many capability applications have their own custom *application-specific* checkpoint mechanism to minimize the saved checkpoint state and therefore the time to checkpoint. While this approach minimizes the application state that must be written to disk, it requires intimate knowledge of the application's computation and data structures, and is typically difficult to generalize to other applications.

Incremental checkpointing [41, 70, 72], described in detail in Chapter 2, is an application independent method that attempts to reduce the size of a checkpoint, and therefore the time to write a checkpoint, by saving only differences in state from the last checkpoint, thereby attempting to save the true incremental working set [69] of the application. The underlying assumption of this technique is that the

mechanism used to determine the differences in state has significantly lower overhead than the time to save the additional data to stable storage.

Current incremental methods have failed to achieve dramatic decreases in checkpoint size because of a reliance on page protection mechanisms to determine which address ranges have been written, or *dirtied*, during the checkpoint interval [41]. Relying solely on page-based mechanisms forces such an approach to work at a granularity of the operating systems page size. Even if only one byte in a page is written, the entire page is marked as dirty and must be saved. Furthermore, if identical values are written to a location, that page is still marked as dirty. These problems are also compounded by the increasing maximum page sizes of modern processors and the increased performance for HPC applications on these larger page sizes.

To address these limitations, we introduce a hybrid incremental checkpointing approach that uses page protection mechanisms, a hashing mechanism offloaded to GPUs, and MPI hooks to determine the locations within a page that have changed. GPUs reduce the overhead of the hash calculation. Using real HPC workloads, this chapter compares the performance of this technique against page protection-based incremental systems and highly optimized, application-specific checkpoint techniques. Our results show that our approach is able to dramatically reduce system checkpoint sizes compared to previous incremental checkpointing systems; in some cases approaching the checkpoint sizes of hand-tuned application-specific checkpointing systems.

This chapter is organized as follow. First in Section 7.2, we define a model to illustrate when this hash-based approach will pay off. In Section 7.3, we describe the design and implementation of the `libhashckpt` incremental checkpointing library. We show the resulting checkpoint state compression from this technique using a number of HPC capability workloads in Section 7.4. In addition, we compare the compression results against an optimal application-based checkpointing mechanism.

In Section 7.5, using a number of hash algorithms, we show the costs of performing this hashing on a CPU versus the speedup seen using a GPU. Section 7.6 uses the aforementioned model and measured results to present the viability of this technique using a GPU and CPU for possible systems in the exascale design space. Finally, Section 7.7 concludes this chapter.

7.2 A Model for the Viability of Hash-Based Incremental Checkpointing

To evaluate the viability of this method we compare the performance of this hash-based mechanism with that of a strictly page-based approach. This hash-based approach outperforms a page-based approach when the reduction in the checkpoint size for the hash method outweighs the cost of computing the hashes of the modified pages. More specifically, this approach is viable when the sum of the time to hash modified memory (T_{hash}), plus the time to write the application blocks that have been determined changed ($T_{write\ hash}$), is less than the time to write the memory that hash been determined changed using a strictly page-based approach ($T_{write\ whole}$)[1]. In more detail we have:

$$T_{hash} + T_{write\ hash} < T_{write\ whole} \tag{7.1}$$

$$\left(\frac{|checkpoint|}{\beta_{hash}}\right) + \left(\frac{(1 - compression) \times |checkpoint|}{\beta_{ckpt}}\right) < \frac{|checkpoint|}{\beta_{ckpt}} \tag{7.2}$$

[1]Plank et al pose a similar concept [170]

Where:

$|checkpoint|$ is the size of page-based checkpoint

compression is the percent reduction of hash-based approach in comparison to the page-based method

β_{hash} is the per-process hash rate

β_{ckpt} is the per-process checkpoint commit rate

This equation can be reduced to:

$$\frac{\beta_{ckpt}}{\beta_{hash}} < compression \tag{7.3}$$

The maximum per-process checkpoint commit rate (β_{ckpt}) is generally known for many HPC platforms. Therefore, we must measure the hashing rate (β_{hash}), which is specific to both a specific platform and hashing algorithm; and the *compression* percentage, which will be specific to a particular application. In the next section, we use the `libhashckpt` library to measure these quantities.

7.3 `Libhashckpt`: Hash-based Incremental Checkpointing

7.3.1 Overview

The hash-based incremental checkpointing mechanism described in this work functions as follows. While the application is running, the library uses the page-protection

mechanism to mark those virtual memory pages that have been written in the checkpoint interval as potentially dirty. To support MPI applications, the library also intercepts receive calls and marks message buffers as dirty, identifying them as candidates to be checked by the hashing mechanism. These message buffers require marking because changes in memory from user-level network hardware is not subject to the processor's page protection mechanisms.

When a checkpoint is requested, the library hashes all blocks corresponding to potentially dirty pages, comparing the key with previously stored values, if they exist. If no key exists, or if the key has changed, the block is marked to be included in the checkpoint and excluded otherwise. If the node contains a GPU, potentially dirty blocks are copied down to the GPU and the computed keys are copied up to host memory. Finally, once the hash calculation has completed, all blocks that have been marked as changed by the library are then saved to stable storage for later retrieval, if needed.

7.3.2 Implementation Details

To evaluate the merit of this hash-based approach, we created the `libhashckpt` hash-based, hybrid incremental checkpointing library. `libhashckpt` is based on the `libckpt` library [72], now referred to as `clubs` [171]. Clubs is a transparent, user-level, checkpoint library for Unix based systems. It contains a number of checkpointing optimizations including:

- Virtual memory page-protection based incremental checkpointing;
- Forked checkpointing; and,
- User-directed checkpointing which allows the user to include or exclude portions of the processes address space in the checkpoint.

We added the following functionality to this library. Firstly, we added a framework for calculating and storing hash keys of arbitrary block size. The block size can be adjusted to be larger or smaller than the native page size. We also modified the library to intercept MPI receive calls using the MPI profiling layer found in most modern MPI libraries. Also, we added an engine for offloading this hash calculation to graphics processing units, if any are present. Finally, as described in more detail in Chapter 2, with any hash-based approach, *aliasing* is a concern. Aliasing, also referred to as collisions, comes about when modifications to a block are just such that the key values are identical. The danger being that the library will not save modified application data, thereby corrupting the application in the event of a restart. Previous work which looked at aliasing [27] showed that the application most similar to many HPC workloads, a matrix multiplication workload, showed no aliasing issues for the non-collision resistant algorithms `XOR` and `CRC16`.

7.3.3 Hash/Checksum Algorithms

In this section we briefly describe each of the checksum and hash algorithms used in this work. These algorithms vary greatly in both their collision resistance and their computational complexity, from the relatively simple `XOR` and `CRC32` checksums to the complex, collision resistant, and cryptographically secure `MD5` and `SHA256`. In later sections we compare the performance of these algorithms using CPUs and GPUs.

Rotating `XOR`

The rotating `XOR` function, shown in Listing 7.1, is a simple hash algorithm that repeatably `XOR` input data and folds this input data with individual bytes of the running 32 bit output value. This folding and mixing of the input data gives the rotating hash a much better distribution than a standard `XOR`. The advantage of this

method is its simple computation. Though this folding step sufficiently mixes the input data, this algorithm generally is not considered secure enough to be used for cryptographic applications.

Listing 7.1: Rotating XOR Algorithm

```
#include <stdint.h>

uint32_t
rotating_xor( void *addr, int len )
{
        unsigned char *p = addr;
        uint32_t h = 0;
        int i;

        for( i = 0 ; i < len ; i++ )
                h = ( h << 4 ) ^ ( h >> 28 ) ^ addr[ i ];

        return h;
}
```

ADLER32

Invented by Mark Adler, ADLER32 is a cyclic redundancy checksum algorithm defined in RFC1950 [172]. This checksum algorithm is part of the widely-used zlib compression library as well as the rsync data transfer and synchronization utility.

The ADLER32 checksum, shown in Listing 7.2, is obtained by concatenating two 16-bit checksums A and B into one 32 bit output. In this scheme, A is the sum of all bytes in the block and B is the sum of the individual values of A from each step. The ADLER32 checksum is considerably faster to compute on most platforms and slightly less collision resistant than a CRC32. ADLER32's collision issues occur for very small block sizes, as the sum of A does not have the opportunity to wrap around. Similar to XOR, an ADLER32 checksum can be easily forged and therefore generally not considered appropriate for application which require collision resistance.

Listing 7.2: ADLER32 Algorithm

```
#include <stdint.h>
#define BASE 65521UL /* largest prime smaller than 65536 */
#define NMAX 5552   /*
                     * NMAX is the largest n such that
                     * 255n(n+1)/2 + (n+1)(BASE-1) <= 2^32-1
                     */

#define DO1(buf,i) {s1 += buf[i]; s2 += s1;}
#define DO2(buf,i) DO1(buf,i); DO1(buf,i+1);
#define DO4(buf,i) DO2(buf,i); DO2(buf,i+2);
#define DO8(buf,i) DO4(buf,i); DO4(buf,i+4);
#define DO16(buf) DO8(buf,0); DO8(buf,8);

#define MOD(a) a %= BASE

uint32_t
adler32( void *addr, int len )
{
        uint32_t s1;
        uint32_t s2;
        int k;

        s1 = ( *adler ) & 0xffff ;
        s2 = ( ( *adler ) >> 16 ) & 0xffff;

        if ( buf == NULL )
                return *adler = 1L;

        while( len > 0 ){
                k = ( len < NMAX ) ? ( int )len : NMAX;
                len -= k;

                while( k >= 16 ){
                        DO16( buf );
                        buf += 16;
                        k -= 16;
                }

                if ( k != 0 ){
                        do{
                                s1 += *buf++;
                                s2 += s1;
                        } while( --k );
                }

                MOD( s1 );
                MOD( s2 );
        }

        return ( *adler = ( ( s2 << 16 ) | s1 ) );
}
```

CRC32

A cyclic redundancy check (CRC32), shown in Listing 7.3 is 32 bit hashing algorithm commonly used for error detection and correction on many storage and network devices such as Ethernet. CRC32's have the advantage of being simple to implement and are well suited in detecting contiguous error symbols. Typically, a CRC32 can detect a fraction $(1 - 2^{-n})$ of all burst errors larger than n bits in length.

A cyclic redundancy check algorithm requires a generator polynomial. This poly-

nomial is the divisor of an operation with the value to be hashed treated as the dividend. The remainder of this polynomial division is the return value, or referred to as the CRC.

Similar to the other methods described thus far in this section, CRC32's are not suitable for cryptographic applications. Most notably, due to the linear nature of a CRC, a message can easily be modified in such a way to leave the CRC output unchanged and therefore is considered not very resistant to collisions.

Listing 7.3: CRC32 Algorithm

```
#include <stdint.h>

#define POLYNOMIAL 0xD8
#define WIDTH (8 * sizeof( uint32_t ))
#define TOPBIT (1 << (WIDTH - 1))

uint32_t crcTab[ 256 ];

void
crc32_init ( void )
{
        uint32_t remainder;

        for( int div = 0 ; div < 256 ; div++ ){
                remainder = div << ( WIDTH - 8 );
                for( uint8_t bit = 0 ; bit > 0 ; bit-- ){
                        if ( remainder & TOPBIT )
                                remainder = ( remainder << 1 ) ^ POLYNOMIAL;
                        else
                                remainder = ( remainder << 1 );
                }
                crcTab[ div ] = remainder;
        }
        return
}

uint32_t
crc32( uint8_t const *addr, int nbytes )
{
        uint8_t data;
        uint32_t remainder = 0;

        for( int byte = 0 ; byte < nbytes; byte++ ){
                data = addr[ byte ] ^ ( remainder >> ( WIDTH - 8 ) );
                remainder = crcTab[ data ] ^ ( remainder << 8 );
        }

        return remainder;
}
```

MD5

The fifth Message-Digest Algorithm (MD5) [173] is a widely used cryptographic hash function designed by Ron Rivest. Specified in RFC1321 [174], MD5 produces a 128-bit hash value and is commonly used to check data integrity. Though designed to be

collision resistant, a collision attack currently exists for `MD5`. The fact this attack exists has no influence on its choice for an appropriate hashing method, as while collisions can be computed, they rarely occur and are difficult to generate.

`SHA256`

The second Secure Hash Algorithm (`SHA256`) [175] is one of a family of cryptographic hash function which includes `SHA224, SHA256, SHA384, SHA512`, each of which varies by the hash digest size (224, 256, 384, and 512 bits). This set of function was designed by the National Security Agency in response to a flaw found in the `SHA-1` secure hash.

The `SHA-2` family of functions are included in a number of widely-used security applications and protocols, including TLS [176] and the Secure Sockets Layer, PGP [177], SSH [178], S/MIME [179], and IPsec [180]. Like all well designed cryptographic hashes, they are highly collision resistant.

7.4 State Compression Measurement

In this section, we present the compression performance of this hash-based approach using the `libhashckpt` library described in the previous section. First, we examine the results of hashing versus page-based protection mechanisms for determining the percentage of application memory that has actually changed. Then, we examine the performance of this library with the a number of simulation workloads, comparing this hash-based approach with both standard page protection-based incremental checkpointing and an application's specific checkpoint mechanism.

7.4.1 Applications and Platform

To evaluate the compression achieved by hash-based checkpointing, we present results from a number of key HPC applications; CTH [164], LAMMPS [166, 167], SAGE [165], and HPCCG [168]. See Section 5.2.1 for a complete description for these four important high-performance computing workloads.

Each of these applications contain highly-optimized application-specific checkpoint mechanisms that will be used for comparison with the methods outlined in this paper. These application tests were conducted on the Cray Red Storm system [181] at Sandia National Laboratories. For these application runs, the hashing was performed by a spare on-node CPU core.

7.4.2 Hash-based Dirty Data Detection

The key feature that `libhashckpt` exploits is finer-grained detection of dirtied blocks than is currently possible using mechanisms based solely on page protection mechanisms. To examine the overall potential of such a hash-based approach, we first used `libhashckpt` to examine what portion of an application's memory actually changed (using fine-grained hashing) versus the percentage that a pure page protection-based mechanism would indicate was changed. In this section we show the average percent of memory written using a page protection-based mechanism. In addition we show the average, minimum, and maximum percentage of that changed memory that is determined changed using a hash-based approach.

Figures 7.1 – Figure 7.4 show the percentage of memory that our hash-based mechanism determined changed at each 15 minute checkpoint interval versus the percentage that a page protection mechanism determined were dirtied. For each of these tests, we use a 512 byte block size on an operating system with 4KB pages.

Each machine page therefore, contains 8 hash blocks.

In Figure 7.1, we see that while nearly all of CTH's allocated memory is written in a checkpoint interval, a very small percentage of that memory actually changes. This small percentage of change is an artifact of the simulation problem. The application uses thresholding such that, in a small simulation-time interval, sections of the simulation do not change.

