ABSTRACT

FIALA, DAVID JEROME. Transparent Resilience Across the Entire Software Stack for High-Performance Computing Applications. (Under the direction of Frank Mueller.)

Proposed exascale systems will present a number of considerable resilience challenges. In particular, DRAM soft-errors, or bit-flips, are expected to greatly increase due to much higher memory density of these systems. Current hardware-based fault-tolerance methods cannot cope by themselves with the expected soft error frequency rate. As a result, additional software is needed to address this challenge. Moreover, not only must an application be protected, but its entire software stack and operating system must also be guarded by software resilience in order to guarantee that it will scale without experiencing memory corruption.

This work provides generalized error protection mechanisms for memory faults of high-performance computing (HPC) applications that are transparent to the user. The work contributes generalized resilience mechanisms for at each level of the HPC software stack:

1. At the application level, on-demand page integrity verification coupled with software-based error correcting codes allows for automatic error recovery in the event of memory failures.

2. At the MPI layer, transparent redundancy in execution protects applications experiencing even high rates of data corruption and prevents faults from spreading to other MPI tasks, which otherwise would cause corruption of all tasks of a job.

3. Within the operating system, key kernel data structures pertaining to the MPI application and runtime system are captured in so-called mini checkpoints while the application data itself remains untouched so that a kernel panic, memory corruption, and other detectable bugs within the kernel can be mitigated by a warm restart of the kernel resulting in only a brief interruption of the application.

These capabilities allow HPC application developers to focus on algorithm design instead of resilience concerns as their new and existing code bases scale to larger computing infrastructure experiencing memory failure rates exceeding hardware protection capabilities.

Transparent Resilience Across the Entire Software Stack for High-Performance Computing Applications

by
David Jerome Fiala

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2015

APPROVED BY:

Vincent Freeh	Xiaohui Gu
Huiyang Zhou	Frank Mueller Chair of Advisory Committee

BIOGRAPHY

David Fiala attended North Carolina State University from 2010 to 2015.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

1

INTRODUCTION

1.1 High Performance Computing

To meet the demands of ever growing scientific simultations, data processing, and other experiments running on computing platforms, the size of computing clusters continues to grow. As individual components that make up supercomputers cannot be sped up infinitely, massive parallel machines are designed to operate in a highly coupled interconnected network. Today, the largest publicly announced supercomputers contain several million computational cores, and we can expect that supercomputers will continue to grow in terms of total component count [5].

Due to the nature and sensitivity of the work being performed on supercomputing facilities, there exist types of workloads that simply cannot afford to fall victim to incorrect results or memory corruption due to faults. This is particularly the case for simulations that run for very long times (e.g., several days). Other workloads may be bound by temporal constraints, which require successful completion by a specific time in order to be useful (e.g., weather/hurricane forecasting).

1.2 Definitions

The sheer quantity of components, density, and heat generated by today's supercomputers increases the chance of component failures in everyday operations. In a highly coupled environment, any individual error may result in incorrect results, loss of computational work, or may cascade causing the entire computing system to report incorrect results or terminate without reporting results. Such errors occur due to manufacturing defects, inadequate heat dissipation, or even cosmic radiation striking the components.

The taxonomy used to describe failures in this work follows established definitions [101] and is as follows:

1. **Fault:** An event that causes an unintended defect in a system is called a fault. This may include unintended state or even power failures.
2. **Memory Fault:** A type of fault that affected memory of some component is called a memory fault.
3. **Error:** A fault that causes a discrepancy between the intended and actual behavior of a system is called an error.
4. **Memory Error:** A type of error wherein referencing of data subject to a memory fault in such a way that the fault is uncorrected is called a memory error.
5. **Failure:** An erroneous outcome wherein an error's effects cause an incorrect action (or outcome) is called a failure (i.e., a crash or wrong output).

The research community and this work refer to Silent Data Corruption (SDC) failures as a fault that has gone unnoticed. While errors may be the result of a hard fault (a fault that persists, such as a "stuck bit") or a soft fault (one that is transient in nature), this work focuses on SDC that is a result of a soft fault. This implies that the SDC events protected by this work are temporary and that a recovery mechanism fixes any single instance (per point in time) of such transient faults. We also note that not all SDCs necessary results in a failure. Depending on where a fault occurs, it is possible that some execution paths will lead to a benign error, which will either not be referenced or even though referenced do not affect the outcome / numerically have an insignificant impact on the outcome. (The latter is sometimes used in algorithm-based fault tolerance.)

Throughout the remainder of this work, SDCs are assumed to have been caused by memory faults that evade hardware correction support such as ECC (and sometimes even hardware

detection). In the context of protecting the software stack, a failure will be used to denote an outcome where a crash occurs or an incorrect output is produced. Colloquially, error, fault, and SDC are often used interchangeably in the community because software-based detection and correction operates at a level higher than hardware and hardware-based protection mechanisms.

1.3 Silent Data Corruption

An individual computer within a cluster that makes up a supercomputer may experience many types of errors. Its components include one or more central processing units (CPU), dynamic memory (RAM), and typically a network device (NIC). When any of these components experiences an error, it is possible that the underlying memory it contains will be changed, such as a zero becoming a one. To remedy this, many components include some degree of protection by incorporating error-correcting codes that provide detection and correction of some errors. However, in high performance computing (HPC) environments, the rate of errors given the number of components in a large-scale system has allowed us to observe memory failures that evade today's hardware protection [96, 89]. We call these types of errors, when they change a system's memory without being detected, silent data corruption (SDC) since the memory is no longer correct and the error is not detected by the hardware.

Beyond hardware protection, the next line of defense against silent data corruption is software based techniques that may detect and optionally correct memory after an error has occured. Developing algorithms to detect and correct possible SDCs is non-trivial, which is why this work focuses on providing generalized mechanisms that may protect any application without complex re-engineering of HPC algorithms [58, 79, 68, 13]. Algorithm-based fault tolerance (ABFT) provides protection to an application by storing additional information about its data in encodings that allow for detection and optional correction of incorrect values. ABFT requires that an algorithm be designed in such a way that it may abstract high level meaning from encodings for verification, correction, and sanity checks.

1.4 Hypothesis

Hardware resilience to memory faults in high-performance computing applications is reaching a point where hardware alone will no longer be effective against data corruption at scale. Protecting applications against memory faults will require additional software to supplement hardware fault tolerance. While algorithm-based fault tolerance approaches may protect some parts of an application's data, they are both non-trivial to develop and not applicable to many types of

software. Not only must an application be protected, but its entire software stack and operating system must also be guarded by software resilience in order to guarantee that it will scale without experiencing memory corruption.

The hypothesis of this dissertation is:

By using generalized software protection mechanisms including redundancy, software-based ECC, and fault-tolerant operating systems, we can protect an entire high-performance computing software stack from memory faults at scale.

The impact of this disseration is:

The use of these mechanisms will alleviate the design burdens of developing hardened high-performance computing algorithms as well as allowing legacy applications to continue scaling without algorithmic refactoring in the face of increased memory faults.

1.5 Contributions

With generalized mechanisms, applications will not require modifications to continue scaling on ever larger supercomputers. Specifically, to provide generalized fault tolerance from memory errors, this dissertation provides the following contributions:

1. First, this work studies the potential for redundancy to detect and correct soft errors in MPI message-passing applications while investigating the challenges inherent to detecting soft errors within message passing (MPI) applications by providing transparent MPI redundancy. By assuming a model wherein corruption in application data manifests itself by producing differing MPI messages between replicas, this work studies the best suited protocols for detecting and correcting corrupted MPI messages. Using a fault injector, this work observes that even a single error can have profound effects on applications by causing a cascading pattern of corruption, which in most cases spreads to all other processes. Results indicate that the proposed consistency protocols can successfully protect MPI applications experiencing even high rates of data corruption.

2. Second, this work introduces FlipSphere, a tunable, transparent silent data corruption detection and correction library for HPC applications that is first in its class to use hardware accelerators such as the Intel Xeon Phi MIC to increase application resiliency. FlipSphere provides comprehensive silent data corruption protection for application memory by implementing on-demand page integrity verification coupled with software-based error correcting codes that allow for automatic error recovery in the event of memory failures.

This work investigates the trade-off of hardware resources for resiliency in terms of runtime overheads and efficiency.

3. Finally, this work investigates mechanisms to protect running applications from memory faults within the operating system given the inherent dependency of all applications on their supervising operating system. This dependency makes it a critical component of the software stack that requires protection. As part of this work, key data structures, operating system instrumentation points and methods to transparently persist application data across operating system failures are investigated to provide generalized requirements for a resilient operating system that allows its supervised applications to survive a failure. This work introduces mini-ckpts, an experimental implementation of a modified Linux kernel that provides resilience to running applications during a kernel panic, memory corruption, and other detectable bugs within the kernel. The effectiveness and costs associated with mini-ckpts are evaluated on scientific benchmarks while being exposed to faults injected within the operating system.

Ultimately, this work provides generalized resilience mechanisms capable of protecting a running HPC application from memory faults at the application level, MPI layer, and operating system. These capabilities allow HPC application developers to focus on algorithm design instead of resiliency concerns as their new and existing code bases scale to larger computing facilities experiencing memory failure rates exceeding hardware protection capabilities.

1.6 Organization

Chapter 2 will provide background on common related work common to all chapters. Chapter 3 describes RedMPI, which provides transparent silent error detection and correction for message passing applications. Chapter 4 addresses generalized error detection and correction for both applications and libraries using hardware accelerators. In Chapter 5, we address memory errors that may occur in the operating system by designing a transparent warm reboot and recovery mechanism. In Chapter 6, we conclude this work.

CHAPTER

2

RELATED WORK

2.1 Fault Tolerance in HPC

Since the early 1990s [30], fault tolerance in large-scale HPC systems is primarily assured through application-level checkpoint/restart (C/R) to/from a parallel file system. Support for C/R at the system software layer exists, such as through the Berkeley Lab Checkpoint Restart (BLCR) [54] solution, but it is only employed at a few HPC centers. Diskless C/R, *i.e.*, using compute-node memory for distributed checkpoint storage, exists as well, like the Scalable C/R (SCR) library [21], but is rarely used in practice. Message logging, algorithm-based fault tolerance, proactive fault tolerance, and Byzantine fault tolerance have all been researched in the past and are also not available in production HPC systems. Redundancy in HPC, as showcased in this work, has only been recently explored (see below).

2.2 Silent Data Corruption Protection

Historically, the primary defense against silent data corruption (SDC) has been error correcting code (ECC) in dynamic random access memory (DRAM). Only very recently, ECC has been deployed in server-market processors, such as in the AMD Opteron and Intel Xeon, to protect

caches and registers as well. Single event upsets (SEUs) [41], *i.e.*, bit flips caused by natural high-energy radiation, are the dominant source of SDC. In today's memory modules and processors, single-error correction (SEC) double-error detection (DED) ECC protects against SEU as well as single event multiple upset (SEMU) scenarios. Chipkill offers additional protection against wear-out and complete failure of a memory module chip by spanning ECC across chips but Bose Chaudhuri-Hocquenghem (BCH) encoding provides better energy-delay characteristics [67]. Manufacturers will continue exploring mitigation strategies and will be able to continue to deliver products with certain soft error resilience. However, high reliability for the latest-generation processors and memories come [55]. Redundancy may provide more extensive SDC protection, especially considering the expected increase in SECDED ECC double-error rates.

Pure software-based solutions [45] try to protect against memory corruption without extending hardware ECC. However, they cannot provide perfect coverage to all memory and are subject to job failure if just a single process terminates due to a fault. In contrast, redundancy for SDC correction survives single process faults more gracefully.

2.3 Silent Data Corruption in HPC

Studies primarily done at Los Alamos National Laboratory (LANL) focused on analyzing the probability and impact of silent data corruption in HPC environments. One investigation [70] showed that a Cray XD1 system with an equivalent number of processors as the ASCI Q system, *i.e.*, ∼18,000 field-programmable gate arrays (FPGAs) with 16.5 TB SECDED-ECC memory, would experience one SDC event within 1.5 hours due to the high vulnerability of the FPGAs. Ongoing work at LANL focuses on radiating new-generation processors, flash, and memory with neutrons at the Los Alamos Neutron Science Center (LANSCE) to measure vulnerability and efficiency of protection mechanisms. Another study [20] at Lawrence Livermore National Laboratory (LLNL) investigated the behavior of iterative linear algebra methods when confronted with SDC in their data structures. Results show that linear algebra solvers may take longer to converge, not converge at all, or converge to a wrong result. These investigations not only point out a high SDC rate when scaling up HPC systems, but also the severe impact SDC has. More extensive SDC protection is needed to assure application correctness at extreme-scale.

CHAPTER

3

REDMPI: REDUNDANCY FOR DETECTION AND CORRECTION OF SILENT ERRORS

3.1 Introduction

In High-End Computing (HEC), faults have become the norm rather than the exception for parallel computation on clusters with 10s/100s of thousands of cores. Past reports attribute the causes to hardware (I/O, memory, processor, power supply, switch failure etc.) as well as software (operating system, runtime, unscheduled maintenance interruption). In fact, recent work indicates that (i) servers tend to crash twice a year (2-4% failure rate) [89], (ii) 1-5% of disk drives die per year [78], (iii) DRAM errors occur in 2% of all DIMMs per year [89], which is more frequent than commonly believed, and (iv) large scale studies indicate that simple ECC mechanisms alone are not capable of correcting a significant number of DRAM errors [61].

Even for small systems, such causes result in fairly low mean-time-between-failures/interrupts (MTBF/I) as depicted in Figure 3.1 [57], and the 6.9 hours estimated by Livermore National Lab for its BlueGene confirms this.

Table 3.1 Reliability of HPC Clusters

System	# CPUs	MTBF/I
ASCI Q	8,192	6.5 hrs
ASCI White	8,192	5/40 hrs ('01/'03)
PSC Lemieux	3,016	9.7 hrs
Google	15,000	20 reboots/day
ASC BG/L	212,992	6.9 hrs (LLNL est.)

In response, long-running applications on HEC installations are required to support the checkpoint/restart (C/R) paradigm to react to faults. This is particularly critical for large-scale jobs; as the core count increases, so does the overhead for C/R, and it does so at an exponential rate. This does not come as a surprise as any single component failure suffices to interrupt a job. As we add system components (multicore chips, memory, disks), the probability of failure combinatorially explodes.

For example, a study from 2005 by Los Alamos National Laboratory estimates the MTBF, extrapolating from current system performance [77], to be 1.25 hours on a petaflop machine. The wall-clock time of a 100 hour job in such as system was estimated to increase to 251 hours due to the C/R overhead implying that 60% of cycles are spent on C/R alone, as reported in the same study.

Prior work [28, 29, 77] has revealed that checkpoint/restart efficiency, *i.e.*, the ratio of useful vs. scheduled machine time, can be as high as 85% and as low as 55% on current-generation HEC systems. Recent work by Sandia [43] shows rapidly decaying useful work for increasing node counts (see Table 3.2). Only 35% of the work is due to computation for a 168 hour job

Table 3.2 168-hour Job, 5 year MTBF

# Nodes	work	checkpt	recomp.	restart
100	96%	1%	3%	0%
1,000	92%	7%	1%	0%
10,000	75%	15%	6%	4%
100,000	35%	20%	10%	35%

on 100k nodes with a node MTBF of 5 years while the remainder is spent on checkpointing,

restarting and then partial recomputation of the work lost since the last checkpoint. Table 3.3 shows that for longer-running jobs or shorter MTBF, useful work becomes *insignificant* as most of the time is spent on restarts.

Table 3.3 100k Node Job, varied MTBF

job work	MTBF	work	checkpt	recomp.	restart
168 hrs.	5 yrs	35%	20%	10%	35%
700 hrs.	5 yrs	38%	18%	9%	43%
5,000 hrs,	1 yr	5%	5%	5%	85%

The most important finding of the Sandia study is that **redundancy in computing can significantly revert this picture.** By doubling up the compute nodes so that every node N has a replica node N', a failure of primary node N no longer stalls progress as the replica node N' can take over its responsibilities. Their prototype, rMPI, provides dual redundancy [43]. And *redundancy scales*: As more nodes are added to the system, the probability for simultaneous failure of a primary N *and* its replica rapidly decreases. Of the above overheads, the recompute and restart overheads can be nearly eliminated (to about 1%) with only the checkpointing overhead remaining — at the cost of having to deploy twice the number of nodes (200,000 nodes in Table 3.2) and four times the number of messages [43]. But once restart and rework overheads exceed 50%, redundancy is actually *cheaper* than traditional C/R at large core counts.

The failure scenarios above only cover a subset of actual faults, namely those due to fail-stop behavior / those detectable by monitoring of hardware and software. Silent data corruption (SDC) is yet a different class of faults, which is the focus of this work. It materializes as bit flips in storage (both volatile memory and non-volatile disk) or even within processing cores. A single bit flip in memory can be detected (with CRC) and even mitigated with error correction control (ECC). Double bit flips, however, force an instant reboot after detection since ECC cannot correct such faults. While double bit flips were deemed unlikely, the density of DIMMs at Oak Ridge National Lab's Cray XT5 causes them to occur on a daily basis (at a rate of one per day for 75,000+ DIMMs) [48].

Meanwhile, even single bit flips in the processor core remain undetected as only caches feature ECC while register files or even ALUs typically do not. Significant SDC rates were also reported for BG/L's unprotected L1 cache [21], which explains why BG/P provides ECC in

L1. Nvidia made a similar experience with its shift to ECC in their Fermi Tesla GPUs. Yet, hardware redundancy remains extremely costly [94, 71, 51, 83]

Today, the frequency of bit flips is no longer believed to be dominated by single-event upsets due to radiation from space [81] but is increasingly attributed to fabrication miniaturization and aging of silicon given the increasing likelihood of repeated failures in DRAM after a first failure has been observed [89]. With SDCs occurring at significant rates, not every bit flip results in faults. Flips in stale data or code remain without impact, but those in active data/code may have profound effects and potentially render computational results invalid without ever being detected. This creates a severe problem for today's science that relies increasingly on large-scale simulations. Redundant computing can detect SDCs where relevant, i.e., when results are impacted. While detection requires dual redundancy, correction is only feasible with triple redundancy. Such high levels of redundancy appear costly, yet may be preferable to flawed scientific results. Triple redundancy is also cheaper than comparing the results of two dual redundant jobs, which would be the alternative at scale given the amount of useful work without redundancy for large systems from Table 3.3. Overall, the state of HEC requires urgent investigation to level the path to exascale computing — or exascale HEC may be doomed as a failure (with very short mean times, ironically).

3.1.1 Modeling Redundancy

Elliott *et al.* [37] combine partial redundancy with checkpointing in an analytic model and experimentally validate it. Results indicates that for an exascale-size machine, more jobs can utilize a cluster under redundancy than would be possible with checkpointing without redundancy for the same number of cores, i.e., redundancy increases capacity computing in terms of HPC job throughput.

We used this model in combination with Jaguar's system MTBF of 52 hours [100] (equivalent to a node MTBF of 50 years) to assess the viability of redundancy. Consider a 128 hour job (without checkpointing). We then assess the time required for such a job without redundancy (1x), dual redundancy (2x) and triple redundancy (3x) at different node counts under weak scaling and with an optimal checkpoint interval to minimize overall execution (see Figure 3.1). At 18,688 nodes (Jaguar), marked as line C, single-node (1x) runs are about 7% faster than dual (2x) and 20% faster than triple (3x) redundancy. The problem is that a job at 1x will have no indication if it had been subjected to an SDC. Consider Jaguar's double bit error rate of once a day again [48], which is silently ignored (to increase system availability) as it cannot be corrected. Scientists will not know if their outputs were affected, i.e., if outputs are flawed

(incorrect science problem).

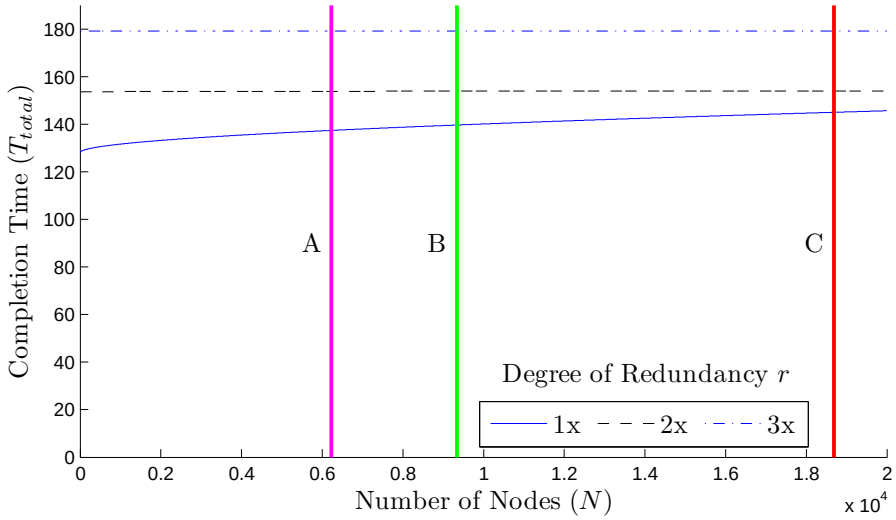


Figure 3.1 Modeled Time to Completion with Redundancy

Let us consider dual redundancy at half the node count of Jaguar (line B in Figure 3.1). In order to ensure absence of SDCs, a user would have to run a single redundant (1x) job twice for a total time of about 280 hours (twice 140) vs. a dual redundant (2x) job at twice the number of nodes (full Jaguar size, line A) with 155 hours. Hence, dual redundancy, results in nearly half the wall-clock time if SDC detection is a requirement for verification and validation of a job's results.

Consider triple redundancy at a third of Jaguar's node count (line A). Running two jobs at 1x takes about 276 hours (twice 138), a dual redundant (2x) job 145 hours and a triple redundant job 180 hours. The output of the two 1x jobs differs, a third run would be required (assuming that two of them produce correct results). If the dual redundant (2x) job detects an error, it also needs to be rerun. The triple redundant (3x) job, in contrast, can correct errors so that no reruns would be needed.

At exascale core counts of one million and a node count of 100,000 (swim lane 1 [90, 35]), dual redundancy would have the lowest cost (lower than single job at 1x). The additional cost of SDC correction at triple redundancy adds another 14% overhead in wall-clock time, with the benefit of no repeated runs for SDCs. This is based on the assumption of Jaguar's system MTBF (52 hours), even though the MTBF could be much smaller given that double bit errors

for a 128 Petabyte system would occur every four minutes (compared to one a day today on Jaguar) [48]. SDC detection, if not correction, may thus become essential at exascale.

3.1.2 Contributions

The main contributions of this work are (i) the design of novel silent data correction detection / correction methods and (ii) a study on the challenges and costs of performing SDC protection using redundancy. By utilizing redundancy, our key to success is to not only rely on reactive resilience requiring restart overheads but to sustain failures with forward computational progress without a need to restart.

Our work makes the following major contributions: (1) We contribute the design and implementation of protocols for SDC detection and correction at the communication layer. (2) We demonstrate the capabilities and assess the cost of redundancy to (a) detect SDC and (b) recover from such corruption in experiments on a real system. While dual redundancy can detect SDCs, triple redundancy can actually correct them through voting. We study the benefits and limitations of the spectrum ranging from no redundancy over dual to triple redundancy in terms of overhead and computing/interconnect resource costs. A key challenge is to limit the overhead for SDC detection by reducing the relevant footprint of computational results, which we explore. (3) We assess the resilience of HEC jobs to faults through injection. Hardware and software failures are studied through injection on an actual cluster. (4) We develop a live SDC tracking and reporting framework to investigate the effects of SDCs on applications in terms of their rate of taint (corruption) progression spreading from node to node via MPI communication. Further, we use this framework to evaluate several application responses to fault injection and classify three types of observed behavior that result in invalid data being generated. In summary, this work contributes to fault detection and recovery by significantly advancing existing techniques by controlling levels of redundancy intervals in the presence of hardware and software faults.

3.2 Design

This work presents RedMPI, an MPI library that is capable of both detecting and correcting SDC faults. RedMPI creates “replica” MPI tasks for each “primary” task and performs online MPI message verification intrinsic to existing MPI communication. The replicas compare received messages, or hashes, from multiple senders and can thus detect if a process’s communication data has been corrupted.

RedMPI can run in double redundant mode and detect divergent messages between replicas.

Such messages are indicative of corruption due to the fact that replicas will be run in a deterministic manner. When RedMPI is run in triple redundant mode, it gains the additional potential to also correct faulty messages from a corrupted replica. RedMPI supports additional levels of redundancy for environments where multiple near-simultaneous faults can occur during data transmission. A voting algorithm is used to determine which of the received messages are correct and should be used by all receivers.

To detect SDCs, RedMPI solely analyzes the content of MPI messages to determine divergence between replicas during communication. Upon divergence, the result deemed to be invalid is discarded on the receiver side and transparently replaced with a known "good" value from another replica.

A different SDC detection approach would be to constantly compare the memory space of replicas' processes and compare results. Such an approach suffers from excessive overhead due to constant traversals of large memory chunks, overhead due to global synchronization to ensure that each process is paused at the exact same spot during a memory scan, and the communication required for replicas to compare their copy of each memory scan while looking for differences. In this case, if corruption is detected, it is not feasible to correct the memory while the application is running as this could interfere with application-side writes to the same memory region. This, in turn, could necessitate a rollback of all tasks to the last "good" checkpoint (assuming that checkpointing was also enabled).

By instead focusing on the MPI messages themselves, we have cut our search area down to only data that is most critical for correctness of an MPI application; i.e., we argue communication correctness is a necessary (but not sufficient) condition for output correctness. Moreover, should an SDC occur in memory that is not immediately communicated over MPI, the fault is eventually detected as the corrupted memory may later be accessed, operated on, and finally transmitted. The same principle holds true for data that became corrupted while residing in a buffer or any other place in memory. If the SDC is determined to eventually alter messages, then RedMPI detects it during transmission, independent of when, where or how the SDC originated.

It is very important to note that RedMPI is designed to protect an an entire application from SDC by using replication. RedMPI is not designed to protect an interconnect. By assuming that an application's most critical data is communicated during/after computation, we have effectively reduced the scope to data that gets communicated and may be compared between replicas to ensure consistency. A process receiving corrupted data that affects important calculations will eventually result in message correction so that uncorrupted replicas are guaranteed to have received correct data, only.

3.2.1 Point-to-Point Message Verification

The core of RedMPI's error detection capabilities are designed around a reliable, verifiable point-to-point communication protocol. Specifically, a point-to-point message (e.g., `MPI_Isend`) sent from an MPI process must be identical to the message sent by other replicas for any given rank. Upon successful receipt of a message, the MPI application is assured that the message is valid (not corrupted).

Internally, a verification message may take the form of a complete message duplicate that is compared byte-by-byte. Alternatively, since MPI messages may be large, it is in many cases more efficient to create a unique hash of the message data and use the hash itself for message verification to reduce network bandwidth. Message data verification can be performed at either the sender or the receiver.

Let us first consider the case of sender-side verification. To perform verification at the sender, all of the replicas need to send a message to communicate with each other and verify their content (through some means) before sending the verified data to the receiving replicas. However, this approach incurs added latency and overhead for each message sent due to the time taken to transmit between replicas and to internally verify messages. Additionally, it is best to optimize for the critical path, i.e., for the case that a sent message is not corrupted and that all senders have matching data. A sender-side approach is subject to additional overhead for every message sent at both sending and receiving nodes. Specifically, while every sent message is treated as suspect, the time required for the senders to agree that each of their own buffered messages is correct represents the time lost on the receiver side before the application can proceed. For this reason, RedMPI's protocols use receiver-side verification resulting in faster message delivery with considerably reduced message latency.

3.2.2 Assumptions

RedMPI does not protect messages over a transport layer, such as TCP or InfiniBand, and assumes the transport to be reliable. An unreliable network could cause undefined behavior and deadlock RedMPI. Fortunately, protection in the transport layer is well understood and is a common feature of modern communication fabrics, e.g. hamming codes or checksums for correction/retransmission.

As the primary focus of this work is to investigate redundancy as a means to protect application data, RedMPI only attempts to protect against corruption in data and not application code or instructions. RedMPI does not protect MPI I/O functionality, but orthogonal work [16] could be combined with RedMPI to also cover this aspect.

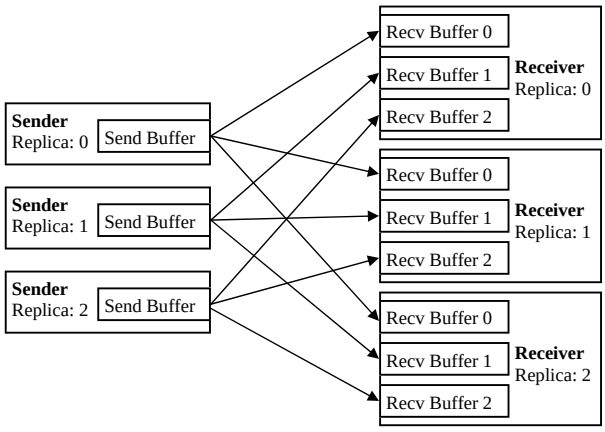


Figure 3.2 All-to-all Method Overview

In this work, we frequently reference RedMPI's `MPI_Isend` and `MPI_Irecv`. Note that RedMPI does, in fact, support 53 standard MPI functions including collectives via interposing, such as `MPI_Send` and `MPI_Recv`, around their non-blocking equivalents within RedMPI.

Deterministic ordering of messages between replicas (for wildcard receives, MPI_Testall/any, MPI_Waitall/any calls or reduction orders) is ensured via back-channel communication.

3.3 Implementation

RedMPI provides the capability of soft error detection for MPI applications by online comparison of results of nearly identical replica MPI processes. To an MPI developer, the execution of replica processes of their original code is transparent as it is handled through MPI introspection within the RedMPI library. This introspection is realized through the *MPI profiling layer* that intercepts MPI function calls and directs them to RedMPI. The profiling layer provides a standard API allowing libraries to wrap MPI calls and add additional or replacement logic for API calls.

To understand how RedMPI functions internally, it is first important to understand how redundancy is achieved within RedMPI. When launching an MPI job with RedMPI, some *multiple* of the original number of desired processes needs to be launched. For example, to launch an MPI job that normally requires 128 processes instead requires 256 or 384 processes for dual or triple redundancy, respectively. RedMPI handles redundancy internally and provides an environment to the application that appears to only have the originally required 128 processes.

The primary difference between replica MPI processes is a replica rank that distinguishes

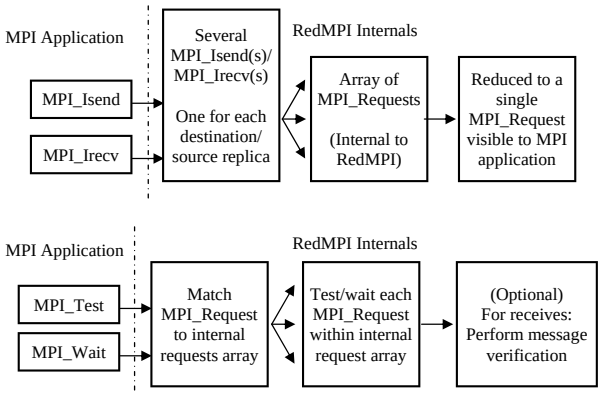


Figure 3.3 All-to-all Function Overrides

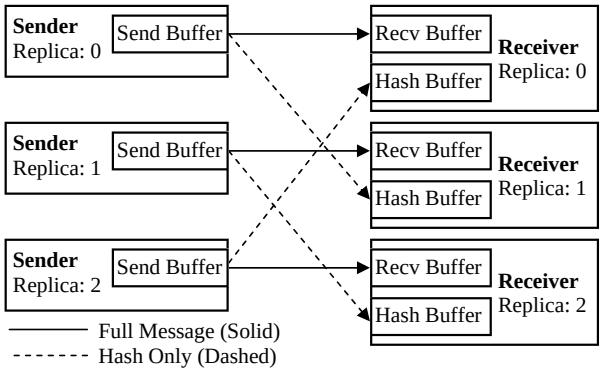


Figure 3.4 MsgPlusHash Method Overview

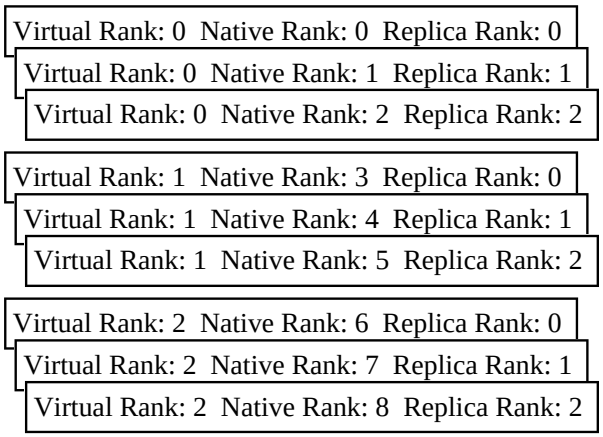


Figure 3.5 Triple Redundancy

redundant processes. For example, for an application to run with three replicas (triple redundancy), it would be started with three times as many MPI ranks as usual. Internally, the number of ranks visible to the MPI application would be divided by three where each redundant rank carries an internal replica rank of 0, 1, or 2. Figure 3.5 shows how triple redundancy may appear within an MPI application expecting a size of three.

