© 2017 by Jon Cameron Calhoun. All rights reserved.

FROM DETECTION TO OPTIMIZATION: IMPACT OF SOFT ERRORS ON HIGH-PERFORMANCE COMPUTING APPLICATIONS

BY

JON CAMERON CALHOUN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Doctoral Committee:

Professor Luke N. Olson, Chair
Professor Marc Snir, Director of Research
Professor William D. Gropp
Dr. Franck Cappello, Argonne National Laboratory

Abstract

As high-performance computing (HPC) continues to progress, constraints on HPC system design forces the handling of errors to higher levels in the software stack. Of the types of errors facing HPC, soft errors that silently corrupt system or application state are among the most severe. The behavior of HPC applications in the presence of soft errors is critical to gain insight for effective utilization of HPC systems. The need to understand this behavior can be used in developing algorithm-based error detection guided by application characteristics from fault injection and error propagation studies. Furthermore, the realization that applications are tolerant to small errors allows optimizations such as lossy compression on high-cost data transfers. Lossy compression adds small user controllable amounts of error when compressing data, to reduce data size before expensive data transfers saving time. This dissertation investigates and improves the resiliency of HPC applications to soft errors, and explores lossy compression as a new form of optimization for expensive, time-consuming data transfers.

To my beautiful wife.

Acknowledgments

This dissertation would not be possible without the support of many people. My advisors, Luke N. Olson and Marc Snir, deserve many thanks for providing valuable insight and guidance in my development as a researcher and as a professional. I also thank my committee members, William D. Gropp and Franck Cappello, for their thoughtful comments and guidance. Thanks to the National Center for Supercomputing Applications and the National Science Foundation for supporting my research with a Blue Waters Graduate Fellowship. Finally, I owe my wife, parents, brothers, family, and friends a debt of gratitude for their constant love and support.

Table of Contents

List of Tables . . . . **vii**

List of Figures . . . . **viii**

Chapter 1 Introduction **1**
1.1 Classification of Errors 1
1.1.1 Hard Errors 2
1.1.2 Soft Errors 4
1.1.3 Going Forward 6

Chapter 2 Soft Error Detection and Recovery for Algebraic Multigrid **8**
2.1 Introduction 8
2.2 Algebraic Multigrid 9
2.3 Transient Error Detection 10
2.3.1 Multi-level Recovery Scheme 11
2.3.2 Soft Error Detectors 12
2.3.3 Segmentation Fault Recovery 14
2.4 Experimental Results 14
2.4.1 Overhead 15
2.4.2 Fault Characteristics 16
2.4.3 Convergence Analysis 18
2.5 Performance Model 20
2.5.1 Basic Model 20
2.5.2 Segmentation Fault Recovery 21
2.5.3 Residual and Energy Check 22
2.5.4 Comparison of Time to Solution 22
2.6 Conclusions 23

Chapter 3 Error Propagation **25**
3.1 Introduction 25
3.2 Motivation 26
3.3 Experimental Results 28
3.3.1 Testing Methodology 28
3.3.2 Micro-level Propagation 30
3.3.3 Macro-level Propagation 36
3.4 Conclusion 46

Chapter 4 Fault Injection **54**
4.1 Introduction 54
4.2 Background 54
4.3 Fault Injector 55
4.3.1 LLVM Compiler Pass Design 55

4.3.2 Usability and Extensibility . . . 59
4.4 Experimental Results . . . 63
4.4.1 Scalability . . . 63
4.4.2 Selective Injection . . . 64
4.5 Conclusion . . . 65

Chapter 5 Soft Error Propagation Tracking . . . 68
5.1 Introduction . . . 68
5.2 Tracking Propagation . . . 68
5.2.1 Micro-level Propagation Tracking . . . 69
5.2.2 Macro-level Propagation Tracking . . . 74
5.2.3 Weak Scaling Performance . . . 75
5.3 Conclusion . . . 76