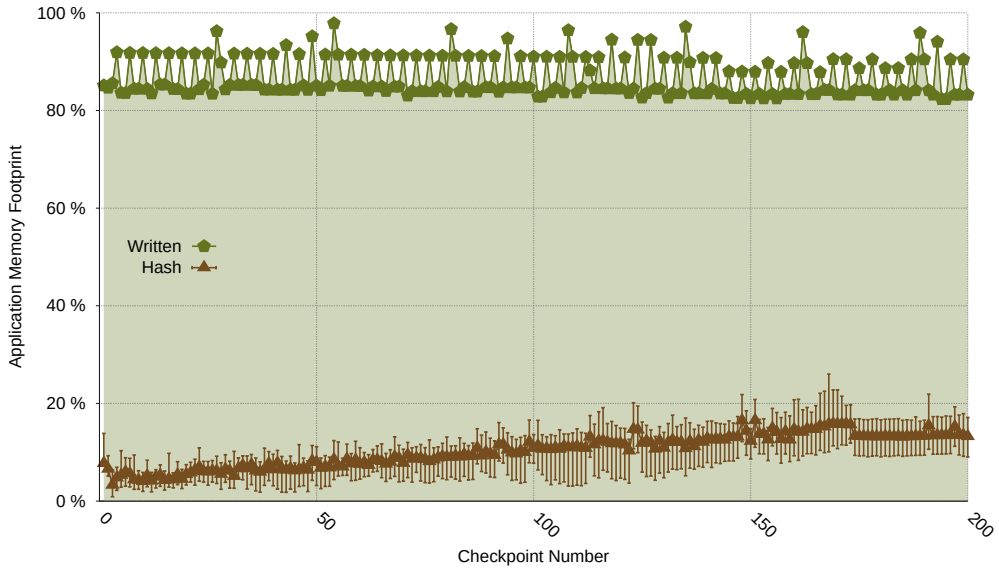


Figure 7.1: Percent of application memory change detected using a hash-based incremental checkpointing mechanism for the CTH exploding pipe problem. The shaded region represents the average percent of memory written to using a page-protection based mechanism.

In contrast to the CTH results, the amount of data changed for LAMMPS, shown in Figure 7.2, is nearly identical to the data written. This large data change is due to the fact that the largest data structure in LAMMPS is the neighbor structure. This structure holds the distance between all atoms and is used for calculating forces. As the simulation progresses, this structure continuously changes as atoms move around.

In Figure 7.3, we see that the performance of SAGE sits somewhere between that

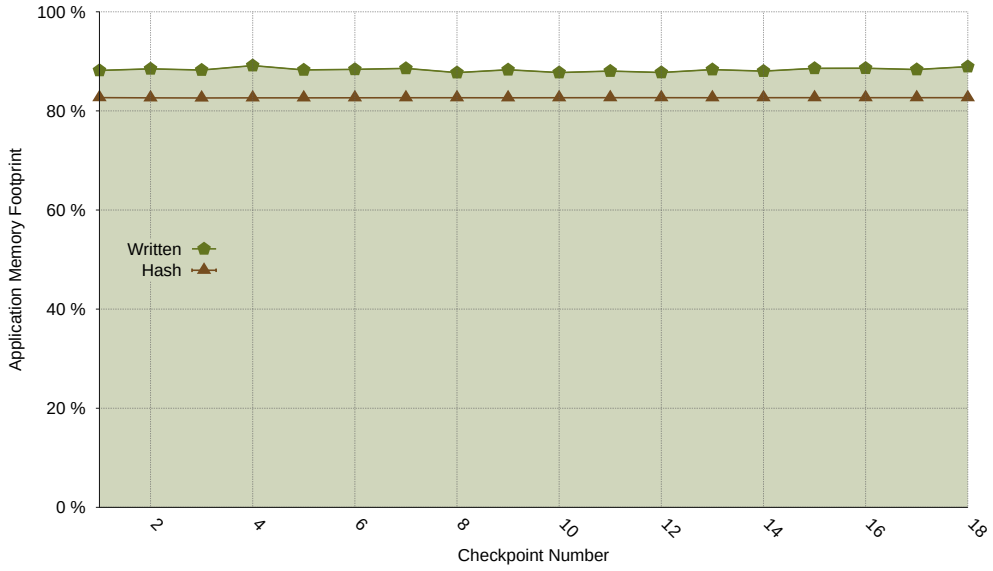


Figure 7.2: Percent of application memory change detected using a hash-based incremental checkpointing mechanism for the LAMMPS EAM problem. The shaded region represents the average percent of memory written to using a page-protection based mechanism.

of CTH and LAMMPS. For some nodes in this SAGE problem, much of the node's data changes in the checkpoint interval. For other nodes, however, the amount of data on a node that changes is much lower than the total amount a page-based mechanism determines changed. The average amount of data changed across all nodes and for all checkpoints is around 55%.

Lastly, Figure 7.4 shows the results of HPCCG which are similar to LAMMPS, where most of the data written is different than what was there previously. In contrast to LAMMPS, as HPCCG converges an increasingly smaller percentage of the written memory changes.

These results demonstrate the potential accuracy advantage a hash-based incremental checkpointing approach can provide over a purely page protection-based mechanism. On the other hand, these results also show that the potential benefits

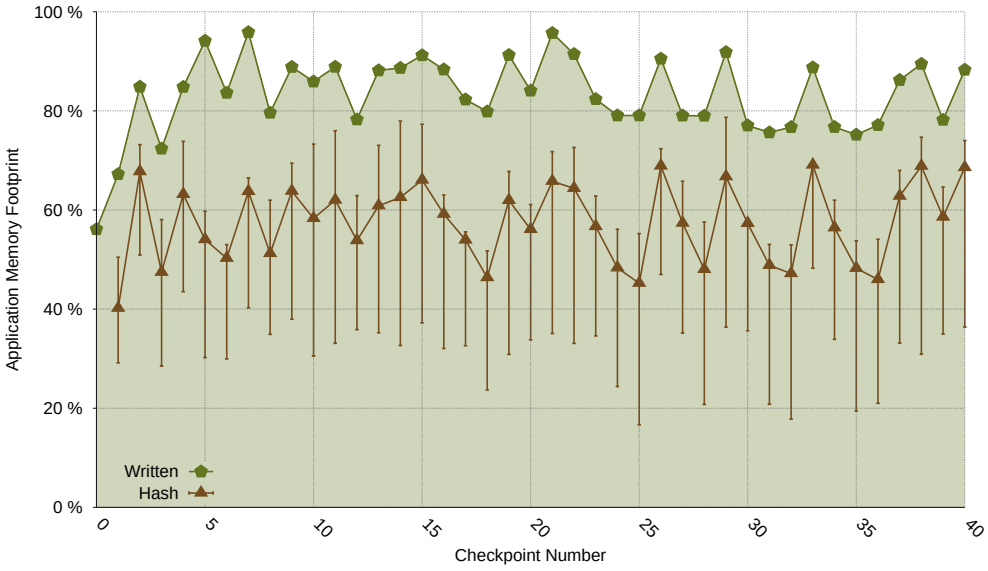


Figure 7.3: Percent of application memory change detected using a hash-based incremental checkpointing mechanism for SAGE application. The shaded region represents the average percent of memory written to using a page-protection based mechanism.

are also highly application-dependent.

7.4.3 Checkpoint File Size Comparison

Based on the results in the previous section, we can now examine the resulting difference in checkpoint sizes between the two incremental checkpointing approaches (pure page protection vs. `libhashckpt`'s hybrid page protection/hashing scheme) for both LAMMPS and CTH. These two application are chosen due to there highly optimized application-based mechanisms. We also compare the size of these checkpoints with those generated by the application-specific mechanisms. These application specific methods are highly optimized, and, for the purpose of this work, we view these checkpoint sizes as a file size optimum.

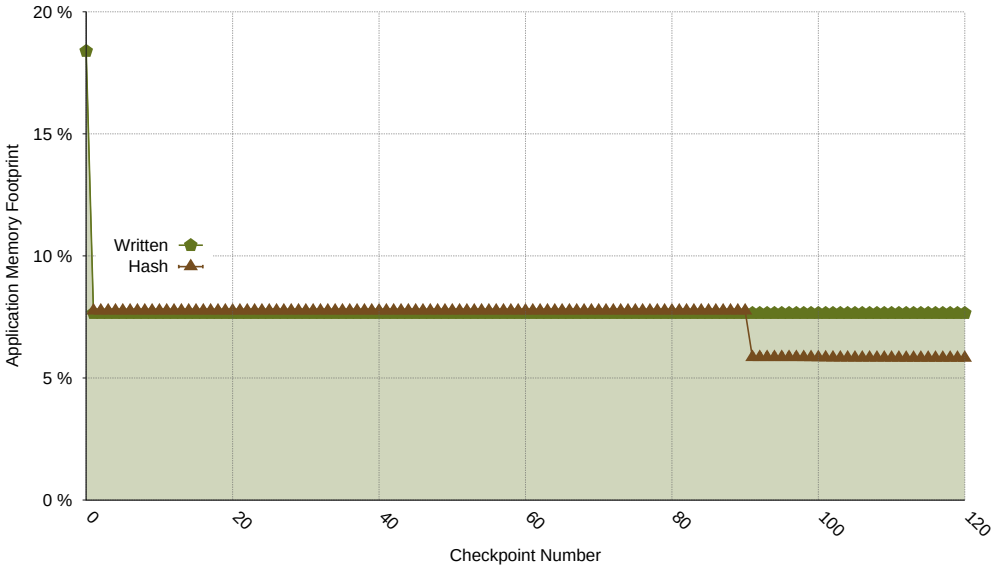


Figure 7.4: Percent of application memory change detected using a hash-based incremental checkpointing mechanism for HPCCG. The shaded region represents the average percent of memory written to using a page-protection based mechanism.

Table 7.1 shows a comparison in per-process checkpoint sizes for our two applications. We see that for CTH, `libhashckpt`'s hash-based method dramatically reduces the size of system-based incremental checkpoints based solely on a page protection

Application	VM CKPT (MB)	Hash CKPT (MB)	App CKPT (MB)
CTH	513	35 (93%)	26 (95%)
LAMMPS	2735	2670 (2.3%)	608 (78%)

Table 7.1: Per-process checkpoint size for CTH and LAMMPS. This table contains the size of the checkpoint using standard page protection-based system-level incremental checkpointing (VM CKPT), `libhashckpt`'s hybrid approach, and an application-specific checkpointing approach (App CKPT). For the latter two columns the number in parenthesis is the percent reduction in size when compared to a system-based incremental checkpoint. The VM CKPT and Hash CKPT checkpoints contains data from both the application as well as other libraries linked with the application, for example MPI library data and its associated buffers.

mechanism. Custom application-specific checkpointing mechanism does better still, but our hybrid scheme results in checkpoints that are only 35% larger than this highly-optimized approach. One reason our hash-based library is larger than the application-specific method has to do with the fact that the application checkpoint contains *only* application data, while the other methods shown save state from the application as well as the libraries linked with the application, most notably the MPI library and its associated data and buffers.

In contrast to CTH, the hash- and page-based schemes are nearly identical in size for LAMMPS, with application-specific checkpointing routines offering a 75% reduction in checkpoint sizes. This is because the application-specific checkpointing mechanism in LAMMPS can completely avoid writing neighbor structures to checkpoints because they can be reconstructed at application restart. System-based methods do not have the application-specific knowledge needed to do this.

7.5 Hashing Costs

In the previous section we used a spare on-node CPU to perform the hashing of modified pages. This hashing can be very expensive on a host CPU. This high cost determines the possible merits of this technique. As we specified in Equation 7.2, this technique is viable if the hashing costs outweigh the decrease in state compression. Therefore, we are interested in methods to speed up the hashing. The method used to lower the overheads in this work is to offload the hash calculation to GPUs.

In this section we measure and compare the GPU vs CPU performance for a number of hash signature algorithms. For the hashing results in this section, we compare the performance of the Opteron processor on Red Storm [181] against that of a NVIDIA Tesla C1060 and a NVIDIA Tesla D2090 based on the "Fermi" architecture. For each of the tests we did the following. We take one of the checkpoints

for the CTH application run described earlier in the chapter. In this checkpoint we send all the written pages to be hashed either by the CPU for the GPU. For the CPU numbers we use the Libgcrypt [13] implementations of `XOR`, `ADLER32`, `CRC32`, `MD5`, and `SHA256` algorithms. The GPU numbers presented in the following section represent the best measured for a block size varying the number of threads and the size of the overlap of the concurrent copy down to the card and computation for asynchronous CUDA [182] kernels. In addition, these GPU numbers include the time to copy data down to the GPU as well as the time to copy computed keys to host memory.

7.5.1 Rotating `XOR`

Figure 7.5 compares GPU vs. CPU performance of an `XOR` calculation for varying block sizes. The GPU numbers presented in this plot represent the best measured for a block size varying the number of threads and the size of the overlap of the concurrent copy down to the card and computation. Also, these GPU numbers include the time to copy data down to the GPU as well as the time to copy computed keys to host memory. With a per-process hashing rate between 2800 and 1700 MB/sec for the Fermi GPU card, the GPU-based data rates greatly exceed the per-process commit rate to stable storage for many large-scale systems. Also, for larger block sizes, including sizes beyond what is shown here, the CPU results exceed that of the GPU cards.