The actual rank assigned to a process by `mpirun/mpiexec` is referred to as the native rank. The rank that is visible to an MPI process via the `MPI_Comm_rank` API is referred to as the virtual rank. Likewise, the size returned by `MPI_Comm_size` is referred to as the virtual size. The number of replicas running per virtual rank describes the redundancy of the application and is referred to as the replication degree. Within RedMPI, a mapping structure is stored in each process that allows the forward and reverse lookup of any processes' native rank, virtual rank, or replica rank.

3.3.1 Rank Mapping

When launching an MPI job, the mapping of native ranks may be specified on the command line with either a custom map file or by specifying a flag to indicate the desired virtual size. When a virtual size is specified on the command line, RedMPI automatically generates a structure that maps native ranks to a virtual rank of $[0 \ldots virtual\_size - 1]$ and assigns replica ranks of $[0 \ldots (native\_size/virtual\_size) - 1]$. Additionally, for each communicator, group, or MPI topology created within the MPI application, another map is created to track ranks within the new group.

Native to virtual mappings:
$virtual\_rank = native\_rank \bmod virtual\_size$ $replica\_rank = native\_rank / virtual\_size$
Virtual to native mapping:
$native\_rank =$ $\quad virtual\_rank + (replica\_rank \times virtual\_size)$

Figure 3.6 Rank Mapping Formulas

Internally, mappings can be translated using a formula (Figure 3.6) or by storing the data in a lookup structure. The formula method provides a simple, deterministic mapping function with low memory requirements, but it is not capable of providing fine-tuned control of rank mapping. By using a custom rank map file passed to RedMPI during startup, the user has the capability to specifically designate which virtual ranks are mapped to a native rank. This is advantageous in particular when the user desires to put replica processes on the same physical host or on neighboring hosts with low network latency. If a custom map file is omitted, the mapping formula is used to build the initial structure upon startup.

3.3.2 Message Corruption Detection & Correction

3.3.2.1 Method 1: All-to-all

RedMPI's first receiver-side protocol, All-to-all, supports both message verification and message voting to ensure that the receiver discards corrupted messages. The All-to-all method requires that each MPI message sent is transmitted from all sender replicas to each and every receiver replica. Thus, for a redundancy degree of three, each sender sends three messages where one message goes to each replica receiver as demonstrated by Figure 3.2. This means that for a degree of 2 or 3 the number of messages sent for a single `MPI_Isend` is 4 or 9, respectively. Each receiver listens for a message from each sender replica and places such messages in separate receive buffers.

Such message verification requires each sender to send *degree* messages for each MPI send encountered. This is realized by interposing `MPI_Isend` via RedMPI using the MPI profiling layer. The new `MPI_Isend` routine determines all replicas for the virtual rank of a message's destination. For each such replica, RedMPI performs a non-blocking send with a payload of the entire message and records the `MPI_Request` for each pending send. Upon completion, the

overridden `MPI_Isend` returns back to the MPI application a single `MPI_Request` that can later be used by `MPI_Test` or `MPI_Wait`. In a similar manner, `MPI_Irecv` is interposed by RedMPI to look up all replicas of the source's virtual rank and internally posts a non-blocking receive for a message from each replica. Every receive is stored into a different, temporary buffer entry. Again, all `MPI_Request` handles originating from non-blocking receives are recorded internally, but only a single `MPI_Request` is returned to the MPI application. Figure 3.3 visualizes this process.

Following an `MPI_Isend` or `MPI_Irecv`, an MPI application usually completes these requests with an `MPI_Test` or `MPI_Wait`. RedMPI interposes these functions as it needs to test not just the single `MPI_Request`, but rather impose a test for each array element of internal MPI requests corresponding to sends/receives from all replicas. The `MPI_Request` is looked up and the test or wait is performed on all outstanding requests. If the test or wait was performed on a request from an `MPI_Isend`, then no further action from RedMPI is required once the requests complete. Only a request from an `MPI_Irecv` requires extra steps to verify message reception from each replica.

When an MPI application receives a message, RedMPI internally waits for all replica MPI receive requests to finish during an `MPI_Test` or `MPI_Wait` before verifying the data. The actual verification occurs before `MPI_Test` or `MPI_Wait` return to the MPI application, but after all replica receives arrive. Verification is performed via memory comparison or computing a `SHA1` hash of each replica receive buffer and then comparing these hashes.

If during message verification a buffer mismatch is detected, RedMPI mitigates in a manner dependent on the degree of replication. With replication degree of two, it is impossible to determine which of the two buffers is corrupt. Hence, an error is logged noting corruption detection, but no corrective action may proceed since the source of corruption is indeterminate. With a replication degree exceeding two, buffers are compared and corrupted messages are voted out upon mismatch with the simple majority (of matching messages). In this event, RedMPI ensures that the MPI application's receive buffer contains the correct data by copying one of the verified buffers if necessary.

3.3.2.2 Method 2: Message Plus Hash (MsgPlusHash)

The MsgPlusHash (message plus hash) corruption detection and correction method provides a key performance enhancement over the All-to-all method by vastly reducing the total data transfer overhead per message and the number of messages in the general case. Similar to the All-to-all method, MsgPlusHash performs message verification solely on the receiver end.

The critical difference is that MsgPlusHash sends one copy of a message originating from an `MPI_Isend` in addition to a very small hash message. This change in protocol allows each sending replica to transmit their message only once, while the additional hash message is later used to verify each receiver's message. MsgPlusHash's contribution is a reduction of messages and thus bandwidth required from n^r to simply $n * r$ where n is the number of messages sent and r is the degree of replication.

Internally, the MsgPlusHash method interposes `MPI_Isend`, `MPI_Irecv`, `MPI_Test`, and `MPI_Wait` similarly to the All-to-all method previously discussed. The following logical overview of MsgPlusHash outlines how the MsgPlusHash implementation differs, while the same level of transparency is provided to MPI applications as for All-to-all. E.g., MsgPlusHash internally utilizes multiple send/receive `MPI_Request` handles, but the MPI application only ever receives one such `MPI_Request` handle.

To detect message corruption, the minimum requirement is a comparison between two different sources. Additionally, the most likely scenario (critical path) is for corruption to not exist. The MsgPlusHash method takes full advantage of these facts by only receiving a single copy of any message transmitted plus a hash from an alternate replica. From an efficiency standpoint, it is not necessary to send two full messages since a hash suffices to verify data correctness without imposing overheads of full message retransmission. Once the full message is received, a hash of the message is generated at the receiver and compared with a hash from a different replica. In the likely event that hashes match, the receiver can be assured that its message is correct, i.e., no corrective action is taken.

As shown in Figure 3.4, each sender replica must calculate where to send its message and where to send a hash of its message. The actual message's destination is simply calculated by finding the receiver with the same replica rank as the sender. The hash message's destination is calculated by taking the sender's replica rank and adding one. In the event that the destination replica rank exceeds the replication degree, the destination wraps around to replica rank 0. This pattern provides a simple and elegant solution to ensure each receiver always gets a copy of the full message plus a hash of the message from a different sender replica over a ring of replicas.

In the event that the message's hash does not match the received hash, it is necessary to determine if either the message is corrupt or if the received hash was produced from a corrupt sender. In any case, if a sender becomes corrupt, it transmits both the corrupted message and a hash of the corrupted message to adjacent receiving replicas. It is important to realize that a single corrupt sender affects both receivers. For example, with a replication degree of three where the middle sender (replica 1) transmits a corrupted message, we can see from Figure 3.4 that both receiver replicas 1 and 2 are affected. In this particular case, receiver replica 1 has received

a corrupt message, but a good hash since sender replica 0 was not corrupted. Conversely, receiver replica 2 has received a valid message, but a hash of a corrupted message. In this scenario, both receiver replicas 1 and 2 cannot yet determine if their message is corrupt, but they are both aware that one of their senders was in fact corrupted. Additionally, receiver replica 0 is unaware of any corruption since both message and hash matched on arrival. If the replication degree had only been two, a corrupt error would be logged at this point, but no corrective action would be available. With larger replication degrees, in contrast, a corrupted message can be corrected.

MsgPlusHash message correction is a multi-step process that takes place on the receiver replicas that have been flagged with potential corruption. In this event, there are always two adjacent receiver replicas that are aware of corruption since both are affected by the same corrupt sender replica. Yet, these receivers cannot easily identify whether their message or their hash was corrupted. By analyzing the communication pattern, it is obvious that the replica with a higher replica rank always contains the corrupt message with a bad hash. Therefore, the two adjacent replicas communicate with one another to determine which of them holds a correct message. After this handshake, the higher replica rank transmits a correction message to the lower ranked replica to complete the correction. (Ranks wrap around 0 and the highest rank in these situations.)

3.3.2.2.1 Corrupted Adjacent Replica Discovery

After a process encounters a message and hash mismatch, it initiates a handshake protocol to discover which of its adjacent replicas are also participating in a discovery. For each adjacent rank also actively trying to discover a potentially corrupted process, the other rank engages in the discovery protocol since its message and hash did not match. Any rank that did not obtain a mismatched message and hash is entirely unaware of the corruption elsewhere and thus not participate in the this protocol. In order for the two searching processes to find each other, they both attempt to send a probe to the rank below them (replica rank - 1) while simultaneously issuing a receive probe from the rank above them (replica rank + 1). After one of the processes receives a probe, an acknowledgment is returned. Figure 3.7 depicts this process. The highest rank's X+1 wraps around to rank 0 and vice versa.

In part (a) of Figure 3.7, the process posts a non-blocking receive to listen for a probe from above. Next, in part (b), the process posts a non-blocking receive to listen for an acknowledgment from the process below. With the receives in place, part (c) posts a non-blocking send as a probe to the rank below. The probe contains a copy of the received message's hash as a means to match this particular probe on the other end. At this point, the process waits for either the probe

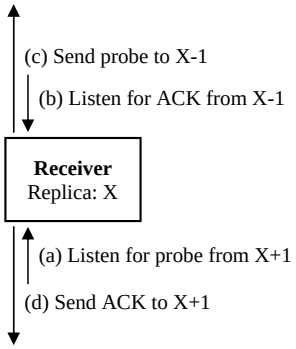


Figure 3.7 MsgPlusHash Correction

or acknowledgment requests to complete as they both result in different outcomes. If a probe message is received then the process can immediately assume that it is the lower-ranked replica and, as such, has a copy of the corrupted message due to the communication patterns. This lower rank then sends an acknowledgment (see part (d)) to signal that the discovery is complete. Meanwhile, if a process receives an acknowledgment instead of a probe message then that rank immediately assumes that it is the higher ranked process with a valid copy of the original message. In both cases, once discovery has completed, any outstanding sends and receives that were posted but left incomplete are now canceled through MPI from RedMPI.

The nature of the discovery process creates a problem: Two unique SDCs detected by adjacent replicas at separate times may send probe messages that are received in a later discovery. RedMPI handles this case by using hashes to identify whether a probe pertains to the SDC at hand. Probes that are unrelated to the current discovery process is safely discarded until an expected probe hash arrives.

With discovery complete, the higher ranked replica sends a full copy of the original, validated message to the lower rank. The lower rank receives a copy of this message within the application's buffer while overwriting the copy that originally was received in a corrupted state. Once this transfer completes, all replicas hold a validated message copy in their buffers and the MPI application may proceed.

Notice that RedMPI may sustain an unlimited number of corrupt messages from a sender provided that the degree of redundancy is greater than two and that no other replica of the same virtual rank becomes corrupt simultaneously. By our design, corruptions remain isolated to a single process, even if never corrected.

3.3.3 MPI Operations

Deterministic Results: RedMPI relies on keeping replica processes running with approximately equal progress in execution. As replicas execute in a deterministic manner, we guarantee that all MPI messages are sent in exactly the same frequency, order, and message content. There are, however, a few factors that might derail the replicas leading to non-deterministic results that would leave RedMPI inoperable, which has to be precluded. In particular, care was taken to ensure any MPI routine with the potential to diverge in execution progress of replicas is instead replaced with logic that provides the same results across all replicas.

One notable MPI routine with the potential to induce divergence is `MPI_Wtime` function. Not only is `MPI_Wtime` extremely likely to return a different value between separate processes and separate hosts, but its usage may guarantee different outcomes across processes especially if used as a random number seed. The divergence problem is solved by allowing only the rank zero replica to call `MPI_Wtime`. Since all replica ranks call `MPI_Wtime` at about the same time, the first replica simply sends a copy of its result to the others, which is then returned to the MPI application.

Another MPI routine with similar potential is the `MPI_Iprobe` function. Unlike `MPI_Wtime`, a probe may result in inconsistent results among replicas due to networking delays. It is possible that all but one replica received a message. To prevent results of `MPI_Iprobe` from diverging, the lowest ranking replica performs a real `MPI_Iprobe` for the requested message. Following the non-blocking probe, the lowest rank then sends a copy of the results to all higher ranking replicas. If `MPI_Iprobe` returned no message, then every replica simply reports that no message was found. Otherwise, if the lowest rank did report probing a message then each higher rank enters a blocking `MPI_Probe` to wait until their copy of the message arrives. As every replica has the same communication pattern, they are guaranteed to return from `MPI_Probe` quickly if the probed message had not, in fact, already arrived. Many other MPI functions such as `MPI_Testany`, `MPI_Testall`, `MPI_Waitany`, and `MPI_Waitall` are handled similar to `MPI_Iprobe` as previously described.

Random number generation may be considered a source of divergence, but in our experience many applications seed their random number generator with `MPI_Wtime`, which has already been protected. If this is not the case for an application then other protective measures would be required to ensure consistency between replicas or the application's source may need to be slightly modified to conform.

Some MPI operations (*e.g.*, `MPI_Recv`, `MPI_Iprobe`) may specify wildcards for their source or tag parameters (*i.e.*, `MPI_ANY_TAG`, `MPI_ANY_SOURCE`). To ensure that replicas do not receive

messages from differing nodes in a receive and to ensure that probe results return the same source and tag, RedMPI handles these wildcards differently than regular operations when they are encountered during program execution. When a wildcard is detected, only the lowest ranking replica actually posts the wildcard while the other replicas await an envelope containing the source and tag from the lowest replica in a manner similar to how `MPI_Wtime` is handled. Additional care must be taken to ensure that only the lowest rank replica can post receives until the wildcard has been resolved. Full technical details are omitted due to space. Overall, RedMPI fully supports wildcard sources, wildcard tags, or both in combination.

Mathematical operations such as those in `MPI_Reduce` will execute in the same order across replicas independent of their virtual rank. For this reason, if a floating point reduction is performed in a tree then although differing virtual ranks might perform different floating operations, our virtual replicas will always perform the same operations in order.

Collectives: MPI collectives pose a unique challenge for corruption detection/correction. First, there is a lack of non-blocking collectives in the MPI-2 standard (to be addressed in the future MPI-3 standard). Without non-blocking collectives, it is impossible to overlap collective operations. Thus, it is possible to sustain a faulty process in a collective that does not participate or encounter other unforeseen problems. These issues may cause other participants to become non-responsive ("hang") or fail ("crash"). A second critical issue with native MPI collectives is the inability to detect/correct messages at the granularity of individual processes.

RedMPI supports two solutions for collectives:

1. Built-in linear collectives that map all collectives to point-to-point messages inside of RedMPI: Our linear collectives are portable to any MPI implementation and reside entirely in the profiling layer. These collectives are not necessarily performance oriented for large scale usage as we tried to avoid direct replication of existing MPI functionality.
2. An Open MPI collectives module that acts as a wrapper to other existing collective modules, such as `linear` or the more efficient `tuned`: Our customized module resides within Open MPI and translates all point-to-point operations that would normally occur without any RedMPI instrumentation into operations that call the RedMPI library. By redirecting collective communication through RedMPI we are able to exploit the enhanced performance of the native collectives while still providing SDC protection. Our solution utilizes the Open MPI wrapper module during experiments. The same algorithmic solution can also be applied to other MPI implementations in place of the portable but less efficient RedMPI linear collectives.

The normal MPI communication stack is shown in Figure 3.8. However, when RedMPI

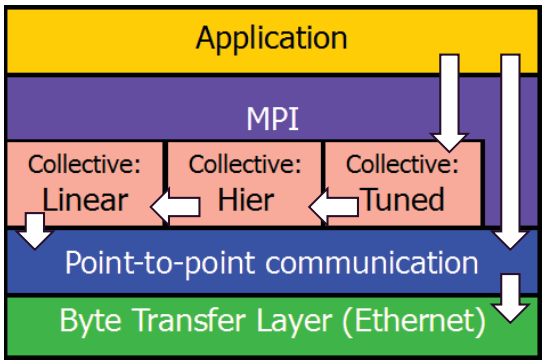


Figure 3.8 MPI stack without RedMPI's collectives module

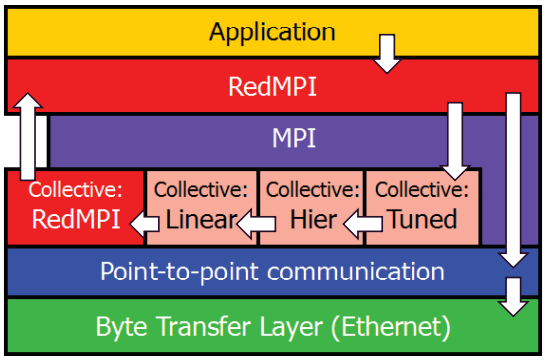


Figure 3.9 MPI stack with RedMPI's collectives module and PMPI layer

is enabled along with its counterpart collectives wrapper module, both the PMPI layer and collectives have additional instrumentation as shown in Figure 3.9. All MPI requests from the application are now directed through the PMPI RedMPI library residing between the application and Open MPI. Application point-to-point communication is interposed into its redundant/deterministic counterparts and sent to Open MPI's point-to-point communication layer. Application collectives are handled quite differently; a collective is routed through RedMPI and then to Open MPI's collective modules. Normally each Open MPI collective module would directly communication with the point-to-point communication layer, but this would circumvent RedMPI's point-to-point communication protocol for SDC protection. To solve this challenge, the RedMPI collective module intercepts point-to-point communication that is generated within Open MPI's collectives and redirects it to the RedMPI layer where it can be protected similar to normal application communication. This allows RedMPI to efficiently take advantage of Open MPI's more featureful and topologically-aware collectives functionality.

3.4 Fault Injector Design

As the research goals of this work include detecting and protecting applications from silent data corruption, an integrated fault injection tool is required to evaluate the effectiveness of RedMPI to detect and correct memory errors during execution. Additionally, the same fault injector can later be used to monitor the adverse effects of SDC on running applications when RedMPI is not actively protecting them, i.e., we specifically injector errors and allow them to propagate.

To experimentally determine the effect of corruption and verify corrective actions, our fault injector was designed to produce data corruption in a manner resembling naturally occurring faults. Namely, single bit flips undetected by ECC are of interest (e.g., within an arithmetic-logic unit of a processor) when their effects eventually propagate into a message transmission over MPI. Alternatively, also of interest is SDC due to multiple bit flips in main memory resulting in a corrupted bit pattern that ECC is unable to detect / correct.

Our fault injector, which is built to co-exist with RedMPI, specifically targets MPI message send buffers to ensure that each injection actually impacts the MPI application while simultaneously reaching message recipients. When activated, the fault injector is given a frequency of $1/x$ during launch, which is the probability that any single message may become corrupted. By using a random number generator with a state internal to RedMPI (without effect on the MPI application), the injector randomly picks messages to corrupt. Once targeted for corruption, RedMPI selects a random bit within the message and flips it prior to sending it out. RedMPI is agnostic to the data type of the message, i.e., the injector calculates the total number of bits within the entire message regardless of type or count before picking a bit to flip.

Note that not only does the fault injector flip a bit in the send buffer, but it actually modifies the application's memory directly. If the MPI application accesses the same memory again, further calculations based on that data will be invalid with a high probability of causing further divergence from non-corrupted replicas.

3.4.1 Targeted Fault Injections

Memory corruption faults may also be specifically targeted to occur within specific sets of replicas, MPI ranks, application timesteps, or frequency. For instance, as we will later investigate the effects of SDC on our experimental applications when SDC errors are allowed to propagate without any protection enabled, our targeted faults will be limited to only one set of replicas such that in dual redundancy half of the nodes may serve as "control" replicas that never experience faults while the other "experiment" replicas will receive faults. By modifying MPI

applications to report back to RedMPI whenever they reach a new timestep, we can also target faults to occur at very specific points in execution such as defining a desired timestep for an SDC injection to occur.

3.5 Experimental Framework

We deployed RedMPI on a medium sized cluster and utilized up to 96 nodes for benchmarking and testing. Each compute node consists of a 2-way SMPs with AMD Opteron 6128 (Magny-Cours) processors of 8 cores per socket (16 cores per node) with 32 GB RAM per node. Nodes are connected via 1Gbps Ethernet for user interactions and management. MPI transport is provided by a 40Gb/s InfiniBand fat tree interconnect. To maximize the compute capacity of each node, we ran up to 16 processes per node.

When launching RedMPI jobs, we map replica processes so that they do not reside on the same physical nodes. This type of mapping is preferred as a fault on a node will not affect multiple replicas of the same process simultaneously (i.e., due to localized power failures for a whole rack).

3.5.1 Time Overhead Experiments

RedMPI allows applications to utilize transparent 2x or 3x redundancy. While the physical cost of redundancy is known (2x or 3x the number of tasks), the additional cost in terms of wall-clock time should be investigated to determine what types of costs are expected, if any. To determine the cost of redundancy in terms of time, we run a variety of applications demonstrating differing scaling, processor counts, communication patterns, and problem sizes.

For our timing experiments, we solely report benchmark results for the MsgPlusHash SDC method as it provides a more efficient communication protocol than All-to-all by design. To provide meaningful metrics, each experiment assesses the run time for regular, unaltered Open MPI (referred to as *1x* in the results tables), RedMPI with dual redundancy (*2x*), and RedMPI with triple redundancy (*3x*). Note that the size reported is the size without redundancy in all experiments and results. Hence, a size 512 job in our results with triple redundancy is actually running across 1536 processors.

We assess both strong and weak scaling when evaluating overheads associated with RedMPI. For each weak scaling application, the input data size remains constant for each process no matter how many processes are run. In contrast, strong scaling applications have a constant problem size for a given class that varies the amount of input data each process receives when

Table 3.4 LAMMPS input chain.scaled

Size	1x [sec]	2x [sec]	3x [sec]	2x OV	3x OV
128	240.5	241.34	242.54	-3.8%	-3.3%
256	244.39	244.61	245.25	0.1%	0.4%
512	250.93	251.89	256.11	0.4%	2.1%

Table 3.5 LAMMPS input chute.scaled

Size	1x [sec]	2x [sec]	3x [sec]	2x OV	3x OV
128	137.50	138.38	139.01	0.6%	1.1%
256	138.26	140.43	140.00	1.6%	1.3%
512	139.19	140.22	140.67	0.7%	1.1%

Table 3.6 SWEEP3D

Size	1x [s]	2x [s]	3x [s]	2x OV	3x OV
128	390.30	389.49	393.05	-0.2%	0.7%
256	428.17	427.53	431.20	-0.1%	0.7%
512	488.08	488.93	494.09	0.2%	1.2%

Table 3.7 HPCCG (uses MPI wildcards)

Size	1x [sec]	2x [sec]	3x [sec]	2x OV	3x OV
128	99.79	99.76	125.75	0.0%	26.0%
256	99.64	128.83	131.02	29.3%	31.5%
512	126.36	146.19	152.26	15.7%	20.5%

Table 3.8 NPB CG

Size	1x [sec]	2x [sec]	3x [sec]	2x OV	3x OV
128-D	201.42	205.87	215.51	2.2%	7.0%
256-D	127.21	132.61	136.64	4.2%	7.4%
512-D	70.10	77.54	83.67	10.6%	19.4%

Table 3.9 NPB EP

Size	1x [s]	2x [s]	3x [s]	2x OV	3x OV
128-D	72.31	72.63	72.74	0.4%	0.6%
256-E	579.94	581.02	581.27	0.2%	0.2%
512-E	289.80	290.83	291.30	0.4%	0.5%

Table 3.10 NPB FT

Size	1x [sec]	2x [sec]	3x [sec]	2x OV	3x OV
32-C	117.45	117.95	118.68	0.43%	1.05%
64-C	68.82	68.62	71.77	-0.29%	4.29%
128-D	222.75	228.76	234.97	2.70%	5.49%

Table 3.11 NPB LU

Size	1x [sec]	2x [sec]	3x [sec]	2x OV	3x OV
128-D	361.78	379.90	375.44	5.0%	3.8%
256-D	179.78	191.97	195.55	6.8%	8.8%
512-D	102.90	115.07	121.01	11.8%	17.6%

Table 3.12 NPB MG

Size	1x [s]	2x [s]	3x [s]	2x OV	3x OV
128-E	339.17	340.41	429.67	0.4%	26.7%
256-E	168.56	170.68	171.48	1.3%	1.7%
512-E	66.97	68.35	69.29	2.1%	3.5%

jobs of differing sizes are run. In effect, we expect strong scaling applications to reduce the amount of data and computation required per process as the number of processors increases.

Our test suite of weak scaling applications includes LAMMPS, ASCI Sweep3D, and HPCCG. LAMMPS is a popular molecular dynamics code that we evaluate with two different problems, "chain" and "chute". Sweep3D is a neutron transport code. Finally, HPCCG is a finite elements application from the Sandia National Labs Mantevo Project. It was chosen because of its use of `MPI_ANY_SOURCE` to demonstrate RedMPI's capability to handle non-deterministic MPI operations.

The strong scaling NAS Parallel Benchmarks (NPB) are also evaluated with varying problem class sizes and number of processes. We use the NPB suite to demonstrate how varying the communication-to-computation ratio affects RedMPI in some cases.

3.6 Results

Tables 3.4-3.12 report execution time for the benchmarked applications. Every application was run with three different MPI sizes. For all of the cases except one, we conducted experiments with 128, 256, and 512 processors for the baseline. The uninstrumented (no RedMPI) version of each application is shown under the *1x* column while the *2x* and *3x* columns represent dual and triple redundancy, respectively, under RedMPI. The final two columns represent the percent overhead incurred by adding dual or triple redundancy relative to the baseline. Runs with redundancy use two or three times as many processes as the uninstrumented baseline runs. Performance is subject to cache effects when running the same application with RedMPI. This effect may vary between degrees of redundancy. This is evident for results that indicate small negative overheads (speedup under redundancy) when averaged, such as in Table 3.4.

We first analyze the runtime results of the weak scaling applications in Tables 3.4-3.7. LAMMPS with input *chain* was run with a dataset size of 32x40x20 for 512 processors and scaled down proportionally for 256 and 128 processors, which explains the relatively consistent runtimes. Likewise, LAMMPS with input *chute* was given a dataset size of 320x88 for 512 processors and also scaled proportionally. Sweep3D had an input size of 320x40x200 and HPCCG had an input size of 400x100x100.

These applications performed very well with RedMPI, i.e., in most cases the RedMPI overhead was not perceptible due to a well balanced communication-to-computation ratio that weak scaling allowed us to retain despite increasing the number of processors as we scaled up the benchmarks with RedMPI.

To demonstrate the effectiveness of RedMPI's wildcard support, HPCCG was chosen as it

makes use of `MPI_ANY_SOURCE` receives. Since RedMPI requires special handling for wildcards, the overheads incurred may vary based on how long it takes the replicas to receive an envelope message that resolves the wildcard. When wildcard resolution is completed quickly, very little performance penalty is seen as in the *2x* results with size 128 (see Table 3.7). Conversely, when wildcard resolution takes a relatively long time, then RedMPI forces MPI to receive all messages in an unexpected queue. Delaying message reception can potentially degrade performance, and MPI wildcards are recommended to be avoided when possible.

The remaining NPB benchmarks are strong scaling applications subjected to input classes C, D, and E, where E is the largest size. For each class, data is equally distributed across all MPI processes. For example, the CG and LU benchmarks were both run with the same class size *D* for 128, 256, and 512 size jobs. We can see that as the number of processes increases, the baseline runtime decreases since there is less computation per process to perform. In turn, as the computation is decreased per process the amount of communication incurred does, in fact, increase. Tables 3.8 and 3.11 clearly demonstrate how the overhead of RedMPI increases as the per-process communication overshadows the per-process computation. Hence, to keep RedMPI overheads reasonable, it is important to choose input classes such that the ratio of communication-to-computation is balanced, e.g., as seen for input sizes for EP and FT. For FT, we were unable to run class E with size 256 and 512 because our experimental setup did not have enough memory available to hold class E. Thus, we chose to run FT with smaller class sizes and report smaller runs.

Overall, RedMPI's runtime overheads are modest for well-behaved applications that can be scaled to exhibit a fair communication-to-computation ratio.

3.7 Fault Injection Studies

RedMPI's fault injector provides two key opportunities for specifically analyzing silent data corruption faults within the scope of running MPI applications. We will use RedMPI to answer these questions:

(1) **Propagation:** Does SDC affect applications messages and correctness when no protection mechanisms (such as RedMPI's voting) are available? How quickly do SDC injections propagate to other processes via communication? Do corrupted processes further disrupt other processes in a cascading manner by sending invalid, divergent MPI messages as compared to the correct execution of a job?