Chapter 6 Lossy Compression . . . 78
6.1 Introduction . . . 78
6.2 Background . . . 80
6.3 Linking Compression Error to Numerical Error . . . 80
6.3.1 Lossy Compressor SZ . . . 81
6.3.2 Method . . . 81
6.4 Understanding Compression Error Behavior on 1D Model Problems . . . 83
6.4.1 1D Heat . . . 84
6.4.2 1D Advection . . . 85
6.5 Lossy Compressing Production Applications . . . 87
6.5.1 PlasComCM . . . 88
6.5.2 Nek5000 . . . 92
6.6 Modeling Time to Checkpoint . . . 96
6.7 Discussion . . . 98
6.8 Conclusion . . . 99

Chapter 7 Related Work . . . 100
7.1 Resilience in Algebraic Multigrid . . . 100
7.2 Error Propagation . . . 101
7.3 Fault Injection . . . 102
7.4 Lossy Compression . . . 103

Chapter 8 Conclusions . . . 104

References . . . 106

List of Tables

2.1 Percentage of injections in AMG components. 16
2.2 Percentage of trials that converge with a single injection. 18
2.3 Percentage of trials that converge with multiple injections. Average of 14 injections per trial for *Low Cost* and *All*. Average of 4 injections per trial for *No Detectors*. 19

3.1 Breakdown of failure symptom by instruction type. 30
3.2 Dynamic LLVM instruction type percentage. 34
3.3 Dynamic LLVM instruction percentage for `WAXPBY` kernel. 35
3.4 `WAXPBY` kernel failure symptom percentage as a function of compiler optimization. 35
3.5 Dynamic LLVM instruction classification percentage for SpMV kernel. 36
3.6 SpMV kernel failure symptom percentage as a function of compiler optimization. 36

4.1 Corrupt function's arguments (left to right). 57
4.2 FlipIt instruction classification types. 59
4.3 User callable functions. 60
4.4 Compiler pass arguments. 61
4.5 Command line arguments for fault injector. 62
4.6 Results of injecting into certain types. 65

List of Figures

1.1 Logical flow of error propagation from initial fault to subsequent failure. 2
1.2 Smaller feature sizes and lower voltage decrease the reliability of components (larger soft error rate). The **Nominal** curve illustrates past and present trends while the **Vscale_L**, **Vscale_M**, and **Vscale_H** curves assume low, medium and high amounts (respectively) of voltage scaling in future deep submicron technologies. The user-visible failure rates highlighted at 45 nm and 16 nm are calculated assuming a 92% system-wide masking rate [38]. 6

2.1 Impact on convergence of a single bit-flip error in a IEEE 754 double precision floating-point element of the residual vector. 9
2.2 AMG V-cycle process used to solve $A\mathbf{x} = \mathbf{b}$. 10
2.3 Overhead in solve time of fault detectors compared to the original AMG code. Each process has $16,384$ unknowns. 15
2.4 Characterization of cost of the energy check compared to total time on each level and cost relative to the energy check on level 0. Total level time is the sum of original level time and time for the energy check on that level. 17
2.5 Breakdown of faults injected into AMG components based on the type of instruction executed. 17
2.6 Convergence of AMG with multi-level recovery scheme. 20
2.7 Relative overhead in converging to a fixed tolerance for various values of β. 23
2.8 Speedup in time to solution relative to non-protected AMG (*No Detectors*). 23

3.1 Propagation of SDC via a sparse matrix-vector multiply. 27
3.2 Propagation of SDC by iterative method style reuse. 28
3.3 Latency (number of LLVM instructions) until a segmentation fault. 31
3.4 Bit positions where injected fault resulted in a segmentation fault. 31
3.5 Latency (number of LLVM instructions) until SDC detection. 32
3.6 Latency (number of LLVM instructions) until control flow diverges. 33
3.7 Average percentage in main variables of Jacobi. 37
3.8 Average corruption in selected variables on rank 3 for Jacobi within 90% confidence. 38
3.9 Log of average 2-norm of the error in main variables of Jacobi. 38
3.10 Average 2-norm of the error in selected variables on rank 3 for Jacobi within 90% confidence. 39
3.11 Log of average max-norm of the error in main variables of Jacobi. 39
3.12 Average max-norm of the error in selected variables on rank 3 for Jacobi within 90% confidence. 40
3.13 Average corruption in selected variables on rank 3 for HPCCG within 90% confidence. 42
3.14 Average percentage of corrupted vector elements for variables in the mini-app HPCCG in iterations after injection. 43
3.15 Average 2-norm of corrupted vector elements for variables in the mini-app HPCCG in iterations after injection. 44
3.16 Average 2-norm of the error in selected variables on rank 3 for HPCCG within 90% confidence. 45
3.17 Average max-norm of corrupted vector elements for variables in the mini-app HPCCG in iterations after injection. 46
3.18 Average max norm of the error in selected variables on rank 3 for HPCCG within 90% confidence. 47