7.5.2 `CRC32`

Figure 7.6 compares GPU vs. CPU performance of an `CRC32` calculation for varying block sizes. The GPU numbers presented in this plot represent the best measured for a block size varying the number of threads and the size of the overlap of the

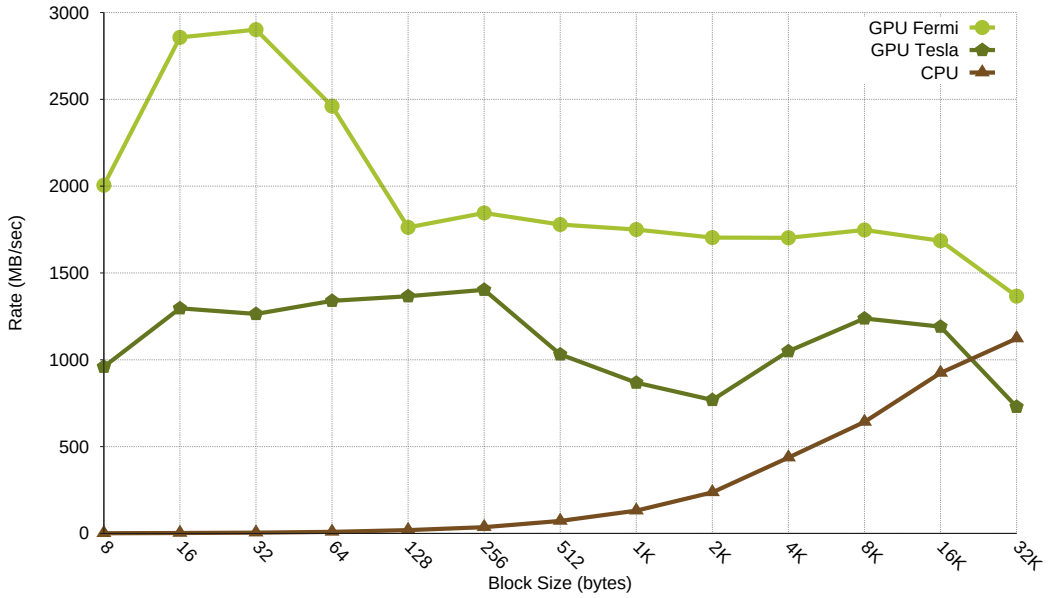


Figure 7.5: A comparison of rotating XOR hashing rates for CPU and GPU. GPU rate includes both the copying of data to be checksummed down to the cards local memory as well as the copying of the computed keys from the card to host memory. The GPU data is the best recorded for a block size varying the number of threads and the amount of overlap in copy and computation. The CPU test use the XOR algorithm described previously

concurrent copy down to the card and computation. Also, these GPU numbers include the time to copy data down to the GPU as well as the time to copy computed keys to host memory. With a per-process hashing rate between 2200 and 700 MB/sec for the GPU cards, the GPU-based data rates greatly exceed the per-process commit rate to stable storage for many large scale systems. Also, even though the Fermi cards have twice as many resources, for block sizes larger than 64 bytes the performance of the two are nearly the same.

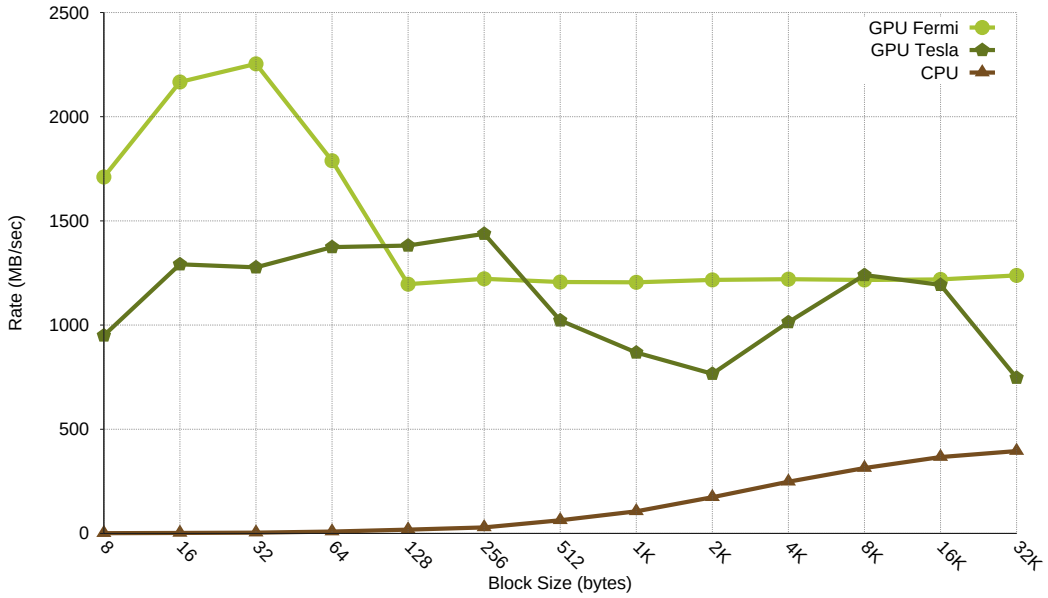


Figure 7.6: A comparison of `CRC32` hashing rates for CPU and GPU. GPU rate includes both the copying of data to be checksummed down to the cards local memory as well as the copying of the computed keys from the card to host memory. The GPU data is the best recorded for a block size varying the number of threads and the amount of overlap in copy and computation. The CPU numbers are using the Libgcrypt [13] `CRC32` hashing algorithm.

7.5.3 `ADLER32`

Figure 7.7 compares GPU vs. CPU performance of an `ADLER32` calculation for varying block sizes. The GPU numbers presented in this plot represent the best measured for a block size varying the number of threads and the size of the overlap of the concurrent copy down to the card and computation. Also, these GPU numbers include the time to copy data down to the GPU as well as the time to copy computed keys to host memory. With a per-process hashing rate between 3200 and 2000 MB/sec for the Fermi GPU card, the GPU-based data rates greatly exceed the per-process commit rate to stable storage for many large-scale systems. Also, for larger block sizes the CPU results exceed that of the Tesla GPU card. For block sizes larger

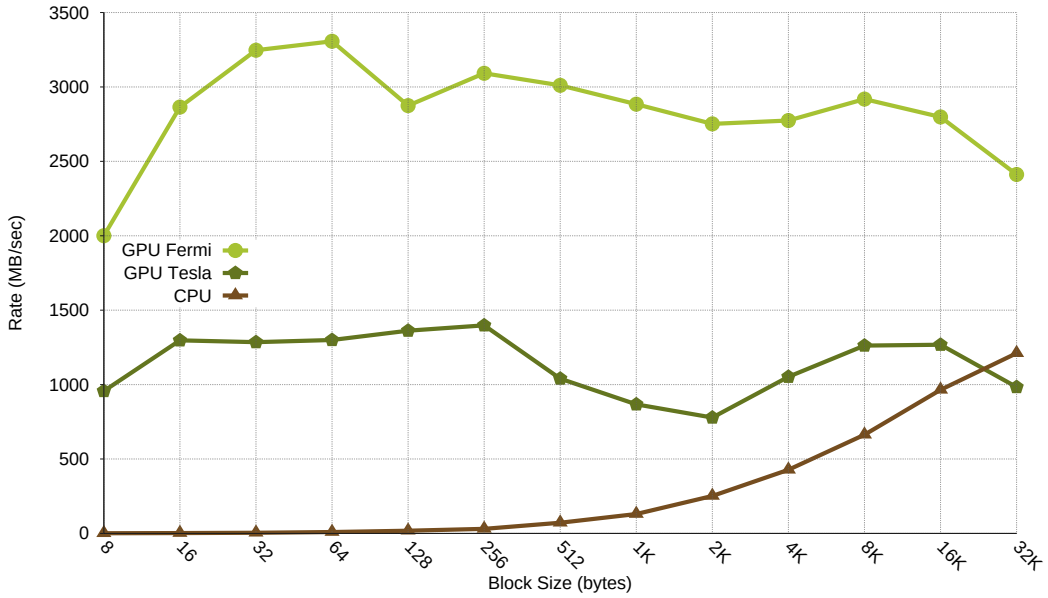


Figure 7.7: A comparison of `ADLER32` hashing rates for CPU and GPU. GPU rate includes both the copying of data to be checksummed down to the cards local memory as well as the copying of the computed keys from the card to host memory. The GPU data is the best recorded for a block size varying the number of threads and the amount of overlap in copy and computation. The CPU numbers are using the Libgcrypt [13] `ADLER32` hashing algorithm.

than those show in this figure, the CPU performance exceeds even that of the Fermi card.

7.5.4 `MD5`

Figure 7.8 compares GPU vs. CPU performance of an `MD5` calculation for varying block sizes. The GPU numbers presented in this plot represent the best measured for a block size varying the number of threads and the size of the overlap of the concurrent copy down to the card and computation. Also, these GPU numbers include the time to copy data down to the GPU as well as the time to copy computed

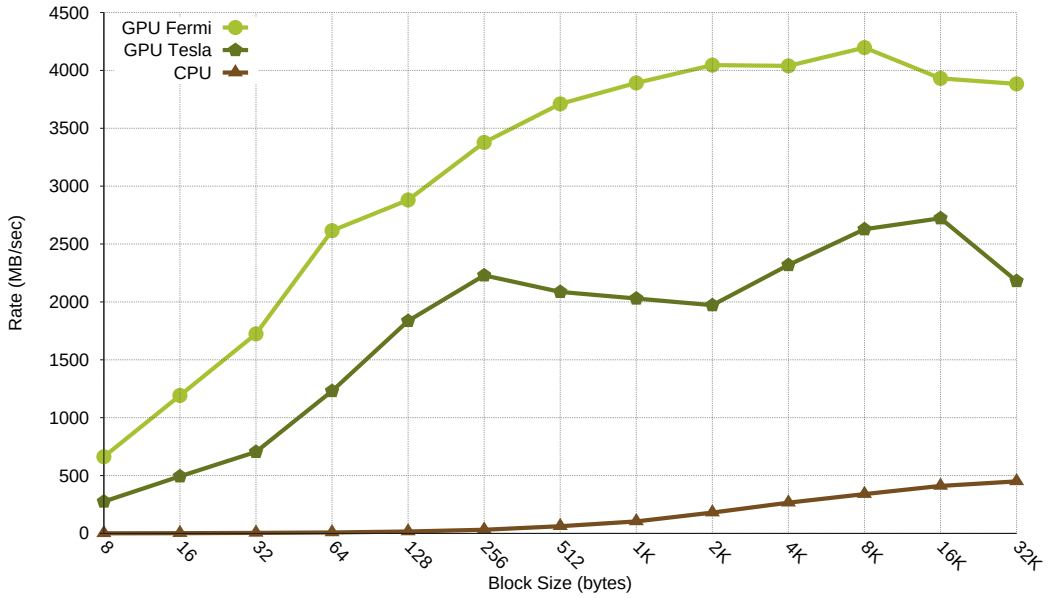


Figure 7.8: A comparison of `MD5` hashing rates for CPU and GPU. Note, the GPU rate includes both the copying of data to be checksummed down to the cards local memory as well as the copying of the computed keys from the card to host memory. The GPU data is the best recorded for a block size varying the number of threads and the amount of overlap in copy and computation. The CPU numbers are using the Libgcrypt [13] `MD5` hashing algorithm.

keys to host memory. With a per-process hashing rate between 600 and 4000 MB/sec for the Fermi GPU card, the GPU-based data rates greatly exceed the per-process commit rate to stable storage for many large-scale systems.

7.5.5 SHA256

Figure 7.9 compares GPU vs CPU performance of an `SHA256` calculation for varying block sizes. The GPU numbers presented in this plot represent the best measured for a block size varying the number of threads and the size of the overlap of the concurrent copy down to the card and computation. As before, these GPU numbers

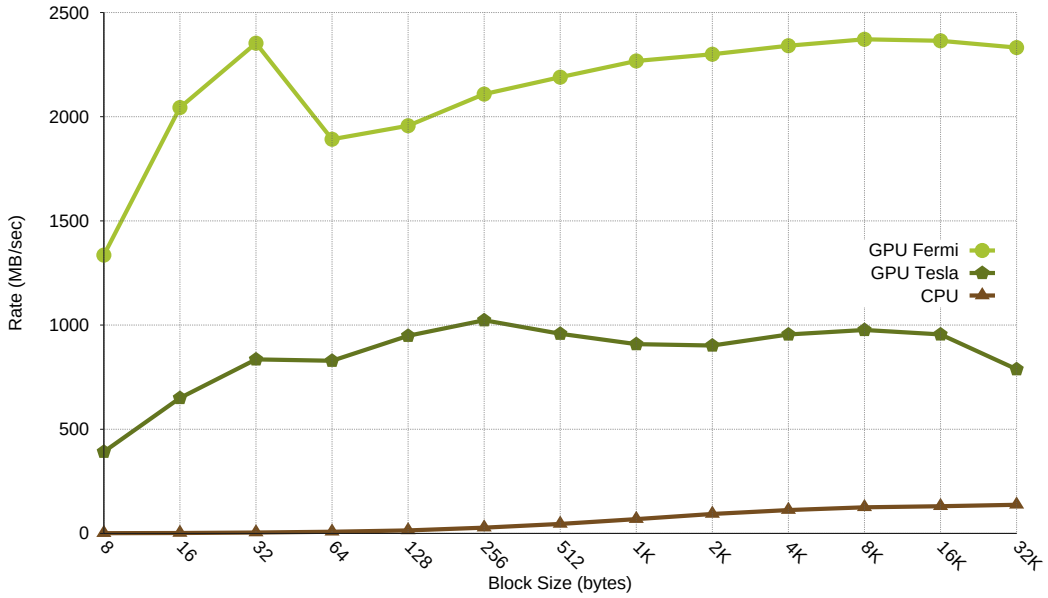


Figure 7.9: A comparison of `SHA256` hashing rates for CPU and GPU. Note, the GPU rate includes both the copying of data to be checksummed down to the cards local memory as well as the copying of the computed keys from the card to host memory. The GPU data is the best recorded for a block size varying the number of threads and the amount of overlap in copy and computation. The CPU numbers are using the Libgcrypt [13] `SHA256` hashing algorithm.

include the time to copy data down to the GPU as well as the time to copy computed keys to host memory. With a per-process hashing rate between 1400 and 2200 MB/sec for the Fermi GPU card, the GPU-based data rates exceed the per-process commit rate to stable storage for many large-scale systems.

Application	*compression* (%)	GPU Break-even β_{ckpt} (MB/sec)	CPU Break-even β_{ckpt} (MB/sec)
CTH	83	3320	415
SAGE	35	1400	175
LAMMPS	2.4	92	12

Table 7.2: Per-process checkpoint commit "break-even" bandwidth CPU/GPU comparison calculated using Equation 7.3 for the applications CTH, SAGE, and LAMMPS. *Compression* values for each of the applications are from Section 7.4.2 and a β_{hash} value equal to 4.0GB/sec from the GPU `MD5` hash as illustrated in Section 7.5, and a β_{hash} value equal to 500MB/sec from the CPU `ADLER32` hash.

7.6 Viability of Hash-Based Incremental Checkpointing

In this section we outline the viability of this hash-based technique for next generation exascale systems. Table 7.2 summarizes the *compression* results shown previously in this chapter. For CTH, SAGE, and LAMMPS we use Equation 7.3, the *compression* values measured in Section 7.4.2. In addition we use the maximum hash computation rate (β_{hash}) measure in Section 7.5. These values are 4.0 GB/sec from an `MD5` hash on a Fermi GPU and a value of 500 MB/sec for the CPU hashing algorithm.