(2) **Protection:** When utilizing triple redundancy with RedMPI, are SDCs successfully detected and corrected? Do applications still complete with correct answers even in the face of

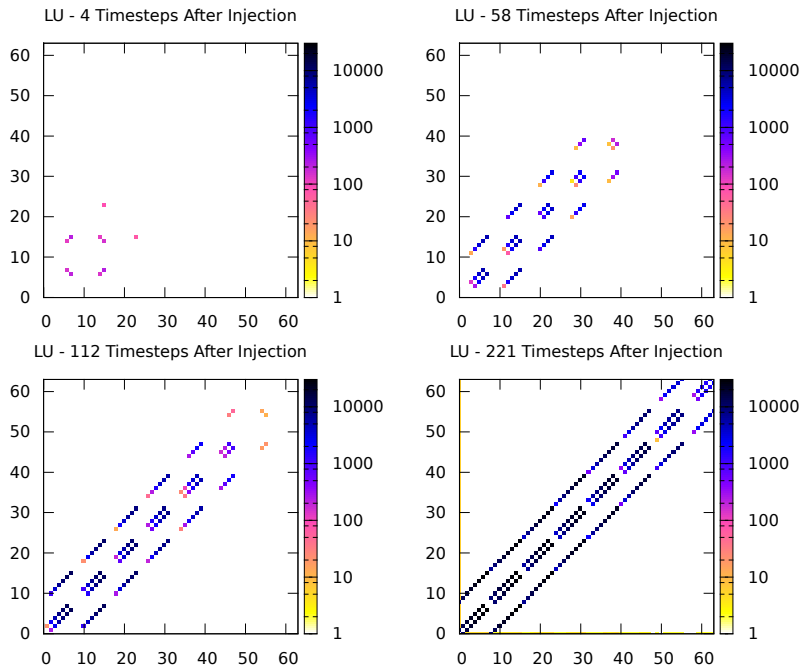


Figure 3.10 NPB LU Corrupted Communication Patterns

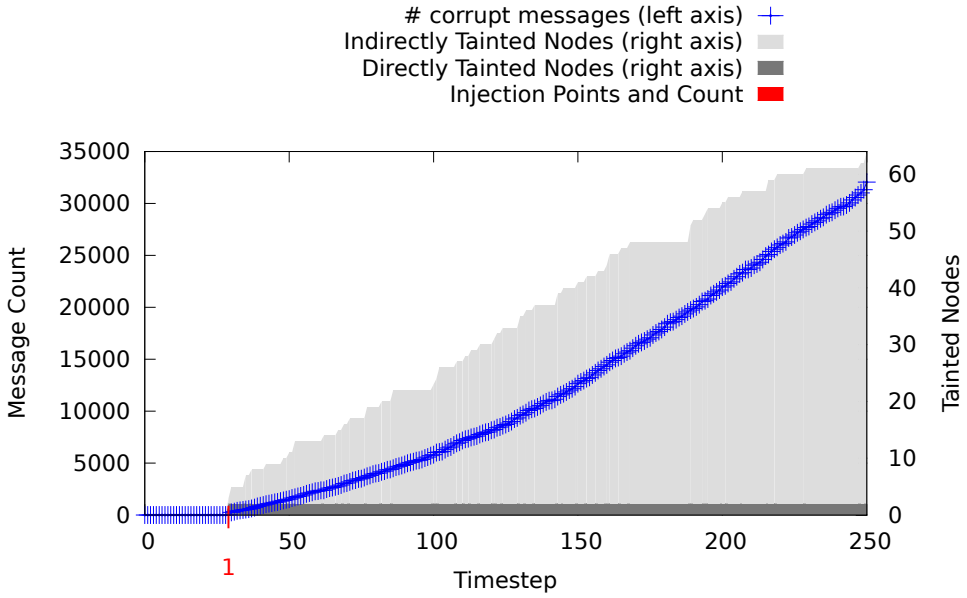


Figure 3.11 NPB LU Overview of Corrupt Nodes and Messages

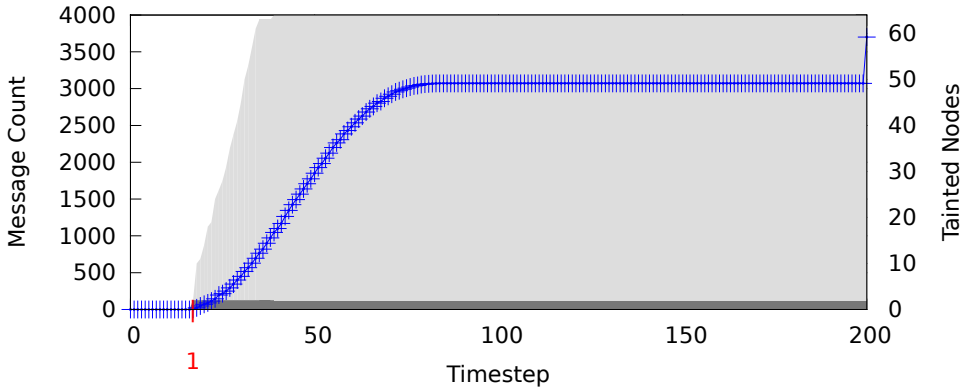


Figure 3.12 NPB BT Overview of Corrupt Nodes and Messages

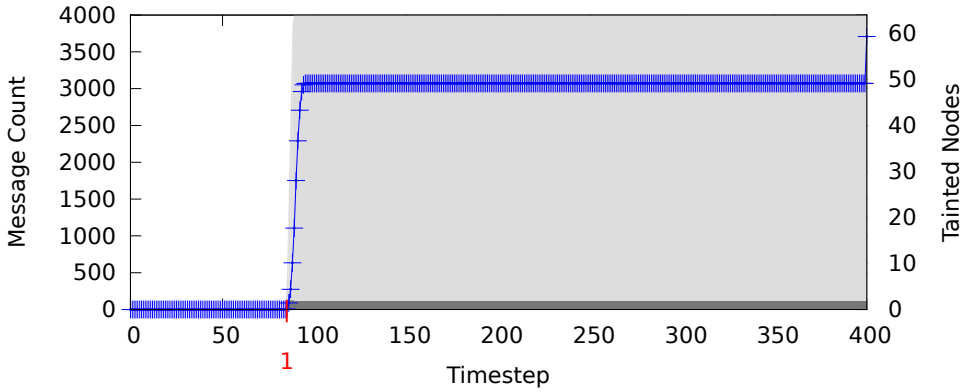


Figure 3.13 NPB SP Overview of Corrupt Nodes and Messages

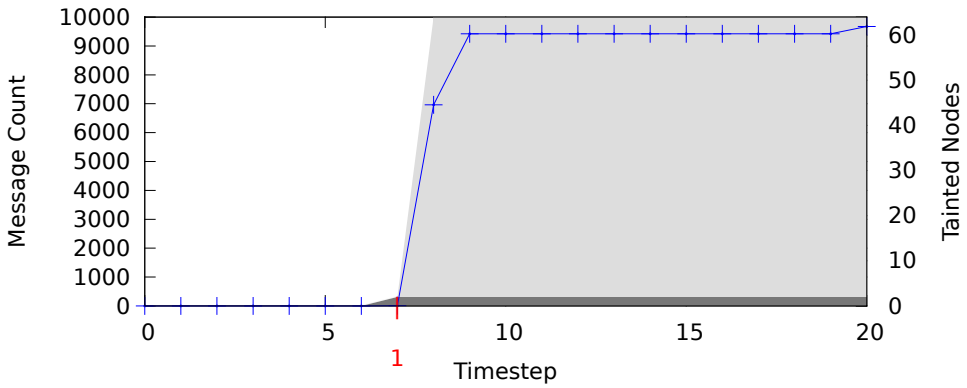


Figure 3.14 NPB MG Overview of Corrupt Nodes and Messages

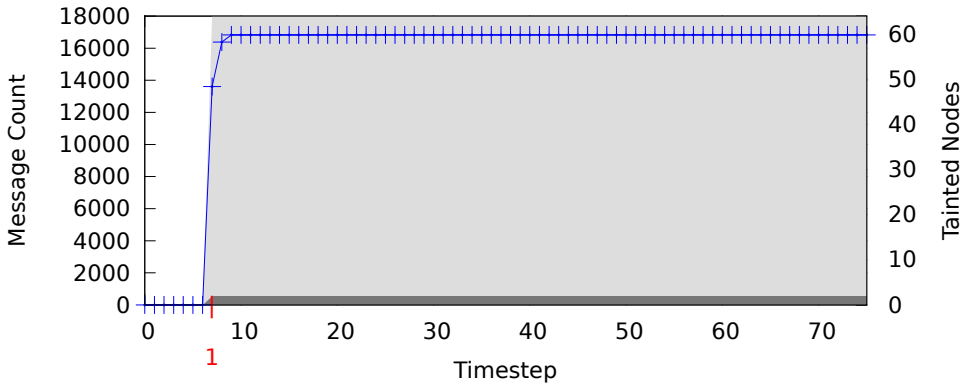


Figure 3.15 NPB CG Overview of Corrupt Nodes and Messages

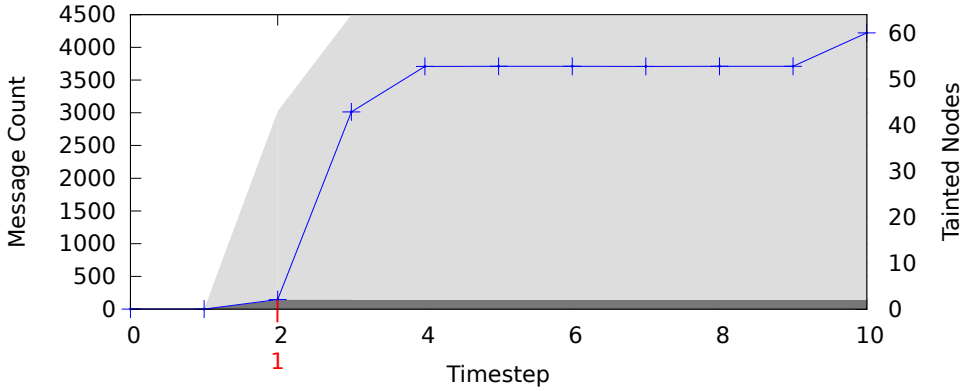


Figure 3.16 ASCI Sweep3D Overview of Corrupt Nodes and Messages

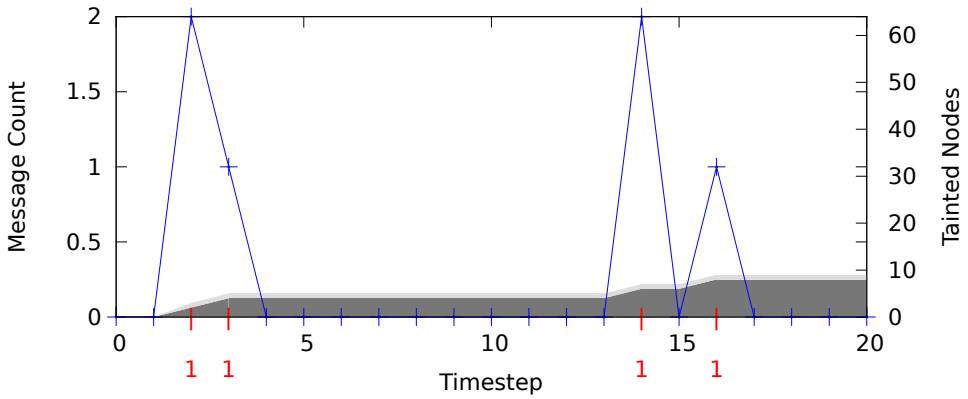


Figure 3.17 NPB FT Overview of Corrupt Nodes and Messages

SDC injections?

3.7.1 SDC Propagation Study

Our first study investigates whether leaving message data unprotected in the face of SDCs does in fact lead to incorrect results. This happens when a single SDC injection in one process will later spread to other nodes causing an overwhelming cascade of invalid data.

As described in Section 3.4.1, we run RedMPI with dual redundancy and assume two sets of replicas. The first set of replicas is a control set and will not receive any SDC injections. The control set will execute normally and should produce correct results upon completion. Our second set of replicas is the test set, which becomes the victim when SDC faults are injected. *Further, during these experiments we also purposefully disable RedMPI's corrective capabilities.* RedMPI still detects divergent messages between the control and test set replicas, but allows application progress to continue. RedMPI tracks live statistics on applications running in this environment such as:

- which processes receive SDC injections;
- which processes send bad messages and where;
- which processes receive corrupt messages and how that corruption further spreads to and from nodes that were indirectly tainted by a bad message; and
- aggregate data at the granularity of a single application-defined timestep.

This type of reporting is made possible by redundancy and is considered a new technique for live application analysis and correctness since we do not need to actually log data for later viewing; this is advantageous for long-running applications or when bandwidth is high and logging to disk for offline analysis is undesirable.

Figures 3.11-3.17 show how SDC injection(s) spread causing bad messages and cascading tainting of other processes via reception of corrupt messages. The x-axis of these graphs denotes progress in the form of application-reported timesteps. The blue line correlates with the left vertical axis, which denotes the number of bad messages that were cumulatively received by all MPI processes in any given timestep. The gray filled areas match to the right axis and denote the number of MPI processes that receive an SDC injection themselves (direct tainting) or become corrupt indirectly (indirect tainting/light gray) by receiving a corrupt message from a previously corrupted sender themselves. Combined, both of the gray filled areas indicate how many of the MPI processes operating in a single job have become corrupt over time.

For all of the experiments except NPB FT (Figure 3.17), only a single SDC injection was performed. NPB FT was subjected to 4 SDC injections. All SDC injections were randomly inserted using the aforementioned RedMPI fault injector. In all cases we ran the experiments with 64 MPI processes under dual redundancy (64 control set processes + 64 test set processes).

These 7 graphs indicate three disjoint trends in response to SDC injections. First, the *progressive* trend is characterized by Figures 3.11 and 3.12 as a single injection that does not spread immediately. Instead, *progressive* applications often communicate with their grid neighbors resulting in a sphere of corruption that grows outward. For example, Figure 3.10 shows a heatmap indicating nodes (both on the x-and y-axes) that communicate tainted messages with one another due to an SDC. The figures depict timesteps 4 (one timestep after first injection), 58 (more indirectly tainted messages), 112 (most nodes tainted), and 221 (all nodes tainted) after the SDC injection. As the application progresses, the final heatmap shown is actually a heatmap of normal communication. This indicates that eventually virtually all communication has succumbed to invalid data due to a single SDC injection.

The next trend we identify is the *immediate/explosion* case. Of the experiments reported, Figures 3.13-3.16 fall in this category. For these applications we noticed that a single injection resulted in corruption that spread across all nodes usually within two timesteps of the initial SDC. The most common reason for the *immediate* trend is the use of collectives or some communication pattern that tends to exchange messages between all nodes in a short period of time. Heatmaps of this trend are not provided as they essentially mirror the full communication pattern of the application almost immediately after the injection occurs.

Our third identified trend is the *localized* case. In Figure 3.17, NPB FT is one such application where injections result in just a few invalid messages. This occurs when the corrupted data is neither reused nor retransmitted, which in turn keeps the sphere of corruption relatively isolated to the processes that were tainted by a direct SDC injection. In this experiment we targeted FT with multiple injections to demonstrate how applications that fit the *localized* trend typically do not result in large aggregates of tainted messages or nodes. Although we did not receive a high degree of corrupt messages or spreading for FT, it is important to note that all benchmarks (including NPB FT) failed to pass their internal verification or complete with correct results that matched their "control" counterparts.

In summary, by observing that just a single SDC injection can induce a profound effect on all communicating processes, we conclude that protecting applications at the MPI message level is an appropriate method to detect, isolate, and prevent further corruption. Had RedMPI's protection not been purposefully disabled for this study, then all of the SDC injections would have been isolated from spreading and no bad messages could have been received by other

processes.

3.7.2 RedMPI SDC Protection Study

To gauge SDC sustainability when RedMPI is active with redundancy, we inject faults into the running benchmarks to determine if the faults are detected and if correction succeeds. Additionally, we experimentally determine if the fault corrections allow the benchmarks to complete their self-verification process successfully.

Next, we analyze the effectiveness of SDC detection and correction protocols. We ran the fault injector with a corruption frequency of $1/5,000,000$ messages to ensure a relatively high likelihood for an injection while running the CG benchmark with 64 processes (virtual ranks) and a replication degree of three (192 physical processes). Note that we restricted the intentional corruption injections to only occur on replicas with a replica rank of zero to control the experiment. This ensures that at least two of the three replicas do not receive an injection in all cases. Thus, voting results in a valid outcome so that invalid messages can be corrected. During ten experiments with a frequency of $1/5,000,000$, we encountered one occasion with two injections, four occasions with a single injection, and five occasions without injections. In every run except one, the corruption resulted in a single bad message that was successfully detected and corrected by the receiving replicas. In one event, a single injection cascaded resulting in 6,242 bad messages originating from the corrupted sender. Nevertheless, the receiving replicas were able to correct the messages as they arrived. Eventually, the corrupted node ceased to send corrupted messages as the application finished traversing data structures until the fault was no longer touched. In these experiments, the applications progressed until completion and successfully passed their built-in verification at the end of processing.

Following that experiment, we performed injections with a frequency of $1/2,500,000$ in another ten runs that were not limited to a single group of replicas. By doubling the odds for an injection and removing the process selection restriction, we detected a significantly larger number of faults. On average, we received 2.5 injections and several thousand invalid messages per run as a result. Nonetheless, RedMPI carried all but two runs to a successful completion with verification. Of the two runs that failed, we observed that when two of the three replicas simultaneously transmitted a corrupt message over RedMPI, it was detected. The voting process is then forced to fail. In this case, RedMPI aborts because voting becomes impossible with three unique messages and hashes. Statistically, as more processes are added to a job, the likelihood of two processes with the same virtual rank becoming corrupt decreases. Therefore, observing results similar to this particular controlled experiment decreases as job sizes and replication

degrees increase. Nonetheless, the longer a process remains in an invalid state (i.e., sending bad messages), the longer the correction features of RedMPI are impaired. However, it is important to note that RedMPI still forces a job to abort if it does detect corruption across two or more nodes while utilizing triple redundancy.

Performance of SDC correction has proved to be quite efficient. During SDC correction overhead experiments, we discovered that with as few as three injections we were able to produce nearly 100,000 invalid messages from corrupted senders. The receiving replicas were able to successfully detect and correct each invalid message while effectively generating no perceived overhead. In fact, while running 20 experimental iterations to gauge the protocol overhead of correcting MsgPlusHash messages during injection, the runtime was 0.31 seconds less than the original experiment runtime without fault injection on average.

Realistically, we do not expect to encounter such a high number of naturally occurring SDCs for a small environment such as our benchmarking cluster. The actual overhead incurred due to SDC correction is a function of the number of invalid messages, and the number of invalid messages is highly dependent on the data reuse patterns of an MPI application.

As empirical evidence of the success of RedMPI, we discovered that RedMPI was detecting and correcting MPI transmission errors during our timing benchmarks even though we had not activated our fault injection module. While investigating, we learned that RedMPI had been properly detecting and correcting faulty memory that was later confirmed to be producing errors, which could not be corrected by ECC alone. These problems were occasionally (but not always) visible through the Linux EDAC monitoring module of the memory controller. Interestingly, EDAC was unable to consistently detect all of the errors that RedMPI detected. Using RedMPI, we discovered which of the 1536 compute cores in the experiment was faulting and were able to reproduce similar experiments that consistently produced faulty MPI messages on this node before removing it from the production system. Without RedMPI, this failing hardware may have gone unnoticed for some time.

3.8 Related Work

3.8.1 Modular Redundancy

In general, modular redundancy (MR) transparently masks any errors without the need for rollback recovery. In case of SDC, detection is achieved through comparison and recovery is performed by majority voting. MR has been used in information technology, aerospace and command & control [93]. Recent software-only approaches [50, 92] focused on thread-level,

process-level and state-machine replication to eliminate the need for expensive hardware. The sphere of replication [72] concept describes the logical boundary of redundancy for a replicated system. Components within such a sphere are protected; those outside are not. Data entering it (input) is replicated, while data leaving (output) is compared. This work relies on this concept directly in the all-to-all and indirectly in the MsgPlusHash protocol.

3.8.2 Checkpoint/Restart vs. Redundancy in HPC

Recent work [40] studied the impact of deploying redundancy in HPC systems. Redundancy can significantly increase system availability and correspondingly lower the needed component reliability. Redundancy applied to a single computer decreases the MTTF of each replica by a factor of 100-1,000 for dual redundancy and by 1,000-10,000 for triple redundancy without lowering overall system MTTF. If a failed replica is recovered through rebooting or replacing with a hot spare, replica node MTTF can be lowered by a factor of 1,000-10,000 for dual and by 10,000-100,000 for triple redundancy. Redundancy applied to each compute node in a HPC system with 1 million nodes allows lowering the node rating from 7 to 3 nines with 2 million dual-redundant nodes and to 2 nines with 3 million triple-redundant nodes. Redundancy essentially offers a trade-off between component quality and quantity. Our work permits this trade-off.

Another compelling study [43] uses an empirical assessment of how redundant computing improves time to solution. The simulation-driven study looked at a realistic scenario with a weak-scaling application that needs 168 hours to complete, a per-node MTTF of five years, a fixed five minutes to write out a checkpoint, and a fixed ten-minute time to restart. Checkpointing is performed at an optimal interval. The results show that at 200k nodes, an application spends eight times the time required to perform the work, reducing the throughput of such a machine to just over 10% compared to a fault-free environment. In contrast, using 400k nodes and dual redundancy, the elapsed wall clock time is 1/8 of that for the 200k-node non-redundant case. The throughput of the 400k-node system is four times better with redundant computing than the non-redundant 200k-node system. The prototype detailed in this work is a step toward achieving this capability.

Elliott *et al.* [37] combine partial redundancy with checkpointing in an analytic model and experimentally validate it. The work shows that exascale core counts of one million, triple redundancy is projected to have the lowest cost (lower than dual redundancy) so that the cost of SDC protection essentially becomes free in terms of resource and power requirements (as triple redundancy is the lowest cost option anyhow). This underscores the importance of investigating

SDC protection within redundant MPI.

3.8.3 Redundant Execution of MPI-based Applications

rMPI [19] is a prototype for redundant execution of MPI applications that uses the MPI profiling interface (PMPI) for interpositioning. rMPI maintains redundant nodes and and each replica duplicates the work of its partners. In case of a node failure, the redundant node continues without interruption. The application fails only when two corresponding replicas fail. The reported impact on actual applications is for the most part negligible [43]. Our work differentiates itself from rMPI in that it takes the research in a new direction using replication to detect and correct silent errors. The protocols necessary for this detection and correction and the performance impacts are quite different.

The modular-redundant Message Passing Interface (MR-MPI) [39] is a similar solution for transparent HPC redundancy via PMPI interpositioning. In MR-MPI, a redundantly executed application runs with $r * m$ native MPI processes, where r is the number of MPI ranks visible to the application and m is the replication degree. Messages are replicated between redundant nodes. The results show the negative impact of the $O(m^2)$ messages between replicas. For low-level, point-to-point benchmarks, the impact can be as high as the replication degree. In realistic scenarios, the overhead can be 0% for embarrassingly parallel or up to 70-90% for communication-intensive applications in a dual-redundant configuration. RedMPI extends beyond the capabilities of MR-MPI by protecting against SDC, lowering the replication overhead, and advancing internal communication protocols (MsgPlusHash). Unlike MR-MPI, RedMPI expands upon MR-MPI's linear collectives by providing internal MPI modifications that exploit native collectives performance.

VolpexMPI [65] is an MPI library implemented from scratch that offers redundancy internally and uses a polling mechanism by the receiver of point-to-point messages to avoid message replication. If a polled sender (of a replicated sender-receiver pair) fails to respond, a different sender (replica of the original sender) is chosen until the receive is successful. Messages are matched with a logical timestamp to allow for late message retrieval. VolpexMPI achieves close to 80% of Open MPI's point-to-point message bandwidth, while latency of small messages increases from 0.5ms to 1.8ms. Using the NAS Parallel Benchmark suite, there is no noticeable overhead for BT and EP for 8 and 16 processes. SP shows a significant overhead of 45% for 16 processes. The overhead for CG, FT and IS is considerably higher as these benchmarks are communication heavy. VolpexMPI does not provide SDC protection, however, it provides good performance as replication protocols are part of the low-level communication inside the MPI

library.

Other parallel debugging tools that utilize redundancy exist such as MPI Echo [86][26].

3.9 Conclusion

Redundant computing is one approach to detect SDC. This study assesses the feasibility and effectiveness of SDC detection and correction at the MPI layer. We presented two consistency protocols, explored the unique challenges in creating a deterministic MPI environment for replication purposes, investigated the effects of fault injection into our framework, and analyzed the costs of performing SDC protection via redundancy.

This study develops a novel, efficient SDC detection and correction protocol (MsgPlusHash) with overheads ranging from 0% up to 30% for dual or triple redundancy depending on the number of messages sent by the application and the communication patterns. In particular, overheads do not change significantly for weak scaling applications as the number of processes is varied. These modest overhead ranges indicate the potential of RedMPI to protect against SDC for large-scale runs.

Our protocol detected and corrected injected faults for processes that continued to completion even when these faults resulted in many thousands of corrupted messages from a sender that experienced one or more SDC faults. In our controlled experiments, injected faults that were targeted to a single set of replicas were successfully isolated from spreading by fixing corrupted messages and allowing the applications to complete without incident. Further, when we injected faults into two or more replicas (of the same rank), RedMPI detected the corruption and was able to abort the application thus preventing invalid results from being reported.

In summary, RedMPI was successful in preventing invalid data from propagating or being transmitted without detection even under extreme scenarios. Our experiments showed profound effects from applications that experience even a single soft error without any form of protection. Without RedMPI, just one injected SDC was observed to quickly spread to other processes and messages, causing the majority of message data to become corrupt, which consistently lead to invalid results at a global scale.

Empirically, RedMPI not only performed exactly as expected in our controlled experiments, but it also pinpointed previously unknown hardware faults on our own experimental cluster nodes that had not been detected until RedMPI alerted us to SDCs occurring on unaltered MPI jobs.

While the cost of double/triple redundancy appears high in terms of power, analytic models show that for large core counts redundancy actually improves job throughput. As both the

likelihood of node failure and silent data corruption increases as we scale up HPC systems, the importance of protecting data becomes obvious and available at a low cost when redundancy is already in place to ensure high throughput of mission-critical/high-consequence large scale applications.

CHAPTER

4

FLIPSPHERE: A SOFTWARE BASED ERROR DETECTION AND CORRECTION LIBRARY

4.1 Introduction

With the increased density and power concerns in modern computing chips, components are shrinking, heat is increasing, and hardware sensitivity to outside events is growing. These variables, combined with the extreme number of components expected to make their way into computing centers as our computational demands expand, are posing a significant challenge to the design and implementation of future extreme-scale systems. Of particular interest are soft errors in memory (spontaneous bit flips) that manifest themselves as silent data corruption (SDC). SDC is of great importance to the reliability of these systems due to its ability to render results invalid in scientific applications without detection of such corruption.

Silent data corruption can occur in many components of a computer system including the processor, cache, and memory due to radiation, faulty hardware, and/or lower hardware tolerances. While cosmic particles are one source of concern, another growing issue resides within

the circuits themselves, due to miniaturization of components. As components shrink, heat becomes a design concern, which in turn leads to lower voltages in order to sustain the growing chip density. Lower component voltages result in a lower safety threshold for the bits that they contain, which increases the likelihood of an SDC occurring. Further, as densities continue to grow, any event that upsets chips (i.e., radiation) is more likely to flip bits.

Current systems use memory with hardware-based ECC that is capable of correcting single bit errors and detecting double bit errors [23] within a region of memory (typically a cache line). Errors in current systems that result in three or more bit flips will produce undefined results including silent data corruption, which may produce invalid results without warning. While the frequency of single and double bit errors is known (8% of systems will incur correctable errors while 2%-4% of will incur uncorrectable errors [89]), the frequency of higher bit errors is still an open research question, and GPUs are known to be more prone to them [32]. While chipkill can detect and correct more errors than ECC [95], it cannot do so for all memory errors and SDCs outside DRAM may not be detected at all [48]. The overall occurrence of bit flips is expected to increase as chip densities increase and feature sizes decrease.

While hardware vendors will address silent data corruption for the consumer and enterprise markets via more sophisticated detection and correction hardware logic, this will come at a price of increased memory overheads, latencies, and power. More importantly, it is not clear that these vendor solutions will be sufficient given the unprecedented scale and failure rates of future extreme-scale systems [42].

To address this SDC issue, we introduce FlipSphere, a software-based, generic memory protection library that increases the resilience of applications by protecting data at the page level using an application transparent, tunable, and on-demand verification system with the following contributions:

- It introduces FlipSphere, which provides transparent protection against SDC for all applications without any program modifications.
- This is, to the best of our knowledge, the first published class of software-based resilience that harnesses hardware accelerators such as Xeon Phis to harden applications. Further, this work evaluates the feasibility of full or shared co-execution for resilience on accelerators.
- We introduce two levels of protection and analyze their costs independently: SDC detection and SDC detection plus automatic correction.
- FlipSphere provides a view into an application's memory access patterns as part of its functionality. In fact, using this feature, its operation is agnostic by tuning to the data

access patterns of an application directly.

- It is extensible with new features, e.g., custom hashing algorithms or further strengthened ECC techniques.

4.2 Related Work

FlipSphere is based on the software of LIBSDC [45]. LIBSDC provides software-based page-level protection by tracking page accesses using the MMU and removing page permissions of a less recently used page every time a new page is accessed. LIBSDC may result in application slowdowns ranging from as little as 1x to as high as 20x due to a combination of its page locking mechanisms, dependence on the application tracing API `ptrace` and hashing methods used. FlipSphere differentiates itself from LIBSDC by providing a full software-based ECC implementation via hardware accelerators, protection of both application heap and BSS, non-blocking performant timer-based relocking instead of LRU, MPI and network device interception to provide protection for memory incurring DMA, and kernel function call tracking to remove dependence on `ptrace`. FlipSphere constitutes a low-overhead, feasible option for software-based SDC protection that uses hardware accelerators.

Research involving GPGPUs to generate ECC codes of GPU memory has been investigated [69]. That work focused on generating and storing ECC codes for GPU memory that resides in GPU global memory only. We differentiate by generating software ECC for data the resides on the host CPU by using hardware accelerators such as a Xeon Phi or a GPU for the computation of the ECC codes.

One method to address silent data corruption is from the field of algorithmic fault tolerance where researchers have proposed methods to protect matrices from SDCs that corrupt elements within a dense matrix [58]. While these methods are effective for some matrix operations, fault tolerant algorithms are incredibly difficult to design for many arbitrary data structures or operations on that data [25]. Worse, this method does not provide comprehensive coverage to the entire application, which leaves anything outside of the algorithm, such as other data and instructions, entirely vulnerable to SDCs. Distributed systems approaches and eventually consistent storage abstractions provide a software solution to harden limited state within protocols [27, 15] or checking code for specific data structures [18], but they do not generalize to large-scale numerical data (matrices) in HPC.

Similar to FlipSphere, another approach uses software-implemented error correction using background scrubbing with software ECC to periodically validate memory and correct errors if

possible [91]. While this approach is transparent to the application, FlipSphere differentiates itself by providing on-demand page-level checking based on the application's data access patterns. In a HPC environment, software-based background scrubbing would likely consume considerable memory bandwidth and generate substantial application jitter [44]. FlipSphere additionally distinguishes itself from scrubbing by providing software ECC when the hardware lacks it or by providing an additional layer of protection for systems that utilize ECC protected memory. For some types of HPC workloads the necessity for correct output demands enhanced protection beyond today's hardware ECC capabilities alone.

While techniques described thus far do not require changes to the application, the next techniques described require modifications of the executable through source-to-source transformations or modifications to the executable with duplicate instructions or control flow verification.

Source-to-source transformation techniques [82] have been investigated that generate a redundant copy of all variables and code at the source level. Throughout the transformed source code are additional conditional checks that verify agreement in the redundant variables after a set of redundant calculations is performed. If during execution redundant variables do not agree, then the application aborts. Unfortunately, this technique is unable to handle pointers, only supports basic data-types and arrays, and doubles the required memory. Due to a high number of conditional jumps for consistency checking, the efficiency of pipelining and speculative execution suffers. FlipSphere differs from this work by requiring source modifications, lowering memory requirement overheads substantially, supporting any type of code (pointers, data-types, etc.), and being able to protect any region of memory at runtime.

Duplicated instructions is another proposed technique to increase SDC resilience in software. Error detection by duplicated instructions (EDDI) [74] duplicates instructions and memory in the compiled form of an application in a manner similar to the source-to-source transformations, but achieves more support for programming constructs at the cost of platform dependence. Unlike the source-to-source transformations, EDDI compiles applications to binary form, redundantly executes all calculations, ensures separation between calculations by using differing memory addresses and differing registers, and attempts to order instructions to exploit super-scalar processor capabilities. During execution, the results of calculations are compared between their redundant variable copies, but as a result, available memory is halved and register pressure is doubled. FlipSphere differentiates itself from this work by being platform-independent, not requiring redundant execution or program modifications, and protecting instruction memory without the need for complex control-flow checks.

Extensions to EDDI have been proposed [84] that achieve better efficiency by assuming reliable caches and memory, but still require redundant registers and instructions. Their experiments

showed an average normalized execution time of 1.41, but without protection for system memory. The similarity to EDDI may indicate that even without protecting memory there is a substantial overhead due to register pressure, additional instructions, and highly frequent conditionals that come with duplicating instructions and registers. This work also showed that compiled executables with the added fault tolerance were 2.40x larger than the original unaltered executables.

Control-flow checking is another area of research that attempts to detect the effects of SDCs in applications [75]. Unfortunately, control-flow integrity (CFI) verification does not necessarily protect against SDCs that only alter data without affecting the execution path of an application. Additionally, CFI incurs its own penalties in terms of increased binary size as well as runtime overheads without the coverage provided by protecting all of application memory.

4.3 Design

This work introduces FlipSphere, a transparent, application-agnostic library that is capable of detecting and optionally correcting silent data corruption (SDC) in the memory of an executing process. FlipSphere works alongside traditional hardware ECC as an additional layer of defense against SDCs that exceed the limitations of hardware protection or it can independently provide protection on systems that lack hardware ECC altogether. Our SDC detection techniques are based on the ability to verify the correctness of memory residing in RAM that has not been accessed (relatively) recently. Specifically, we are most interested in ensuring that memory has not become perturbed by an SDC event before the application is allowed to read a region of memory. When FlipSphere guards memory accesses and verifies them before use, it is possible to ensure that memory read by a process has not been changed due to any external events, such as memory corruption.[1]

Error Model: FlipSphere currently protects an application's heap, BSS, and data sections but not that of the operating system. Although the FlipSphere methodology applies to all process sections, we protect neither the stack nor code (instructions) in the implementation. Within the protected regions, we assume a uniform error distribution.

4.3.1 Memory Access Tracking

The underlying tenant of FlipSphere is that an application's working-set of pages, which is the subset of virtual memory pages read or written to over a given period of time, is considerably

[1] An external event is defined as any occurrence that involves direct memory modification without FlipSphere's knowledge. For instance, FlipSphere must be made aware of DMA transfers before/after they occur, as they may bypass the CPU and thus FlipSphere.

smaller than the entire set of pages an application's data resides in. Further, FlipSphere assumes that SDC is likely to occur in memory that has not been accessed recently. We assume that the SDCs we are guarding against are more likely to occur in memory not recently accessed by the processor. In particular, FlipSphere is designed to protect memory in DRAM, as opposed to processor faults.

Next, we define two terms that relate to an application's data within FlipSphere: (1) *locked* memory, which has not been accessed recently, resides in RAM, and must be verified by FlipSphere before the application may read it, and (2) *unlocked* memory, which has been recently accessed and has undergone verification by FlipSphere to ensure its integrity before being read/used by the application. While the verification process will be explained later, the transition between *locked* and *unlocked* must be considered first. Let us assume that when an application is launched with 3 pages of data, all pages of its memory start in a *locked* state. Let us also assume an access pattern of P1, P3, P3, P2. Upon the first memory access to read memory in P1, FlipSphere will interrupt execution to verify P1 since it is *locked*. FlipSphere verifies that P1 is free from SDC (correct) via verification, marks it as *unlocked*, and returns control to the application. Next, P3 is accessed by the application, but since it is *locked*, the same aforementioned process occurs for P3 before control is returned to the application. When P3 is accessed a second time, no interruption occurs since it has already been marked as *unlocked* to indicate it was verified. Upon access of P2, the verification and transition to *unlocked* occurs again. At this point, all memory is marked as *unlocked*, and no further interruptions from FlipSphere occur since all pages are in this state.