3.19 Average percentage of corrupted vector elements for variables in the mini-app CoMD in iterations after injection. . . . 48
3.20 Average corruption in selected variables on rank 3 for CoMD. . . . 49
3.21 Average log of 2-norm for variables in the mini-app CoMD in iterations after injection. . . . 50
3.22 Average log of 2-norm in selected variables on rank 3 for CoMD. . . . 51
3.23 Average log of max-norm for variables in the mini-app CoMD in iterations after injection. . . 52
3.24 Average log of max-norm in selected variables on rank 3 for CoMD. . . . 53

4.1 Code transformation by FlipIt to inject faults. . . . 56
4.2 Partitioning of compressed argument for FlipIt's corrupting functions. . . . 57
4.3 ASCII version compiler log file. . . . 59
4.4 Blue text shows required changes to HPCCG source files (`HPCCG.cpp` and `Makefile`) and additional libraries to FlipIt should link against (`FlipIt/scripts/config.py`) to enable fault injection. . . . 60
4.5 Steps in compiling and instrumenting code with `flipit-cc` 61
4.6 Weak scaling of applications with and without compilation with FlipIt. . . . 66
4.7 Overhead of instrumentation of applications compared to reference code. . . . 67
4.8 Selective injection in residual calculation on rank 0. 16 processes with approximately 16,384 unknowns per process. . . . 67

5.1 Overview of the instrumentation process that transforms source code to be able to track propagation and inject faults. . . . 68
5.2 Overview of the code transformation for micro-level propagation tracking. . . . 70
5.3 Logic of store instrumentation call. . . . 72
5.4 Logic of load instrumentation call. . . . 72
5.5 Overhead of weak scaling applications instrumented to track corruption propagation. . . . 75
5.6 Weak scaling of applications with and without instrumentation for corruption propagation tracking. . . . 77

6.1 Overview of method to select the lossy compression error tolerance. . . . 82
6.2 Error at each grid-point in a 1D heat equation between numerical solution, u^h, and a numerical solution restarted from lossy checkpoints, $\widetilde{u}^h$, at time $t = 0.6, 1.1$. . . . 85
6.3 Maximum absolute error in the compressed numerical solution, $\widetilde{u}^h$ due to restarting from a lossy compressed checkpoint for the 1D heat equation (6.2). . . . 86
6.4 Error at each grid-point in a 1D advection equation between a normal numerical solution, u^h, and a numerical solution restarted from lossy checkpoints, $\widetilde{u}^h$, at times $t = 1.25$, 2.5, and 3.75. 87
6.5 Maximum absolute error in a compressed numerical solution, $\widetilde{u}^h$ due to restarting from lossy compressed checkpoints for the 1D advection equation (6.6). Restarts from a lossy checkpoint occur at times $t = 1.25$, 2.5, and 3.75. . . . 88
6.6 Overset mesh in PlasComCM adds a curvilinear mesh around cylindrical object (white circle). 89
6.7 Momentum magnitude after 60,020 time-steps for PlasComCM. The fixed cylinder object (white circle) obstructs the flow causing momentum to slow around the object in the direction of flow (to the right). . . . 90
6.8 Compression factors for PlasComCM. . . . 91
6.9 Compression time for PlasComCM. . . . 91
6.10 Propagation of momentum error in PlasComCM. . . . 92
6.11 Max-norm between numerical solution u^h and compressed numerical solution $\tilde{u}^h$ for PlasComCM. . . . 93
6.12 Magnitude of velocity after 30,000 time-steps for Nek5000. . . . 93
6.13 Smaller compression tolerances lead to higher compression factors for Nek5000. . . . 94
6.14 Time to compress for Nek5000 increases as the error tolerance decreases. Easy to compress data compresses faster, Figure 6.13. . . . 95