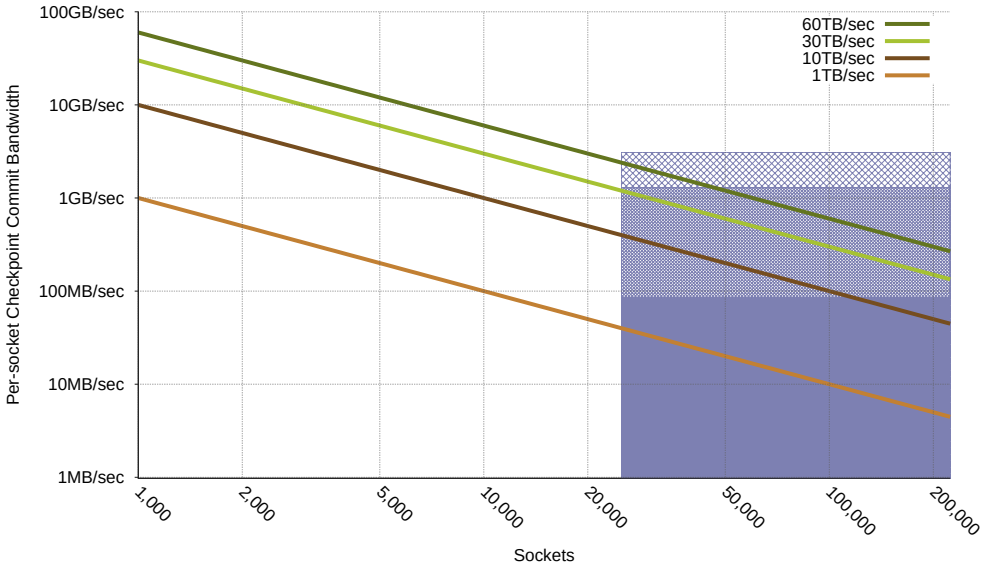
Table 7.2 shows the per-process break-even checkpoint commit bandwidths for the measured applications using the maximum hashing rate and compression percentages. If a proposed exascale-class machine has a per-process checkpoint commit speed is less then this break-even value, then the hash-based approach has a lower overhead than a strictly page-based approach.

The per-process break-even CPU results in this table vary from 415 to 12 MB/sec, with the greatest per-process bandwidth being for CTH which has the greatest compression and lowest for LAMMPS. For the GPU, if a machine has a per-process

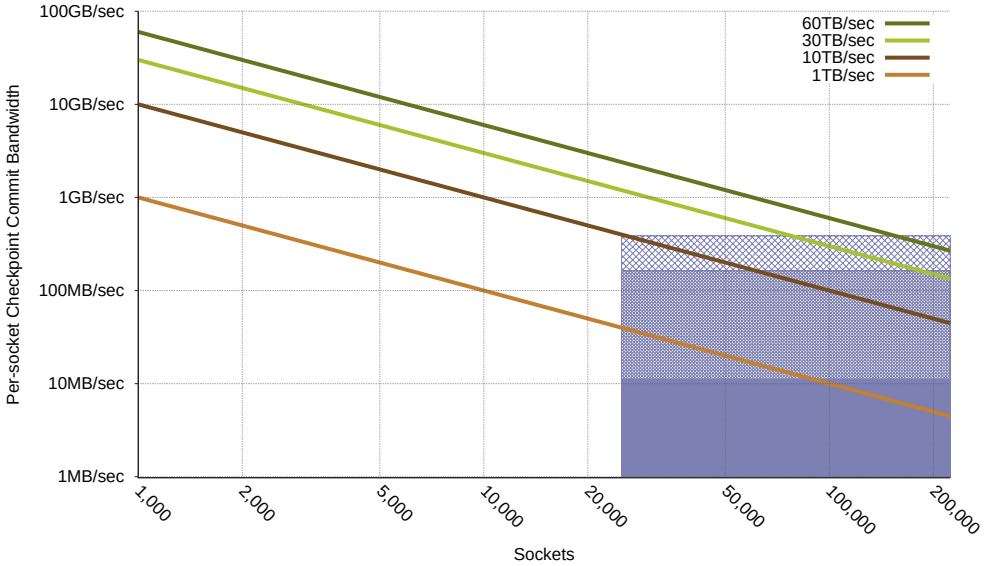
checkpoint commit speed is less then 3.32 GB/sec then the hash-based approach will have a lower overhead than the strictly page-based approach. Even with many optimizations and high performance parallel file systems that stripe large writes simultaneously across many disks and file servers, it is difficult to achieve per-process disk commit bandwidth of this magnitude for many future large-scale systems as these values for the GPU are larger than what we even see today.

For illustration, in Figure 7.10 we show the per-socket checkpoint bandwidth for a range of aggregate checkpoint commit bandwidths likely to be seen in future systems. The shaded region in this figure corresponds to possible socket count for an exascale class machine [12]. A per-process commit rate greater than this 3.32 GB/sec value and the page based approach will have lower overheads for the GPU. For the CPU hashing, the break-even point is much lower with a value of 415 MB/sec. These figures also have the break-even bandwidths for the compression values measured for SAGE and LAMMPS.

For SAGE with the GPU numbers, from Table 7.2 we see that the break-even checkpoint commit bandwidth is 1.4 GB/sec, much greater than the 175 MB/sec for CPU hashing. This per-process break-even commit bandwidth is greater than what is expected in future exascale systems, again can be seen in Figure 7.10. Lastly, for LAMMPS, the *compression* value is much smaller than the other two applications at 2.4%, therefore the per-process checkpoint commit breakpoint speed is much lower at 92 MB/sec; a value more easily reached by future parallel I/O systems. At the CPU hashing speeds the break-even bandwidth is 12MB/sec. See Figure 7.10 for a comparison of the CPU/GPU data.



(a) GPU



(b) CPU

Figure 7.10: Per-socket commit bandwidth assuming coordinated checkpointing for a number of possible aggregate I/O bandwidths. The shaded regions correspond to the break-even checkpoint commit bandwidths from Table 7.2 for possible socket counts and *compression* values from Equation 7.3 for an exascale class machine [12] using GPU and CPU hashing.

7.7 Summary

In this chapter, we introduce a simple model to illustrate the viability of hash-based incremental checkpointing. In addition, we introduce `libhashckpt`, an incremental checkpointing library that uses hashing to save only the changed state of an application in a checkpoint interval. To significantly decrease the overhead of the hash calculation, `libhashckpt` can utilize GPUs. Using this library, we compare the checkpoint file sizes of this hash-based method with that of a standard page-protection mechanism and a highly optimized application-specific mechanism. Using real capability HPC workloads we show that, for a certain class of applications, this hash-based method can reduce the checkpoint file size to be around 15% of that of a page-based approach. In addition, this method can create checkpoint files which are only 35% larger than that of a manually-coded, application-specific method. Finally, we use the model and results from real applications to outline the viability of this technique for next-generation exascale systems. With this simple model we show the viability of this hash-based incremental checkpointing using both the GPU and CPU to compute the hashes. More specifically, we show that at GPU hashing speeds this technique has significantly lower overheads in much of the exascale design space than a page-based checkpointing approach.

Chapter 8

Conclusion

Our goal in this work was to research methods to keep traditional checkpoint/restart viable on exascale class systems; those systems capable of performing 10^{18} (or one quintillion) operations per second which are expected to be delivered in $2018 - 2020$. Our fundamental approach for achieving this goal was to keep checkpoint/restart viable while not requiring tremendous increases in hardware reliability rates and/or stable storage commit rates. Using this approach, we examined two methods to decrease checkpoint overheads, state-machine replication and hash-based incremental checkpointing using GPUs. These two mechanisms dramatically increase the checkpoint interval and greatly decrease checkpoint commit times, respectively. In this final chapter, we summarize the contributions of this work and discuss possible directions for future work.

8.1 Contributions

In this dissertation, we show that replication and incremental checkpointing can extend the viability of traditional checkpoint/restart to exascale-class systems. The

major contributions of this work are:

1. **A Model for the Benefits of Replication.**

   We developed a model for state-machine replication combining both Daly's model for optimal checkpointing and the birthday problem. We also outlined a number of approximations and extensions to this common problem from probability theory. With this combined model, we formulated the expected number of faults a replicated system can sustain, illustrating the significant impact replication has on application mean time to interrupt and efficiency. Also we outlined a coordinated checkpoint simulator and validated this model using this simulator. These results all showed that this replication technique has a higher efficiency in comparison to traditional checkpoint/restart at the socket counts expected in exascale systems assuming no run time overheads. In addition, using this simulator we show that for more realistic distributions, the overheads of checkpoint/restart are more dramatic than seen with the commonly used exponential model.

2. ***r*`MPI`: a Portable and Transparent Replication Library for MPI.**

   We developed a portable, transparent replication library called *r*`MPI`. This library utilizes the MPI profiling layer to enable transparent redundant computation for MPI applications. In this work we described the design of this library, detailing the techniques that are necessary to maintain MPI semantics, especially managing message ordering and active replica consistency protocols. Additionally, we presented the run time protocol overheads for *r*`MPI`, showing that while the protocol overheads are quite high for a number of communication micro-benchmarks, there is a low overhead protocol choice for each of the tested, real-world HPC capability workloads, with that choice being dependent on the computational pattern of the application. We incorporate this overhead

in our replication model to more accurately examine the cost associated with state machine replication.

3. **Analysis of Checkpoint/Restart Viability for Exascale Scale Systems.**

   We showed the viability of state-machine replication as the *primary* exascale fault tolerance mechanism, with checkpoint/restart providing *secondary* fault tolerance when necessary. Using the aforementioned model we show that this fault-tolerance mechanism's "break-even" point, the point at which the nodes hours (for efficiency) used for this method is less than the projected sizes of next-generation exascale systems. A combination of modeling, empirical evaluation, and simulation were used to study the various costs and benefits of state machine replication over a wide range of potential system parameters. This included both the hardware and software costs of state machine replication for MPI applications, and covered different failure distributions, system mean time to interrupt ranges, and I/O speeds.

   Our results show that a state machine replication approach to exascale resilience outperforms traditional checkpoint/restart approaches for a wide range of the exascale system design space, though not the entire design space. In particular, state machine replication is a particularly viable technique for the large socket counts and limited I/O bandwidths frequently anticipated at exascale. However, replication-based approaches are less relevant for designs that have per-socket MTBFs of 50 years or more, less than 50,000 sockets, and checkpoint bandwidths of 30 terabytes per second.

   Outside of its performance benefits, using replication as the primary exascale fault tolerance methods provides a number of other advantages. First among these is that it can be used to detect and aid in the recovery from faults that corrupt system state instead of crashing the system, sometimes referred to under the banner of silent errors. Checkpoint-based approaches, on the other

hand, potentially preserve such errors. In addition, while the extra hardware nodes needed to support replication-based approaches can also be used to increase the capacity of exascale systems when it runs more but smaller (e.g. 1-10 petaflop-scale) jobs.

4. **`Libhashckpt`: a Hash-based Incremental-Checkpointing Library using GPU Accelerators**

   We developed a hybrid incremental checkpointing library that uses both OS page protection mechanisms and a hash mechanism to determine the location within a page that has changed, and therefore ensure only changed application state is saved in a checkpoint interval. To significantly decrease the overhead of the hash calculation, `libhashckpt` can utilize GPUs. To illustrate the possible benefits of this technique we created a simple model. Using this model, we show that the viability of this technique is dependent on a platforms per-process checkpoint commit rate (β_{ckpt}) and hash rate (β_{hash}) to the percent reduction in state size of the hash based approach (*compression*). Using this library, we compare the checkpoint file sizes of this hash-based method with that of a standard page-protection mechanism and a highly optimized application-specific mechanism. Using real capability HPC workloads we show that, for a certain class of applications, this hash-based method can reduce the checkpoint file size to be around 15% of that of a page-based approach. In addition, this method can create checkpoint files which are only 35% larger than that of a manually-coded, application-specific method. Finally, we use the model and results from real applications to outline the viability of this technique for next-generation exascale systems. With this model, we showed that this approach has significant performance advantages for proposed exascale architectures at GPU hash rates. At measured CPU hashing rates, this approach has a more limited viability within the exascale design space.

8.2 Future Directions

While the research described outlines most of the potential costs and benefits of these techniques, there are several avenues for future work. In terms of *state machine replication*, more work is needed quantifying the software costs of using replication to detect silent errors. While such techniques are well known in other communities, it is unclear what their cost would be for HPC applications; the quantitative results in this paper do not attempt to measure these costs and focus only on using replication to mask the pressing issue of frequent crash failures on exascale systems. In addition, more detailed studies of the scaling, benefits, and hardware costs of the various alternative methods to scaling exascale fault tolerance described in Chapter 2 are needed. While state machine replication appears viable at exascale, other approaches may still be superior; careful investigation of such approaches is needed to understand their comparative costs and benefits. Lastly, we are investigating an alternative method to enable redundant computing that has a lower resource overhead then what is presented here. Rather than having a replica specific to a particular rank, we are looking into methods that would aggregate a number of replicated ranks onto one node and spread state throughout all of these aggregate replicas in a job. This will allow an aggregate replica, on demand, to replace a certain rank and could possibly allow the application to avoid checkpointing altogether.

In terms of the *hash-based incremental checkpointing*, more work is needed in order to evaluate the merit of this technique to a broader set of large-scale applications. In addition, we propose further research investigating other hash and checksum algorithms. For this study we used a range of hashes; from cryptographically secure hash algorithms to such simpler checksum algorithms. Neither of these algorithms may be ideal for determining block changes. Ideally, we want the collision resistance of the cryptographic hashes, yet not have the other cryptographic guarantees. These ideal hashes should may have significantly lower overheads. In addition, much work

could be done in improving the performance of the GPU hash algorithms. While these CUDA implementation perform well in comparison to the CPU methods, they may not be fully utilizing the full potential of the GPU. Lastly, we need to compare this hash-based method with other checkpoint optimization techniques, such as compiler-assisted incremental checkpoint methods.

Appendices

Appendix A

*r*MPI Micro-benchmark Performance

In this chapter, we present the performance impact of our state machine replication protocols for `MPI_Allreduce()`, `MPI_Reduce()`, `MPI_Bcast()`, `MPI_Barrier()`, and `MPI_Alltoall()`. Each of these micro-benchmarks are from the OSU MPI benchmark suite(OMB) [183]. For each of the micro-benchmarks we use the *r*`MPI` library described in Chapter 4. For each of the tests we present mirror and parallel results described in Section 4.2.1. In addition, we use the replica placement options described in Section 5.1.1, showing results for forward, reverse, and shuffle options. Similar to the results presented in Chapter 4, we ran multiple tests with applications on the Cray Red Storm system at Sandia National Laboratories compiled with both *r*`MPI` and the original unmodified Cray MPI library. Red Storm is a XT-3/4 series machine consisting of over 13,000 nodes, with each compute node containing a 2.2 GHz quad-core AMD Opteron processor and 8 GB of main memory. Additionally, each node contains a Cray SeaStar [163] network interface and high-speed router. The SeaStar is connected to the Opteron via a HyperTransport link. The current generation SeaStar is capable of sustaining a peak unidirectional injection

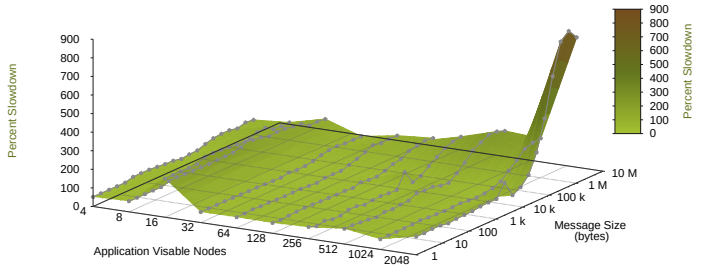
bandwidth of more than 2 GB/s and a peak unidirectional link bandwidth of more than 3 GB/s. Lastly, to ensure leader and replica are on separate physical nodes, and to avoid memory and bandwidth bottlenecks on the nodes themselves, we only used one CPU on each node.