FlipSphere strives to both minimize and ensure that the only pages allowed in an *unlocked* state are pages that have been recently accessed. A simple way to conceptualize this is to define a fixed-length least recently used (LRU) table of pages. Whenever a page is accessed, it will be added to the LRU table, and once the table is full, the oldest page entry in the table would be evicted to make room for the most recent page accessed. The eviction would require that FlipSphere transition the evicted page from an *unlocked* to a *locked* state, and simultaneously store some correctness metadata on the page so that it may later verify the page when it is accessed again in the future. As maintaining a LRU table and performing high frequency evictions/lookups would be prohibitively expensive in an HPC context, FlipSphere trades off the accuracy of an LRU table for the performance of a timer-based approach, wherein a *relock interval* is used to periodically transition all *unlocked* pages into a *locked* state. This approach allows for an unbounded number of pages to be accessed between each *relock interval* before all become *relocked*; however, in practice we tune the *relock interval* to a value that correlates to a target percent of memory being in the *unlocked* state. For example, in some applications, a

Listing 4.1 "bench" example code

```
int* mem = malloc(sizeof(int) * outerloopmax * innerloopmax);
for(i=0..outerloopmax)
  for(BusyWorkLoop=0..100)
    for(j=0..innerloopmax)
      mem[i][j] += 1; //touch memory
```

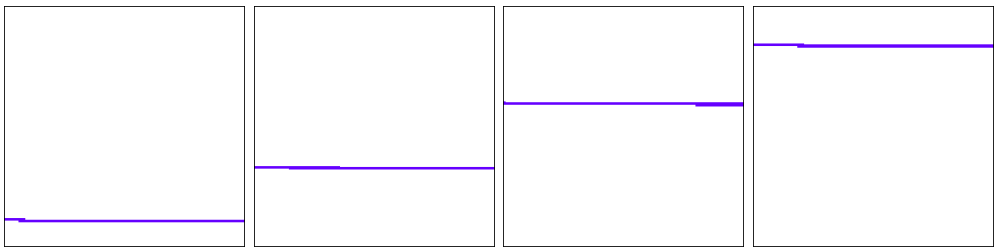
Listing 4.2 "bench-rand" example code

```
int* mem = malloc(sizeof(int) * outerloopmax * innerloopmax);
for(i=0..outerloopmax)
  v = rand() % outerloopmax;
  for(BusyWorkLoop=0..100)
    for(j=0..innerloopmax)
      mem[v][j] += 1; //touch memory
```

relock interval of 0.1 seconds may correlate to no more than 10% of memory being *unlocked* on average.

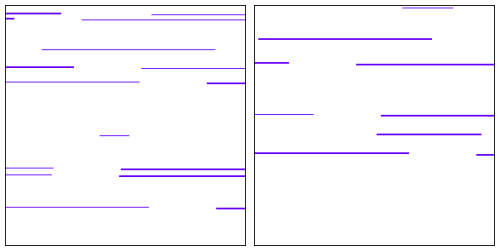
Listing 4.1 demonstrates a sample application, henceforth "bench", that allocates a large array of memory and then proceeds to touch all memory in several iterations over the i loop. Using bench with FlipSphere, we can visualize the memory access patterns by utilizing FlipSphere's page access tracking functionality.

In Figure 4.1 we see four individual snapshots of the memory access pattern of bench during execution. To generate these graphs, we ran FlipSphere with bench while setting its *relock*



(a) f 10/83 (b) f 30/83 (c) f 50/83 (d) f 70/83

Figure 4.1 Memory access patterns of bench: *relock interval* 0.2 secs, frames 10,30,50,70 out of 83, blue=pages accessed in each frame, lower left corner=memory offset 0 with increasing addresses left to right and top to bottom.



(a) f 12/93 (b) f 42/93

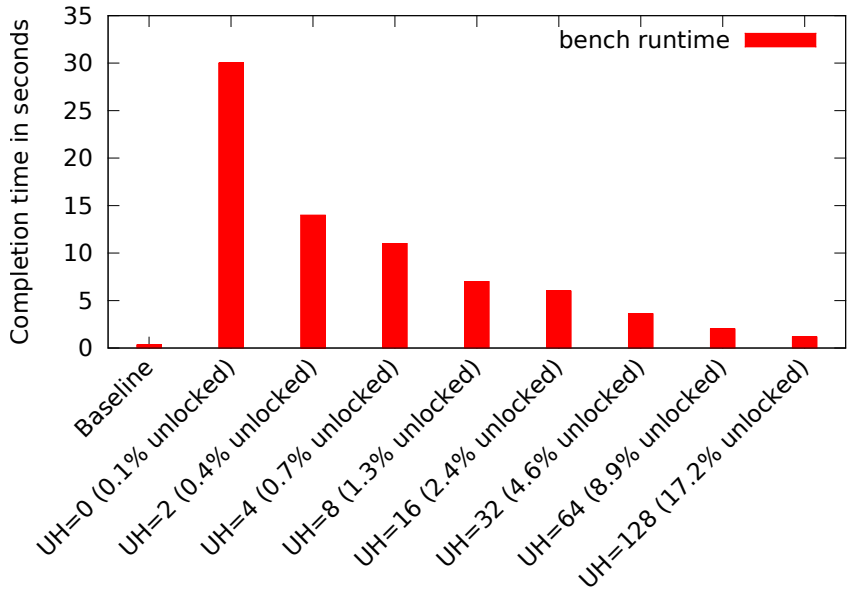
Figure 4.2 Memory access patterns of bench-rand: *relock interval*=0.2 sec.

interval to 0.2 seconds. At each interval, FlipSphere captured the status of every page's status. For instance, from Figures 4.1a to 4.1b we see that the blue *unlocked* pages have moved from lower addresses to higher addresses, just as expected from Listing 4.1.

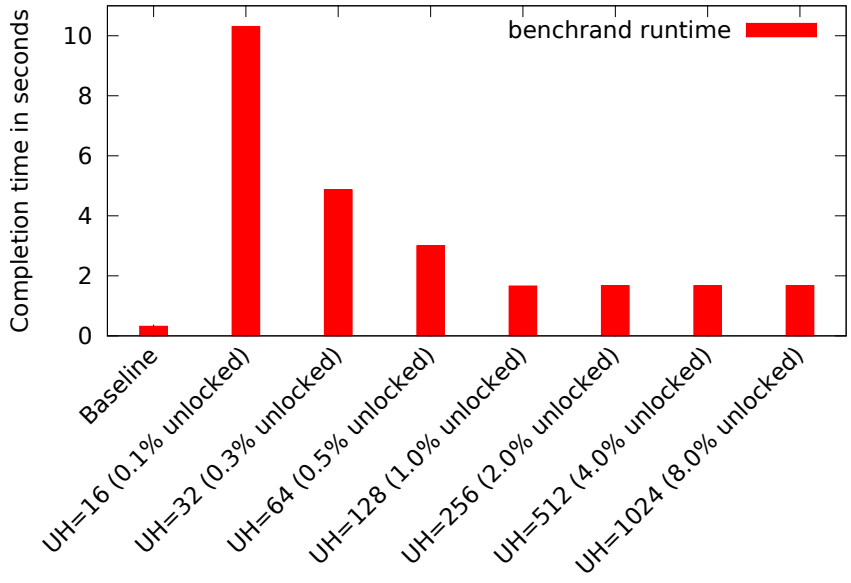
As we will later explore in Section 4.4.1, the implementation of page-level tracking is relatively expensive. In the example under consideration at the moment, an uninstrumented run of bench completed in 0.34 seconds, whereas the tracked execution in Figure 4.1 was 100 times slower. Although the runtime overhead was severe, the amount of memory in the *unlocked* state stayed on average at approximately 0.1%.

To counter the high overhead of page-level tracking, we next introduce the optimization parameter *unlock ahead*. *unlock ahead* specifies a number of additional pages to unlock linearly ahead of the current page accessed. E.g., if *unlock ahead* is 3 while P10 is accessed, then pages P10, P11, P12, and P13 would all be unlocked at the same time. This reduces the number of interruptions an application will receive during execution, at the trade-off of a speculative memory access pattern. Tuning *unlock ahead* depends on the application's characteristics. For bench, memory is scanned from low addresses to high addresses without many jumps, so higher *unlock ahead* values will directly correlate to both an increase in performance and in the number of pages *unlocked* on average (see Figure 4.3a). However, applications that perform strides through memory will be best served by finding an *unlock ahead* parameter with a matching size of each stride's chunk of memory read. Listing 4.2/Figure 4.2 show an application that reads 512KB chunks at random memory locations.

Figure 4.3b demonstrates optimal performance (in terms of execution time) first at *unlock ahead*=128, since 128 * 4KB (VM Page Size) = 512KB. We also see that increasing the *unlock ahead* value beyond the working-set size of an application will **increase** the amount of memory *unlocked* without increasing performance. Specifically, we see that as we move from an *unlock ahead* of 128 to 256, 512, or 1024 that the only notable difference is a significant increase in the



(a) bench: *relock interval*=0.2 secs



(b) benchrand: *relock interval*= 0.01 secs

Figure 4.3 Runtimes of benchmarks: UH=*unlock ahead* value, "% unlocked"=avg. % pages *unlocked* between each relock interval.

unlocked pages from 1% to 8% with no runtime effect. This effect underscores the importance of proper tuning since we do not want to speculatively unlock pages that will not actually be used by an application in the relatively near future.

One last consideration is in choosing the optimal combination of *relock interval* and *unlock ahead*. In most applications, there will be a many-to-one mapping of *relock interval* and *unlock ahead* values to the exact same execution speed. Yet, there will likely be variance in the number of pages in the *unlocked* state for each of these combinations. For this reason, applications should be tuned not to the minimal runtime necessarily, but rather the global minimum that achieves both a desirably low number of *unlocked* (lower counts of *unlocked* yields higher protection) matched with an acceptable execution time.

4.3.2 Error Detection (Memory Verification)

By assuming the model for the previous section wherein memory accesses are efficiently tracked by FlipSphere, we next move to begin protecting memory from silent data corruption. In network communications and many file systems, cyclic redundancy check (CRC) codes are used to guard against changes to raw data. CRC codes are one-way hashes that are particularly well suited to detect corruption of data as they are designed to be computationally cheap, fast to compute since they are used for verification instead of security. In FlipSphere we use the CRC32 hash, which requires only 32bits of storage per hash generated.

We can now extend the infrastructure provided by FlipSphere's page tracking concepts to define a means of memory verification that occurs during the transition between *unlocked* and *locked* states. First let us assume that for every page tracked by FlipSphere, we store additional metadata that contains both the *state* (*unlocked* or *locked*) and the *CRC*. If a page is in the *locked* state, then it is assumed to have a corresponding *CRC* saved in metadata that holds the CRC32 value of that page at the time when it previously transitioned from *unlocked* to *locked*. Further, when a page is accessed at the transition from *locked* to *unlocked*, we generate a new CRC32 of the page being accessed while FlipSphere interrupts the execution (Listing 4.3). The new CRC32 is compared against the previously stored CRC32. If they match then FlipSphere marks the page as *unlocked* and returns control to the application. However, if they do not match then the only conclusion is that the data was modified while it resided in RAM, which may indicate that an SDC has occurred. If we only perform CRC verification, FlipSphere may notify the user and/or process, and optionally terminate it immediately as to not perform further computations on invalid data. A rollback to a previously known good checkpoint may be desired, if checkpointing is available. However, we will later introduce automatic error correction

Listing 4.3 Unlocking and Relocking (Detection only)

```
#define PAGE_SIZE 4096 /*Typical 4KB page size*/

int StoredCRCValues[TOTAL_PAGES]; /* 32bit int per page */
char* MemoryStart; /* Protected memory */

//Called upon page access. Transitions page from locked->unlocked
   .
void UnlockPage(int page) {
  void *address = MemoryStart + (page * PAGE_SIZE);
  if( GenerateCRC32(address, PAGE_SIZE) != StoredCRCValues[page]
     )
    ERROR(SDC_DETECTED, "Page␣%d␣had␣an␣SDC\n", page);
  MarkPageAsUnlocked(page);
}
//Called by a timer at relock-interval. Transitions ALL pages to
   locked.
void RelockAllPages() {
  for(int page = 0; page < TOTAL_PAGES; page++) {
    if(PageIsUnlocked(page)) {
      void *address = MemoryStart + (page * PAGE_SIZE);
      StoredCRCValues[page] = GenerateCRC32(address, PAGE_SIZE);
      MarkPageAsLocked(page);
} } }
```

provided by FlipSphere (in Section 4.3.3).

Recall that after a page transitions to an *unlocked* state, there are no further interruptions from FlipSphere until it is returned to the *locked* state during the next relock interval. Since the CRC value stored by FlipSphere is immediately outdated once a page has been written to, FlipSphere must calculate and store a new CRC value every time the page transitions back to a *locked* state, as shown in Listing 4.3. This process occurs indefinitely throughout the execution of any application protected by FlipSphere.

4.3.3 Error Correction

In the previous section, we referred to FlipSphere's ability to store a hash of pages that are under its protection. While the comparison of hash values cannot correct errors, we can still provide correction capabilities by computing and storing error correcting codes (ECC), such as hamming

Table 4.1 Storage Overheads of FlipSphere Error Detection and Correction Codes

Algorithm	Overhead per 4KB page	Storage Overhead %
CRC32 Hash	4 bytes	0.10%
72/64 ECC	512 bytes	12.5%
Total Cost	**516 bytes**	**12.6%**

codes alongside our CRC hashes. For example, the 72/64 hamming code, frequently used in hardware, may be employed inside of FlipSphere to provide single error correction, double error detection (SECDED) capabilities at the expense of the additional storage required for the ECC codes. In fact, combining FlipSphere with hardware ECC can provide not only the ability to detect triple bit errors or greater (dependence on the capabilities of any particular ECC chip), but can also provide correction capabilities as the software-layered protection in FlipSphere may still retain viable error correcting codes once hardware protection has been exceeded.

Using FlipSphere extended with hashing plus ECC codes, it is possible to enjoy the protection and speed of hashing while limiting ECC code recalculation during unlocking only to times when a page has become corrupt during execution resulting in a mismatched hash. Listing 4.4 revises the pseudo-code for error detection (CRC) and correction (ECC). FlipSphere extends basic error detection by using 72/64 hamming codes for parity, which requires 1 byte (8bits) of storage per 8 bytes (64bits) of data, as shown in Table 4.1.

4.3.4 Memory Layout

Internally, FlipSphere maintains several separate sections of memory, the largest of which is the memory of the protected application. At a high level, FlipSphere is capable of protecting any contiguous range of virtual memory, such as the application's heap or BSS data section exclusively. Beyond that, storage is kept for internal data, namely for *locked*/*unlocked* status, CRC, and ECC data of every managed page. The internal data is held in a separate range of virtual addresses distant from the application's data. Application data may be either (1) unprotected, which is outside of FlipSphere's range of protection, (2) protected but *unlocked* noting it has been accessed recently, or (3) protected and *locked* indicating that it has not been accessed recently and has up-to-date CRC and ECC bits stored.

Listing 4.4 Error Detection and Correction

```
#define PAGE_SIZE 4096 /*Typical 4KB page size*/

/* 72/64 Hamming stores 1 byte of ECC per 8 bytes of data */
#define ECC_GROUPS_PER_PAGE (PAGE_SIZE/8)
char StoredECCValues[TOTAL_PAGES][ECC_GROUPS_PER_PAGE];

int StoredCRCValues[TOTAL_PAGES]; /* 32bit int per page */
char* MemoryStart; /* Protected memory */

//Called upon page access. Transitions page from unlocked->locked
   .
void UnlockPage(int page) {
  void *address = MemoryStart + (page * PAGE_SIZE);
  if( GenerateCRC32(address, PAGE_SIZE) != StoredCRCValues[page]
     ) {
    PerformECCRecovery(address, StoredECCValues[page]);
    /* See if ECC was successful after attempted recovery */
    if( GenerateCRC32(address, PAGE_SIZE) != StoredCRCValues[page
       ] )
      ERROR(SDC_DETECTED, "Page␣%d␣had␣an␣unrecoverable␣SDC\n",
         page);
    else
      printf("Page␣%d␣RECOVERED\n", page);
  }
  MarkPageAsUnlocked(page);
}
//Called by a timer per relock-interval. Transitions ALL pages to
    locked.
void RelockAllPages() {
  for(int page = 0; page < TOTAL_PAGES; page++) {
    if(PageIsUnlocked(page)) {
      void *address = MemoryStart + (page * PAGE_SIZE);
      StoredCRCValues[page] = GenerateCRC32(address, PAGE_SIZE);
      GenerateECCBits(address, StoredECCValues[page]);
      MarkPageAsLocked(page);
} } }
```

4.3.5 Regions of Memory Protected

Internally, FlipSphere maintains its own alternative protected heap space for the application and interposes memory allocation functions, such as `malloc`, `realloc`, and `memalign`. These interposed functions will allocate memory from FlipSphere's protected heap and provide addresses for the application. Later, these heap pages will always be either *locked* or *unlocked*.

Alternatively, for applications that allocate the bulk of their memory in a data or BSS section of the executable, FlipSphere protects those sections of data as well. When an application begins execution, all memory in FlipSphere's protected heap or data/BSS sections are *locked* by default. When an application allocates memory (i.e., calls `malloc`), the pages returned will become *unlocked* on future read/write memory accesses until locked by the library later on. Memory outside of the scope of FlipSphere's protected memory will by default not be protected or altered in any way.

4.3.6 Acceleration: Xeon Phi and SSE4.2

The heavy use of CRC hashing and software-based generation of ECC bits would normally prohibit the efficiency of any software error detection and correction mechanism. FlipSphere takes advantage of recent hardware developments by performing CRC generation on the host CPU by utilizing the new, performant CRC32 instruction[6] added in the SSE4.2 instruction set. Furthermore, one of FlipSphere's significant contributions is the use of accelerators with an asynchronous kernel written for the Intel Xeon Phi (Intel Many-Integrated Core) co-processors[7]. Each Phi co-processor is an Intel chipset with approximately 60 cores capable of over a teraflop of performance through a high degree of parallelism. As each co-processor is a dedicated PCI-x card, FlipSphere's custom kernel is responsible for data transfer of application memory via DMA between the host memory and the Phi's own on-board memory. From there, FlipSphere is able to generate new ECC bits (previously shown as a function in Listing 4.4) that are kept until the next relocking interval and the process is repeated. We will explore both full and partial Xeon Phi utilization for resilience such that a co-processor may optionally be partially co-executing an application and our resilience algorithm at the same time on a Phi.

4.3.7 Assumptions and Limitations

FlipSphere's protection extends only to memory and is not designed to protect against faults that occur in the CPU or other attached devices, including any attached Xeon Phi co-processors.

Since protection is provided for data stored in main memory, FlipSphere requires the

capability to detect memory accesses. FlipSphere achieves this by altering process page tables and removing read/write page permissions in order to receive OS signals that indicate which memory addresses are being accessed upon a page fault. This requires a memory management unit (MMU) and the ability for applications to install a signal handler that detects access violations.

For simplicity, FlipSphere at present only protects memory that is dynamically allocated using previously mentioned functions such as `malloc` or static sections of data such as the BSS. While the benchmarks evaluated under FlipSphere handle BSS and `malloc` data, the approach generalizes and could be extended to protect all data regions (including code and initialized data).

As FlipSphere verifies page contents upon transitioning from the *locked* to the *unlocked* state, any SDCs that affect *unlocked* memory during the window in which they are not protected are vulnerable. For this reason it is important to ensure that the relocking timer fires frequently enough as to not needlessly leave more pages than necessary in an *unlocked* state when they are not being utilized. Specifically, it is desirable to ensure that the *relock interval* does not allow substantial amounts of memory to become *unlocked* when only a smaller working set of pages is needed for execution.

Any application that depends on DMA with devices such as network interconnects must ensure that buffers are in an *unlocked* state before DMA begins. This assumption is necessary since DMA avoids the MMU and thus FlipSphere is never notified of page accesses to buffers. Data written through DMA would appear as corruption to FlipSphere because changes made via DMA occur while the pages are in a *locked* state. As described in the next section, we ensure that MPI and other libraries depending on DMA safely work within this requirement via MPI library interpositioning. FlipSphere ensures that MPI safely works with *locked* pages used as pointers in MPI operations by tracking all outstanding MPI requests and any associated pointers.

4.4 Implementation

4.4.1 Memory Tracking Technique

To ensure protection of memory, FlipSphere has to receive a notification when a page is accessed, which is implemented via a `mprotect` system call to remove read and write access of protected pages. Removing these permissions ensures that a segmentation fault (`SIGSEGV`) violation is raised when the page is accessed. The library installs a signal handler for this `SIGSEGV` violation

for notification.

Upon notification, FlipSphere uses an internal table to verify that the addressed page is under its protection. Then, as stated previously, verification is performed by comparing hash values. After verification, the page's read and write bits are restored, again using the `mprotect` call, and control is returned to the application.

FlipSphere's internal table stores the following information for each protected page.

- A status flag to indicate locked, unlocked, or permanently[2] unlocked pages;
- Storage for the page's last known good hash;
- (Optional) Storage for the page's software ECC bits.

Also, each locked page's hash and ECC storage is tunable to accommodate the size of whichever hashing algorithm is used.

4.4.2 Synchronized mprotect/TLB Flushing

In order for FlipSphere to track accesses to memory, it depends on the memory management unit (MMU) to trigger access violations whenever pages in the *locked* state are read or written. During the transition from *locked* to *unlocked*, which occurs after FlipSphere verifies an accessed page's hash, the `PROT_READ` and `PROT_WRITE` page permissions are added to the target page. In the x86_64 architecture, in which we evaluated FlipSphere, the process of adding additional access rights to a page does not involve flushing the TLB, which makes the cost of transitioning from `PROT_NONE` to read and write permissions relatively cheap. Unfortunately, the reverse process in which we return all pages' access rights to `PROT_NONE` during a relocking cycle is much more expensive and, in fact, causes a TLB flush for each call to `mprotect`.

To minimize the disruption caused by frequent TLB flushes, FlipSphere's unlock and relock algorithms are extended to perform as few `mprotect` calls as possible, by pre-computing which neighboring virtual pages will receive the same permissions in order to coalesce all identical page permissions into as few system calls as possible.

Further, if desired, FlipSphere may synchronize all protected processes on a node when the relocking timer is triggered. Instead of independently tracking the next relocking trigger per process, one server process simultaneously notifies all protected processes to temporarily

[2]Pages considered permanently unlocked refer to any region of memory that is either undergoing a potential DMA transfer (MPI transmission, for instance) or a system call that may directly change application memory). Pages are no longer considered permanently unlocked as soon as the operation that placed them in this state completes, such as the completion of an MPI message receive to a buffer.

suspend execution while pages are re-*locked* and `PROT_NONE` is applied to their page permissions. Synchronizing across all processes not only avoids costly TLB flushes during computation but also ensures that memory access performance remains as consistent as possible to reduce the system noise of TLB misses.

4.4.3 Hashing and ECC Implementations

FlipSphere currently supports the CRC32 hashing algorithm and the 72/64 hamming code (ECC) for both CPUs and the Xeon Phi accelerator. Additional hashing and/or ECC algorithms can be easily added to FlipSphere, for example, utilizing external libraries such as libgcrypt or writing new kernels for the Xeon Phi.

Our own 72/64 hamming code implementation is capable of single-error-correct-double-error-detect on each group of 64 bits in protected memory. Our kernel is capable of execution on either a host processor with SSE4.2 (we rewrote our algorithm to use instructions like the bit population counter in SSE4.2) or preferably on a Xeon Phi by substituting the population count op-code for a comparable one of the Phi's ISA.

We parallelize both the host and Phi code using OpenMP, substitute the normal parity computation with compact look-up tables, generate pipelined code by removing instruction branching, and finally unroll our computations to match the size of a virtual page on the target host. Note that we also implement verification/correction for the ECC codes to fix errors on-demand when a hash mismatch occurs.

4.4.4 Optimization: Background Relocking

Recall that FlipSphere can synchronously trigger all protected processes to relock their unprotected pages all at once to line up anticipated TLB flushes due to `mprotect`. Unfortunately, this leaves applications with a brief window of time in which all progress is stopped, including communication. This is FlipSphere's primary source of overhead to a protected application's completion time. Mitigating this, to allow applications to make forward progress in both computation and any outstanding communication, FlipSphere provides an optimized version of its relocking algorithm that operates in parallel to the application. By temporarily serializing the process of `mprotect`ing protected pages and optionally DMA copying memory to an accelerator, such as a Xeon Phi, FlipSphere can delegate the task of generating ECC bits to run in parallel to the application as soon the application's memory is safely copied via DMA. Since accelerators operate in the background, the application continues with forward progress immediately.

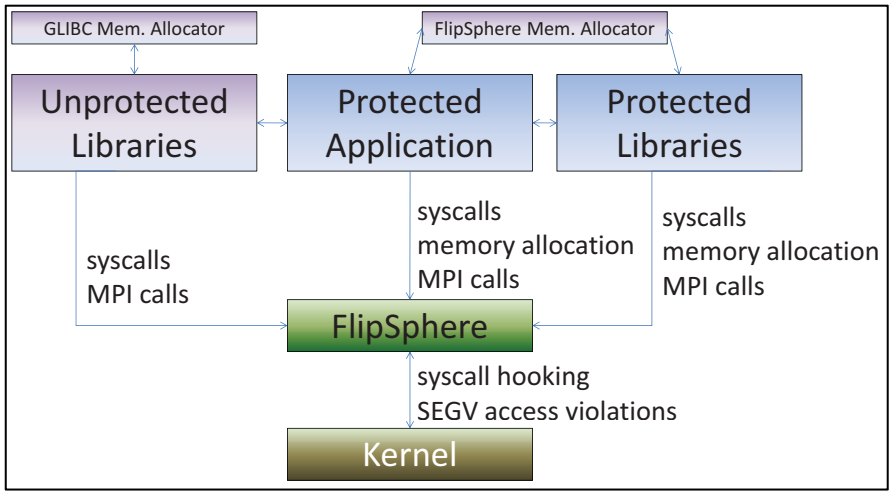


Figure 4.4 FlipSphere Component Interactions

4.4.5 Protected Memory: Heap and BSS

There are two potential types of memory that FlipSphere protects: the heap or BSS. First, heap data that is dynamically allocated via `malloc`, `realloc`, `free`, etc. is protected by interposing the standard memory allocation functions. Internally, a separate heap is created by FlipSphere and managed by the dlmalloc memory allocator. Returning addresses that are within a predefined protected range of memory, FlipSphere's allocator effectively provides SDC protection for dynamically allocated memory. Since many dynamically linked libraries will make use of dynamic memory, our interposed functions will perform a back trace using the stack to identify if the caller should receive a pointer to the regular application heap or FlipSphere's protected heap. For example, MPI implementations will commonly dynamically allocate memory, but special care must be taken to never return pointers to a protected heap address due to the potential for DMA accesses between the MPI implementations and communication devices. Application/library `malloc` calls will receive protected memory.

Note that since MPI calls and system calls always carry the potential to include pointers to protected memory, FlipSphere monitors these calls from both protected and unprotected libraries as well as the application itself.

Figure 4.4 visualizes the interactions between the various software layers and components with FlipsSphere.

Alternatively, many applications opt to allocate large arrays of memory in their BSS section. For these types of applications, FlipSphere inspects the program during startup (using

`LD_PRELOAD` and the `constructor` attribute we preempt even the `main()` function) and determines what range of virtual memory is allocated for the BSS section in order to directly apply page protection to this range. When FlipSphere is protecting the BSS, dynamic memory allocation interpositioning on `malloc` is disabled and thus no longer returns specially allocated *locked* memory to the application on heap requests. In this sense, FlipSphere mutually exclusively protects either the heap or BSS.

4.4.6 Client/Server Model of FlipSphere

As a single server may host multiple protected processes on it, FlipSphere has adopted a client/server model that allows it to provide services such as notification relock interval timers, multiplexing of Xeon Phi co-processors, and simplified logic abstracted away from running in the same address space as the protected application.

One of FlipSphere's primary goals is to be application agnostic. We accomplish this by bootstrapping the startup of a process to be protected by utilizing Linux's `LD_PRELOAD` environment to attach a FlipSphere shared library to the address space of a process during startup. This library proceeds to immediately intercept execution as soon as the process is created. A series of steps occur next:

Config: FlipSphere reads configuration data from environment variables. This includes heap size, *unlock ahead* tuning, *relock interval* value, and other tuning parameters, e.g., thread/core counts, and whether to track memory, provide error detection, and/or utilize the Phi for error correction codes.

Memory: FlipSphere allocates shared memory for the protected heap (if applicable) or unmaps the original BSS from the process's virtual mappings and then reallocates it as a shared region. It also creates shared memory for all internal data, such as page state, CRC values, and ECC bits. Shared semaphores and communication pipes are created for the next step.

Fork: The original process forks, sharing only data segments allocated in the previous step. If a forked server already exists, the client instead attaches to the server, thus connecting to all of the client's shared memory segments.

Server: The server tests communication with the client, and optionally "warms up" the Xeon Phi by allocating buffers which will later be filled with application memory prior to using the ECC calculation kernel.

Client: The client installs signal handlers for `SIGSEGV` and a signal for *relock interval* timer events from the server. Special hooks are installed in all Linux signal manipulation functions to ensure the process cannot overwrite FlipSphere's signal handlers. Instead, they may be installed as chains by FlipSphere. Thereafter, FlipSphere code preloaded into the application will only execute in (1) the context of a signal handler for FlipSphere or `SIGSEGV`, (2) one of many hooked wrapper functions that ensure MPI and system calls have their pointers unlocked prior to execution, or (3) our alternative protected heap allocator, if applicable.

Proceed: Full control is returned to the application, which begins to run `main()`.

One point of interest is that between the client (application) and server (FlipSphere), all virtual memory is shared via Linux shared memory. This removes the need for any redundancy in host memory or any memory copies, since both processes have access to the same physical memory, albeit in different virtual address spaces. By adopting this model, the server initiates communication via triggers and timers that are received as signals by the client. Utilizing inter-process communication (IPC), the server is able to control or multiplex the Xeon Phi for the client, which provides the added benefit that a protected application may still be able to offload computation on to a Phi if it desires since the server is an entirely separate process.

4.4.7 Handling User Pointers with System Calls

The use of the `SIGSEGV` handler allows FlipSphere to track an application's memory accesses at the page level during execution. Unfortunately, if an application or one of its libraries makes a system call with a pointer to userspace memory, the kernel will not invoke the userspace `SIGSEGV` handler when it is unable to dereference a pointer. E.g., a system call to `read` will directly place data into a userspace process's memory. Since the kernel is directly writing to a pointer of a process, a `SIGSEGV` will not be generated if the kernel is given a pointer to a region of memory that does not have write permission. For this reason, we must wrap relevant system calls and preemptively unlock any pointers to protected userspace memory that the kernel receives as parameters. To achieve this, FlipSphere includes hooks for syscalls via a wrapper that directs the userspace FlipSphere library to unlock all pointers passed to the kernel prior to starting the actual system call's logic.

Not all system calls are interposed, and many do not even need to be since they lack userspace pointers. In choosing which system calls to support initially, we profiled many MPI applications and determined which system calls were needed for our MPI library and for the applications themselves to function as a proof-of-concept.

It should be noted that there are other, less portable solutions that may accomplish system call interpositioning, but would require extensive per-platform work such as binary rewriting. Our goal was to provide a platform for gauging the viability and costs of SDC protection through hashing and page protection while avoiding writing a complex platform-specific system call interpositioning scheme that would not add to the research contributions.

4.4.8 MPI Support and DMA

Any libraries (such as MPI) that depend on DMA to transmit data to hardware will operate outside the bounds of the MMU and thus will go undetected by FlipSphere. To ensure that any memory undergoing DMA accesses will not be perceived as corruption, FlipSphere interposes all calls to MPI functions using the MPI profiling layer. Any MPI function that takes as arguments a pointer to memory that will be read/written will be treated as permanently *unlocked* before control is transferred to MPI. Additionally, since non-blocking MPI sends and receives may leave a region of memory volatile to change for an indeterminate amount of time, FlipSphere tracks all outstanding MPI Request objects to ensure that any pending non-blocking MPI requests will be prevented from returning the to the *locked* state until the MPI Request is completed. This specifically includes asynchronous send/receive operations, which FlipSphere identifies as outstanding until a matching `MPI_Test`, `MPI_Wait`, or related function is called for each request. Memory that may be modified by MPI and DMA operations will always remain in an *unlocked* state until the MPI operation that triggered it is finished. FlipSphere has support for all common MPI calls in C, C++, and Fortran.

4.5 Experimental Results

To gauge FlipSphere's effectiveness, we performed experiments to demonstrate both range of coverage in memory and cost as application runtime overhead. These experiments were carried out on compute nodes of the Stampede cluster at TACC with each node consisting of 16 cores with Intel Xeon E5 E5-2680 processors and 32GB memory.

For our experiments involving Xeon Phi co-processors, we ran FlipSphere's kernels on Xeon Phi SE10P (Knights Corner) co-processors. Our results will cover performance metrics under utilization of an entire co-processor for resilience as well as partial resource sharing to allow co-execution of resilience algorithms and computation on the same accelerator.

As one of the contributions of FlipSphere is ECC generation in a Xeon Phi kernel, we first wrote our ECC algorithm on an x86_64 host CPU and evaluated the engine to measure the

throughput of ECC generation. Next, we ported the optimized version of our algorithm to the Xeon Phi and evaluated it by varying the number of threads available. Table 4.2 shows the performance of both the host CPU and Phi. As our code is highly parallel, both configurations show peak performance when the maximum number of hardware threads is used. While raw throughput when all host CPUs are used does outperform a Xeon Phi at full capacity, it is critical to keep in mind that we may only dedicate a subset of the available computation resources to resilience while the majority of the resources will be for application progress. In FlipSphere, ECC generation occurs in tandem with regular computation, i.e., compute cores are used for application forward progress while others may be reserved and utilized for resilience. Crucially, if resources are not split between computation and resilience, then applications would be forced to wait for periodic resilience tasks to complete when in fact the tasks may be safely and efficiently split. Therefore, let us now look at the most important metric of Table 4.2 by noting that when 25% of the host threads are reserved for resilience, we achieve only 26% of the potential ECC generation (resilience) throughput. But we can nearly double the efficiency to 48% of optimal Phi performance when only 25% of Phi cores are reserve for resilience. The marked difference indicates that systems based on Xeon Phi are better suited for software-based resilience than traditional architectures.