6.15 Max-norm between numerical solution u^h and compressed numerical solution $\tilde{u}^h$ for pressure and velocity for Nek5000. There is no error before time-step 1,500 as the simulation is not yet restarted. 96
6.16 Propagation of velocity error in Nek5000. 96
6.17 Fastest I/O configuration with various file system and compression bandwidths. 97

Chapter 1

Introduction

High-performance computing (HPC) has become a fundamental tool of science and discovery for many fields of science and industry. HPC resources allow for solving previously intractable computational problems. In doing so, HPC has cemented itself as the tool of scientific advancement for the twenty-first century.

HPC centers seek to optimize their impact on science by maintaining fast and efficient machines with high availability. HPC systems are composed of physical components that have a finite lifetime and are susceptible to environmental conditions. Current flagship HPC systems are composed of thousands of nodes each composed of many processing cores and memory modules. As new HPC systems grow ever larger and more powerful, design constraints such as cost and power consumption limits the reliability of future systems [96, 13, 98]. Decreasing the mean time between failure (MTBF), which can range from 4–8 hours [75, 95], forces the handling/consideration of errors to higher levels in the software stack. Mitigating the impact of errors on HPC applications is critical for efficient utilization of future machines.

1.1 Classification of Errors

This dissertation uses the definitions of fault, error, and failure from [5]:

Fault: hypothesized cause for an error;

Error: part of the system/application state that may lead to a subsequent failure; and

Failure: an event that occurs when an operation's result deviates from the intended result.

Figure 1.1 shows a logical flow of how a fault occurring in the system/application transitions to a failure. Once a fault has been activated, error is present in the system/application. Over time this error is masked by the system/application, or propagates corrupting more of the system/application state. If the error is not masked, a failure can occur. Failure detection is essential to ensuring usability and reliability for systems/applications. Moreover, if errors are detected before they cause a noticeable failure, then recovery time may be reduced.

Faults that occur in systems/applications come in two forms: permanent and transient [5]. Permanent

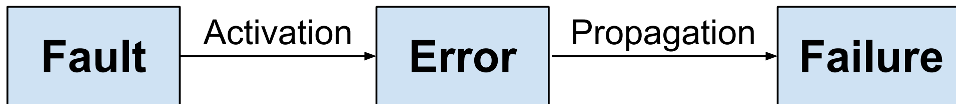


Figure 1.1: Logical flow of error propagation from initial fault to subsequent failure.

faults produce systematic and repeatable *hard errors* that are reproducible on retry or re-execution — e.g. inability to communicate with an offline node, an input file is not found, etc. Transient faults are not systematically reproducible and produce *soft errors* — e.g. radiation induced bit-flips, a packet drops in the network, etc.

1.1.1 Hard Errors

Hard errors are the result of permanent faults in the system. Their permanent nature allows them to be readily detected and corrected. Commonly they are handled in the fail-stop recovery model. In the fail-stop model, when a component fails, application progress halts due to the failed component no longer providing its intended service. In HPC, failure of a node or inability to communicate with a node in a parallel job is sufficient for most applications to experience a hang or a crash that results in a fail-stop failure of the application. Without an ability to recover the execution of the application, the application needs to be restarted from the beginning, wasting computational time and resources. The prevalent method of recovering from fail-stop failures in HPC is checkpoint-restart.

Checkpoint-restart

HPC checkpoint-restart relies on a short detection latency to minimize wasted work and traditionally employs synchronous roll-back recovery. To minimize wasted time, work has been done to find the optimal checkpointing period [105, 30]. System-level checkpoint-restart [58, 87, 45] offers the ability to recover from fail-stop failures — e.g. a process/node crash — in a method transparent to the application. However, system level checkpoints are expensive in time/memory, system dependent, and do not allow for migration between computing systems.