A.1 `MPI_Allreduce()`

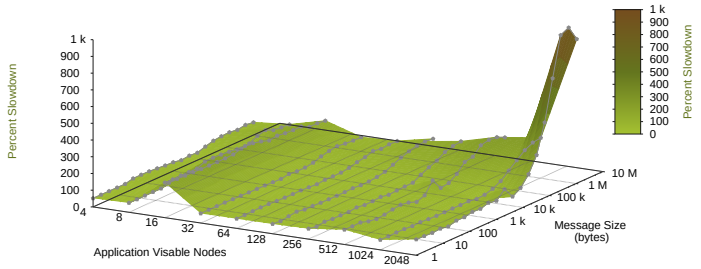
Figure A.1 and Figure A.2 show the percent slowdown of the `MPI_Allreduce()` micro-benchmark due to *r*`MPI`'s mirror and parallel consistency protocols, respectively. In both of these figures we also show the impact of the replica placement protocols. Since these tests were run on the Red Storm platform which has a 3-D torus network topology, there is typically little difference between placement options. In general we see that for this micro-benchmark, the parallel protocol has lower overheads than mirror.

A.2 `MPI_Reduce()`

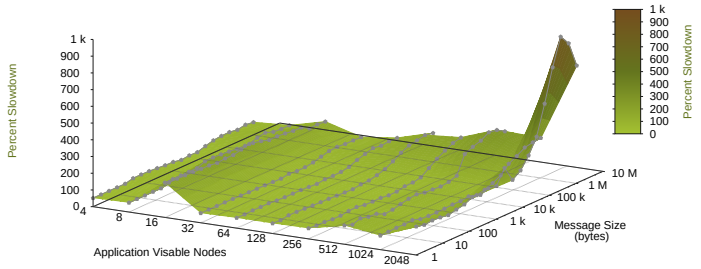
Figure A.3 and Figure A.4 show the slowdown of the `MPI_Reduce()` micro-benchmark due to *r*`MPI`'s mirror and parallel consistency protocols, respectively. In both of these figures we also show the impact of the replica placement protocols. Since these tests were run on the Red Storm platform which has a 3-D torus network topology, there is typically little difference between placement options. In general we see that for this micro-benchmark, the parallel protocol has lower overheads than mirror.



(a) Forward

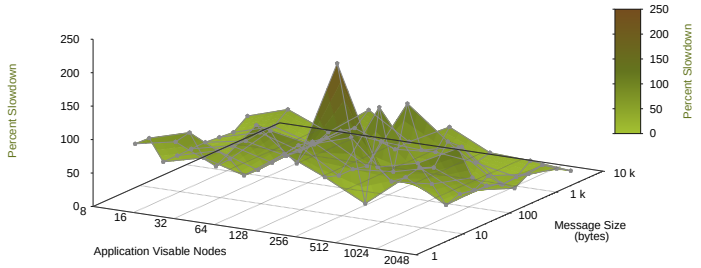


(b) Reverse

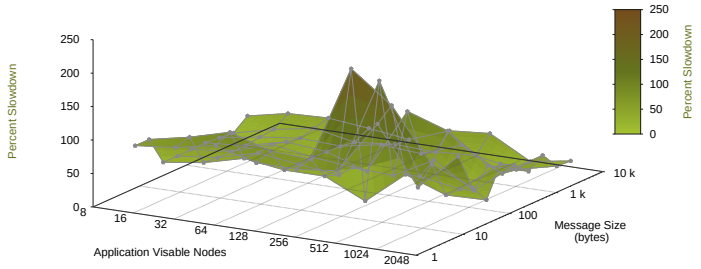


(c) Shuffle

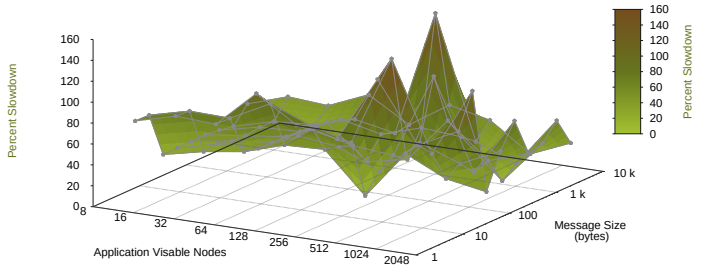
Figure A.1: `MPI_Allreduce()` micro-benchmark percent slowdown in comparison to the native MPI performance for the mirror protocol.



(a) Forward

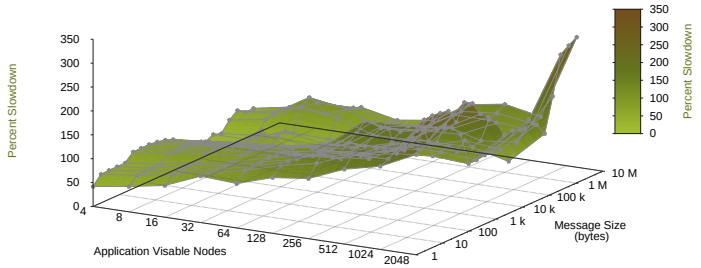


(b) Reverse

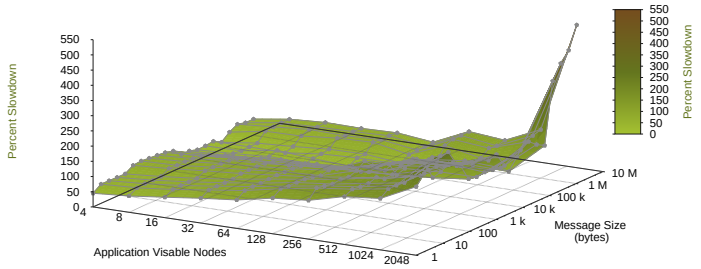


(c) Shuffle

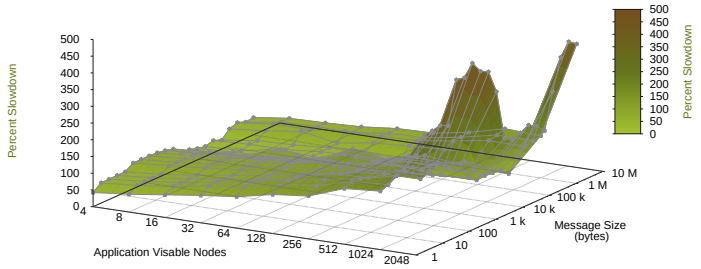
Figure A.2: `MPI_Allreduce()` micro-benchmark percent slowdown in comparison to the native MPI performance for the parallel protocol.



(a) Forward

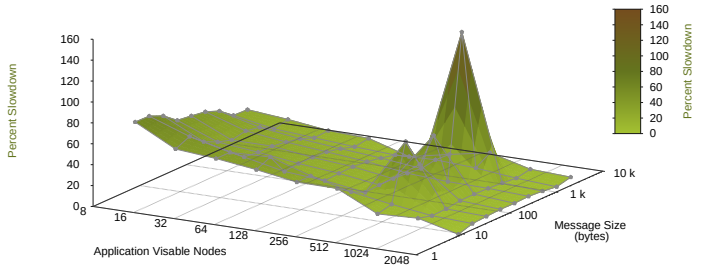


(b) Reverse

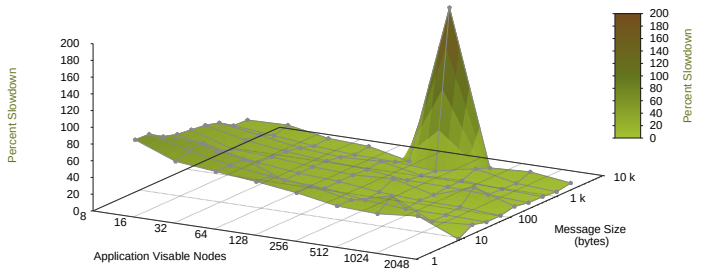


(c) Shuffle

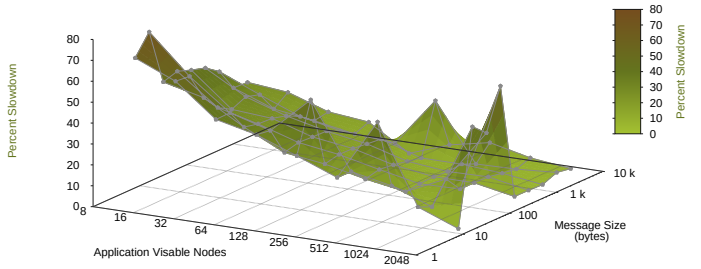
Figure A.3: `MPI_Reduce()` micro-benchmark percent slowdown in comparison to the native MPI performance for the mirror protocol.



(a) Forward



(b) Reverse



(c) Shuffle

Figure A.4: `MPI_Reduce()` micro-benchmark percent slowdown in comparison to the native MPI performance for the parallel protocol.

A.3 MPI_Bcast()

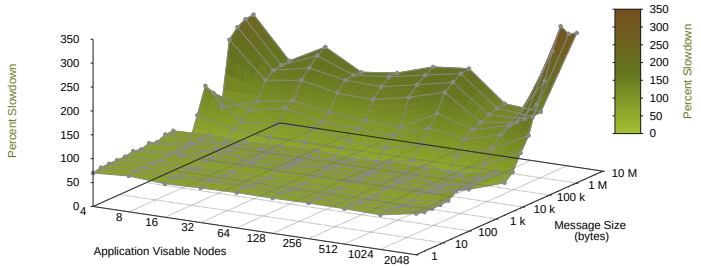
Figure A.5 and Figure A.6 show the slowdown of the MPI_Bcast() micro-benchmark due to *r*MPI's mirror and parallel consistency protocols, respectively. In both of these figures we also show the impact of the replica placement protocols. Since these tests were run on the Red Storm platform which has a 3-D torus network topology, there is typically little difference between placement options. In general we see that for this micro-benchmark, the parallel protocol has lower overheads than mirror.

A.4 MPI_Alltoall()

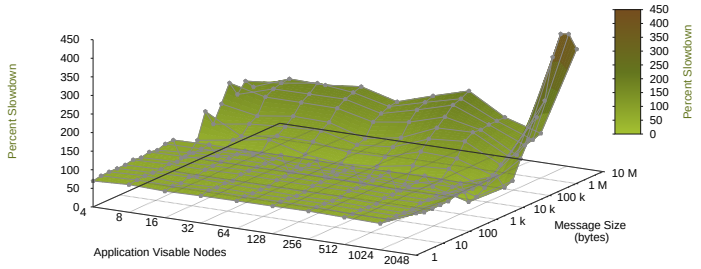
Figure A.7 shows the slowdown of the MPI_Alltoall() micro-benchmark due to *r*MPI's parallel consistency protocols. *r*MPI mirror results are not show due to limited system time on Red Storm. In this figure we show the impact of the replica placement protocols. Since these tests were run on the Red Storm platform which has a 3-D torus network topology, there is typically little difference between placement options.

A.5 MPI_Barrier()

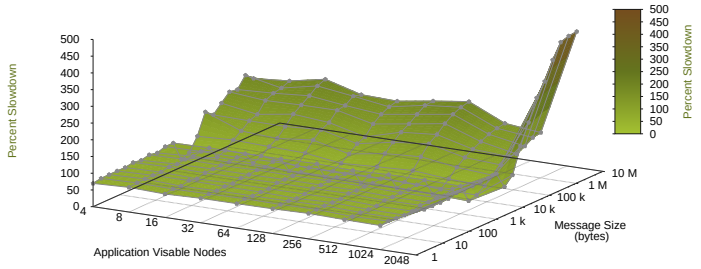
Figure A.8 shows the slowdown of the MPI_Barrier() micro-benchmark due to *r*MPI's mirror and parallel consistency protocols. In this figure we also show the impact of the replica placement protocols. Since these tests were run on the Red Storm platform which has a 3-D torus network topology, there is typically little difference between placement options. In general we see that for this micro-benchmark, the parallel protocol has lower overheads than mirror.



(a) Forward

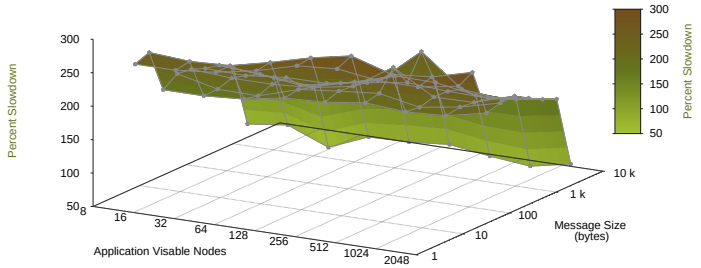


(b) Reverse

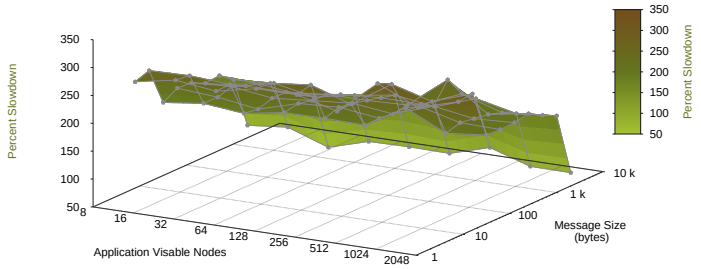


(c) Shuffle

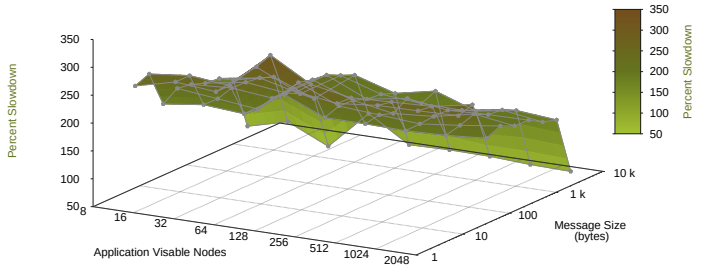
Figure A.5: `MPI_Bcast()` micro-benchmark percent slowdown in comparison to the native MPI performance for the mirror protocol.