Although today's Intel Xeon Phi Knights Corner (KNC) co-processors are separate compute units that communicate through the PCI-express and maintain their own DRAM (separate from the host's DRAM), the next generation Xeon Phi (Knights Landing — KNL) will be available in two forms, as a co-processor and as the next generation host CPU. For FlipSphere's purposes, this change provides a key variable in the way we evaluate our performance. In KNC, which we evaluated FlipSphere on, all memory must incur a DMA between the host CPU and the co-processor via PCI-express. However, the next generation KNL simply will not incur the performance penalty as the host DRAM is directly attached to the Xeon Phi. While Xeon Phi's computational performance will likely improve in each generation, from FlipSphere's perspective we are most concerned with the fact that DMA overhead is not relevant and any clock speed performance gains or additional threads will only increase performance.

FlipSphere's Xeon Phi implementation of ECC generation was profiled earlier as shown in Table 4.2. Our first result to report is an observed DMA transfer rate between the host CPU and Xeon Phi (KNC) of 6271 MB/s, which is lower than the kernel's ECC generation throughput of 9690 MB/s. Although the DMA transfer rate appears reasonable, we notice that the serial portion of FlipSphere requires a DMA copy from the application's data to the Xeon Phi before the application may resume progress. For instance, copying 512 MB of application data imposes a serial requirement of at least 0.08 seconds at every *relock interval*. For an application running

Table 4.2 ECC Generation Performance

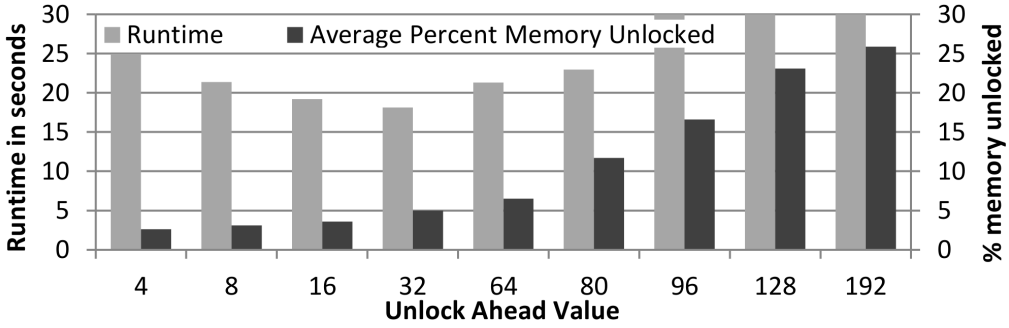
Number of Threads	Throughput	% of Optimum Throughput
1/16 CPU Thds.	1148 MB/s	7%
2/16 CPU Thds.	2161 MB/s	13%
4/16 CPU Thds.	4352 MB/s	26% **(25% host thds. used)**
8/16 CPU Thds.	8691 MB/s	51%
15/16 CPU Thds.	16016 MB/s	94%
16/16 CPU Thds.	**17035 MB/s**	100% **(Host Optimum)**
17/16 CPU Thds.	14800 MB/s	87%
61/244 Phi Thds.	4372 MB/s	48% **(25% Phi thds. used)**
122/244 Phi Thds.	7226 MB/s	76%
244/244 Phi Thds.	**9560 MB/s**	100% **(Phi Optimum)**
305/244 Phi Thds.	7754 MB/s	81%

for one hour with a *relock interval* of 0.1 seconds, we expect 72000 interrupts for DMA for a total costed time of 48 minutes, which would be unreasonable. As discussed, the DMA penalty will not apply to the KNL generation of Xeon Phi. Thus we will report results for KNC (DMA added), and KNL (no DMA, derived as an interpolation from the former) in the results section, although the pertinent result is the effectiveness of FlipSphere's Xeon Phi integration, not the cost of DMA, which will be phased out from Xeon Phi.

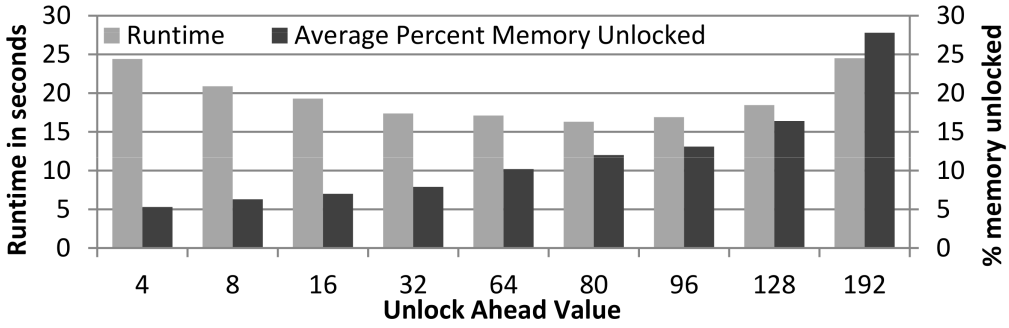
As FlipSphere protects applications from silent data corruption by employing a generalized technique of hashing pages to provide on-demand integrity verification, we chose to evaluate our library on two different applications from the NAS Parallel Benchmarks (NPB) suite. Our first experiment is NPB's FT (a discreet 3D fast Fourier Transform) benchmark with a customized input size 75x75x75 to extend execution across 10 iterations. The second experiment is NPB's CG (conjugate gradient solver) with a customized problem size of 150,000 and 10 iterations. Each application has different memory access patterns so that FlipSphere is evaluated in both an environment where memory is touched relatively infrequently per time unit (FT) and very frequently (CG). For reference, to consider how much memory will later be protected and transferred over DMA, CG processes used 475MB of memory, and FT 640MB.

4.5.1 Error Detection (CRC) Only

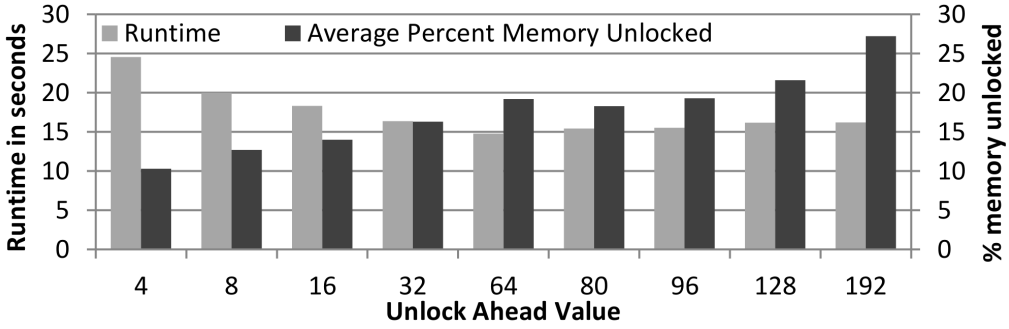
We first analyze the performance of FlipSphere with error detection (CRC) enabled on the NPB-FT benchmark as shown in Figure 4.5. In this particular experiment the host CPU is used for CRC generation, and Phi co-processor is not utilized. The three subfigures show multiple experiments with different *relock interval* values. Figure 4.5a shows the results of varying the



(a) Relock interval 0.025 sec

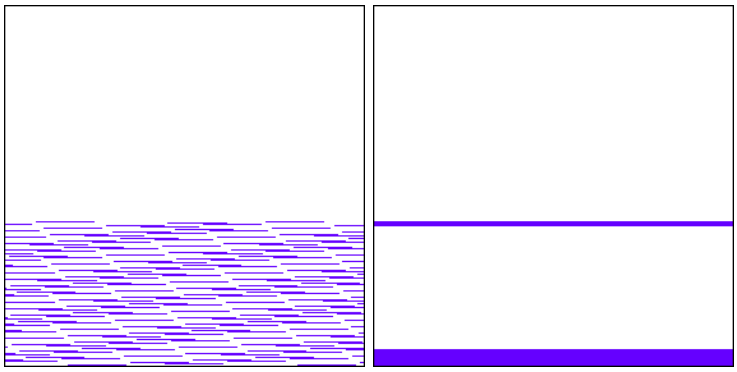


(b) Relock interval 0.05 sec



(c) Relock interval 0.10 sec

Figure 4.5 NAS FT CRC (SDC Detection Only)



(a) Strided (b) Linear

Figure 4.6 Two common, periodic memory access patterns of NPB-FT

unlock ahead (UH) parameter while keeping a constant *relock interval* of 0.025 seconds. We observe a convex trend with a minimum runtime at UH=32 of 18 seconds (left y-axis) and approximately 5% memory unlocked (right y-axis) on average, or, inversely said, up to 95% of memory is protected against silent data corruption at this datapoint. *This is a substantial degree of protection.* For reference, the original benchmark completed in 10 seconds, which indicates that with RL=0.025 seconds and UH=32 we observed a 70% runtime overhead, which is relatively high albeit with a high degree of protection.

As FlipSphere is designed to be tuned to an application, it is important to point out that the reported results show the affects of tuning. By observing the effects of varied input parameters we see that Figure 4.5c shows no added runtime benefit (in terms of time to completion) when UH is increased beyond 32. In fact, as UH is increased, we only notice a steep decline in the amount of memory protected. Figures 4.5a and 4.5b both show sensitivity to *unlock ahead* and *relock interval*, which demonstrates the potential ranges a user would explore to find their desired trade-off of performance vs. resilience. Generally, decreasing the *relock interval* increases the resilience of an application and its time to completion, while the *unlock ahead* value is then subsequently used to further tune the application (decrease overhead by improving time to completion) to match both the working set size (per unit *relock interval*) and amount of contiguous memory read in stride per memory region/structure.

4.5.2 Error Detection and Correction

We now evaluate FlipSphere's detection and correction contributions in two configurations: (1) All Xeon Phi threads are used for resilience while the protected application runs on the host,

Table 4.3 NAS FT - CRC plus ECC - With DMA, 100% Xeon Thds. (Full Load on Knights Corner)

Row	Relock Interval	Unlock Ahead Value	Normalized Completion Time	Percent Memory Unlocked
1	0.05s	16	1.62x	17.8%
2	0.05s	32	1.46x	20.6%
3	0.05s	64	1.44x	23%
4	0.05s	128	1.51x	24.5%
5	0.1s	16	1.59x	18.3%
6	0.1s	32	1.5x	20.5%
7	0.1s	64	1.44x	23.1%
8	0.1s	128	1.45x	25.2%
9	0.15s	16	1.59x	22.8%
10	0.15s	32	1.51x	24.2%
11	0.15s	64	1.44x	26.7%
12	0.15s	128	1.52x	27.6%

Table 4.4 NAS FT - CRC plus ECC - Without DMA, 25% Xeon Thds. (Approximating Load Splitting)

Row	Relock Interval	Unlock Ahead Value	Normalized Completion Time	Percent Memory Unlocked
1	0.05s	16	1.56x	8.8%
2	0.05s	32	1.40x	10.8%
3	0.05s	64	1.58x	10.8%
4	0.05s	128	1.51x	15.4%
5	0.1s	16	1.47x	16.8%
6	0.1s	32	1.54x	16.8%
7	0.1s	64	1.43x	19.4%
8	0.1s	128	1.30x	23.3%
9	0.15s	16	1.54x	23.6%
10	0.15s	32	1.51x	24.6%
11	0.15s	64	1.37x	27.6%
12	0.15s	128	1.31x	29.5%

Table 4.5 NAS CG - CRC plus ECC - With DMA, 100% Xeon Thds. (Full Load on Knights Corner)

Row	Relock Interval	Unlock Ahead Value	Normalized Completion Time	Percent Memory Unlocked
1	0.05s	16	1.80x	22.2%
2	0.05s	32	1.55x	31.3%
3	0.05s	64	1.35x	43.6%
4	0.05s	128	1.40x	50.4%
5	0.1s	16	1.69x	39.3%
6	0.1s	32	1.46x	53.4%
7	0.1s	64	1.33x	62%
8	0.1s	128	1.30x	66.3%

Table 4.6 NAS CG - CRC plus ECC - Without DMA, 25% Xeon Thds. (Approximating Load Splitting)

Row	Relock Interval	Unlock Ahead Value	Normalized Completion Time	Percent Memory Unlocked
1	0.05s	16	1.85x	21.9%
2	0.05s	32	1.52x	34.7%
3	0.05s	64	1.43x	42.1%
4	0.05s	128	1.21x	48%
5	0.1s	16	1.80x	41.8%
6	0.1s	32	1.43x	53.4%
7	0.1s	64	1.32x	62.7%
8	0.1s	128	1.38x	62.1%

which obligates DMA transfer between the host and Xeon Phi, and (2) we approximate a Xeon Phi running as the host processor (no DMA required) with 75% of threads reserved for the application while 25% are earmarked for resilience. Today's Knights Corner co-processors are used in the former and are an accurate representation of performance. An environment with a Knights Landing host processor is approximated by reserving only 25% of hardware threads for FlipSphere. The ratio of compute to resilience threads is variable, but we have chosen 25% from Table 4.2. This configuration does not require DMA, hence, performance is increased without the overhead of DMA. At the same time, the resilience throughput is decreased. Depending on the access patterns of applications, in some occasions the net difference results in no performance change in terms of protection and overhead. For FT, Rows 2 of Tables 4.3 and 4.4 show similar completion time, for instance, but protection is halved in the latter. For CG, rows 1 and 2 of Tables 4.5 and 4.6 show almost no difference in performance as the effect of moving the overheads from DMA to computation was balanced out.

For all results in Tables 4.3 to 4.6, we have highlighted what we deem to be the best performing configurations. Each row contains as tunable input parameters the *unlock ahead* value and the *relock interval* resulting in a normalized runtime (1.0x is an unmodified execution without resilience) and the percent of memory unlocked, on average. To find the degree of protection, we invert the percent of memory unlocked, i.e., lower unlocked is better. Although not shown, we have evaluated FlipSphere in many unreported configurations (to conserve space), but have reported the best performing values and their neighbors to provide perspective.

As shown in Figure 4.6, FT predominantly experiences a strided access memory pattern and periodic scanning pattern. Compared to CG (which has rapid, linear scanning of all its address space), FT's performance with FlipSphere is highly dependent on a tuned *unlock ahead* value. While Table 4.3 is representative of performance with DMA, a similar result also appears

in Table 4.4. We can see that in each set of 4 rows we only vary the *unlock ahead* value and always observe peak memory coverage between values 16-32 for *unlock ahead*. This is indicative of proper tuning to match the strides shown in Figure 4.6. Beyond that range, memory coverage rapidly decreases for values such as 64-128. Although not shown, one can increase the memory coverage by lowering the *relock interval*, but this is prohibitively expensive as overheads quickly exceed 2-3x, which would then begin to favor double modular redundancy for SDC detection alone (albeit without the corrective capabilities FlipSphere provides). For FT, we conclude that the highlighted row 2 is the best balanced configuration when both the completion time and memory protection are considered as it provides 90% memory coverage at a 40% overhead when Xeon Phi is the host CPU for both computation and resilience. Alternatively, it provides 80% coverage with a 46% overhead when traditional CPUs are used on the host for computation.

CG's performance is reported in Tables 4.5 and 4.6. While the tables feature different resilience configurations, the completion time and memory coverage is similar in CG's case since it is more memory bound than FT. While Table 4.6 utilizes only 25% of the Phi's hardware threads for resilience, its loss of resilience throughput is balanced out by a lack of DMA transfer overheads (Knights Landing). In contrast to FT, in both configurations of CG we find that due to high frequency of memory scanning the best reasonably achievable memory coverage of 88% is possible at a runtime overhead of 85%. CG illustrates the impact of high speed memory accesses on FlipSphere while still providing protection at less than 2x overheads. It is important to note the observed overheads are more performant than double redundancy and feature error correction, whereas the best case redundancy merely provides error detection.

4.6 Conclusion

FlipSphere realizes a silent data corruption detection and correction library. It protects pages of memory with known good hashes per page coupled with software based ECC codes. Memory is verified on demand and FlipSphere confirms the integrity of each page upon access while fixing any potential errors using the precomputed ECC codes.

We showed that using Intel Xeon Phi co-processors to offload software-based resilience is feasible for adding a variable amount of memory resilience to an application using a generic approach, free of any type of algorithmic fault-tolerance requirements. Our research is the first to indicate that when a portion of computational resources is set aside for resilience, Intel Xeon Phi is superior relative to a traditional host CPU for software ECC resilience computation.

FlipSphere exceeds the standard of protection provided by ECC by ensuring that errors evading even hamming codes will always be detected, even if not fixed, by performing CRC32

verification after hamming code correction is attempted.

Our results indicated that FlipSphere's error detection and correction can achieve up to 90% coverage with a 40% runtime overhead for some classes of applications. Even highly memory bound applications still achieve up to 88% coverage at an overhead of 85%, which provides significant benefit over double redundancy (100% cost) or triple redundancy (200% cost) as FlipSphere further provides error correction, in contrast to detection only under dual redundancy. With costs between 40%-85% vs 200%, our results may indicate an opportunity to disable ECC for DRAM and rather run ECC-unprotected when FlipSphere provides protection for kernels (not just for ECC-detectable events but also extra protection against silent data corruption), particularly since turning off ECC may result in lower memory latency and power consumption; yet, FlipSphere does not requiring algorithmic changes as in Li's work [66].

CHAPTER

5

MINI-CKPTS: NON-STOP EXECUTION IN THE FACE OF OPERATING SYSTEM FAILURES

5.1 Introduction

The core of an operating system, known as the kernel, is responsible for supervising running applications on a computer as well as providing services such as isolation, memory management, file input/output, network communication, interprocess (multithreaded) management and communication, and hardware device access. These tasks are accomplished by a predefined set of system calls that provide an application with a portable interface to acquire and utilize system resources as needed. While applications are isolated from each other and a failure in one application does not necessarily affect others, a failure in the kernel results in a system crash that will cause all running applications to forcibly terminate without warning. Today, most operating systems do not provide much or any resilience against failures within the kernel and are instead subject to fail-stop behavior: When an unexpected error is encountered, the operating system immediately stops thus preventing any continued progress of its running applications as well as

complete loss of unsaved application memory.

In the context of High-Performance Computing (HPC) environments where clusters have 100s of thousands of cores and 100s of terabytes of main memory, faults are becoming increasingly common. Faults include failures in memory, processors, power supplies, and other hardware. Research indicates that servers tend to crash twice per year (2-4% failure rate) and that DRAM errors occur in 2% of all DIMMs per year [89]. This trend does not appear to be slowing down as recent work shows that current generation DDR-3 memory modules maintain an approximately constant fault rate per device relative to previous generation memory, which continues to motivate research mitigating the effects of memory faults [14, 97]. Further, large studies indicate that simple ECC mechanisms alone are not capable of correcting a significant number of DRAM errors [61, 96].

This work will focus on protecting applications from faults affecting the Linux kernel that originate within the processor and main memory of a system. While the kernel occupies a relatively small (10-100s of megabytes) portion of memory relative to the HPC applications they support (typically up to 16 or 32 gigabytes per node today), the importance of protecting the kernel remains critical for the same reason that HPC clusters experience faults [5]. Despite a low likelihood of any individual component failing, the vulnerability of a single failure by an application spanning the cluster statistically increases at scale. The kernel too is a critical individual component in the HPC software stack that currently lacks any inherent error recovery mechanisms. Our work will target the Linux operating system due to its widespread use in HPC clusters — currently 97% of the Top500 as of November 2014 — although the observations and methodology presented herein will apply to any commodity operating system [5].

Faults due to hardware failure are a concern in large clusters, yet software bugs pose an equally fatal outcome in the kernel, regardless of scale. Bugs may result in kernel crashes (a kernel "panic"), CPU hangs, memory leaks resulting in eventual panics, or other outcomes leaving the system only partially functional. This work will experimentally show that this work improves upon the situation by recovering the system in the face of bugs that create memory leaks and hung CPUs in the kernel.

The primary objective of mini-ckpts is to show that it is possible to preserve the state of an HPC application and its memory across voluntary or failure-induced system reboots by serializing and isolating the most essential parts of the application to persistent memory locations. This is achieved in our experimental implementation by creating minimal modifications to the operating system for saving the CPU state during transitions between userspace and kernel space while migrating application virtual memory to an execute-in-place (XIP) directly mapped physical memory region. Finally, failures are coped with by warm booting a previously

loaded identical copy of the kernel and resuming the protected applications via restoration of the previous process' saved state inside the newly booted kernel.

While other research, such as Otherworld [31], has proposed the idea of a secondary crash kernel (activated upon failure) that parses the data structures of the original, failed kernel, this is the first work to demonstrate complete recovery independence from the data structures of an earlier kernel. This work notes that a failure in a kernel may corrupt the kernel's content. The failure itself could be due to a corruption or bug in the first place, which could impede access to the old kernel's data. mini-ckpts's unique contribution is that it has no dependencies on the failed kernel, and that all state required to transparently resume a protected process can be proactively managed prior to a failure. In turn, this allows HPC applications to avoid costly rollbacks (during kernel panics) to previous checkpoints, especially when considering that rollbacks typically require all communicating nodes to rollback together unless coupled with other more costly techniques such as message logging [63].

This work makes the following contributions:

- It identifies the minimal data structures required to save a process' state during a kernel failure.
- It identifies the common instrumentation locations generically needed in operating systems for processor state saving.
- It presents mini-ckpts, an experimental framework that implements process protection against kernel failures.
- It evaluates the overhead and time savings of mini-ckpts' warm reboots and application protection.
- It experimentally evaluates application survival of mini-ckpts protected programs in the face of injected kernel faults and bugs in both OpenMP and MPI parallelized workloads.

5.2 Related Work

Many hypervisors in both the open source (KVM) and commercial (VMWare) world provide live migration, which enables a virtual machine to be transferred to a new physical host transparently. During migration, there may be a period of unresponsiveness while the virtual machine's memory is copied over the network from the source hypervisor to the destination hypervisor. If a failure is expected to occur on the source hypervisor, then a preemptive live migration may provide a

means to prevent the virtual machine from failing alongside its host hypervisor. Unfortunately, this technique does not protect a virtual machine from unanticipated failures, nor will it protect a virtual machine if the hypervisor fails during an incomplete live migration. Likewise, redundant concurrency [34] within the same operating system suffers from potential failures if a failure occurs prior to migration or checkpointing.

Traditional checkpointing of applications is a well researched field. Normally checkpointing involves saving the state of one or more processes to disk, including its memory, memory layout, open files, threads, and handles to other services provided by the kernel. Checkpointing usually comes in two flavors: (1) System-level checkpointing, such as BLCR [36], which aims to provide transparent checkpoint and restart support to applications via kernel modules without requiring userspace application modifications, and (2) User-level checkpointing, such as MTCP [85] and others, which also aims to provide transparent or assisted checkpointing to processes without requiring kernel modifications or kernel modules. Although all traditional checkpointing methods may vary in how they are implemented or the services they provide, one common theme is that they all restore a process to an **earlier** state. For long running applications that are infrequently checkpointed due to the overhead associated with taking a checkpoint, the loss of progress due to a restart from failure may dramatically reduce the efficiency of an application. To mitigate some of the overhead incurred by checkpointing, various optimizations have been studied, such as incremental checkpoints saving only the changed portions of a program, compressing the checkpoints, and moving incremental checkpoint logic to the kernel [8, 102, 80, 49]. Nevertheless, even with such technology, when checkpoints, restarts, and rework are modeled to match future exascale systems' expected time to failure, studies show that applications may spend more than 50% of their time in checkpoint/restart/rework phases instead of making forward progress [38, 76, 88]. Today, failures that occur within the operating system require an application to rollback and rework as there simply is no other recourse. However, mini-ckpts may eliminate unnecessary restart/rework when the application itself remains recoverable, despite a failed kernel.

The idea of an operating system resilient to software error is not new [9, 62, 22, 56, 33], nor is system hardening using fault injection [87]. The MVS [9] operating system was designed with fault tolerance in mind for both software and hardware errors by requiring that all services be accompanied by an error recovery routine that executes in the event of a fault. This increases both the complexity and size of the operating system, and even in the case of MVS, recovery routines were provided for only 65% of the critical jobs. Despite the efforts of the recovery routines that were provided, in the cases where they were activated due to a fault, they were only successful 50% of the time in preventing an abort of critical jobs [99]. Noting that many

kernel failures occur due to bugs within device drivers, the NOOKS [98] framework wrapped calls between the core Linux kernel and device drivers to isolate the effects of one from the other. For cases where bugs are known to exist in drivers excluding the core kernel, NOOKS provided a transparent solution, but does incur a latency and bandwidth cost due to the wrapping of communication to and from drivers.

The Rio File Cache [24] provides the operating system with a reliable write-back file cache in memory that survives warm reboots. Rio guarantees that a file write operation buffered in the kernel during a failure will not be lost, provided that the kernel is able to perform a warm reboot, recover its unperturbed contents, and resume the write operation to stable storage on subsequent reboots. Rio is an example of an independent part of the operating system making guarantees about forward progress regardless of failures by providing a near-zero cost virtual write-through resilience cache.

Protecting the operating system in other ways, such as remote patching in Backdoors [17] or recovering from disk and memory, have been studied, but they are not focused on generalized, transparent high performance application restart in an HPC environment where automatic, low-latency recovery is essential for the efficiency of the cluster [12]. Additionally, work that requires remote intervention or local disk access may not be able to function in circumstances where drivers or kernel support for either the network or disk are no longer functional. mini-ckpts is specialized in that it aims to ensure minimal internal kernel dependencies during fault handling.

Otherworld [31] is perhaps the most similar work that attempts to recover applications in the event that a kernel fails. Otherworld maintains a second "crash kernel" that is loaded in a separate memory space and receives control of the system upon kernel failures. Their crash kernel uses known offsets of data structures within the old kernel to attempt to parse and reconstruct old kernel data structures. This includes a wide variety of traversals and parsing of memory mappings, process structures, file structures, and so forth. While mini-ckpts and Otherworld both aim to continue execution of an application after a crash, Otherworld heavily depends on an intact kernel state upon failure. This is both a key assumption and shortcoming. The event that caused a kernel to fail might either be an effect of a corrupt data structure (i.e., it would continue to exist upon restart) or may even create corruption within kernel data structures during the process of crashing. If any single leg of Otherworld's data dependencies has become corrupt, then reconstructing the application postmortem may be impossible. mini-ckpts aims to minimize the likelihood that a corrupt kernel may impede continued execution of an application by decreasing dependencies on kernel data structures both during the crash and the subsequent restart/rejuvenation of the kernel.

mini-ckpts serves a secondary role beyond reactive fault tolerance for operating systems

by providing a means of fast kernel rejuvenation, if used proactively outside the existence of an imminent crash/fault. Software rejuvenation conceptually involves restarting a program to combat process aging — namely the "gradual degradation" of application performance over time due to memory leaks, unreleased locks, bugs, etc. — that may lead to premature program termination [59]. Outside of operating system designs involving non-volatile memory, to the best of our knowledge, this work is the first in class to provide fast operating system restarts without checkpointing or interrupting an application [11]. Rejuvenating operating systems in HPC workloads and virtual machine monitors has been shown to be beneficial in reducing downtime provided that the rejuvenation either occurs between the execution of applications or that virtual machines may be paused/resumed [73, 64]. mini-ckpts advances this research direction by allowing a restart at any point in time during execution.

5.3 Overview

mini-ckpts protects applications by taking a checkpoint of a process and its resources, modifying its address space for persistence, and then periodically saving the state of the process' registers whenever the kernel is entered. With the checkpointed information and register state, mini-ckpts later recovers a process after a kernel crash occurs. Recovery is performed by recreating the process in the exact state it was in prior to the crash using the same well established techniques employed by traditional checkpoint/restart software such as Berkeley Lab Checkpoint/Restart (BLCR). In fact, we adopt some of BLCR's codebase directly for checkpoint and restoration of process resources. The basic checkpointing features mini-ckpts supports are shown in Table 5.1. Additionally, mini-ckpts support for persistence virtual memory is comprehensive. A summary is available in Table 5.2.

To use mini-ckpts, one first needs to boot a kernel with mini-ckpts modifications followed by additional bootstrapping steps during start-up:

- The mini-ckpts kernel is booted.
- Filesystems such as */proc* and */dev* are mounted.
- The PRAMFS persistence memory filesystem is either initialized as empty or mounted if it already exists from a prior boot.
- A fresh copy of the mini-ckpts kernel is loaded into a reserved section of memory for later use (if the system crashes).

Table 5.1 Summary of mini-ckpts core checkpoint features protected during failures

Feature	Status
Single Threaded Processes	yes
Multi-Threaded Processes	yes
Mutex/Conditions Variables	yes
Process ID	yes
Process UID & GID	yes
System Calls	yes; defaults to -EINTR on failure
Regular Files	yes; but file seek position requires new checkpoint
Unflushed File Buffers	no; Rio File Cache could provide this
Signal Handlers & Masks	yes
Pending Signals	no; any pending are not tracked
Stdin/Out/Err	yes
mmap'd Files	yes
mprotect	yes
FPU State	yes
CPU Registers	yes
Network Connections	no; applications must support restarting connections
Process Credentials	yes
Block Devices	partial; /dev/null, /dev/zero allowed
Special FD's	no; (no signalfd, no eventfd)

- If we are booting after a failure, the last checkpoint is restarted automatically. The mini-ckpts protected application resumes exactly where it left off.
- A shell server (SSHD) is started for remote access.

The first time the system is booted, a user may launch a program that they wish to protect with mini-ckpts. This can be performed manually via a remote shell or through a clustering tool that manages running programs. Once the application to protect is launched, it will begin receiving protection after its first checkpoint. A checkpoint is either self triggered programmatically via a call to a shared library (preloaded with $LD\_PRELOAD$) or by sending a specific signal to the process through an external checkpointing tool. The initial checkpoint migrates the process' virtual memory to PRAMFS, stores its open files, memory mappings, signal handlers, and so forth as shown in Tables 5.1 and 5.2. Once the first checkpoint is complete, the system will ensure that the CPU registers and FPU (Floating Point Unit) state are always saved in persistent memory when the kernel is entered. In order for the kernel to execute code that results in a fault, it must be entered first. Therefore, any entry to the kernel through a

Table 5.2 Persistent virtual memory support protected during failures

Memory Type	Notes
Executable	yes*
BSS Section(s)	yes
Data Section(s)	yes
Heap	yes
Stack	yes (plus each thread's)
Shared Libraries	yes*
Shared Library BSS & Data	yes
vdso & vsyscall	yes, provided by kernel
anonymous mmap'd regions	yes
file-based mmap'd regions	yes*
*Original mapping is migrated to PRAMFS.	

system call or interrupt handler will now update the checkpoint with the CPU and FPU state. Once the checkpoint is complete, any fault in the kernel will automatically warm reboot the system and trigger the application to resume exactly where it failed. This process can repeat indefinitely until the mini-ckpts protected application completes execution successfully.

5.4 Persistent Virtual Memory Across Reboots

Similar to how Rio File Cache protects file buffers (see Section 5.2), achieving persistence of virtual memory associated with a process is essential to mini-ckpts. This work will now explore how userspace memory persistence is designed in mini-ckpts.

5.4.1 Traditional Anonymous/Private Mappings and tmpfs

Achieving persistence of an application's virtual memory across kernel reboots stands as a key contribution of mini-ckpts. An application's virtual memory consists of file-mapped memory regions, anonymous ("private" or lacking a file backing) memory regions, and special (vsyscall or VDSO) regions. Additionally, file-mapped memory regions may be either private CoW (copy on write) or shared between processes. If a process were made up purely of file-mapped memory regions, then restarting it might be conceptually simple as you would merely need to remap all of the files back to their original virtual addresses. Unfortunately, this would not fully capture the essence of an application if the file mappings were privately modified or if anonymous memory regions were used. Of course, all applications fall into the latter case by making use of stack and

heap space, which are always anonymous memory regions. As HPC applications make use of large amounts of private memory via the stack and heap, anonymous memory provided by the kernel is extensively utilized.

Internally, the kernel provides applications with anonymous memory virtual pages by allocating them from previously unused physical memory as needed. The physical to virtual layout and structure is stored within the kernel and is volatile should the kernel be rebooted, lost, or corrupted. Anonymous pages provided by the kernel need not be linear in physical memory, and they are impossible to "recover" without the volatile mapping stored within the kernel. Beyond anonymous memory used by processes, another frequently used type of volatile memory is $tmpfs$, which provides a 100% in-memory RAM file system. Although $tmpfs$ is entirely stored in physical memory, it is incapable of surviving a kernel restart because the mapping that describes both the layout and contents of the file within it are not backed by any permanent medium. In fact, all requests for storage within $tmpfs$ come from the kernel's page cache, which cannot be reconstructed without knowledge of the physical to virtual mappings as well as various data structures that make up the file nodes within a $tmpfs$ file system, just as the aforementioned anonymous memory. Ultimately, without a perfect — and valid — copy of the kernel's live data structures, it is not possible to recover the contents of $tmpfs$.

5.4.2 PRAMFS: Protected and Persistent RAM Filesystem

As this work will show, the need for a 100% in-memory RAM filesystem that can survive a kernel crash and reboot becomes obvious as this work investigated a method to persist anonymous and/or private memory regions across kernel failures without the need for perfect kernel data structure state.