At large scales, system-level globally coordinated checkpoint-restart limits application performance due to coordination and I/O time [88]. Since most HPC applications achieve low file system bandwidth [73] during I/O, reducing the amount of checkpoint data written to the parallel file system improves performance. Application-based checkpointing schemes lower the amount of data in a checkpoint by allowing

HPC applications to select the specific data to be checkpointed [107].

Current application-level checkpointing methods leverage multiple levels in the memory hierarchy to improve time to checkpoint [11, 79]. Multi-level checkpointing schemes keep the recent, frequently-taken checkpoints in memory and only write a subset of all checkpoints to the parallel file system. To reduce coordination overhead, asynchronous checkpoint-restart schemes have been developed [83, 94, 42], but are susceptible to a domino effect of rolling back to older and older checkpoints to correctly recover [7]. Message logging [77] avoids a global restart by storing communication in in-memory checkpoints, allowing for recovery of the failed process by replaying the logged communication.

Although application-based checkpoint-restart and asynchronous checkpoint-restart do improve performance by reducing the volume of data that is checkpointed, eventually restart files need to be written to a stable non-volatile medium — e.g. a parallel file system. As new HPC systems come online, they expand the amount of main memory and increase the number of levels in the memory hierarchy, but retain the same file system bandwidth.

Current checkpointing schemes are limited by file system bandwidth as checkpoints eventually need to be written to the parallel file system for long term persistence and recoverability. This is exacerbated on newer machines as increasing core counts and reliance on cheaper commodity parts cause a decrease in the mean time between failure (MTBF) [13, 96]. Decreasing the MTBF causes an increase in the number of checkpoints taken putting increased pressure on the parallel file system. Consequently, new machines provide on-node burst buffers and NVRAM to cache I/O operations, thus increasing memory bandwidth to persistent storage [69]. Techniques such as data compression [99, 51, 6, 53] offer exciting avenues for reducing checkpoint sizes and mitigating negative effects of file system bandwidth.

Forward Recovery

To avoid the need to re-execute some portion of the computation, forward recovery seeks to create a new valid state by leveraging application specific properties or redundancies to recover/reconstruct data belonging to a failed process. The most well know version of forward recovery is RAID [86]. RAID allows for reconstruction of a failed disk by leveraging redundant data or encodings stored on other disks in the array. In the context of HPC applications that typically solve a time dependent partial differential equation (PDE) or ordinary differential equation (ODE), fail-stop failure of a process or node results in values for a region of the domain being missing. Missing data from the crashed processes are interpolated from non-crashed processes, or replaced with initial values [1, 27, 54].

1.1.2 Soft Errors

Another class of faults affecting HPC systems are transient faults that lead to soft errors. This class of faults has received an increased interest in the resilience community due to the complexity of their detection/recovery and their expected prevalence on future systems [23, 98].

Soft hardware errors arise from two distinct sources. The first source is through chip manufacturing where failure arises due to manufacturing defects, component aging, and contamination with radioactive materials. In 1978, Intel was unable to deliver chips to AT&T due to chip packaging modules being contaminated with trace amounts of uranium [76]. In 1986, IBM faced a similar problem of contamination. IBM traced their problem to a radioactive contaminate used to clean bottles that stored an acid needed in chip manufacturing. Chip manufacturing companies seek to obtain material from non-contaminated sources, but it is difficult to remove all sources radioactive contamination.

The second source is from alpha particles and neutrons from cosmic radiation from the sun, distant stars, and supernova that is constantly bombarding Earth. A 1979 survey [108] predicted the occurrence of soft errors at various locations and elevation. IBM later confirmed these predictions, and internally released the first report on cosmic radiation induced soft errors in 1984 [109]. Alpha particles interact directly with electrons in silicon based transistors, while neutrons interact via elastic or inelastic collisions. Inelastic collisions cause the majority of silent errors as neutrons break apart into smaller particles that interact with electrons in the transistors. Interference with electrons causes an incorrect signal to be generated or it could be masked. The incorrect signal is a soft error that is propagated to other components and can lead to silent data corruption (SDC) in system/application state if a failure does not occur. Eventually this can result in corruption of the application's output. Large scale DRAM studies show bit-flip events occur thousands of times per day [101] with variability between DRAM manufactures [100].