(a) Forward

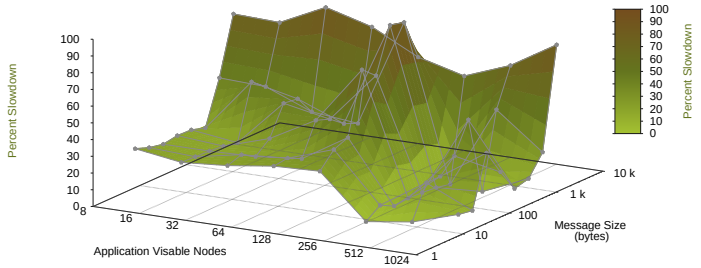


(b) Reverse

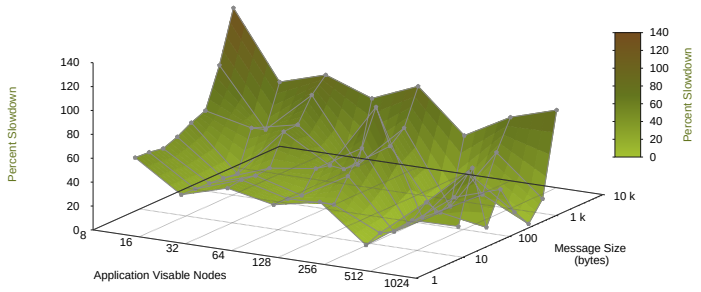


(c) Shuffle

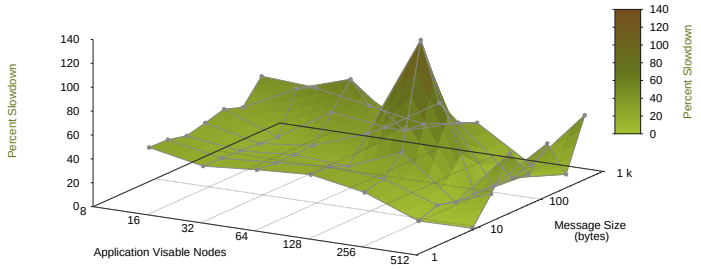
Figure A.6: `MPI_Bcast()` micro-benchmark percent slowdown in comparison to the native MPI performance for the parallel protocol.



(a) Forward

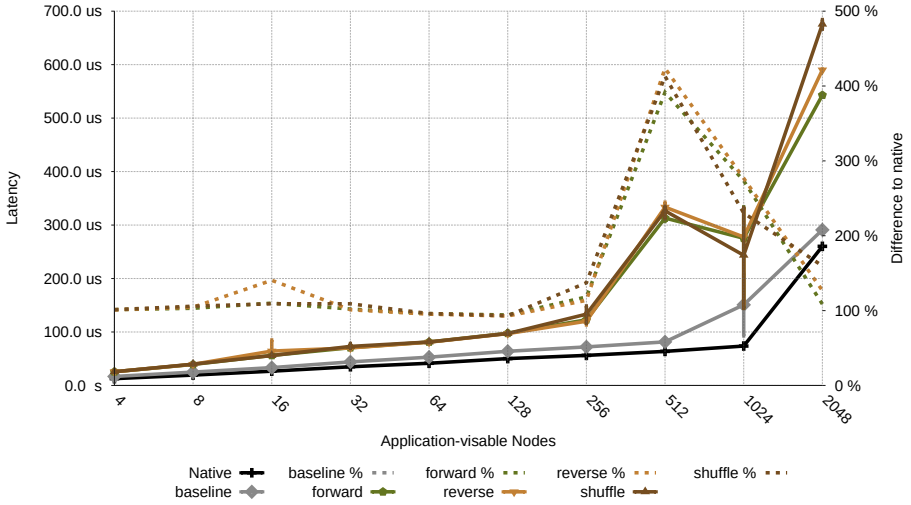


(b) Reverse

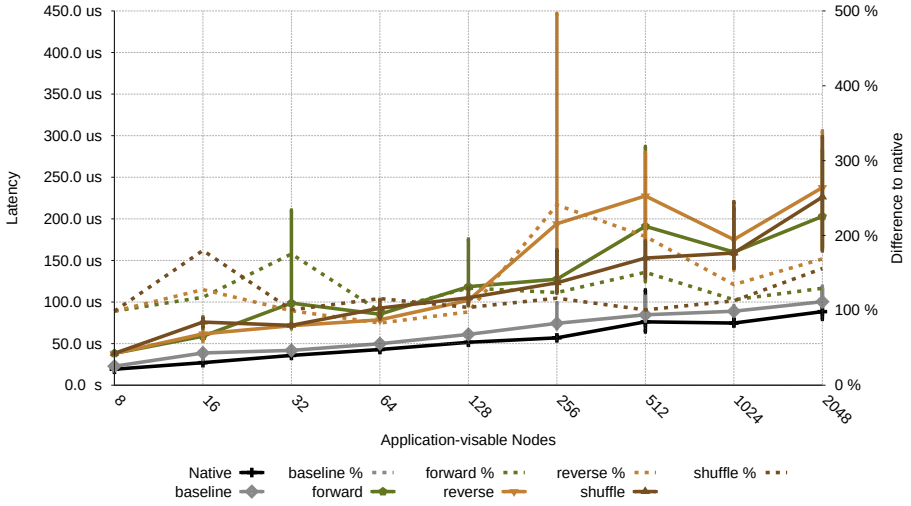


(c) Shuffle

Figure A.7: `MPI_Alltoall()` micro-benchmark percent slowdown in comparison to the native MPI performance for the parallel protocol.



(a) Mirror



(b) Parallel

Figure A.8: `MPI_Barrier()` micro-benchmark percent slowdown in comparison to the native MPI performance for the parallel and mirror protocols.

References

[1] (2010, November) Top 500 web site. [Online]. Available: http://www.top500.org

[2] (2011, June) Blue waters. [Online]. Available: http://www.ncsa.illinois.edu/BlueWaters/system.html

[3] (2011, June) Cielo. [Online]. Available: http://www.lanl.gov/science/NSS/issue2_2010/story4b.shtml

[4] (2011, June) Sequoia. [Online]. Available: http://en.wikipedia.org/wiki/IBM_Sequoia

[5] S. Amarasinghe, D. Campbell, W. Carlson, A. Chien, W. Dally, E. Elnohazy, M. Hall, R. Harrison, W. Harrod, K. Hill, J. Hiller, S. Karp, C. Koelbel, D. Koester, P. Kogge, J. Levesque, D. Reed, V. Sarkar, R. Schreiber, M. Richards, A. Scarpelli, J. Shalf, A. Snavely, and T. Sterling, "Exascale software study: Software challenges in extreme scale systems," DARPA IPTO, Air Force Research Labs, Tech. Rep., September 2009.

[6] R. A. Oldfield, S. Arunagiri, P. J. Teller, S. Seelam, M. R. Varela, R. Riesen, and P. C. Roth, "Modeling the impact of checkpoints on next-generation systems," in *24th IEEE Conference on Mass Storage Systems and Technologies*, Sep. 2007, pp. 30–46.

[7] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers," *Journal of Physics: Conference Series*, vol. 78, no. 1, p. 012022, 2007.

[8] G. Gibson, B. Schroeder, and J. Digney, "Failure tolerance in petascale computers," *CTWatch Quarterly*, vol. 3, 2007.

[9] C.-h. Hsu and W.-c. Feng, "A power-aware run-time system for high-performance computing," in *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2005, p. 1.

[10] J. Stearley and R. Ballance, "A preliminary report on red storm RAS performance," in *Proceedings of the 2006 Cray Users Group Meeting*, 2006.

[11] T. J. Hacker, F. Romero, and C. D. Carothers, "An analysis of clustered failures on large supercomputing systems," *J. Parallel Distrib. Comput.*, vol. 69, pp. 652–665, July 2009.

[12] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, P. Kogge, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick, "Exascale computing study: Technology challenges in achieving exascale systems," http://www.science.energy.gov/ascr/Research/CS/DARPAexascale-hardware(2008).pdf, Sep. 2008.

[13] (2010, July) libgcrypt web page. [Online]. Available: http://directory.fsf.org/project/libgcrypt/

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

[15] (2011, June) Jaguar. [Online]. Available: http://www.nccs.gov/jaguar/

[16] (2011, June) ASCI red. [Online]. Available: http://www.sandia.gov/ASCI/Red/RedFacts.htm

[17] A. Oliner and J. Stearley, "What supercomputers say: A study of five system logs," in *DSN '07: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 575–584.

[18] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN2006)*, Jun. 2006. [Online]. Available: http://www.pdl.cmu.edu/PDL-FTP/stray/dsn06_abs.html

[19] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM errors in the wild: a large-scale field study," in *Proceedings of the eleventh international*

joint conference on Measurement and modeling of computer systems, ser. SIGMETRICS '09. New York, NY, USA: ACM, 2009, pp. 193–204. [Online]. Available: http://doi.acm.org/10.1145/1555349.1555372

[20] D. McEvoy, "The architecture of tandem's nonstop system," in *ACM '81: Proceedings of the ACM '81 conference*. New York, NY, USA: ACM, 1981, p. 245.

[21] J. F. Bartlett, "A nonstop kernel," in *SOSP '81: Proceedings of the eighth ACM symposium on Operating systems principles*, 1981, pp. 22–29.

[22] F. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

[23] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 4, pp. 398–461, Nov. 2002.

[24] K. Uhlemann, C. Engelmann, and S. L. Scott, "JOSHUA: Symmetric active/active replication for highly available HPC job and resource management," in *Proceedings of the 8th IEEE International Conference on Cluster Computing (Cluster) 2006*. Barcelona, Spain: IEEE Computer Society, September 2009.

[25] H.-C. Nam, J. Kim, S. Hong, and S. Lee, "Probabilistic checkpointing," in *Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on*, jun 1997, pp. 48 –57.

[26] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira, "Adaptive incremental checkpointing for massively parallel systems," in *Proceedings of the 2004 International Conference on Supercomputing*, St. Malo, France, 2004.

[27] E. N. Elnozahy, "How safe is probabilistic checkpointing?" in *Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, ser. FTCS '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 358–. [Online]. Available: http://portal.acm.org/citation.cfm?id=795671.796882

[28] H. chang Nam, J. Kim, S. J. Hong, and S. Lee, "A secure checkpointing system," in *Dependable Computing, 2001. Proceedings. 2001 Pacific Rim International Symposium on*, 2001, pp. 49 –56.

[29] L. Holst, "The general birthday problem," in *Random Graphs 93: Proceedings of the sixth international seminar on Random graphs and probabilistic methods*

in combinatorics and computer science. New York, NY, USA: John Wiley & Sons, Inc., 1995, pp. 201–208.

[30] F. H. Mathis, "A generalized birthday problem," *SIAM Review*, vol. 33, no. 2, pp. 265–270, 1991.

[31] D. Wagner, "A generalized birthday problem," in *Proceedings of the 22nd Annual International Cryptology Conference on Advances in Cryptology*, ser. CRYPTO '02. London, UK: Springer-Verlag, 2002, pp. 288–303. [Online]. Available: http://portal.acm.org/citation.cfm?id=646767.704294

[32] N. Henze, "A poisson limit law for a generalized birthday problem," *Statistics and Probability Letters*, vol. 39, no. 4, pp. 333–336, 1998.

[33] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI - The Complete Reference, Volume 1: The MPI Core*, 2nd ed. Cambridge, MA, USA: MIT Press, 1998.

[34] F. C. Gärtner, "Fundamentals of distributed computing in asynchronous environments," *ACM Comput. Surv.*, vol. 31, no. 1, pp. 1–26, 1999.

[35] P. Jalote, *Fault Tolerance in Distributed Systems*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1994.

[36] F. Cristian, "Understanding fault-tolerant distributed systems," *Commun. ACM*, vol. 34, pp. 56–78, February 1991. [Online]. Available: http://doi.acm.org/10.1145/102792.102801

[37] F. B. Schneider, *What good are models and what models are good?* New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1993, pp. 17–26. [Online]. Available: http://portal.acm.org/citation.cfm?id=302430.302432

[38] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, pp. 382–401, July 1982. [Online]. Available: http://doi.acm.org/10.1145/357172.357176

[39] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden, "Tolerating byzantine faults in transaction processing systems using commit barrier scheduling," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 59–72. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294268

[40] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative byzantine fault tolerance," in *Proceedings of twenty-first*

ACM SIGOPS symposium on Operating systems principles, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 45–58. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294267

[41] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, 2002.

[42] J.-M. Hélary, R. H. B. Netzer, and M. Raynal, "Consistency issues in distributed checkpoints," *IEEE Trans. Softw. Eng.*, vol. 25, pp. 274–281, March 1999. [Online]. Available: http://portal.acm.org/citation.cfm?id=630823.631237

[43] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The design and implementation of zap: a system for migrating computing environments," *SIGOPS Oper. Syst. Rev.*, vol. 36, pp. 361–376, December 2002. [Online]. Available: http://doi.acm.org/10.1145/844128.844162

[44] V. C. Zandy and B. P. Miller, "Reliable network connections," in *Proceedings of the 8th annual international conference on Mobile computing and networking*, ser. MobiCom '02. New York, NY, USA: ACM, 2002, pp. 95–106. [Online]. Available: http://doi.acm.org/10.1145/570645.570657

[45] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, "C^3: A system for automating application-level checkpointing of MPI programs," in *LCPC*, ser. Lecture Notes in Computer Science, L. Rauchwerger, Ed., vol. 2958. Springer, 2003, pp. 357–373.

[46] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, pp. 63–75, February 1985. [Online]. Available: http://doi.acm.org/10.1145/214451.214456

[47] B. Randell, "System structure for software fault tolerance," in *Proceedings of the international conference on Reliable software*. New York, NY, USA: ACM, 1975, pp. 437–449. [Online]. Available: http://doi.acm.org/10.1145/800027.808467

[48] Y. Tamir and C. H. Squin, "Error recovery in multicomputers using global checkpoints," in *In 1984 International Conference on Parallel Processing*, 1984, pp. 32–41.

[49] J. Hursey, T. I. Mattox, and A. Lumsdaine, "Interconnect agnostic checkpoint/restart in open MPI," in *HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing*, 2009, pp. 49–58.

[50] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (blcr) for linux clusters," *Journal of Physics: Conference Series*, vol. 46, no. 1, p. 494, 2006. [Online]. Available: http://stacks.iop.org/1742-6596/46/i=1/a=067

[51] T. H. Lai and T. H. Yang, "On distributed snapshots," *Inf. Process. Lett.*, vol. 25, pp. 153–158, May 1987. [Online]. Available: http://dx.doi.org/10.1016/0020-0190(87)90125-6

[52] C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello, "Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant mpi ," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006, p. 127.

[53] F. Cristian and F. Jahanian, "A timestamp-based checkpointing protocol for long-lived distributed computations," in *SRDS*, 1991, pp. 12–20.