Originally, the Linux kernel had no support for non-volatile RAM filesystems. The use of the Linux Page Cache inherently restricted the widely used $tmpfs$ to a lifetime of a single boot since the page cache's contents are discarded upon reboot. Meanwhile, growth in embedded systems with attached non-volatile memory drove the development of non-volatile RAM (byte-addressable NVRAM) filesystems that exist within memory independent of the running kernel's state. A persistent memory filesystem for NVRAM may be best used by directly mapping the physical memory of the NVRAM and accessing it using memory read/write operations. In contrast to a block device such as spinning disks, a persistent filesystem is able to read and write its contents like native memory. Further, any files that are opened and then mapped into a process' address space can be accessed directly instead of through the kernel's page cache. The direct I/O bypassing the page cache is a key difference: Traditionally, all writes to file-backed memory

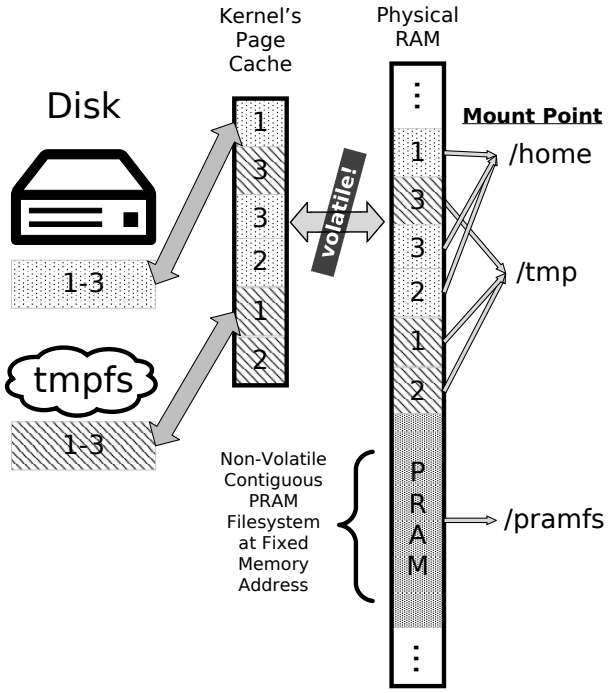


Figure 5.1 PRAMFS vs. Page Cache Dependent Filesystems

mappings spend at least a small amount of time residing in the kernel's page cache. During the window of write operations residing in the kernel's page cache prior to flushing to disk, they are considered volatile. Subsequently, if the kernel crashes or reboots then any unflushed changes would be lost.

The PRAMFS Filesystem [4] is a persistent NVRAM filesystem for Linux that is capable of storing both the filesystem's metadata as well as contents 100% in-memory while bypassing the Linux Page Cache. A PRAMFS partition can be utilized on volatile RAM as well as NVRAM, although the lifetime of a PRAMFS partition would then be restricted to the duration of power being applied to the RAM (i.e., a PRAMFS filesystem in RAM cannot survive once the power is turned off). PRAMFS can be combined with file mapping in processes as well as execute-in-place (XIP) support to allow for a process' file-backed mappings to not only read/write directly to physical memory but also execute directly from PRAMFS [60]. Since memory mappings to PRAMFS are one-to-one with physical memory, an application whose executable, data, stack, and heap are mapped to a file based in PRAMFS would, in fact, be entirely executing, reading, and writing within the PRAMFS storage without any volatility should the process be terminated at any point. To put this another way, if all of a process' memory was mapped to PRAMFS files

and the process was terminated at any arbitrary point, then the image in PRAMFS would be very similar to a process' core dump, even if this work completely erases all buffers within the kernel.

Figure 5.1 demonstrates how the page cache interacts with traditional disk storage file systems, in-memory $tmpfs$, and PRAMFS. Since disks are block based devices, to provide memory mapped I/O to the contents of a file on disk the kernel first reads the contents of a file off the disk into the page cache. In the figure, 3 pages worth of content from disk are shown loaded into the page cache. Since the page cache resides in physical RAM, these pages are drawn twice in the figure. A subsequent modification to the file's contents via memory mapped I/O would be performed by changing the contents of the page cache, which eventually are flushed back to disk; however, this flush is not immediate. Next, this work also sees that a $tmpfs$ has 3 pages of data. Unlike the disk example, the 3 $tmpfs$ pages exist only in the page cache and thus physical RAM. Any change to $tmpfs$ files is immediate as there is no backing store to flush to. However, since the page cache allocates pages in a non-deterministic order, the locations of pages 1-3 are not contiguous. Finally, this work observes PRAMFS, which has allocated a physically contiguous region of memory. Since its size and location are always constant, anyone can read its contents without any external state. Additionally, since PRAMFS does not have a backing store besides physical RAM, any changes to it are immediate. The volatility of PRAMFS, in this example, is only dependent upon keeping the RAM powered on and not becoming overwritten.

5.4.2.1 Protecting PRAMFS from Spurious Writes During Faults

While enumerating the outcomes of all possible software faults, it is entirely possible to reason that some types of faults may result in spurious writes to parts of memory that were indirectly related to or referenced by the functionality of the faulting code. For instance, imagine a corrupted pointer that is written to as it is dereferenced, or perhaps a write operation that overflows beyond its allowed boundary. In both cases the end result is undesired memory changes to arbitrary locations. As this work will show, mini-ckpts assumes that the memory stored in PRAMFS is guarded by userspace protection mechanisms, such as ABFT, software fault tolerance, redundancy, or some other means outside the scope of operating system resilience [58, 46]. Whether or not an application's temporary memory is stored within PRAMFS or anonymously mapped pages, there is always an inherent risk that the data may become corrupt as part of the side-effects of a failing kernel. However, this work will point out that studies have shown that toggling the write bit in the page tables in an as-needed/on-demand basis may be used to trap and prevent unintended writes. This technique could easily be applied to mini-ckpts, but it will come at a

cost. Otherworld investigated the use of write-protected page tables to protect userspace memory and reported a runtime overhead between 4%-12% for this additional protection [31]. Other works that investigated protecting NVRAM filesystems also found that page table protection mechanisms result in 2x to 4x runtime overheads for applications that experience frequent I/O operations, which in terms of kernel fault tolerance could be directly translated to the frequency of kernel entries/exits invoked via hardware interrupts or user system calls [52, 53]. mini-ckpts does not implement page protection mechanisms. However, the underlying PRAMFS used by mini-ckpts does support write protection if desired. Interestingly, it was found that even the page table protection technique does not block spurious writes during a failure between 1.5%-2.2% of the time, which further motivates the need for applications to provide their own data integrity protection mechanisms [52].

5.4.3 Preparing a Process for mini-ckpts

Processes that are protected by mini-ckpts must undergo transformations that modify the backing store for all of their virtual memory. Consider a simple program such as the one shown in Listing 5.1. This "hello world" program makes use of a stack, heap, data, executable code, and shared libraries. Once running, the memory mappings of this process can be observed to confirm that both anonymous and file-based mappings are indeed active, as shown in Listing 5.2.

Listing 5.1 Hello World

```
#include <unistd.h>
char str[] = "hello\n";
int main()
{
  write(STDOUT, str, 6);
  while(1)
    ; /*noop infinite loop*/
  return 0;
}
```

Listing 5.2 Hello World Before Remapping

```
1 00400000-00401000           r-xp  /hello
2 00600000-00601000           rw-p  /hello
3 77beef00000-77beef01000     r-xp  [anonymous][1]
4 7ffff764a000-7ffff764c000   r-xp  /libdl.so
5 7ffff764c000-7ffff784c000   ---p  /libdl.so
```

[1] 0x77beef00000 is a special executable region added by mini-ckpts shown in Listing 5.4.

```
6  7ffff784c000-7ffff784d000 r--p  /libdl.so
7  7ffff784d000-7ffff784e000 rw-p  /libdl.so
8  7ffff784e000-7ffff79d0000 r-xp  /libc.so
9  7ffff79d0000-7ffff7bd0000 ---p  /libc.so
10 7ffff7bd0000-7ffff7bd4000 r--p  /libc.so
11 7ffff7bd4000-7ffff7bd5000 rw-p  /libc.so
12 7ffff7bd5000-7ffff7bda000 rw-p  [anonymous]
13 7ffff7bda000-7ffff7bdc000 r-xp  /libcr_run.so
14 7ffff7bdc000-7ffff7ddb000 ---p  /libcr_run.so
15 7ffff7ddb000-7ffff7ddc000 r--p  /libcr_run.so
16 7ffff7ddc000-7ffff7ddd000 rw-p  /libcr_run.so
17 7ffff7ddd000-7ffff7dfd000 r-xp  /ld.so
18 7ffff7ff1000-7ffff7ff4000 rw-p  [anonymous]
19 7ffff7ff8000-7ffff7ffa000 rw-p  [anonymous]
20 7ffff7ffa000-7ffff7ffc000 r-xp  [vdso]
21 7ffff7ffc000-7ffff7ffd000 r--p  /ld.so
22 7ffff7ffd000-7ffff7ffe000 rw-p  /ld.so
23 7ffff7ffe000-7ffff7fff000 rw-p  [anonymous]
24 7fffffffde000-7ffffffff000 rw-p  [stack]
25 ffffffffff600000-ffffffffff601000 r-xp [vsyscall]
```

Listing 5.3 Hello World After Remapping

```
1  00400000-00401000           r-xs  /pramfs/temp001
2  00600000-00601000           rw-s  /pramfs/temp002
3  77beef00000-77beef01000     r-xs  /pramfs/temp003
4  7ffff764a000-7ffff764c000 r-xs  /pramfs/temp004
5  7ffff764c000-7ffff784c000 ---s  /pramfs/temp005
6  7ffff784c000-7ffff784d000 r--s  /pramfs/temp006
7  7ffff784d000-7ffff784e000 rw-s  /pramfs/temp007
8  7ffff784e000-7ffff79d0000 r-xs  /pramfs/temp008
9  7ffff79d0000-7ffff7bd0000 ---s  /pramfs/temp009
10 7ffff7bd0000-7ffff7bd4000 r--s  /pramfs/temp010
11 7ffff7bd4000-7ffff7bd5000 rw-s  /pramfs/temp011
12 7ffff7bd5000-7ffff7bda000 rw-s  /pramfs/temp012
13 7ffff7bda000-7ffff7bdc000 r-xs  /pramfs/temp013
14 7ffff7bdc000-7ffff7ddb000 ---s  /pramfs/temp014
15 7ffff7ddb000-7ffff7ddc000 r--s  /pramfs/temp015
16 7ffff7ddc000-7ffff7ddd000 rw-s  /pramfs/temp016
17 7ffff7ddd000-7ffff7dfd000 r-xs  /pramfs/temp017
18 7ffff7ff1000-7ffff7ff4000 rw-s  /pramfs/temp018
19 7ffff7ff8000-7ffff7ffa000 rw-s  /pramfs/temp019
20 7ffff7ffa000-7ffff7ffc000 r-xp  [vdso]
```

```
21 7ffff7ffc000-7ffff7ffd000 r--s /pramfs/temp020
22 7ffff7ffd000-7ffff7ffe000 rw-s /pramfs/temp021
23 7ffff7ffe000-7ffff7fff000 rw-s /pramfs/temp022
24 7fffffdd000-7fffffff000 rw-s /pramfs/temp023
25 ffffffffff600000-ffffffffff601000 r-xp [vsyscall]
```

As mini-ckpts' goal is to ultimately preserve process state in the event of a kernel crash, it is imperative that the contents of all memory mappings are safely retained. However, by looking at the memory mappings from Listing 5.2, it can be seen that there are several regions of memory that will be lost during a kernel crash, mostly due to their dependence on the volatile page-cache:

- Line 2: Copy-on-write (CoW) mapping to the data section (initialized variables) of the hello world executable. This includes the string "hello\n". Changes would permanently reside in the page cache.
- Lines 7, 11, 16, 22: All CoW mappings to shared library data.
- Lines: 12, 18, 19, 23: All private data mappings.
- Line 24: The stack is also a private, anonymous data mapping and thus volatile.

In fact, the only "safe" regions of the process are the ones loaded from disk and never modified. Those recovered easily are lines 1, 4, 6, 8, 10, 13, 15, 16, 17, and 21 because they are marked read only and have a file-based backing. Of course, if the process were to ever mark them read/write and then modify them, then the changes would be volatile. In practice these mappings do not undergo modification normally, and so it is safe to say that all written memory by this process would be lost if the kernel crashes in its current state, unless there was some way to recreate the virtual to physical mappings of the process at the moment before the kernel failed.

To provide mini-ckpts with an always consistent, non-volatile backing store for its protected applications, this work designed a technique to migrate all memory mappings to PRAMFS. The goal of this migration is to move all file-backed and anonymous memory regions to shared memory mappings of PRAMFS files so that the process' memory may survive a kernel crash. To achieve this, this work added functionality to the kernel that pauses all running threads of a process, iterates through each virtual memory mapping of the process, and either copies or moves each memory mapping to a new, distinct file in PRAMFS. For each memory mapping this work created a new file in PRAMFS named /pramfs/tempXXX where XXX is monotonically increasing for each new file. The new file is sized to match each distinct memory region. Then, for each region this work copies the contents of the existing memory to the file, unmap the old

region, and then remap the new PRAMFS file into the exact same location with the exact same combination of read, write, and/or execute flags enabled. The one key change this work makes to all memory regions is that they will be mapped back as shared instead of private, which ensures that future writes are written immediately back to the PRAMFS physical memory backing. Listing 5.3 shows what the virtual memory of the hello world program looks like immediately after the migration to PRAMFS. The VDSO and vsyscall regions are managed by the operating system and do not need to be migrated to PRAMFS. It is important to note that once the PRAMFS files are virtually mapped into a process then there are no further dependencies within the kernel to maintain the mappings. The hardware accesses the page tables directly, and even a failing kernel would not necessarily affect an application's direct access to its memory as there is no buffering or caching taking place between the application, kernel, and physical memory.

5.5 Initial Checkpointing

For an application to be protected by mini-ckpts, it must first undergo an initial checkpoint. An initial checkpoint performs two major functions. First, it remaps all memory into shared memory that is backed in PRAMFS. Second, it stores a serialized version of the process' state.

The checkpoint can be triggered either locally or through another process. An application linked with a userspace mini-ckpts library is provided with an API that allows synchronous checkpointing of itself. Alternatively, a mini-ckpts library will install a signal handler that allows a remote process to trigger a checkpoint by sending the application a specific signal. In either case, the process by which an initial checkpoint is taken proceeds in the same manner.

The initial checkpoint will make critical modifications to an application, such as modifying its memory mappings. To ensure that the application is not running during this time, mini-ckpts interrupts all threads of the target process with a signal and invokes a system call within the threads' signal handlers. The system call traps all threads in the kernel and puts them to sleep until the checkpoint is complete.

Once within the kernel, mini-ckpts remaps all process memory to PRAMFS, as previously described. All memory regions except the VDSO and vsyscall are individually mapped to separate files in PRAMFS. However, since memory regions may be several gigabytes in size, which would exceed PRAMFS' limits, mini-ckpts chunks large contiguous mappings into separate files no larger than 512MB.

After remapping process memory, mini-ckpts records process state information to a separate persistent region of memory. This recording largely mirrors the BLCR checkpoint library to store information such as open file descriptors, threads, virtual memory layout and protections,

process IDs, signal handlers, etc. Since this work borrows from BLCR for functionality relating to basic process state, a full list of supported kernel state is available in [36] while key features are condensed in Table 5.1. Although this work supports file descriptors and memory mapped files, mini-ckpts does not currently track changes to the current pointer (seek location) in open files. The most notable absence in both BLCR and other checkpoint libraries is support for TCP sockets during a restart. This work specifically addresses this for MPI applications later in this work.

Before allowing the application to resume, mini-ckpts marks each running thread as tracked within the kernel. This will enable mini-ckpts to capture the registers of this thread every time it is interrupted by the kernel. In this way, the kernel will always maintain a safe copy of each thread's registers prior to a possible kernel crash. This is a key contribution of mini-ckpts as it allows the threads to restart immediately without any loss in computation if the running kernel fails.

5.6 Capture and Restoration of Registers

A mini-ckpt must be taken at any time a thread loses control over a processor. From a userspace perspective, a thread's loss of control of a processor is either transparent or voluntary through a system call. This work first broadly categorize the types of transitions that take place: (1) system call, (2) interrupt, (3) non-maskable interrupt. Although different in mechanism, each type of transition makes a guarantee as to the state of the processor and registers when control is returned to the interrupted process. All hardware interrupts guarantee that the state of the registers will be exactly as they were prior to the interrupt occurring[2]. From the process' perspective, the interrupt executes transparently. On x86_64, which is our focus, system calls are the second source of transition from userspace to the kernel, and they may change the state of registers when control is returned to the process. Even though some registers change, it is part of an API contract with the kernel. For example, the kernel guarantees that certain registers will be used for passing arguments and returning error codes back from the kernel as a result of the system call. Other registers are guaranteed to be left untouched as part of the system call.

Differentiating between a regular interrupt and non-maskable interrupt is critical to understanding how mini-ckpts succeeds in taking a mini-ckpt at any point during the lifetime of execution. A regular, maskable, interrupt is triggered by hardware when information is available for the kernel, and a software interrupt is triggered by a process when it needs to signal the

[2]Note that software interrupts may change the state of the registers if they are used in place of the SYSCALL/SYSRET instructions. Using *int* $0x80$ was standard practice in x86 prior to x86_64.

kernel. In some critical parts of the kernel it may be necessary for a kernel thread to ensure that execution of some region of code completes without interruption. The hardware provides a bitmask that may be used to temporarily disable any interrupts from occurring. This is commonly found in areas of the kernel that need to be non-preemptive during synchronized operations. However, this raises an interesting question: What happens when the hardware encounters an error, or when there is a software bug such as a stuck processor? If interrupts are disabled when the bug occurs, then it becomes impossible to receive critical processor notifications or to "unstick" a hung thread. Non-maskable interrupts offer a recourse in this case by providing specialized interrupts that are not affected by the normal CPU interrupt bitmask. The key difference between these interrupts is that non-maskable interrupts may be triggered at any time and cannot be prevented by the kernel. In either case, once the interrupt completes, control is returned to the userspace process that was interrupted without any change in register state.

From a programmer's perspective, control appears to be yielded under a diverse set of circumstances such as signals, scheduling, sleeping, synchronization (waiting for mutual exclusion), and both blocking and non-blocking[3] system calls. However, each and every one of these scenarios resides on the building blocks of the system call instruction and hardware/software interrupts. This is why the only locations that require kernel-level instrumentation in order to save the registers of an interrupted process are the entry points to both system call and interrupt handlers within the kernel.

5.6.1 Implementing mini-ckpts at the Interrupt and System Call Level

In Linux, the entry points for all interrupts and system call instructions reside within the *entry_64.S* file. Preparing the kernel to handle an interrupt or system call is complex and requires a hand-coded assembly routine to construct a valid state for the kernel prior to jumping to any C-coded functionality. Interrupts and system calls are handled slightly different:

The entry to an interrupt requires setting up the kernel stack, detecting if the interrupt was a double interrupt (interrupt during interrupt), and by default saving all general purpose registers to the kernel's stack. Once the interrupting C handler has completed, the handler returns back to the assembly routine where the processor's registers are returned to their original state. Control finally returns back to userspace using a special interrupt return instruction. Since mini-ckpts inherently needs to save the general purpose registers while entering the kernel, an

[3]Non-blocking system calls do not exist, but rather a system call may be asked to return immediately if it would normally block.

interrupt already provides most of the functionality needed. mini-ckpts simply calls a routine that copies the saved registers to the mini-ckpt location in persistent memory. If a kernel crash occurs during the handling of an interrupt, then the most recent register state is safely stored. Although not discussed at length, the state of the floating point registers is also copied whenever the general purpose registers are saved.

System calls are handled in a slightly different manner to interrupts due to the potential performance gains by ignoring (skipping over) general purpose registers that actually contain values as arguments to the requested system call. For example, the registers $\%rdi, \%rsi, \%rdx, \%r10, \%r8, \%r9$ are all potentially used for system call arguments. $\%rax$ is used to indicate the system call number as well as the return value. Additionally, $\%rcx$ is clobbered as it stores the return address (RIP) to userspace, and $\%r11$ is clobbered in order to save the processor's flags (RFLAGS). Figure 5.2 summarizes the use of these registers. To make the unmodified Linux kernel more efficient, the normal assembly entry routine simply bypasses the act of saving these registers initially. mini-ckpts depends on saving the state of all registers, so the entry point for system calls required modifications to allocate kernel stack space for the complete set of registers as well as to both save and restore the registers during the system call entry and exit points. These modifications required close examination and consideration to be correct, namely because the assembly routines have many code paths and exit points, depending on the nature of the system call invoked, the kernel's determination on whether to reschedule the process, and whether the process will even exist at the completion of the system call due to an $exit()$ or termination signal. Only once the entire set of general purpose registers is safely captured may the kernel begin processing the system call normally.

If a system call ultimately results in a kernel crash, then mini-ckpts will later restore the process' registers, but it still needs to consider what to do about the system call that was being attempted. This work provides two solutions:

- Return to the instruction immediately after the $SYSCALL$ instruction. Simulate the interruption of a system call by setting the register containing the system call return value to $EINTR$, which is a standard Linux return value indicating that the kernel was interrupted and that the user process should try to repeat the system call.

- Attempt to repeat the system call using the exact same arguments immediately after restoring the thread.

From our observations, this work recommend the former approach because (1) best programming practices require developers to check and handle the return values from system calls, (2) libraries

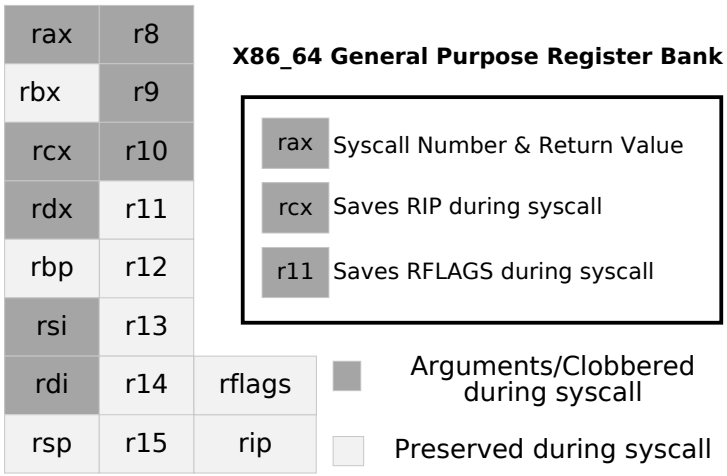


Figure 5.2 In x86_64, unlike interrupt handlers which save all general purpose registers, many are either clobbered or used as system call arguments when the *syscall* instruction is executed.

may already repeat the call for the programmer if they detect $EINTR$, and (3) if the system call is repeated automatically by mini-ckpts, then it risks potentially containing a handle to an object that no longer exists (such as a socket) after a restore. A programmer or supplemental library aware of mini-ckpts could restore such handles before attempting the system call again.

The aforementioned requirement on storing system call argument registers brings up an interesting discussion: If the registers used for system call arguments are considered volatile in the perspective of the user space process, then why does mini-ckpts need to store them at all? The answer resides in the observation that not all system calls require using all available registers as parameters. For example, the *exit* system call requires only one argument — a return code. The *select* system call uses 5 arguments. Since the operating system only clobbers registers used for arguments (plus $\%rax$, $\%rcx$ and $\%r11$), a compiler will assume that any registers unused during a system call will remain unchanged. So if this work only partially restores registers after a kernel crash during a system call that only used one register as an argument, then the other registers will now contain garbage values that the compiler expected to have been preserved across a system call.

5.6.2 Restoring Registers during Restart

For each thread of a process that is restored after a restart, its general purpose registers must be returned to their state immediately prior to the previous failure. A helper routine is used to recreate each thread using standard *clone* system calls, and then each newly created thread

requests its registers to be restored through a system call via *ioctl* to mini-ckpts. As this work sees in Figure 5.2, a system call will always return with several registers clobbered, which is an architectural limitation of the *syscall* instruction. As a solution, this work developed a trampoline technique that uses a combination of the userspace stack and injected code to simulate a transparent restart. This trampoline approach was chosen because it has few architectural dependencies which are available on all common platforms: (1) the ability for a system call to return to a specific address, and (2) the potential for the kernel to modify the virtual memory of a process.

Very similar to a *setjmp/longjmp* approach, our technique depends on using the stack to feed registers to an injected trampoline code:

- mini-ckpts injects a small region of executable helper code into the process being restored.
- A new thread is created for each one lost during the failure.
- mini-ckpts is requested to restore a thread's registers via a system call.
- The userspace stack pointer is retrieved from the registers stored during the system call entry.
- mini-ckpts advances the stack pointer as it writes the values of all previously saved register values directly to the userspace stack.
- mini-ckpts modifies the kernel's saved return instruction pointer to a region of injected code at 0x77beef00000. The system call will now return to a new location.
- The injected code (shown in Listing 5.4) restores each register by popping it from the stack, including the saved flags. The final *retq* instruction returns execution back to the point immediately prior to the previous kernel failure.

Listing 5.4 The mini-ckpts trampoline code is a userspace counterpart that enables a process to retrieve and restore its previous register values from the stack, provided that they were placed there by the mini-ckpts kernel component during a restore.

```
/* Another version could call a user-space
 * library here to notify that a restart is
 * in progress.
 ... call notify_restart ...

 * Note: %rsp was set by the kernel,
 * which makes the stack pointer usable.
```

```
 */
popq %r15
popq %r14
popq %r13
popq %r12
popq %r11
popq %r10
popq %r9
popq %r8
popq %rbp
popq %rax
popq %rbx
popq %rcx
popq %rdx
popq %rsi
popq %rdi
popfq /* restore FLAGS */
retq /* restore last instruction pointer */
```

Once the trampoline finishes restoring the thread's registers, the process is unable to discern that it was interrupted at all except for time measurements spanning between the last mini-ckpt during failure, restart, and time up until the *retq* completes in the trampoline code.

5.7 Handling Kernel Faults (Panics)

A failure in the kernel should result in a software fault handler known as a kernel panic. Within the Linux kernel, the default action of the panic handler is to attempt a backtrace of the current stack, print the current registers, print an error message if provided, and then halt the system. A kernel panic is one of many types of faults including memory hardware failure, software bugs, or assertion failures within kernel code. Depending on the severity of the fault, it is impossible at times for the kernel to even perform its final debugging print messages prior to halting. Additionally, in multicore (SMP) systems, the panic handler is tasked with attempting to halt all other processors.

5.7.1 Warm Rebooting

The Linux kernel provides a feature known as *kdump*, which is a mechanism wherein the panic handler attempts to perform a warm reboot into a secondary crash kernel and simultaneously preserve a copy of its own memory for later observation and debugging from the crash kernel.

Because the intent of a crash kernel is merely to provide the most basic services in order to inspect or save the memory of the previous kernel, the system is warm booted in an unusual configuration: SMP is unavailable, and the core on which the new kernel is running is typically the same core that previously experienced a kernel panic. For the purposes of HPC applications, any warm reboot would require a stable system with all SMP cores available. Once in this state, recovering would normally require a full reboot through the BIOS.

To provide a fully functional SMP system after a kernel panic, mini-ckpts implements a migrate-and-shutdown protocol that is used during a failure. First, since a failure may occur on any core, mini-ckpts installs a specialized NMI (Non-Maskable Interrupt) handler once a panic is detected. Next, since other cores running in parallel may not yet be aware of the failure, the failing core sends an NMI interrupt to all other cores. This NMI cannot be ignored, and it forces all cores to immediately jump to the NMI handler. As part of the NMI handler, any mini-ckpts protected processes have their registers saved as the NMI handler is executed. Once in the NMI handler, core 0 takes the lead (if it was not already the first core to panic) since it must be the first CPU to perform a warm reboot. Once core 0 has detected all other cores to have signaled that they are halted, core 0 begins unpacking and relocating a fresh copy of the kernel, sets up basic page tables, passes memory mapping information for the PRAMFS and persistent regions, and jumps to the entry point of the new kernel. It has essentially performed an emergency shutdown of all cores and then completed the same steps a traditional bootloader would do to start an operating system.

5.7.2 Requirements for a Warm Reboot

Minimizing the data structures required to perform a warm reboot is critical to mini-ckpts' success. The code paths needed for a successful warm reboot begin at a call to *panic*. The paths to reaching *panic* are very broad but may include dereferencing invalid pointers or assertion failures within the kernel. Once panic is reached, mini-ckpts ensures that no further memory allocation will be required. Its dependencies then involve functions that typically execute between 1-5 instructions, such as shutting down the local APIC, high performance timers, and then signaling NMI interrupts for all cores. For NMI interrupts to work, we must also assume that the code paths and interrupt vectors themselves remain valid.

Within the NMI handler, this work must assume that the per-CPU interrupt stacks remain available. This also assumes that hardware registers were not corrupted as part of the failure. Once in the NMI handler, if the panic has interrupted a mini-ckpts protected task, then this work must be able to access a 64-bit pointer in the thread's $thread\_info$ struct. It is fine if any

other thread or task related members have been corrupted. The single pointer both indicates (if non-zero) that the thread is protected and, in doing so, points directly to where the interrupted threads' registers should be saved in persistent memory. Locating the $thread\_info$ struct is fortunately trivial if the thread's registers are valid since it is calculated based on the kernel's stack pointer.

Finally, this work must also assume that the persistent region of memory where mini-ckpts stores a serialized copy of the protected process remains safe. The checkpoint itself is typically under 100KB, which makes it a small target relative to today's main memory size. Additionally, software correction codes could be applied to the persistent memory if desired.

5.8 Supporting MPI Applications

Traditional checkpoint/restart tools do not support automatic reconnection of network sockets for applications during restart. Instead, these tools usually provide a callback procedure for applications that moves the burden (and programming logic required) to the developer, if they wish to support the checkpoint restart paradigm. There are a number of reasons why network connections are not automatically restored (i.e., new IP addresses and reconnection ordering), but from mini-ckpts' perspective, restarting a lost network connection is not a desirable feature: Unlike a traditional checkpoint/restart, when mini-ckpts saves a process from a panic, it must forgo saving any potentially corrupt state in the kernel. Kernels typically buffer both file I/O and network traffic. This means that an application may believe that it has successfully sent data over the network, but the data remains in the kernel for some time, which leaves it potentially volatile should a kernel crash occur. As a result, MPI implementations that do support checkpoint/restart may require network traffic to quiesce and all buffers drained before a checkpoint is taken. Alternatively, message logging may be used in tandem with the checkpoint. Moreover, MPI implementations consider the loss of communication with a peer process as a critical error. This means than an entire job will terminate and all unsaved computation will be lost should any type of unexpected communication failure occur.

mini-ckpts is designed for HPC workloads, and to provide run-through fault tolerance with mini-ckpts and MPI applications, this work designed librlmpi to specifically handle transient network failures, network buffer loss, and incomplete message transmission, which all may occur during a local or remote (peer) kernel failure. librlmpi is both ReLiable in that is supports handling lost messages either on the network wire or kernel buffer, and it is libeRaL in that it can tolerate a network failure during any point of its execution or any of its system calls.

5.8.1 librlmpi Internals

Internally, during typical execution, librlmpi depends on *poll*, *writev*, and *readv*. It uses a threaded progress engine that monitors the return values of system calls. It can detect when mini-ckpts has transparently warm rebooted the system by watching for an $EINTR$ system call error, which was discussed earlier. Upon detection of a warm reboot, librlmpi resets any message buffers in transit (pessimistically assumes they failed), and reestablishes connections with its peers. Likewise, a peer that has not undergone a failure is able to accept a new connection from a lost peer and resume communication where it left off. Any messages that were cut off during a failure are restarted. Any data that had been buffered in kernel send or receive buffers will be resent, as librlmpi employs positive acknowledgements between peers.

This work uses internal message IDs to uniquely identify and acknowledge transmissions between peers. After a failure, the peer node that failed (and subsequently warm rebooted) begins its restoration by reestablishing connections with its peers. Upon connection, both peers send recovery details on the last messages they received even if they were not acknowledged earlier. This allows each peer to resend only the messages that were either (1) not yet sent, or (2) lost in the kernel buffer or network to be resent. During normal execution and during a recovery, any message IDs that preceded the most recent acknowledgement are marked as complete (i.e., MPI_Wait or MPI_Test finish) and the space occupied by the message may be reused, per the normal MPI interface.

librlmpi employs the equivalent of a ready-send transmission protocol. Although ready-send semantics are not enforced at the application level, librlmpi will not transmit an outgoing message until its peer has sent a receive envelope indicating it is ready and has a buffer available to receive the message. If an MPI application attempts to send a message prior to a receive being posted, the transmission will be delayed until the receive is posted.

Overall, librlmpi combined with mini-ckpts *transparently* allows an MPI application to be interrupted by a kernel panic and resumes execution without loss of progress. It does not require any modification to an MPI application. If an MPI application wishes to begin its mini-ckpts protection automatically (instead of an external signal), then a one line function call may be added to the application.

5.8.1.1 Message Verification and Debugging

librlmpi uses two layers of message envelopes during a transmission: An outer layer identifies the incoming message's ID as well as the most recent acknowledgement from the peer. An inner layer contains MPI message details such as tag, datatype, and size. An optional message payload

(data) follows the inner layer. Both layers may optionally be protected by red zones (which will increase message size overhead) to verify that proper protocol is being followed. In the inner envelope, librlmpi may optionally include a hash of the message payload, if the MPI application would like addition message protection beyond that of the underlying communication layer.