Figure 1.1 shows a logical flow of how a fault occurring in the system transitions to a failure. Once a fault as been activated, error is present in the system. During the execution of the remaining instructions this error can be masked due to programmatic or algorithmic properties, lead to a system detectable event such as a segmentation fault, or propagate to corrupt more of the execution state. As the detection latency increases, the chances of error propagation also increases.

Hardware Solutions

To mitigate long latencies and to provide protection against soft errors, extra hardware can be added that allows for efficient detection/correction of soft errors. For mission critical systems such as flight control computers and satellites, this redundancy often takes the form of triple module redundancy (TMR) [74]. In

TMR, results are computed by three copies of the hardware each using their own data. Results from each of the three components are sent to a majority voter. This allows for detection of errors in the system and not permitting them to become SDC.

Although TMR is effective on small mission critical systems, at the technique is to costly at the scale of HPC systems. HPC systems rely on commodity components that do not provide the full protections of TMR. Because SRAMs, DRAMs, and data-paths comprise a larger area than combinational logic for computation, historically they have been more vulnerable to soft errors. High-end server quality DRAMs, processor caches, and data-paths employ coding techniques — such as parity, single-error correction double-error detection (SEC-DED) ECC, Chipkill [32], cyclic redundancy check (CRC), to detect soft errors in stored or transmitted data. More complex hardware solutions, such as Relax [31] and SWAT [66], leverage application properties to recover at the micro architecture level.

Because of the higher cost in terms of transistors and power to protect, processor logic utilizes weaker protection mechanisms than used in memory technology. As feature size continues to shrink along with supply voltage, processor logic becomes increasingly susceptible to bit upset events that lead to SDC [15, 55]. Moreover, as HPC systems are becoming more complex and are using more commodity parts that do not provide high levels of protection in hardware, the rates of soft errors impacting applications on future machines is expected to increase and become a daily event [23, 38].

Software Solutions

Although hardware solutions to soft errors offer short detection latencies, fully protecting HPC systems from soft errors in hardware is prohibitively expensive. If algorithms and applications detect the occurrence of soft errors, then it is possible to estimate the severity of the error, and if beneficial, to correct and recover.

General methods for SDC detection leverage redundancy at the process level [39, 82], redundancy at the instruction level [25, 59, 81, 72, 84], or leverages anomalies in spatial and temporal similarities for detection [14, 9, 10, 12]. Techniques that utilize redundancy at the instruction level use various methods — e.g. machine-learning, fault injection experiments, probability models — to identify key instructions to replicate and add code to detect SDC. Higher level techniques that leverage spatial and temporal similarities use prediction techniques, such as curve-fitting or lower-order methods, to evaluate the validity of newly calculated data values.

Algorithm based fault tolerance (ABFT) has the potential to reduce the overhead and cost for detecting and recovering from SDC by leveraging algorithm dependent heuristics or invariants to detect and recover from SDC [48, 28, 19, 35, 54, 46]. Because ABFT is based on properties/heuristics of the application,

the severity of an error is accessed to determine if recovery is necessary. With the increased focus on detecting SDC inside the application, recent work has looked at its impact on HPC applications and how SDC propagates inside them [36, 22, 4, 65, 21].

1.1.3 Going Forward

Future HPC systems are expected to be constrained in many ways. Power consumption limits the size and types of components in the system [57, 13]. As chips are built with smaller feature sizes run at near-threshold voltage [34] to conserve power, the become more susceptible to soft errors [55, 15]. Figure 1.2 presents the historical trend along with estimates for current technology of the connection between feature size and voltage scaling on the soft error rate.

Next generation systems at Lawrence Livermore National Laboratory[1] and Oak Ridge National Laboratory[2] increase the amount of system memory by 5–9x without a commensurate increase in file system bandwidth. To improve I/O performance, systems will incorporate NVRAM. Reducing the volume of data communicated with the parallel file system helps further alleviate I/O bottlenecks.