[54] Z. Tong, R. Y. Kain, and W. T. Tsai, "Rollback recovery in distributed systems using loosely synchronized clocks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 3, pp. 246–251, March 1992. [Online]. Available: http://dx.doi.org/10.1109/71.127264

[55] L. Lamport, "Using time instead of timeout for fault-tolerant distributed systems." *ACM Trans. Program. Lang. Syst.*, vol. 6, pp. 254–280, April 1984. [Online]. Available: http://doi.acm.org/10.1145/2993.2994

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

[57] Y. Liu, R. Nassar, C. Leangsuksun, N. Naksinehaboon, M. Paun, and S. L. Scott, "An optimal checkpoint/restart model for a large scale high performance computing system," in *IPDPS*. IEEE, 2008, pp. 1–9.

[58] M.-S. Bouguerra, T. Gautier, D. Trystram, and J.-M. Vincent, "A flexible checkpoint/restart model in distributed systems," in *Proceedings of the 8th international conference on Parallel processing and applied mathematics: Part I*, ser. PPAM'09. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 206–215. [Online]. Available: http://portal.acm.org/citation.cfm?id=1882792.1882818

[59] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Commun. ACM*, vol. 17, pp. 530–531, September 1974. [Online]. Available: http://doi.acm.org/10.1145/361147.361115

[60] N. Naksinehaboon, Y. Liu, C. B. Leangsuksun, R. Nassar, M. Paun, and S. L. Scott, "Reliability-aware approach: An incremental checkpoint/restart model in hpc environments," in *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 783–788. [Online]. Available: http://portal.acm.org/citation.cfm?id=1371605.1372487

[61] A. Bouteiller, F. Cappello, T. Herault, G. Krawezik, P. Lemarinier, and F. Magniette, "MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging," in *Proceedings of the ACM/IEEE International Conference on High Performance Computing and Networking*, November 2003.

[62] J. Hursey, J. Squyres, T. Mattox, and A. Lumsdaine, "The design and implementation of checkpoint/restart process fault tolerance for open MPI," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, March 2007.

[63] G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill, "Recent advances in checkpoint/recovery systems," Apr. 2006.

[64] T.-C. Chiueh and P. Deng, "Evaluation of checkpoint mechanisms for massively parallel machines." in *Annual Symposium on Fault Tolerant Computing*. Sendai, Japan: IEEE Computer Society Press, June 1996, pp. 370–379.

[65] E. N. Elnozahy and J. S. Plank, "Checkpointing for peta-scale systems: a look into the future of practical rollback-recovery," *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, no. 2, pp. 97–108, Apr. 2004.

[66] J. F. Ruscio, M. A. Heffner, and S. Varadarajan, "Dejavu: Transparent user-level checkpointing, migration, and recovery for distributed systems," in *International Parallel and Distributed Processing Symposium/International Parallel Processing Symposium*, 2007, pp. 1–10.

[67] G. Bronevetsky, D. J. Marques, K. K. Pingali, R. Rugina, and S. A. McKee, "Compiler-enhanced incremental checkpointing for openmp applications," in *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2008, pp. 275–276.

[68] R. Gioiosa, J. C. Sancho, S. Jiang, and F. Petrini, "Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, ser. SC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 9–. [Online]. Available: http://dx.doi.org/10.1109/SC.2005.76

[69] J. C. Sancho, F. Petrini, G. Johnson, J. Fernandez, and E. Frachtenberg, "On the feasibility of incremental checkpointing for scientific computing," *Parallel and Distributed Processing Symposium, International*, vol. 1, p. 58b, 2004.

[70] Y. Chen, J. S. Plank, and K. Li, "Clip: a checkpointing tool for message-passing parallel programs," in *Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, ser. Supercomputing '97. New York, NY, USA: ACM, 1997, pp. 1–11. [Online]. Available: http://doi.acm.org/10.1145/509593.509626

[71] K. Li, J. F. Naughton, and J. S. Plank, "Low-latency, concurrent checkpointing for parallel programs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 5, pp. 874–879, August 1994. [Online]. Available: http://dx.doi.org/10.1109/71.298215

[72] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: transparent checkpointing under unix," in *Proceedings of the USENIX 1995 Technical Conference Proceedings*, ser. TCON'95. Berkeley, CA, USA: USENIX Association, 1995, pp. 18–18. [Online]. Available: http://portal.acm.org/citation.cfm?id=1267411.1267429

[73] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel, "The performance of consistent checkpointing." in *11th Symposium on Reliable Distributed Systems*. Houston, TX, USA: IEEE Computer Society Press, October 1992, pp. 39–47.

[74] J. S. Plank, Y. Chen, K. Li, M. Beck, and G. Kingsley, "Memory exclusion: optimizing the performance of checkpointing systems," *Softw. Pract. Exper.*, vol. 29, pp. 125–142, February 1999. [Online]. Available: http://portal.acm.org/citation.cfm?id=309087.309095

[75] S. I. Feldman and C. B. Brown, "Igor: a system for program debugging via reversible execution," in *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*, ser. PADD '88. New York, NY, USA: ACM, 1988, pp. 112–123. [Online]. Available: http://doi.acm.org/10.1145/68210.69226

[76] J. Leon, A. L. Fisher, and P. Steenkiste, "Fail-safe PVM: A portable package for distributed programming with transparent recovery," Pittsburgh, PA, USA, Tech. Rep., 1993.

[77] K. Li, J. F. Naughton, and J. S. Plank, "Real-ti concurrent checkpoint for parallel programs," in *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, ser. PPOPP '90. New York, NY, USA: ACM, 1990, pp. 79–88. [Online]. Available: http://doi.acm.org/10.1145/99163.99173

[78] G. Stellner, "CoCheck: Checkpointing and process migration for MPI," in *Proceedings of the 10th International Parallel Processing Symposium*, ser. IPPS '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 526–531. [Online]. Available: http://portal.acm.org/citation.cfm?id=645606.660853

[79] V. C. Zandy, B. P. Miller, and M. Livny, "Process hijacking," in *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, ser. HPDC '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 32–. [Online]. Available: http://portal.acm.org/citation.cfm?id=822084.823234

[80] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. 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*, ser. SC'10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.18

[81] B. K. Bhargava and S.-R. Lian, "Independent checkpointing and concurrent rollback for recovery in distributed systems - an optimistic approach," in *SRDS*, 1988, pp. 3–12.

[82] Y.-M. Wang, "Reducing message logging overhead for log-based recovery." in *ISCAS'93*, 1993, pp. 1925–1928.

[83] B. Randell, "System structure for software fault tolerance," *SIGPLAN Not.*, vol. 10, pp. 437–449, April 1975. [Online]. Available: http://doi.acm.org/10.1145/390016.808467

[84] R. H. B. Netzer and J. Xu, "Necessary and sufficient conditions for consistent global snapshots," *IEEE Trans. Parallel Distrib. Syst.*, vol. 6, pp. 165–169, February 1995. [Online]. Available: http://dx.doi.org/10.1109/71.342127

[85] D. L. Russell, "State restoration in systems of communicating processes," *IEEE Trans. Softw. Eng.*, vol. 6, pp. 183–194, March 1980. [Online]. Available: http://portal.acm.org/citation.cfm?id=1313319.1313517

[86] Y.-M. Wang, "Consistent global checkpoints that contain a given set of local checkpoints," *IEEE Trans. Comput.*, vol. 46, pp. 456–468, April 1997. [Online]. Available: http://portal.acm.org/citation.cfm?id=254340.254371

[87] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[88] D. Briatico, A. Ciuffoletti, and L. Simoncini, "A distributed domino-effect free recovery algorithm," in *Symposium on Reliability in Distributed Software and Database Systems*, 1984, pp. 207–215.

[89] J.-M. Helary, A. Mostefaoui, and M. Raynal, "Preventing useless checkpoints in distributed computations," in *Proceedings of the 16th Symposium on Reliable Distributed Systems*, ser. SRDS '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 183–. [Online]. Available: http://portal.acm.org/citation.cfm?id=829522.830921

[90] J. Ahn, "2-step algorithm for enhancing effectiveness of sender-based message logging," in *SpringSim '07: Proceedings of the 2007 spring simulation multiconference*, 2007, pp. 429–434. [Online]. Available: http://portal.acm.org/citation.cfm?id=1404748

[91] Q. Jiang and D. Manivannan, "An optimistic checkpointing and selective approach for consistent global checkpoint collection in distributed systems," Mar. 2007.

[92] D. B. Johnson and W. Zwaenepoel, "Recovery in distributed systems using asynchronous and checkpointing," in *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, 1988, pp. 171–181.

[93] A. Bouteiller, G. Bosilca, and J. Dongarra, "Redesigning the message logging model for high performance," *Concurr. Comput. : Pract. Exper.*, vol. 22, pp. 2196–2211, November 2010. [Online]. Available: http://dx.doi.org/10.1002/cpe.v22:16

[94] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov, "MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, ser. Supercomputing '02. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 1–18. [Online]. Available: http://portal.acm.org/citation.cfm?id=762761.762815

[95] R. Strom and S. Yemini, "Optimistic recovery in distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, pp. 204–226, August 1985. [Online]. Available: http://doi.acm.org/10.1145/3959.3962

[96] M. L. Powell and D. L. Presotto, "Publishing: a reliable broadcast communication mechanism," *SIGOPS Oper. Syst. Rev.*, vol. 17, pp. 100–109, October 1983. [Online]. Available: http://doi.acm.org/10.1145/773379.806618

[97] A. Borg, J. Baumbach, and S. Glazer, "A message system supporting fault tolerance," in *Proceedings of the ninth ACM symposium on Operating systems principles*, ser. SOSP '83. New York, NY, USA: ACM, 1983, pp. 90–99. [Online]. Available: http://doi.acm.org/10.1145/800217.806617

[98] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. 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.

[99] J. S. Plank, Y. B. Kim, and J. J. Dongarra, "Algorithm-based diskless checkpointing for fault tolerant matrix operations." in *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers*. Pasadena, CA, USA: Los Alamitos, CA, USA : IEEE Comput. Soc. Press, 1995, June 1995, pp. 351–360.

[100] J. S. Plank, Y. Kim, and J. J. Dongarra, "Fault-tolerant matrix operations for networks of workstations using diskless checkpointing," *J. Parallel Distrib. Comput.*, vol. 43, pp. 125–138, June 1997. [Online]. Available: http://dx.doi.org/10.1006/jpdc.1997.1336

[101] L. M. Silva and J. G. Silva, "An experimental study about diskless checkpointing." in *24th EUROMICRO Conference*. Vasteras, Sweden: IEEE Computer Society Press, August 1998, pp. 395 – 402.

[102] C. Engelmann and A. Geist, "A diskless checkpointing algorithm for super-scale architectures applied to the fast fourier transform," in *Proceedings of the 1st International Workshop on Challenges of Large Applications in Distributed Environments*, ser. CLADE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 47–. [Online]. Available: http://portal.acm.org/citation.cfm?id=792760.793177

[103] X. Dong, N. Muralimanohar, N. Jouppi, R. Kaufmann, and Y. Xie, "Leveraging 3d pcram technologies to reduce checkpoint overhead for future exascale

systems," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York, NY, USA: ACM, 2009, pp. 1–12.

[104] M. Prvulovic, J. Torrellas, and Z. Zhang, "Revive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors." in *ISCA'02*, 2002, pp. 111–122.

[105] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood, "Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery," in *Proceedings of the 29th annual international symposium on Computer architecture*, ser. ISCA '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 123–134. [Online]. Available: http://portal.acm.org/citation.cfm?id=545215.545229

[106] F. C. Gärtner, "Fundamentals of fault-tolerant distributed computing in asynchronous environments." *ACM Computing Surveys*, vol. 31, no. 1, pp. 1–26, March 1999.

[107] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta, "Hive: fault containment for shared-memory multiprocessors," in *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 1995, pp. 12–25.

[108] A. Borg, J. Baumbach, and S. Glazer, "A message system supporting fault tolerance," *SIGOPS Oper. Syst. Rev.*, vol. 17, pp. 90–99, October 1983. [Online]. Available: http://doi.acm.org/10.1145/773379.806617

[109] M. L. Powell and D. L. Presotto, "Publishing: a reliable broadcast communication mechanism," in *Proceedings of the ninth ACM symposium on Operating systems principles*, ser. SOSP '83. New York, NY, USA: ACM, 1983, pp. 100–109. [Online]. Available: http://doi.acm.org/10.1145/800217.806618

[110] F. P. S. Frolund, "Pronto: A fast failover protocol for off-the-shelf commercial databases," in *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems*. Washington, DC, USA: IEEE Computer Society, 2000, pp. 176–. [Online]. Available: http://portal.acm.org/citation.cfm?id=829525.831076

[111] P. A. Alsberg and J. D. Day, "A principle for resilient sharing of distributed resources," in *Proceedings of the 2nd international conference on Software engineering*, ser. ICSE '76. Los Alamitos, CA, USA: IEEE Computer Society Press, 1976, pp. 562–570. [Online]. Available: http://portal.acm.org/citation.cfm?id=800253.807732

[112] M. Chérèque, D. Powell, P. Reynier, J.-L. Richier, and J. Voiron, "Active replication in delta-4," in *FTCS*, 1992, pp. 28–37.

[113] Y. Zhou, P. M. Chen, and K. Li, "Fast cluster failover using virtual memory-mapped communication," in *Proceedings of the 13th international conference on Supercomputing*, ser. ICS '99. New York, NY, USA: ACM, 1999, pp. 373–382. [Online]. Available: http://doi.acm.org/10.1145/305138.305215

[114] F. Bouabache, T. Herault, G. Fedak, and F. Cappello, "Hierarchical replication techniques to ensure checkpoint storage reliability in grid environment," in *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 475–483. [Online]. Available: http://portal.acm.org/citation.cfm?id=1371605.1372508

[115] S. Genaud and C. Rattanapoka, "Evaluation of replication and fault detection in P2P-MPI," in *IPDPS*. IEEE, 2009, pp. 1–8.

[116] S. Genaud, E. Jeannot, and C. Rattanapoka, "Fault-management in P2P-MPI," *Int. J. Parallel Program.*, vol. 37, no. 5, pp. 433–461, 2009.

[117] H. Packard, "HP NonStop computing," http://h20338.www2.hp.com/NonStopComputing.

[118] M. Pillai and M. Lauria, "A high performance redundancy scheme for cluster file systems," *Int. J. High Perform. Comput. Netw.*, vol. 2, pp. 90–98, February 2004. [Online]. Available: \url{http://portal.acm.org/citation.cfm?id=1359710.1359713}

[119] ——, "Csar: Cluster storage with adaptive redundancy," in *ICPP*. IEEE Computer Society, 2003, pp. 223–230.