5.8.2 MPI and Language Support

librlmpi supports basic MPI peer to peer communication, linear collectives (including Alltoall, Alltoallv, Allreduce, and many more basic ones), a profiling layer, and both C and Fortran bindings. Additionally, librlmpi provides specialized routines to allow an MPI application to easily begin a checkpoint and protection for itself with mini-ckpts, and it may also send a trigger (for testing) that will inject a kernel fault resulting in a panic and warm reboot recovery.

5.9 Experimental Setup

mini-ckpts is evaluated in both physical and virtualized environments to demonstrate mini-ckpts effectiveness and application performance overheads. Fault injections will be provided as callable triggers that deliberately corrupt kernel data structures or directly invoke a kernel panic. mini-ckpts is evaluated on both OpenMP parallelized applications and MPI-based applications using a prototype MPI implementation, librlmpi, that is designed to handle network connection loss at any point during execution.

Our test environments are (1) 4 nodes with AMD Opteron 6128 CPUs, (2) 1 node with an Intel Xeon E5-2650 CPU, and (3) KVM virtualized environments running on the same AMD and Intel hosts. The first two configurations will be referred to as bare metal. The bare metal environments are all booted in a diskless NFS root environment. The virtual machine environments use a "share9p" root filesystem with their hosts, which, for practical purposes, is similar to a diskless NFS root for the virtual machine. In all configurations this work provides 128MB of memory for storing persistent checkpoint information (although typically <100KB is used) and an additional 4GB of memory for the PRAMFS partition, which is where the memory of protected processes will reside during execution.

All systems have been configured conservatively to optimize boot time: the system's boot order prioritizes booting directly to our custom kernel, which bypasses network controller initialization and any potential scanning for other boot options. This is important since other cluster environments may have even longer boot times than our experimental setup, which would only serve to enhance the reported gains of mini-ckpts during a warm reboot vs. cold boot.

Benchmark	BT	CG	EP	FT	IS	LU	MG	SP	UA
Class	A	B	B	B	C	A	B	A	A

Table 5.3 Class Sizes Used for NPB OpenMP Version — Thread Count is 8

App.	NPB CG	NPB EP	NPB IS	NPB LU	PENNANT	Clover Leaf
Input	C	C	C	B	sedov (400 iter.)	clover_bm2_short.in

Table 5.4 Class Sizes Used/Input for MPI Benchmarks — Processor Count is 4

The experimental operating system modified for mini-ckpts is based on Linux 3.12.33, and our modified userspace checkpoint library is a branch of BLCR [36] version 0.8.6_b4. Our bare metal and virtual machine experiments both use the same root environment based on Fedora 22. The kernel is configured for loadable module support, but all drivers, etc. are compiled directly into the kernel to tune it for a cluster environment. The only modules that are loaded at runtime are our modified BLCR module and small modules that are used to inject faults during our evaluation. As a result, when this work evaluated boot times and methods, it omitted the use of an initial ramdisk. Our hypervisor is qemu-kvm 1.6.2, and this work uses the share9p filesystem for its rootfs (conceptually, it is similar to a network filesystem, but for virtualized guests and their hypervisor), instead of a typical file-based disk image.

The NAS Parallel Benchmarks (NPB) OpenMP Version [10] is used to evaluate the effectiveness of mini-ckpts in a multithreaded environment. NPB DC is excluded as it depends on heavy disk I/O. The class sizes used are listed in Table 5.3.

5.9.1 librlmpi: An Experimental MPI Implementation for mini-ckpts

To evaluate the runtime overhead associated with librlmpi and librlmpi in combination with mini-ckpts, this work will use the NAS Parallel Benchmarks, MPI Version [10]. The CG, IS, LU, and EP benchmarks will be evaluated, while others are excluded due to their dependence on MPI functionality not implemented within our librlmpi prototype, such as MPI_Comm_split. librlmpi is additionally evaluated with PENNANT version 0.7, an unstructured physics mesh mini-application from Los Alamos National Laboratory [3]. The Clover Leaf mini-application version 1.0 from Sandia National Laboratories, an Euler equation solver, is also evaluated [1]. All inputs used are listed in Table 5.4. The OSU Micro Benchmarks version 4.4.1 is used to evaluate librlmpi's peer to peer latency, bandwidth, and collective operations [2]. Open MPI version 1.8.5 is included in the experiments to compare the performance of a mainstream MPI implementation against our prototype implementation, librlmpi, and to establish any relative

(measured in seconds)	BIOS Boot Time	**Kernel Boot Total**	Network Driver & NFS-Root Mounting	Kernel Misc	Software Stack Total	Cold Total w/ BIOS	**Warm Reboot Total**
AMD Bare Metal	37.4	**5.3**	1.5	4.8	0.7	50.3	**6.0**
Intel Bare Metal	50.8	**6.7**	3.0	3.7	0.7	73.0	**7.4**
AMD VM	—	**0.8**	< 0.2	< 0.6	3.0	—	**3.8**
Intel VM	—	**0.7**	< 0.2	< 0.5	1.3	—	**1.9**

Table 5.5 Observed Times for Both Cold and Warm Booting a mini-ckpts-enabled System

difference. All MPI benchmarks are run across four AMD Opteron 6128 nodes using gigabit ethernet for communication.

5.10 Results

This work first measures the time required for a full reboot on both a bare metal and virtualized systems. Table 5.5 shows the costs of both booting and warm booting. The cost of booting measured from the BIOS to the stage that the bootloader is reached is approximately 37 and 50 seconds on our bare metal test systems. As explained in the experimental setup, the systems' boot order was optimized by directly booting to our kernel. Once the bootloader is reached and the kernel begins executing, the boot process requires between 5-7 seconds before the kernel is fully initialized. Of this, device initialization accounts for any single majority of the time. The network card drivers requires between 1-3 seconds to initialize (even though IP addressing is statically configured vs. DHCP and the addressing is configured by the kernel instead of userspace utilities). The remaining time is spread out among the initializations of various kernel subsystems and other devices that are configured relatively quickly. In virtualized environments on the same systems as the AMD and Intel bare metal measurements, this work observes that the same kernel boots in less than one second. The primary difference is the lack of slowly initializing hardware devices: The ethernet driver required less than 0.1 seconds to complete, and the disk driver (a share9p filesystem shared with the host device) completed initialization and mounting in under 0.2 seconds. mini-ckpts never requires a full reboot in either a virtualized or bare metal environment regardless whether or not it is activated to simply rejuvenate a kernel or in response to a failure. The important metric to consider is the difference in boot times excluding the BIOS, which is between 5-7 seconds vs. approximately two to four seconds for a virtual machine (VM). This number represents the approximate downtime incurred by a kernel panic when combined with the cost of booting our software stack. This work observes that the total cost of warm rebooting either bare metal environment is between 6-7.4 seconds.

Once the kernel is booted and userspace programs begin launching, our software stack usually requires about one second to complete the final setup: (1) mounting filesystems, such as */proc*,

/dev, and */sys*, (2) mounting the PRAMFS partition, (3) loading a new copy of our mini-ckpts kernel into memory in preparation of the next failure, (4) loading our modified BLCR kernel module, (5) starting an SSH server daemon, and (6) optionally resuming an application that was previously rescued by mini-ckpts during the last failure.

5.10.1 Basic Applications

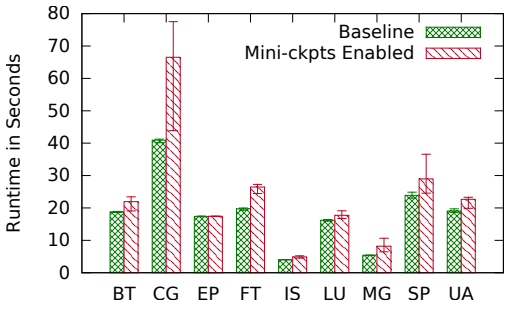
To ensure mini-ckpts basic correctness, this work developed test applications that iterated through hardware registers while setting them to known values in a deterministic pattern in single threaded and multithreaded environments. As the test was executed, this work periodically injected a kernel panic through the kernel's *sysrq* trigger. Through over 100 repeated kernel panics while executing the same test, mini-ckpts was able to continuously warm reboot the system and continue executing the application without affecting its state.

This work then developed a similar test that repeatedly performed floating point (FPU) calculations on multiple cores. It was designed to only print a failure notice if the FPU operations (multiply, divide, add, subtract) diverged from their expected value by more than 10^{-5}, to account for floating point rounding errors. Again, mini-ckpts was able to protect this FPU test against failures for over 100 consecutive panics during the execution of the same application.

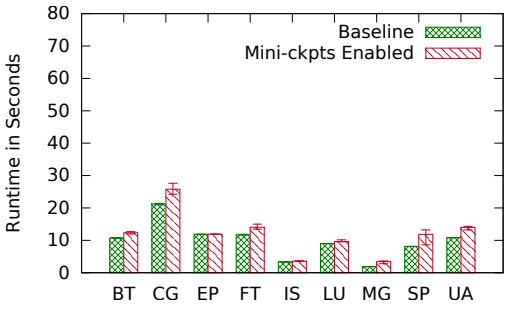
mini-ckpts was then evaluated against the *sh* shell, *vi* text editor, and *python* interpreter. While mini-ckpts has not been extended to support updating file descriptor seek pointers, it was able to continue executing these applications correctly provided they did not perform extensive file I/O. The *vi* editor did require a command to be blindly typed to reset its terminal display after each panic, since the terminal would not be aware that it needs to refresh after a transparent warm reboot.

5.10.2 OpenMP Application Performance

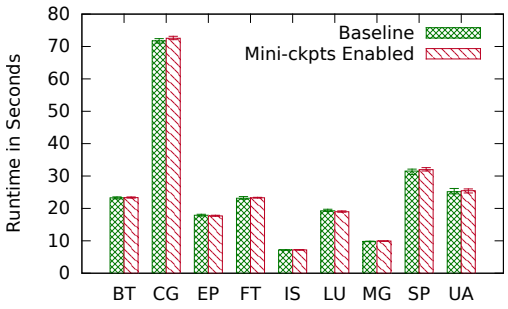
This work next evaluates mini-ckpts with HPC benchmarks from the NAS Parallel Benchmarks (NPB) suite, OpenMP version 3.3 [10]. This work has evaluated each benchmark on AMD CPUs, Intel CPUs, and a QEMU/KVM virtual machine hosted within the aforementioned AMD and CPU systems. This work first ran each benchmark without any protection or instrumentation and then compared it against the same run in a mini-ckpts protected environment. The results for both bare metal and virtual machine executions are shown in Figure 5.3. The results were gathered from eight consecutive runs of each experiment including a warm-up. The bars represent the average runtime per benchmark, excluding intialization, and the error bars indicate the observed minimum and maximum runtimes.



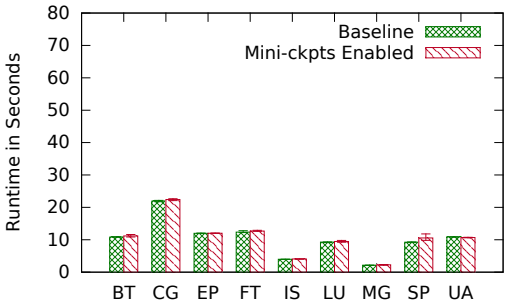
(a) Bare Metal AMD Opteron 6128



(b) Bare Metal Intel E5-2650



(c) KVM Hypervisor on AMD Opteron 6128 Host



(d) KVM Hypervisor on Intel E5-2650 Host

Figure 5.3 Runtime Performance of the NPB OpenMP Benchmarks Running with 8 Threads - No Core Pinning

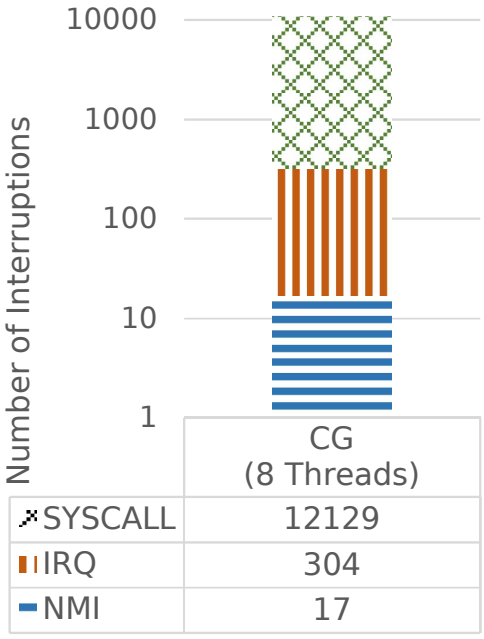


Figure 5.4 Interruption Types Encountered by a Typical Run of NAS CG with 8 OpenMP Threads

During our first OpenMP experiments, on average, mini-ckpts protection was observed to impart an average 26.3% runtime overhead on the AMD CPUs, with a range between 0% (EP) to 63% (CG). For the Intel CPUs, the average overhead was 25% with a range of 0% (EP) to 80% (MG). For the Intel VM, the average was 3.4% with a range of 0% (EP, UA) to 14% (SP). The AMD VM saw an average runtime overhead of 1% for mini-ckpts. However, when you consider the mini-ckpts performance on a virtual machine, it must also be compared against the bare metal performance. In that context, the AMD VM with mini-ckpts protection incurred a 40% runtime overhead relative to bare metal. The Intel VM had a runtime overhead of 11% relative to bare metal.

Hardware architectures create a potential for performance variability in terms of the cost of mini-ckpts. This work next discusses two impactful requirements mini-ckpts imposes on a protected application.

5.10.2.1 System Calls and Interruptions

mini-ckpts only affects an application when it is interrupted or during a kernel panic, so the costs associated with mini-ckpts are most noticeable for system calls during the normal execution of an application. Let us consider the CG benchmark. Figure 5.4 depicts the frequency (y-axis, on

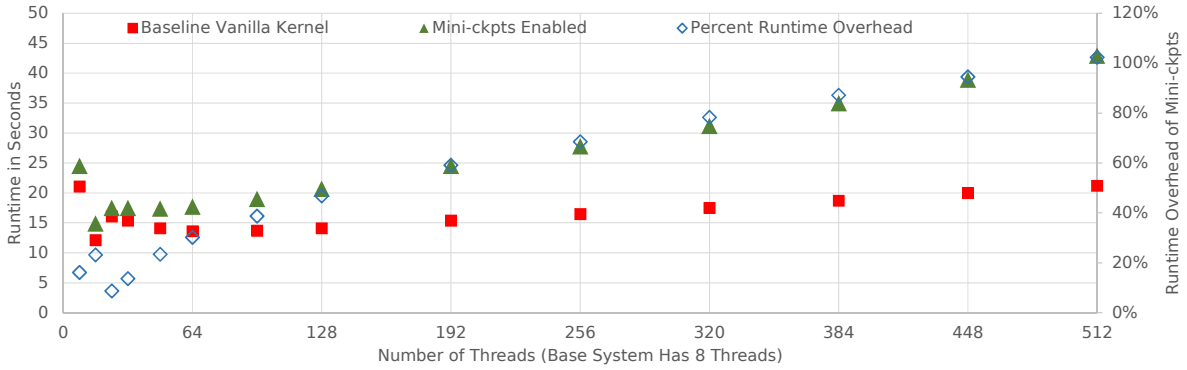


Figure 5.5 Extreme Over-provisioning of Threads Demonstrate a Linear Slowdown with mini-ckpts Due to Increased Systems Calls

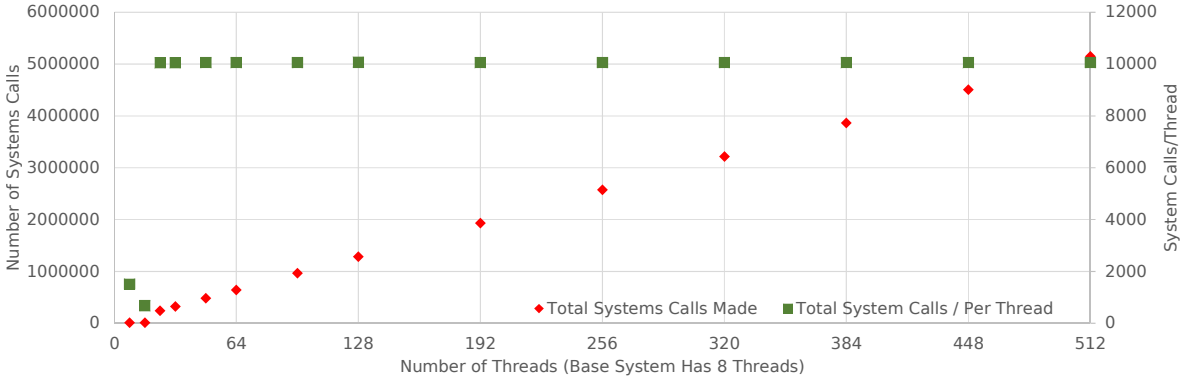


Figure 5.6 NPB CG Growing System Call Usage (Predominantly *futex*) — Each data-point correlates with the same X-axis in Figure 5.5

a logarithmic scale) and types of interruptions it experiences. This work sees that IRQs and NMIs are relatively rare compared to the frequency of system calls. During experimentation this work traced this benchmark and noted that nearly all of its system calls are to a *futex* used for synchronization between the threads. To ascertain how mini-ckpts scales under the increasing load of system calls and higher thread counts, this work progressively over-committed the hardware's thread resources from 8 threads to 512 threads. As this work increased the thread count, this work measured the time to completion for both a regular benchmark and a mini-ckpts protected one. This work also recorded the number of system calls made during each experiment. The results are shown in Figure 5.5. This work observes a linear relationship between the number of threads (x-axis) and the execution time (y-axis). This work also notes that while the baseline benchmark grows in thread count, its runtime is nearly flat. Since the only interaction between a mini-ckpts protected application and mini-ckpts is during an interruption, this work also plots the number of system calls during each experiment in Figure 5.6. While the number of system calls per thread (2^{nd} y-axis) remains constant after 32 threads, the cumulative total (y-axis) linearly grows, which explains why this work sees a linear runtime increase for mini-ckpts protected applications experiencing high numbers of system calls.

Indirectly, this experiment also demonstrates the scalability of mini-ckpts. This work additionally tested panic injections while running NPB CG at thread counts of 512 and were able to successfully recover and complete the application after warm reboots in all instances.

5.10.2.2 NUMA Constraints Imposed by PRAMFS

The second requirement imposed on an application protected by mini-ckpts is the remapping of all application memory to PRAMFS. In Linux, anonymous memory mappings (i.e., heap and stack) may be provided by any free memory within the system. A specialized memory allocator may even request memory that is local to a specific CPU (using the closest memory controller). PRAMFS, however, is allocated to a specific region of physical memory. In non-uniform memory access (NUMA) architectures, such as our experimental platform, this guarantees that PRAMFS mapped memory regions will have differing latencies between memory accesses on varying cores. From Figure 5.3a this work sees that the minimum and maximum run times vary by 62% for the CG benchmark. This particular experiment did not enforce any core pinning between OpenMP threads. As a result, as the scheduler moved threads between cores, the overall runtime of the application varied based on NUMA locality. While CG had the most pronounced effect, the other benchmarks displayed some degree of variance as well.

This work was able to confirm that NUMA was responsible for the runtime variance through

Cores	0-3	4-7	8-11	12-16
AMD	1.42	2.04	3.25	3.30
AMD VM	3.2 - 3.4			
Intel	0.90		1.12	
Intel VM	0.95			

All Times in Seconds

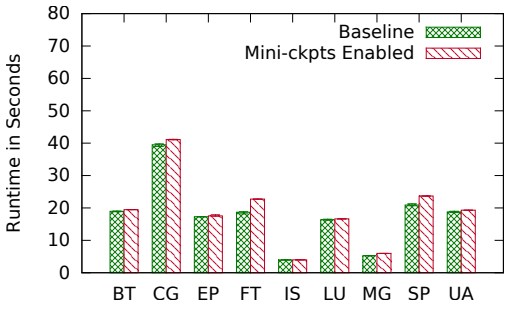
Table 5.6 NUMA Microbenchmark Interacting with PRAMFS Memory Mappings

two experiments:

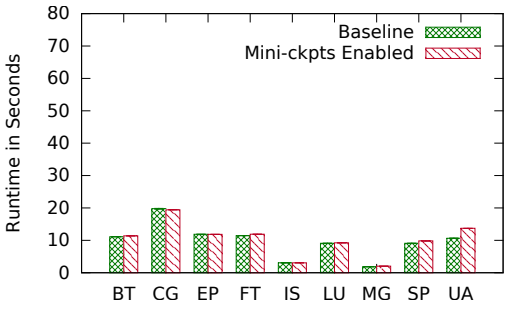
1. This work created a microbenchmark that used *mmap* to map 64MB of memory to our application. In a linear fashion, this work then wrote 6GB worth of cumulative writes over this region. Using core pinning, this work timed this experiment running on each core. This experiment was repeated for both the AMD and Intel bare metal systems as well as within virtual machines on both. The results of each experiment are shown in Table 5.6. This work sees locality for PRAMFS, and this work also observed that by changing the physical location of the PRAMFS the NUMA localities also move, as expected. Additionally, this work compared the runtime of writing to a PRAMFS mapping vs. an *mmap* anonymous memory mappings and found the runtimes had a percent difference of 2.7% on average (in either direction) for the AMD machine and a 1% difference (in favor of PRAMFS) on the Intel machine. On the virtual machines this work found that regardless of core pinning within the VM, the hypervisor treats each virtual CPU as a thread, which is subject to migration by the hypervisor's scheduler. As shown in Table 5.6, the AMD VM scheduler experienced performance near the its bare metal host's worst NUMA mappings. The Intel VM showed a performance between both its host's best and worst performance as the scheduler migrated the virtual cores during execution.

2. To isolate the effects of PRAMFS from mini-ckpts, this work repeated the benchmarks by modifying mini-ckpts to only perform PRAMFS remappings, but to not make any further modifications or checkpoints on the process. When combined with core pinning and knowledge of the NUMA latency of each core, this work was able to repeat both the best and worst case runtimes for each experiment in Figure 5.3. Repeated experiments with best-case NUMA mappings and core pinning are shown in Figure 5.7.

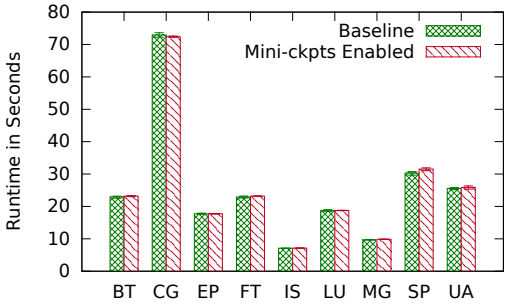
In comparing the two overall OpenMP experiments with (Figure 5.7) and without (Figure 5.3) core pinning, this work observed that mini-ckpts in combination with intelligent physical memory



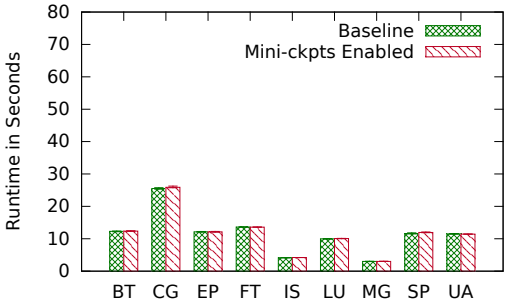
(a) Bare Metal AMD Opteron 6128



(b) Bare Metal Intel E5-2650



(c) KVM Hypervisor on AMD Opteron 6128 Host



(d) KVM Hypervisor on Intel E5-2650 Host

Figure 5.7 Runtime Performance of the NPB OpenMP Benchmarks Running with 8 Threads - Core Pinning Applied

	A	**B**	**C**		
Benchmark	% difference librlmpi	% difference librlmpi+ PRAMFS	% difference librlmpi+ PRAMFS+ mini-ckpts	**B-A** Only PRAMFS	**C-B** Only mini-ckpts
NPB CG	-1.0%	-0.2%	1.0%	0.8%	1.2%
NPB LU	2.0%	5.5%	6.5%	3.5%	1.1%
NPB EP	0.7%	0.7%	0.8%	0.0%	0.1%
NPB IS	-6.0%	-5.1%	-6.1%	0.9%	-1.0%
PENNANT	3.1%	1.3%	2.8%	-1.8%	1.5%
Clover Leaf	7.9%	11.2%	13.4%	3.2%	2.2%
Averages →	**1.1%**	**2.2%**	**3.1%**	**1.1%**	**0.9%**
Diff. Geo. Mean →	**1.0%**	**2.1%**	**2.9%**	**1.0%**	**0.8%**

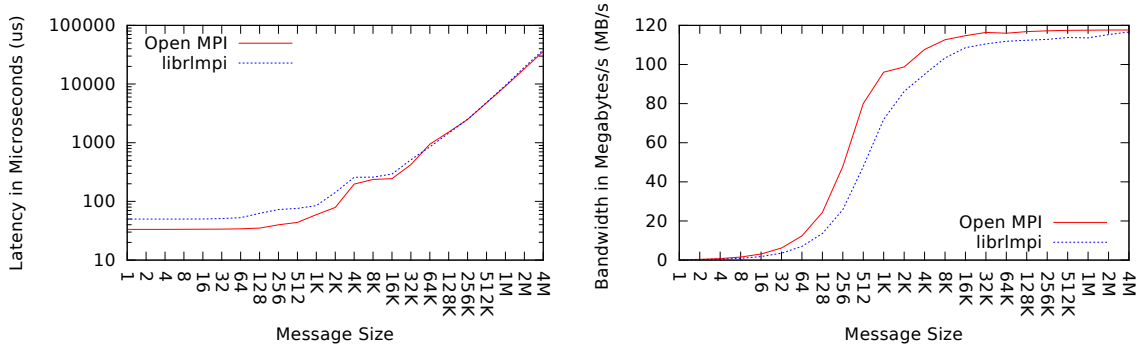
Table 5.7 Runtime Percent Differences and Geoemetric Mean Differences of MPI Benchmarks with librlmpi Relative to Open MPI

mapping of PRAMFS reduced the AMD host runtimes on average from 26% to 6.9%. The Intel host runtimes were reduced on average from 25% to 5.9%.

5.10.3 MPI Application Performance

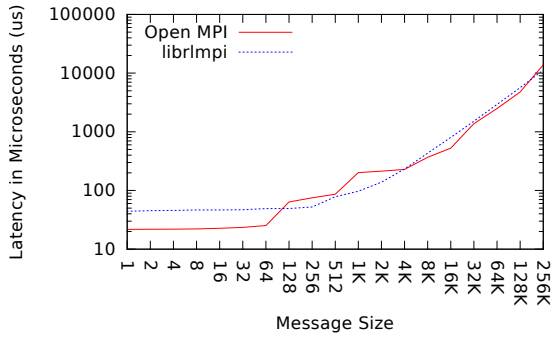
mini-ckpts requires specialized MPI implementation support to handle lost network connections and lost network buffers on both the sender and receiver side. An MPI implementation must also be prepared to handle unexpected failures during system calls by anticipating an $EINTR$ return value from failed system calls that mini-ckpts recovered from after a warm reboot due to kernel panic. Our prototype implementation, librlmpi, supports these requirements. To provide a meaningful evaluation of the performance of mini-ckpts with MPI, this work will also show performance comparisons between vanilla librlmpi (without mini-ckpts enabled) and a mainstream MPI implementation, Open MPI. Additionally, mini-ckpts requires remapping all process memory to PRAMFS, so we include an experiment that triggers PRAMFS remapping, but does not activate mini-ckpts protection. This experiment provides insight into the effects of PRAMFS memory effects (NUMA) without incurring any mini-ckpts overhead. Open MPI currently cannot handle recovery after a mini-ckpt, but the relative performance comparison provides the reader with a grasp of librlmpi's performance against a known implementation.

All MPI micro benchmarks and mini application runtimes are reported as always based on 8 runs per datapoint. Error bars in figures referenced indicated observed minimum and maximum values.

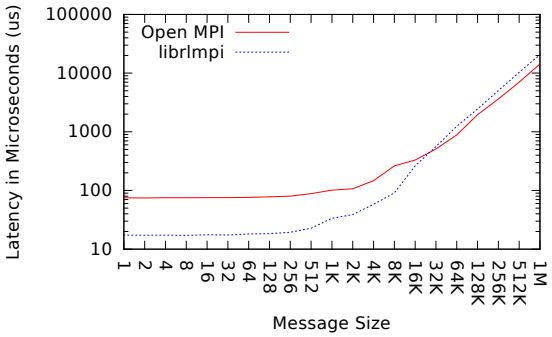


(a) Latency

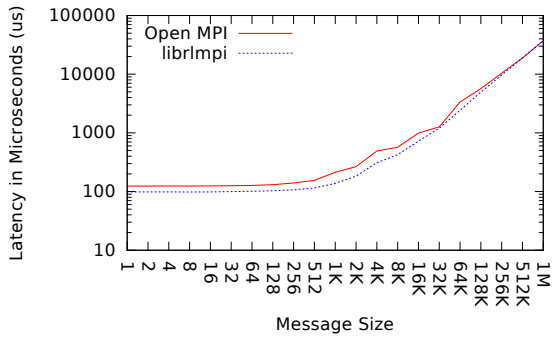
(b) Bandwidth



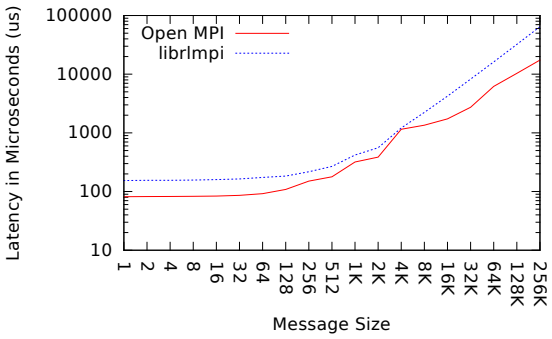
(c) Reduce



(d) Bcast



(e) Alltoallv



(f) Allreduce

Figure 5.8 OSU Benchmarks Comparing Open MPI and librlmpi Performance

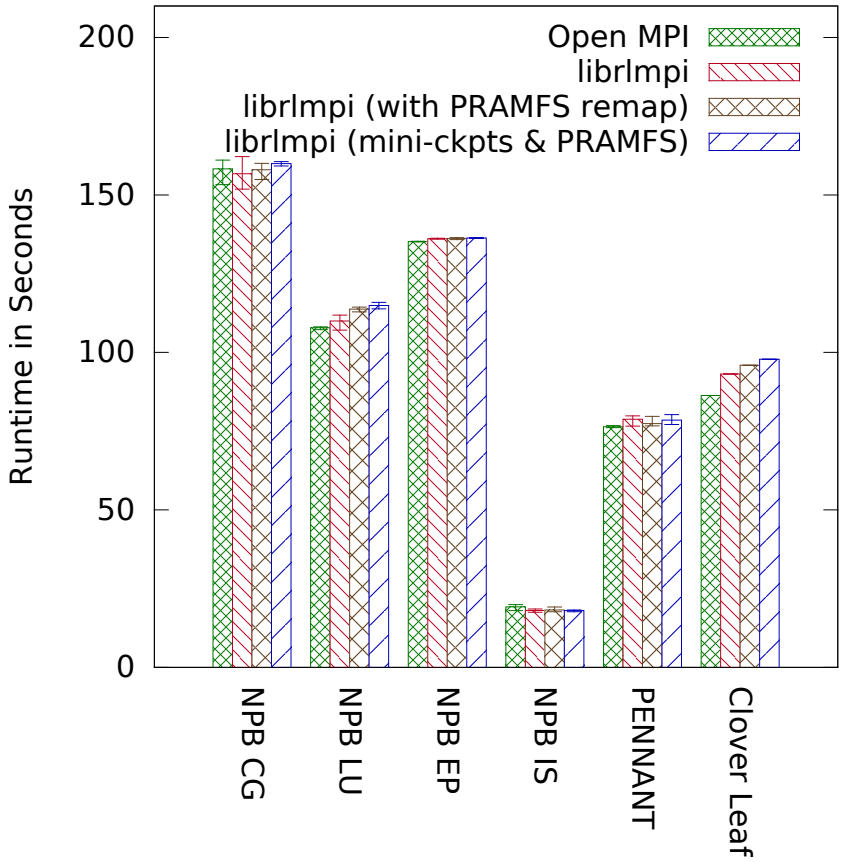


Figure 5.9 Runtime Comparisons of Various MPI Benchmarks in Differing MPI Stack Configurations

5.10.3.1 Micro Benchmarks

This work first evaluates the peer-to-peer and collective performance of librlmpi. Figure 5.8 shows log-scale[4] results from the OSU Micro Benchmark including MPI latency (MPI_Send/MPI_Recv), bandwidth, and latencies for MPI_Reduce, MPI_Bcast, $MPI\_Alltoallv$, and $MPI\_Allreduce$. As a prototype implementation, librlmpi's focus is to support MPI applications within a mini-ckpts environment, which makes MPI performance a secondary concern. Nevertheless, this work sees that librlmpi peer-to-peer performance closes in on Open MPI as message sizes increase beyond four kilo-byte message sizes, indicating that the relative performance of the production-quality Open MPI implementation begins to be dwarfed by network transmission rather than implementation efficiency for larger messages. Collective operations indicate that librlmpi is more performant for small messages in $MPI\_Alltoallv$ and MPI_Bcast. However, librlmpi collective performance does not scale as the node count increases due to the simple, linear implementation of all collectives. Mainstream MPI implementations typically provide tree-based communication, topology aware mappings, and skewed communication patterns to account for anticipated node distance. librlmpi aims to provide a working subset of MPI to show feasibility rather than optimized communication. $MPI\_Barrier$ performance (not depicted in the Figures) was observed to be 156 microseconds with librlmpi and 82 microseconds with Open MPI.