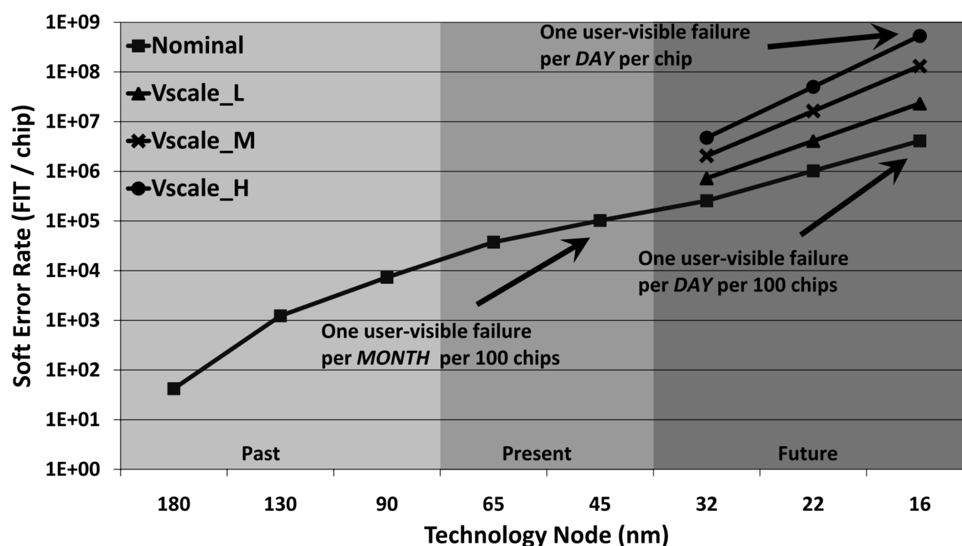


Figure 1.2: Smaller feature sizes and lower voltage decrease the reliability of components (larger soft error rate). The **Nominal** curve illustrates past and present trends while the **Vscale_L**, **Vscale_M**, and **Vscale_H** curves assume low, medium and high amounts (respectively) of voltage scaling in future deep submicron technologies. The user-visible failure rates highlighted at 45 nm and 16 nm are calculated assuming a 92% system-wide masking rate [38].

In an environment where failure is a reality, mitigation strategies must be imposed to ensure usefulness of HPC resources. These mitigation strategies need to provide a high degree of detection for errors that

[1]https://asc.llnl.gov/coral-info
[2]https://www.olcf.ornl.gov/summit/

cause disruption in the application's solution or time to solution at low cost. Moreover, data compression may be required in order to reduce the volume of data transferred since data movement is a major source of power consumption in HPC systems.

This dissertation makes the following contributions to the field of HPC:

- an LLVM based bit-flip style fault injection tool for HPC;
- an efficient detection and recovery scheme for the algebraic multigrid linear solver;
- an LLVM based error propagation tracking tool that emulates lock-step execution;
- study of error propagation HPC applications and a discussion of localized recovery options; and
- methodology for lossy compression error tolerance selection based on *a priori* bounds on simulation accuracy.

This dissertation consists of two parts. The first part concerns SDC detection and propagation, and is divided into four chapters: Chapter 2 presents an algorithmic based detection and recovery scheme for the linear solver algebraic multigrid; Chapter 3 investigates SDC propagation inside HPC applications, common symptoms when soft errors occur, and the ability current detection schemes to contain error propagation; the final two chapters, Chapter 4 and 5, present tools that are developed to aid HPC fault tolerance research. The second part of this dissertation, Chapter 6, explores how lossy compression is used to improve HPC checkpoint restart. Related work is presented in Chapter 7. Finally, this dissertation concludes in Chapter 8.

Chapter 2

Soft Error Detection and Recovery for Algebraic Multigrid

Portions of this chapter are taken from the publication "Towards a More Fault Resilent Multigrid Solver" [20]

2.1 Introduction

Many scientific applications, from modeling blood flow to electromagnetics, depend on linear algebra computations on sparse matrices. These types of computations often consume a sizable percent of a high-performance computing (HPC) resources. In particular, one crucial operation is the sparse, linear solve. Scalability and convergence of such methods are well studied, but the study of their behavior in the presence of hardware faults is less developed. Emerging HPC architectures are expected to experience higher levels of faults than previous architectures and understanding the impact of fault(s) on the *algorithm* is an important component in fully utilizing their resources. Consequently, resiliency techniques need to be developed and analyzed to allow linear solvers to remain efficient and scalable on emerging HPC architectures.