[120] S.-K. Hung and Y. Hsu, "Modularized redundant parallel virtual file system," in *Advances in Computer Systems Architecture*, ser. Lecture Notes in Computer Science, T. Srikanthan, J. Xue, and C.-H. Chang, Eds. Springer Berlin / Heidelberg, 2005, vol. 3740, pp. 186–199. [Online]. Available: http://dx.doi.org/10.1007/11572961_16

[121] ——, "DPCT: Distributed parity cache table for redundant parallel file system," in *High Performance Computing and Communications*, ser. Lecture Notes in Computer Science, M. Gerndt and D. Kranzlmller, Eds. Springer Berlin / Heidelberg, 2006, vol. 4208, pp. 320–329. [Online]. Available: http://dx.doi.org/10.1007/11847366_33

[122] K. Limaye, B. Leangsuksun, Z. Greenwood, S. Scott, C. Engelmann, R. Libby, and K. Chanchio, "Job-site level fault tolerance for cluster and grid environments," in *Cluster Computing, 2005. IEEE International*, sept. 2005, pp. 1 –9.

[123] C. Engelmann, S. L. Scott, C. Leangsuksun, and X. He, "Symmetric active/active high availability for high-performance computing system services: Accomplishments and limitations," in *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 813–818. [Online]. Available: http://portal.acm.org/citation.cfm?id=1371605.1372497

[124] ——, "Transparent symmetric active/active replication for service-level high availability," in *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, ser. CCGRID '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 755–760. [Online]. Available: http://dx.doi.org/10.1109/CCGRID.2007.116

[125] Z. Zheng and Z. Lan, "Reliability-aware scalability models for high performance computing,," in *Cluster'09: Proceedings of the IEEE conference on Cluster Computing*, 2009.

[126] C. Engelmann, H. H. Ong, and S. L. Scott, "The case for modular redundancy in large-scale high performance computing systems," in *Proceedings of the 8th IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN) 2009*. Innsbruck, Austria: ACTA Press, Calgary, AB, Canada, Feb. 16-18, 2009, pp. 189–194. [Online]. Available: http://www.csm.ornl.gov/~engelman/publications/engelmann09case.pdf

[127] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Comput. Surv.*, vol. 36, pp. 372–421, December 2004. [Online]. Available: http://doi.acm.org/10.1145/1041680.1041682

[128] R. Baldoni, S. Cimmino, and C. Marchetti, "Total Order Communications: a Practical Analysis," *5th European Dependable Computing Conference (EDCC), LNCS, 38-54*, 4 2005. [Online]. Available: http://www.dis.uniroma1.it/~midlab

[129] M. F. Kaashoek, A. S. Tanenbaum, and K. Verstoep, "Group communication in amoeba and its applications," *Distributed Systems Engineering*, vol. 1, no. 1, p. 48, 1993. [Online]. Available: http://stacks.iop.org/0967-1846/1/i=1/a=006

[130] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, pp. 133–169, May 1998. [Online]. Available: http://doi.acm.org/10.1145/279227.279229

[131] Z. Bar-Joseph, I. Keidar, and N. A. Lynch, "Early-delivery dynamic atomic broadcast," in *Proceedings of the 16th International Conference on Distributed Computing*, ser. DISC '02. London, UK, UK: Springer-Verlag, 2002, pp. 1–16. [Online]. Available: http://portal.acm.org/citation.cfm?id=645959.676132

[132] D. Dolev, S. Kramer, and D. Malki, "Early delivery totally ordered multicast in asynchronous environments," in *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*, jun 1993, pp. 544 –553.

[133] K. Bettina, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann, "Using optimistic atomic broadcast in transaction processing systems," *IEEE Trans. on Knowl. and Data Eng.*, vol. 15, pp. 1018–1032, July 2003. [Online]. Available: http://portal.acm.org/citation.cfm?id=1435677.858981

[134] P. Vicente and L. Rodrigues, "An indulgent uniform total order algorithm with optimistic delivery," in *Reliable Distributed Systems, 2002. Proceedings. 21st IEEE Symposium on*, 2002, pp. 92 – 101.

[135] L. Moser, Y. Amir, P. Melliar-Smith, and D. Agarwal, "Extended virtual synchrony," in *Distributed Computing Systems, 1994., Proceedings of the 14th International Conference on*, jun 1994, pp. 56 –65.

[136] D. A. Agarwal, L. E. Moser, P. M. Melliar-Smith, and R. K. Budhia, "A reliable ordered delivery protocol for interconnected local area networks," in *Proceedings of the 1995 International Conference on Network Protocols*, ser. ICNP '95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 365–. [Online]. Available: http://portal.acm.org/citation.cfm?id=850933.852338

[137] K. P. Birman and R. V. Renesse, *Reliable Distributed Computing with the ISIS Toolkit*, R. V. Renesse, Ed. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994.

[138] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella, "The totem single-ring ordering and membership protocol," *ACM Trans. Comput. Syst.*, vol. 13, pp. 311–342, November 1995. [Online]. Available: http://doi.acm.org/10.1145/210223.210224

[139] D. Dolev and D. Malki, "The transis approach to high availability cluster communication," *Commun. ACM*, vol. 39, pp. 64–70, April 1996. [Online]. Available: http://doi.acm.org/10.1145/227210.227227

[140] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Commun. ACM*, vol. 17, pp. 643–644, November 1974. [Online]. Available: http://doi.acm.org/10.1145/361179.361202

[141] A. Geist and C. Engelmann, "Development of naturally fault tolerant algorithms for computing on 100,000 processors," 2002.

[142] G. E. Fagg, A. Bukovsky, and J. Dongarra, "Fault tolerant MPI for the HARNESS meta-computing system," in *Proceedings of the International Conference on Computational Sciences-Part I*, ser. ICCS '01. London, UK, UK: Springer-Verlag, 2001, pp. 355–366. [Online]. Available: http://portal.acm.org/citation.cfm?id=645455.654207

[143] G. E. Fagg, A. Bukovsky, and J. J. Dongarra, "HARNESS and fault tolerant MPI," *Parallel Comput.*, vol. 27, pp. 1479–1495, October 2001. [Online]. Available: http://portal.acm.org/citation.cfm?id=543586.543591

[144] J. Dongarra, G. E. Fagg, A. Geist, J. A. Kohl, P. M. Papadopoulos, S. L. Scott, V. S. Sunderam, and M. Magliardi, "Harness: Heterogeneous adaptable reconfigurable networked systems," in *HPDC*, 1998, pp. 358–359.

[145] M. Beck, J. J. Dongarra, G. E. Fagg, G. Geist, P. Gray, J. Kohl, M. Migliardi, K. Moore, T. Moore, P. Papadopoulous, S. L. Scott, and V. Sunderam, "HARNESS: A next generation distributed virtual machine," *Future Generation Computer Systems*, vol. 15, no. 5-6, pp. 571 – 582, 1999. [Online]. Available: http://www.sciencedirect.com/science/article/B6V06-405TF92-5/2/f56763e0b6628eb7240c29ebc8bd835f

[146] C. Engelmann, S. Scott, and G. A. Geist, II, "Distributed peer-to-peer control in harness," in *Proceedings of the International Conference on Computational Science-Part II*, ser. ICCS '02. London, UK, UK: Springer-Verlag, 2002, pp. 720–728. [Online]. Available: http://portal.acm.org/citation.cfm?id=645458.655464

[147] Z. Chen and J. 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, p. 10 pp.

[148] Z. Chen, "Extending algorithm-based fault tolerance to tolerate fail-stop failures in high performance distributed environments," in *IPDPS*. IEEE, 2008, pp. 1–8.

[149] H. Kuang-Hua and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. Comput.*, vol. 33, pp. 518–528, June 1984. [Online]. Available: http://dx.doi.org/10.1109/TC.1984.1676475

[150] C. Engelmann and G. A. A. Geist, "Super-scalable algorithms for computing on 100,000 processors," in *Lecture Notes in Computer Science: Proceedings of the* 5th *International Conference on Computational Science (ICCS) 2005, Part I*, vol. 3514. Atlanta, GA, USA: Springer Verlag, Berlin, Germany, May 22-25, 2005, pp. 313–320. [Online]. Available: http://www.christian-engelmann.info/publications/engelmann05superscalable.pdf

[151] S. Chakravorty, C. L. Mendes, and L. V. Kal, "Proactive fault tolerance in mpi applications via task migration," in *In International Conference on High Performance Computing*, 2006.

[152] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott, "Proactive fault tolerance for hpc with xen virtualization," in *Proceedings of the 21st annual international conference on Supercomputing*, ser. ICS '07. New York, NY, USA: ACM, 2007, pp. 23–32. [Online]. Available: http://doi.acm.org/10.1145/1274971.1274978

[153] Y. Liang, Y. Zhang, A. Sivasubramaniam, M. Jette, and R. Sahoo, "Bluegene/l failure analysis and prediction models," in *Proceedings of the International Conference on Dependable Systems and Networks*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 425–434. [Online]. Available: http://portal.acm.org/citation.cfm?id=1135532.1135723

[154] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott, "Proactive process-level live migration in hpc environments," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 43:1–43:12. [Online]. Available: http://portal.acm.org/citation.cfm?id=1413370.1413414

[155] Y. Li, P. Gujrati, Z. Lan, and X.-h. Sun, "Fault-driven re-scheduling for improving system-level fault resilience," in *Proceedings of the 2007 International Conference on Parallel Processing*, ser. ICPP '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 39–. [Online]. Available: http://dx.doi.org/10.1109/ICPP.2007.42

[156] F. H. Mathis, "A generalized birthday problem," *SIAM Review*, vol. 33, no. 2, pp. 265–270, June 1991.

[157] D. E. Knuth, *The art of computer programming, volume 3: (2nd ed.) sorting and searching.* Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998.

[158] P. Flajolet, P. J. Grabner, P. Kirschenhofer, and H. Prodinger, "On Ramanujan's Q-function," *J. Comput. Appl. Math.*, vol. 58, no. 1, pp. 103–116, 1995.

[159] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition.* The MIT Press and McGraw-Hill Book Company, 2001.

[160] R. Riesen, K. Ferreira, J. Stearley, R. Oldfield, J. H. L. III, K. Pedretti, and R. Brightwell, "Redundant computing for exascale systems," Sandia National Laboratories, Technical report SAND2010-8709, Dec. 2010.

[161] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, September 1996.

[162] W. Gropp, "MPICH2: A new start for MPI implementations," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 9th European PVM/MPI Users' Group Meeting*, ser. Lecture Notes in Computer Science, D. Kranzlmuller, P. Kacsuk, J. Dongarra, and J. Volkert, Eds., vol. 2474, September/October 2002.

[163] R. Brightwell, T. Hudson, K. T. Pedretti, and K. D. Underwood, "SeaStar Interconnect: Balanced bandwidth for scalable performance," *IEEE Micro*, vol. 26, no. 3, May/June 2006.

[164] J. E. S. Hertel, R. L. Bell, M. G. Elrick, A. V. Farnsworth, G. I. Kerley, J. M. McGlaun, S. V. PetneY, S. A. Silling, P. A. Taylor, and L. Yarrington, "CTH: A software family for multi-dimensional shock physics analysis," in *Proceedings of the 19th International Symposium on Shock Waves*, Jul. 1993, pp. 377–382.

[165] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings, "Predictive performance and scalability modeling of a large-scale application," in *Proceedings of the ACM/IEEE conference on Supercomputing*, 2001, pp. 37–48.

[166] S. J. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *J Comp Phys*, vol. 117, no. 1, pp. 1–19, 1995.

[167] Sandia National Laboratories. (2010, April) The LAMMPS molecular dynamics simulator. [Online]. Available: http://lammps.sandia.gov

[168] ——. (2010, April) The mantevo project home page. [Online]. Available: https://software.sandia.gov/mantevo

[169] K. T. Pedretti, C. Vaughan, K. S. Hemmert, and B. Barrett, "Application sensitivity to link and injection bandwidth on a Cray XT4 system," in *Proceedings of the 2005 Cray User Group Annual Technical Conference*, Helsinki, Finland, May 2008.

[170] J. S. Plank and K. Li, "ickp: A consistent checkpointer for multicomputers," *Parallel & Distributed Technology: Systems & Applications, IEEE*, vol. 2, no. 2, pp. 62–67, 1994.

[171] (2011, Feb.) Libckpt web page. [Online]. Available: http://web.eecs.utk.edu/~plank/plank/www/libckpt.html

[172] P. Deutsch and J.-L. Gailly, "ZLIB Compressed Data Format Specification version 3.3," RFC 1950 (Informational), Internet Engineering Task Force, May 1996. [Online]. Available: http://www.ietf.org/rfc/rfc1950.txt

[173] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot, *Handbook of Applied Cryptography*, 1st ed. Boca Raton, FL, USA: CRC Press, Inc., 1996.

[174] R. Rivest, "The MD5 Message-Digest Algorithm," RFC 1321 (Informational), Internet Engineering Task Force, Apr. 1992, updated by RFC 6151. [Online]. Available: http://www.ietf.org/rfc/rfc1321.txt

[175] R. Housley, "A 224-bit One-way Hash Function: SHA-224," RFC 3874 (Informational), Internet Engineering Task Force, Sep. 2004. [Online]. Available: http://www.ietf.org/rfc/rfc3874.txt

[176] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246 (Proposed Standard), Internet Engineering Task Force, Aug. 2008, updated by RFCs 5746, 5878, 6176. [Online]. Available: http://www.ietf.org/rfc/rfc5246.txt

[177] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer, "OpenPGP Message Format," RFC 4880 (Proposed Standard), Internet Engineering Task Force, Nov. 2007, updated by RFC 5581. [Online]. Available: http://www.ietf.org/rfc/rfc4880.txt

[178] T. Ylonen and C. Lonvick, "The Secure Shell (SSH) Transport Layer Protocol," RFC 4253 (Proposed Standard), Internet Engineering Task Force, Jan. 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4253.txt

[179] B. Ramsdell, "Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.1 Message Specification," RFC 3851 (Proposed Standard), Internet Engineering Task Force, Jul. 2004, obsoleted by RFC 5751. [Online]. Available: http://www.ietf.org/rfc/rfc3851.txt

[180] V. Manral, "Cryptographic Algorithm Implementation Requirements for Encapsulating Security Payload (ESP) and Authentication Header (AH)," RFC 4835 (Proposed Standard), Internet Engineering Task Force, Apr. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc4835.txt

[181] W. J. Camp and J. L. Tomkins, "Thor's hammer: The first version of the Red Storm MPP architecture," in *In Proceedings of the SC 2002 Conference on High Performance Networking and Computing*, Baltimore, MD, November 2002.

[182] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, pp. 40–53, March 2008. [Online]. Available: http://doi.acm.org/10.1145/1365490.1365500

[183] Network-Based Computing Laboratory, Ohio State University, "OSU MPI benchmarks (OMB)," http://mvapich.cse.ohio-state.edu/benchmarks/, Apr. 19 2010.