Figure 5.9 shows the runtimes of four NPB MPI benchmarks, Los Alamos' PENNANT Mini-App, and Sandia's Clover Leaf Mini-App running across 4 processors on 4 AMD nodes using Gigabit Ethernet. Compared to Open MPI, librlmpi on its own performs better than Open MPI for IS (6% faster). Performance is similar for CG (-1%), EP (0.7%), LU (2%), PENNANT (3%) and is 8% slower for Clover Leaf. The results as both percent differences and difference in geometric mean (librlmpi in various configurations vs. Open MPI) are shown in Table 5.7. These results show that librlmpi provides comparable peer-to-peer message performance to Open MPI at least at our MPI job size when message latency is not a bottleneck to application performance.

Relative to Open MPI, overall average percent difference of just librlmpi for all benchmarks is 1.1% with a difference in geometric mean of 1.0%. As we enable PRAMFS with librlmpi, the average percent difference becomes 2.2% with a difference of geometric mean of 2.1%. Finally, with full librlmpi, PRAMFS, and mini-ckpts enabled, the average percent difference becomes 3.1% with a difference of geometric mean of 2.9%.

Table 5.7 additionally compares librlmpi against itself. Since librlmpi is a prototype demon-

[4]Figure 5.8b is shown with linear Y-axis.

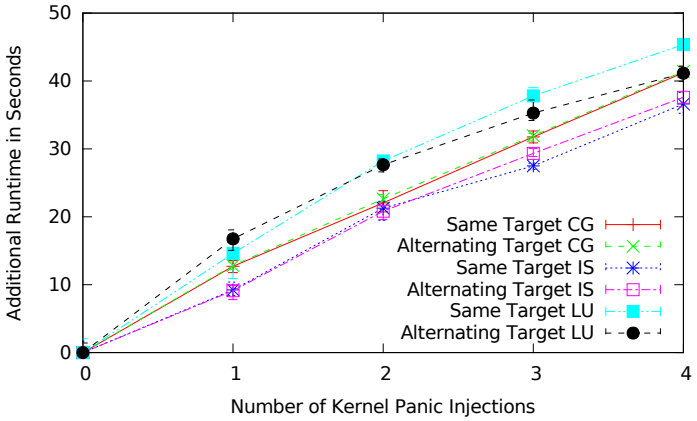


Figure 5.10 Runtime Costs per Kernel Panic Injection for MPI Applications Demonstrating Linear Slowdown

strating the capability of mini-ckpts to work with an MPI implementation, the overheads of PRAMFS and mini-ckpts relative to baseline librlmpi (instead of Open MPI) are now discussed. By subtracting column B from column A, it is observed that on average running an application with data migrated to PRAMFS accounts for 1.1% of the overhead. Further, only considering the costs of mini-ckpts by subtracting column C from column B, it is observed that on average mini-ckpts accounts for 0.9% of the overhead.

5.10.3.2 Injections During MPI Execution

While librlmpi has been shown to be comparable to Open MPI in terms of peer-to-peer messaging performance, the key metric that this work investigates is the scalability of mini-ckpts. It is assumed that any MPI implementation may be adopted to support mini-ckpts requirements, so the performance of our prototype librlmpi is merely used as a tool to demonstrate mini-ckpts rather than its MPI performance.

mini-ckpts provides protection at a per-node level. To be scalable, this work investigated if failures incur a constant cost, independent of the number of processes in a job as well as independence of the type of job running. This work designed injection experiments for MPI applications to evaluate how mini-ckpts scales with MPI. The first experiment involves picking a single node to repeatedly inject kernel failures into. This work varies the number of injections seen per run of the application from zero (failure free case) to four. Our second experiment type alternates between all nodes each time picking a new node to fail. For instance, in our four process jobs, this work first fails node 1, followed by 2, then 3, and finally node 4.

Number of Kernel Panics	1	2	3	4
Avg. Time Increase	12.53	11.89	10.76	10.14

All Times in Seconds

Table 5.8 Average Additional Runtime With Varying Number of Kernel Panics in MPI Applications

Figure 5.10 summarizes our MPI injections. As each benchmark's failure-free runtime differs, this work starts the y-axis at 0 and measure only the additional time required when one or more kernel failures are injected. Each MPI application was run with between zero and four failures, and each failure was repeated in (1) a scenario where one node receives all of the failures, or (2) the failures were distributed ("alternating") between all nodes. A separate experiment was run for every possible failure count and target. Each experiment was repeated 8 times resulting in error bars that show the observed minimum and maximum runtime, while the points represent the average. This work observes a linear runtime increase as the number of injections is incremented for each benchmark. On average, the additional time increase per injection was between 10-13 seconds, as shown in Table 5.8. These experiments were run on AMD nodes where the expected total warm reboot time (as shown in Table 5.5) is previously observed to be 6 seconds, excluding the time to restart an application. This work expects that warm reboots will incur an additional slight overhead besides the time taken for the reboot itself since the CPU caches will have been flushed when the operating system restarts. This is similar to a benchmark performing a "warm-up" prior to starting the primary computation. A mini-ckpts restart is equivalent to jumping straight into main execution without warming up the cache. The restart times from the MPI injections experiments show that mini-ckpts is scalable across multiple nodes and that failures increase the runtime of an application in a linear manner.

5.10.4 Fault Injections

Fault injections were performed by directly modifying kernel memory from within a kernel module that was triggered by *ioctl* system calls, which identifies the type of injection. Ultimately, this work measures mini-ckpts' effectiveness based on a successful recovery from an injection that impacted the kernel in such a way that is causing a crash or system hang. For example, some types of kernel injections will either be benign or cause the system to behave in a way that (while incorrect) does not cause a panic or hang. These types of injections, if they were to occur in the wild, would only be remedied by mini-ckpts if the kernel implements a sanity check or bug check that ultimately detects and reacts (by panicing). This is an assumption on our work, but this work will point out that — even during the development of mini-ckpts — there were

several times that mini-ckpts misbehaved only to be caught one of the kernel's safe guards such as hang detection or file I/O errors. This ultimately caused mini-ckpts to reboot the system to a sane state immediately, making kernel development easier than on bare metal without our improvements.

5.10.4.1 Invalid Pointers

The most common type of injections this work performed were pointer injections, where this work either set a pointer to NULL or switched it to a random value. NULL pointers are the easiest to catch and were protected 100% of the time since NULL pointer dereferencing is an easily caught exception. Changing a pointer with a non-NULL value had differing outcomes: If the pointer was not a valid address, then it was caught as an exception (leading to a panic) just as easily as a NULL pointer. However, changing the low order bits on a valid kernel pointer typically resulted in a call chain that would randomly jump through the kernel until an invalid reference or sanity check was reached.

During each of the NAS benchmarks this work performed the following experiments:

- This work forced a direct call to *panic*.
- This work injected a NULL or invalid pointer to task_struct fs member, signal handlers, *parent* member, and/or $files$ member.

This work was able to detect and recover from each injection. To further test mini-ckpts, this work also scheduled forced panics via a NULL pointer on each core of the system. This test showed that mini-ckpts is able to recover from a panic and reboot the system in a valid manner regardless of which OS core it originated on. This was a critical test as it demonstrated the effectiveness of mini-ckpts NMI migration protocol during a fault.

5.10.4.2 Memory Allocation

Memory allocation within the Linux kernel is essential for it to operate properly, and some types of memory allocations are reserved for contexts that cannot sleep. A failure to allocate memory when it is required but unavailable can force the kernel to panic. mini-ckpts does not allocate memory during a panic. However, it is able to induce a kernel panic by exhausting the available memory and waiting for an allocation request to arrive that cannot be fulfilled. This work performed allocation experiments repeatedly via *kmalloc* exhausting available memory. Kernel crashes that resulted were able to be caught and mitigated through a warm reboot. However, while this was deliberate, mini-ckpts shows promise for mitigating kernel drivers that

contain memory leaks on long-running systems. With mini-ckpts, a leak could temporarily be coped with by allowing a warm reboot to take place whenever it has exceeded available memory. This is not recommended in practice, but it shows a potential use case for mini-ckpts.

5.10.4.3 Hard Hangs

Some types of faults result in hanging the system, such as an infinite loop in a kernel driver. The Linux kernel is now preemptible, however, a kernel thread may at times disable preemption if it does not wish to be interrupted during some work. If a fault or bug occurs during this time, it is possible that the CPU thread associated with that work will become indefinitely hung without the possibility of being interrupted. On its own, mini-ckpts cannot protect against this type of failure.

There is a feature that does provide a solution to this exact problem: An NMI watchdog is provided by modern hardware to periodically send a non-maskable interrupt to the operating system. The handler for the watchdog typically has a counter that is reset to zero at common code paths within the kernel. During normal operation, the counter is continually reset. However, during a hang the counter is not reset, and eventually the watchdog interrupt notices this. At this point, the watchdog can directly call a kernel panic (which mini-ckpts uses to free the system from its hang and warm reboots). As part of our testing, we injected hangs in regular system calls and interrupt handlers. This work was able to successfully detect and recover from all failures. However, the delay associated with these fault injections is variable since the NMI watchdog only executed periodically. In some configurations a system may remain hung for up to a minute during our tests. This work recommends configuring the watchdog interval to match your responsiveness needs, although shorter intervals will potentially cause performance degradation.

5.10.4.4 Soft Hangs

Unlike hard hangs, a soft hang is a software loop that is not making forward progress but is still on a code path that clears the NMI watchdog timer. These types of hangs are much more difficult to detect because the operating system appears to be making progress despite being indefinitely trapped in a loop. While this work did not explicitly test for these types of hangs, but this work did notice that the read-copy-update (RCU) mechanism was able to detect hangs within mini-ckpts' file I/O routines taking several minutes to complete during debugging and development.

The existence of soft and hard hang detection and sanity checks are essential to the effec-

tiveness of mini-ckpts. On its own, mini-ckpts is capable of protecting an application only if an actual crash occurs.

5.11 Future Work

As we present mini-ckpts today, this proposed operating system resilience framework is functioning in a purely reactive manner. However, software tools or a user may force a reboot and subsequent restart of a mini-ckpt protected process by programmatically causing the kernel to panic in the absence or a real fault. Linux allows a superuser to do this through the command line or via insertion of a one-line kernel module that simply calls *panic*(...). This work sees a potential to use this technique to preemptively rejuvenate a running kernel either based on an expected time to failure or based on predictions from anomaly detection tools [47]. If rejuvenation is further coupled with advanced knowledge of communication patterns of a multi-node HPC job, then there is the possibility of timing these rejuvenations to avoid generating cluster noise or cascading communication delays.

5.12 Conclusion

Today's operating systems are currently not designed with fault tolerance in mind. The default mechanism to handle a failure within the Linux kernel and other commodity operating systems is to print an error message and reboot, which causes a loss of all unsaved application data. For HPC applications, where checkpointing, rollback, and rework are expensive operations, mitigating an operating system crash by allowing warm reboots and recovering of running applications without data loss can provide a safe guard against memory corruption, system hangs, and other unexpected failures. This work has identified the key points of instrumentation within the Linux kernel required to save the state of an application during operating system failure, and it has provided mechanisms to safely persist application data across system reboots. Additionally, based on these designs, this work provides an experimental implementation of a checkpointing library capable of protecting HPC applications from crashes within the kernel by providing non-stop forward progress immediately after a short warm reboot. This work shows that a mini-ckpts kernel can can successfully protect an application from computation loss (due to a kernel failure), perform a warm reboot, and transparently restart execution all within 6 seconds of a fault occurring. This work demonstrates the effectiveness of our implementation by injecting common faults into OpenMP and MPI HPC benchmarks and observe runtime overheads that average overall between 5% to 6%, depending on the host architecture for

OpenMP threaded applications and 3.1% overall for MPI applications.

CHAPTER

6

CONCLUSION

Protecting today's high performance computing applications from silent data corruption is an important task. As computing systems continue to grow, so will the risk of data corruption. Protecting applications from silent data corruption is not trivial. Developing new, safe algorithm-based fault tolance is either difficult or impossible for some classes of application, and legacy requirements may require some degree of reengineering to ensure continued scalability. To address these problems, this work has developed and evaluated methods of generalized fault tolerance for silent data corruption. Our first work, RedMPI, was shown to be effective in providing transparent redundancy to message passing applications. RedMPI not only provides error detection and correction, but it also provides insight into the effects of even a single bit flip corruption. This work showed that redundancy was a viable method to protect message passing applications.

The second work, FlipSphere, aimed to provide transparent error detection and correction for classes of applications beyond message passing applications. We showed that on-demand memory integrity verification using software generated error correction codes is able to both detect and correct errors without redundancy. FlipSphere also demonstrated that it is possible to protect an application as well as its associated runtime libraries. Further, it allowed flexibility in the software error codes and hashes it uses internally. This allows an application to selectively

increase protection for specific regions of memory. Finally, FlipSphere showed the potential for resilience tools to utilize hardware accelerators such as graphics processing units (GPUs) or specialized hardware accelerators such as Intel's Xeon Phi co-processors to accelerate the generation of error correcting codes.

The final work, mini-ckpts, provides fault protection for a running operating system. While the first two contributions protect applications and their libraries, the final piece of the software stack remained the operating system. mini-ckpts outlined modifications necessary for a commodity operating system to protect, restore, and resume its running applications in the event of a failure within the operating system. Our experimental prototype was able to protect running applications from all injected kernel failures that resulted in a kernel panic, without loss in computational progress of running applications. mini-ckpts was shown to restore all common classes of HPC applications, including threaded and message passing applications. Results indicated that failures within an operating system may be mitigated in seconds without progress loss, thus avoiding costly rollbacks, restarts, and rework of otherwise lost data.

This dissertation has shown that software redundancy, software-based ECC, and fault tolerant operating systems may be used to provide protection to the HPC software stack from memory errors and even some other types of faults at scale. These results thereby validate the hypothesis of this dissertation.

BIBLIOGRAPHY

[1] Clover leaf mini-app. `http://uk-mac.github.io/CloverLeaf/`.

[2] Osu micro benchmarks. `http://mvapich.cse.ohio-state.edu/benchmarks/`.

[3] Pennant mini-app. `https://github.com/losalamos/PENNANT`.

[4] Protected and persistent ram filesystem. `http://sourceforge.net/projects/pramfs/`.

[5] Top 500 list. http://www.top500.org/, June 2002.

[6] Fast crc computation for iscsi polynomial using crc32 instruction. White Paper, April 2011.

[7] Intel xeon phi co-processor brief. White Paper, 2013.

[8] Saurabh Agarwal, Rahul Garg, Meeta S. Gupta, and Jose E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th Annual International Conference on Supercomputing*, ICS '04, pages 277–286, New York, NY, USA, 2004. ACM.

[9] Marc A. Auslander, David C. Larkin, and Allan L. Scherr. The evolution of the mvs operating system. *IBM Journal of Research and Development*, 25(5):471–482, 1981.

[10] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[11] Katelin Bailey, Luis Ceze, Steven D. Gribble, and Henry M. Levy. Operating system implications of fast, cheap, non-volatile memory. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association.

[12] Mary Baker and Mark Sullivan. The recovery box: Using fast recovery to provide high availability in the unix environment. In *In Proceedings USENIX Summer Conference*, pages 31–43, 1992.

[13] P. Banerjee, J.T. Rahmeh, C. Stunkel, V.S. Nair, K. Roy, V. Balasubramanian, and J.A. Abraham. Algorithm-based fault tolerance on a hypercube multiprocessor. *Computers, IEEE Transactions on*, 39(9):1132–1145, Sep 1990.

[14] R.C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *Device and Materials Reliability, IEEE Transactions on*, 5(3):305–316, Sept 2005.

[15] Diogo Behrens, Christof Fetzer, Flavio P. Junqueira, and Marco Serafini. Towards transparent hardening of distributed systems. In *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems*, HotDep '13, pages 4:1–4:6, New York, NY, USA, 2013. ACM.

[16] Swen Böhm and Christian Engelmann. File i/o for mpi applications in redundant execution scenarios. In *Euromicro International Conference on Parallel, Distributed, and network-based Processing*, February 2012.

[17] Aniruddha Bohra, Iulian Neamtiu, Pascal Gallard, Florin Sultan, and Liviu Iftode. Remote repair of operating system state using backdoors. In *International Conference on Autonomic Computing (ICAC-04)*, New-York, NY, May 2004. Initial version published as Technical Report, Rutgers University DCS-TR-543.

[18] C. Borchert, H. Schirmeier, and O. Spinczyk. Generative software-based memory error detection and correction for operating system data structures. In *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, June 2013.

[19] Ron Brightwell, Kurt B. Ferreira, and Rolf Riesen. Transparent redundant computing with MPI. In Rainer Keller, Edgar Gabriel, Michael M. Resch, and Jack Dongarra, editors,

EuroMPI, volume 6305 of *Lecture Notes in Computer Science*, pages 208–218. Springer, 2010.

[20] Greg Bronevetsky and Bronis R. de Supinski. Soft error vulnerability of iterative linear algebra methods. In *Proceedings of the* 21^{st} *ACM International Conference on Supercomputing (ICS) 2008*, Island of Kos, Greece, June 7-12, 2007. ACM Press, New York, NY, USA.

[21] Greg Bronevetsky and Adam Moody. Scalable I/O systems via node-local storage: Approaching 1 TB/sec file I/O. Technical Report TR-JLPC-09-01, Lawrence Livermore National Laboratory, Livermore, CA, USA, August 2009.

[22] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot — a technique for cheap recovery. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

[23] "C Chen and M Hsiao". Error-correcting codes for semiconductor memory applications: A state-of-the-art review. *IBM Journal of Research and Development*, 28(2), march 1984.

[24] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. The rio file cache: Surviving operating system crashes. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 74–83, New York, NY, USA, 1996. ACM.

[25] Zizhong Chen. Online-abft: An online algorithm based fault tolerance scheme for soft error detection in iterative methods. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2013.

[26] Guy Cobb, Barry Roundtree, Henry Tufo, Martin Schulz, Todd Gamblin, and Bronis de Supinski. Mpiecho: A framework for transparent mpi task replication. Technical Report CU-CS-1082-11, Dept. of Computer Science, University of Colorado at Boulder, June 2011.

[27] Miguel Correia, Daniel Gómez Ferro, Flavio P. Junqueira, and Marco Serafini. Practical hardening of crash-tolerant systems. In *USENIX Conference on Annual Technical Conference*, 2012.

[28] John T. Daly. ADTSC nuclear weapons highlights: Facilitating high-throughput ASC calculations. Technical Report LALP-07-041, Los Alamos National Laboratory, Los Alamos, NM, USA, June 2007.

[29] John T. Daly, Lori A. Pritchett-Sheats, and Sarah E. Michalak. Application MTTFE vs. platform MTTF: A fresh perspective on system reliability and application throughput for computations at scale. In *Proceedings of the Workshop on Resiliency in High Performance Computing (Resilience) 2008*, pages 19–22, May 2008.

[30] Nathan DeBardeleben, James Laros, John T. Daly, Stephen L. Scott, Christian Engelmann, and Bill Harrod. High-end computing resilience: Analysis of issues facing the HEC community and path-forward for research and development. Whitepaper, December 2009.

[31] Alex Depoutovitch and Michael Stumm. Otherworld: Giving applications a chance to survive os kernel crashes. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 181–194, New York, NY, USA, 2010. ACM.

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

[33] Christian Dietrich, Martin Hoffmann, and Daniel Lohmann. Cross-kernel control-flow-graph analysis for event-driven real-time systems. In *Proceedings of the 2015 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '15)*, New York, NY, USA, June 2015. ACM Press.

[34] Björn Döbel and Hermann Härtig. Can we put concurrency back into redundant multithreading? In *Proceedings of the 14th International Conference on Embedded Software*, EMSOFT '14, pages 19:1–19:10, New York, NY, USA, 2014. ACM.

[35] J Dongarra, P Beckman, T Moore, P Aerts, G Aloisio, J C Andre, D Barkai, J Y Berthou, T Boku, B Braunschweig, and et al. The international exascale software project roadmap. *International Journal of High Performance Computing Applications*, 25(1):3–60, 2011.

[36] J. Duell. "the design and implementation of berkeley lab's linux checkpoint/restart". Tr, Lawrence Berkeley National Laboratory, 2000.

[37] James Elliot, Kishor Kharbas, David Fiala, Frank Mueller, Christian Engelmann, and Kurt Ferreirar. Combining partial redundancy and checkpointing for HPC. In *International Conference on Distributed Computing Systems*, page (accepted), 2012.

[38] 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*, 1(2):97–108, April 2004.

[39] Christian Engelmann and Swen Böhm. Redundant execution of hpc applications with mr-mpi. In *Proceedings of the* 10^{th} *IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN) 2011*, Innsbruck, Austria, February 15-17, 2011. ACTA Press, Calgary, AB, Canada.

[40] Christian Engelmann, Hong H. Ong, and Stephen L. Scott. The case for modular redundancy in large-scale high performance computing systems. In *Proceedings of the* 8^{th}

IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN) 2009, pages 189–194, Innsbruck, Austria, February 16-18, 2009. ACTA Press, Calgary, AB, Canada.

[41] Joe Fabula, Jason Moore, and Andrew Ware. Understanding neutron single-event phenomena in FPGAs. *Military Embedded Systems*, 3(2), 2007.

[42] Kurt Ferreira, Rolf Riesen, Patrick Bridges, Dorian Arnold, Jon Stearley, James H. Laros III, Ron Oldfield, Kevin Pedretti, and Ron Brightwell. Evaluating the viability of process replication reliability for exascale systems. In *Supercomputing*, November 2011.

[43] Kurt Ferreira, Jon Stearley, James H. Laros III, Ron Oldfield, Kevin Pedretti, Ron Brightwell, Rolf Riesen, Patrick Bridges, and Dorian Arnold. Evaluating the viability of process replication reliability for exascale systems. In *Supercomputing*, nov 2011.

[44] Kurt B. Ferreira, Patrick G. Bridges, and Ron Brightwell. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *Supercomputing*, November 2008.

[45] D. Fiala, K. Ferreira, F. Mueller, and C. Engelmann. A tunable, software-based dram error detection and correction library for hpc. In *Workshop on Resiliency in High Performance Computing (Resilience) in Clusters, Clouds, and Grids*, pages 110–121, September 2011.

[46] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 78:1–78:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[47] Ana Gainaru, Franck Cappello, Marc Snir, and William Kramer. Failure prediction for hpc systems and applications: Current situation and open issues. *Int. J. High Perform. Comput. Appl.*, 27(3):273–282, August 2013.

[48] A. Geist. What is the monster in the closet? Invited Talk at Workshop on Architectures I: Exascale and Beyond: Gaps in Research, Gaps in our Thinking, August 2011.

[49] 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 *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 9–9, Nov 2005.

[50] Amit Golander, Shlomo Weiss, and Ronny Ronen. DDMR: Dynamic and scalable dual modular redundancy with short validation intervals. *IEEE Computer Architecture Letters*, 7(2):65–68, 2008.

[51] Mohamed Gomaa, Chad Scarbrough, T. N. Vijayjumar, and Irith Pomeranz. Transient-fault recovery for chip multiprocessors. In *International Symposium on Computer Architecture*, pages 98–109, May 2003.

[52] Kevin Greenan and Ethan L. Miller. Reliability mechanisms for file systems using non-volatile memory as a metadata store. In *Proceedings of the 6th ACM & IEEE Conference on Embedded Software EMSOFT 06*, pages 178–187, Oct 2006.

[53] Kevin M. Greenan and Ethan L. Miller. Prims: Making nvram suitable for extremely reliable storage. In *Proceedings of the 3rd Workshop on on Hot Topics in System Dependability*, HotDep'07, Berkeley, CA, USA, 2007. USENIX Association.

[54] Paul H. Hargrove and Jason C. Duell. Berkeley Lab Checkpoint/Restart (BLCR) for Linux clusters. In *Journal of Physics: Proceedings of the Scientific Discovery through Advanced Computing Program (SciDAC) Conference 2006*, volume 46, pages 494–499, Denver, CO, USA, June 25-29, 2006. Institute of Physics Publishing, Bristol, UK.

[55] Tino Heijmen, Philippe Roche, Gilles Gasiot, Keith R. Forbes, and Damien Giot. A comprehensive study on the soft-error rate of flip-flops from 90-nm production libraries. *IEEE Transactions on Device and Materials Reliability (TDMR)*, 7(1):84–96, 2007.

[56] Martin Hoffmann, Florian Lukas, Christian Dietrich, and Daniel Lohmann. dOSEK: The design and implementation of a dependability-oriented static embedded kernel. In *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications (RTAS '15)*, Los Alamitos, CA, USA, April 2015. IEEE Computer Society Press. Best Paper.

[57] Chung-H. Hsu and Wu-C. Feng. A power-aware run-time system for high-performance computing. In *Supercomputing*, 2005.

[58] Kuang-Hua Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Comput.*, 33(6):518–528, June 1984.

[59] Y. Huang, C. Kintala, N. Kolettis, and N.D. Fulton. Software rejuvenation: analysis, module and applications. In *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, pages 381–390, June 1995.

[60] Jared Hulbert. Introducing the advanced xip file system. In *Linux Symposium*, page 211, 2008.

[61] Andy A. Hwang, Ioan A. Stefanovici, and Bianca Schroeder. Cosmic rays don't strike twice: Understanding the nature of dram errors and the implications for system design. *SIGPLAN Not.*, 47(4):111–122, March 2012.

[62] D. Jewett. Integrity s2: a fault-tolerant unix platform. In *Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., Twenty-First International Symposium*, pages 512–519, June 1991.

[63] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using asynchronous message logging and checkpointing. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 171–181, New York, NY, USA, 1988. ACM.

[64] Kenichi Kourai and Shigeru Chiba. A fast rejuvenation technique for server consolidation with virtual machines. In *DSN*, pages 245–255. IEEE Computer Society, 2007.

[65] Troy LeBlanc, Rakhi Anand, Edgar Gabriel, and Jaspal Subhlok. Volpexmpi: An MPI library for execution of parallel applications on volatile nodes. In *Lecture Notes in Computer Science: Proceedings of the* 16^{th} *European PVM/MPI Users' Group Meeting (EuroPVM/MPI) 2009*, volume 5759, pages 124–133, Espoo, Finland, September 7-10, 2009. Springer Verlag, Berlin, Germany.

[66] Dong Li, Zizhong Chen, Panruo Wu, and Jeffrey S. Vetter. Rethinking algorithm-based fault tolerance with a cooperative software-hardware approach. In *Supercomputing*, 2013.

[67] Sheng Li, Ke Chen, Ming-Yu Hsieh, Naveen Muralimanohar, Chad D. Kersey, Jay B. Brockman, Arun F. Rodrigues, and Norman P. Jouppi. System implications of memory reliability in exascale computing. In *Supercomputing*, pages 46:1–46:12, 2011.

[68] Franklin T. Luk and Haesun Park. An analysis of algorithm-based fault tolerance techniques. *J. Parallel Distrib. Comput.*, 5(2):172–184, April 1988.

[69] Naoya Maruyama, Akira Nukada, and Satoshi Matsuoka. Software-based ECC for GPUs. July 2009.

[70] Sarah E. Michalak, Kevin W. Harris, Nicolas W. Hengartner, Bruce E. Takala, and Stephen A. Wender. Predicting the number of fatal soft errors in Los Alamos National Laboratory's ASC Q supercomputer. *IEEE Transactions on Device and Materials Reliability (TDMR)*, 5(3):329–335, 2005.

[71] Subhasish Mitra, Norbert Seifert, Ming Zhang, Quan Shi, and Kee Sup Kim. Robust system design with built-in soft-error resilience. *Computer*, 38(2):43–52, 2005.

[72] Shubhendu S. Mukherjee, Michael Kontz, and Steven K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the* 29^{th} *Annual International Symposium on Computer Architecture (ISCA) 2002*, pages 99–110, Anchorage, AK, USA, May 25-29, 2002. IEEE Computer Society.

[73] Nichamon Naksinehaboon, Narate Taerat, Chokchai Leangsuksun, Clayton Chandler, and Stephen L. Scott. Benefits of software rejuvenation on hpc systems. In *ISPA*, pages 499–506. IEEE, 2010.

[74] N. Oh, P.P. Shirvani, and E J McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, 51(1), mar 2002.

[75] N. Oh, P.P. Shirvani, and E.J. McCluskey. Control-flow checking by software signatures. *IEEE Transactions on Reliability*, 51(1), mar 2002.

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

[77] Ian Philp. Software failures and the road to a petaflop machine. In *HPCRI: 1st Workshop on High Performance Computing Reliability Issues, in Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA-11)*. IEEE Computer Society, 2005.

[78] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure trends in a large disk drive population. In *USENIX Conference on File and Storage Technologies*, 2007.

[79] James S. Plank, Youngbae Kim, and Jack J. Dongarra. Fault-tolerant matrix operations for networks of workstations using diskless checkpointing. *Journal of Parallel and Distributed Computing*, 43(2):125 – 138, 1997.

[80] James S Plank, Jian Xu, and Robert HB Netzer. Compressed differences: An algorithm for fast incremental checkpointing. *University of Tennessee, Tech. Rep. CS-95-302*, 1995.

[81] Heather Quinn and Paul Graham. Terrestrial-based radiation upsets: A cautionary tale. In *Symposium on Field-Programmable Custom Computing Machines (FCCM) 2005*, pages 193–202, April 18-20, 2005.

[82] M. Rebaudengo, M.S. Reorda, M. Violante, and M. Torchiano. A source-to-source compiler for generating dependable software. In *Workshop on Source Code Analysis and Manipulation*, 2001.

[83] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient fault detection via simultaneous multithreading. In *International Symposium on Computer Architecture*, pages 25–36, 2000.

[84] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. Swift: Software implemented fault tolerance. In *International Symposium on Code Generation and Optimization*, 2005.

[85] Michael Rieker, Jason Ansel, and Gene Cooperman. Transparent user-level checkpointing for the native posix thread library for linux. In *The International Conference on Parallel and Distributed Processi ng Techniques and Applications*, Las Vegas, NV, Jun 2006.

[86] B Roundtree, G Cobb, T Gamblin, M Schulz, B Supinski, and H Tufo. Parallelizing heavyweight debugging tools with mpiecho. In *High-performance Infrastructure for Scalable Toolsi, WHIST 2011, Held as part of ICS '11, Tucson, Arizona*, pages 803–808, 2011.

[87] Horst Schirmeier, Martin Hoffmann, Christian Dietrich, Michael Lenz, Daniel Lohmann, and Olaf Spinczyk. FAIL*: An open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance. In *Proceedings of the 11th European Dependable Computing Conference (EDCC '15)*, pages 245–255. IEEE Computer Society Press, September 2015.

[88] Bianca Schroeder and Garth A Gibson. Understanding failures in petascale computers. *Journal of Physics: Conference Series*, 78(1):012022, 2007.

[89] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: A large-scale field study. In *Proceedings of the* 11^{th} *Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS) 2009*, pages 193–204, Seattle, WA, USA, June 11-13, 2009. ACM Press, New York, NY, USA.

[90] J. Shalf. Simulation challenge: Exascale planning overview. Invited Talk at HEC FSIO R&D Workshop, August 2010.

[91] P.P. Shirvani, N.R. Saxena, and E.J. McCluskey. Software-implemented edac protection against seus. *IEEE Transactions on Reliability*, 49(3), sep 2000.

[92] Alex Shye, Joseph Blomstedt, Tipp Moseley, Vijay Janapa Reddi, and Daniel A. Connors. PLR: A software approach to transient fault tolerance for multicore architectures. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 6(2):135–148, 2009.

[93] Daniel P. Siemwiorek. Architecture of fault-tolerant computers: An historical perspective. *Proceedings of the IEEE*, 79(12):1710–1734, 1991.

[94] Joel R. Sklaroff. Redundancy management technique for space shuttle computers. *IBM Journal of Research and Development*, 20(1):20–28, 1976.

[95] "V Sridharan and D Liberty". A study of dram failures in the field. In *Supercomputing*, Nov 2012.

[96] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. Memory errors in modern systems: The good, the bad, and the ugly. *SIGPLAN Not.*, 50(4):297–310, March 2015.

[97] Vilas Sridharan, Jon Stearley, Nathan DeBardeleben, Sean Blanchard, and Sudhanva Gurumurthi. Feng shui of supercomputer memory: Positional effects in dram and sram faults. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 22:1–22:11, New York, NY, USA, 2013. ACM.

[98] Michael M. Swift, Steven Martin, Henry M. Levy, and Susan J. Eggers. Nooks: An architecture for reliable device drivers. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, pages 102–107, New York, NY, USA, 2002. ACM.

[99] Paola Velardi and Ravishankar K. Iyer. A study of software failures and recovery in the mvs operating system. *IEEE Trans. Computers*, 33(6):564–568, 1984.

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

[101] Wikipedia. Dependability — wikipedia, the free encyclopedia, 2015. [Online; accessed 15-June-2015].

[102] Sangho Yi, Junyoung Heo, Yookun Cho, and Jiman Hong. Adaptive page-level incremental checkpointing based on expected recovery time. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, SAC '06, pages 1472–1476, New York, NY, USA, 2006. ACM.