Modern scientific computing relies on solving large, *sparse* systems of linear equations. Sophisticated solvers are employed to take advantage of this sparsity, and as computing capabilities continue to progress so do the demands on the linear solver. One solver that has shown to be flexible across a range of different architectures is algebraic multigrid (AMG), due to its potential scalability, robustness, and efficiency as an $\mathcal{O}(n)$ complexity method, where n is the number of degrees of freedom in the problem. The focus of this chapter is on utilizing ABFT to improve resiliency of AMG. Understanding the level of resiliency that is provided by AMG on emerging architectures is important for AMG and other sparse, linear solvers.

To motivate the need for silent data corruption (SDC) detection and recovery in AMG, Figure 2.1 shows the residual history for the solution of a diffusion problem in 2D (cf. Section 2.4). In this example, a single fault is injected into the residual calculation before restriction during the second iteration on the finest level. This fault is a single bit-flip in the first element of the residual vector, an IEEE 754 double precision floating-point number, on process 0 of a 16 process MPI run. The fault does not lead to a segmentation fault or *segfault*, and is thus silent, but impacts the execution. Depending on *which* bit is flipped in this

location, the number of extra iterations required to achieve the convergence tolerance of $1e{-7}$ ranges from 0 (bits in the mantissa and the sign bit[1]) to more than double the original amount (bits in the exponent), thus motivating the need for SDC detection and recovery.

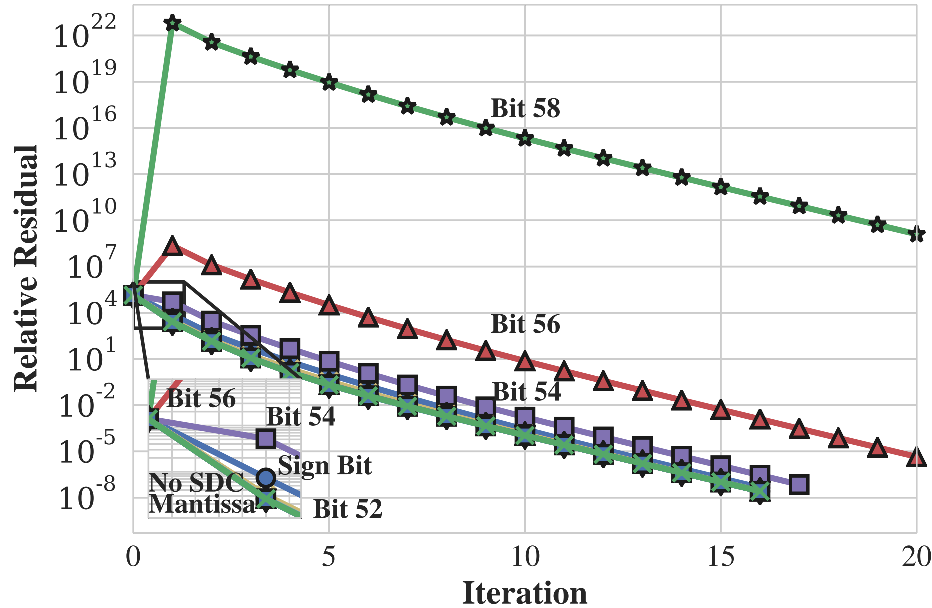


Figure 2.1: Impact on convergence of a single bit-flip error in a IEEE 754 double precision floating-point element of the residual vector.

To this end, we improve and analyze the resilience of AMG in the presence of soft errors. In particular, this chapter makes the following contributions:

- low overhead algorithmic based recovery for AMG;
- low cost SDC detectors for iterative linear solvers;
- AMG specific SDC detectors; and
- a performance model to analyze the time to solution in faulty environments.

2.2 Algebraic Multigrid

Consider the sparse matrix problem $A\mathbf{x} = \mathbf{b}$, where A is $n \times n$. In a parallel setting, iterative solution techniques are preferred due to the memory and complexity requirements to solve. One such approach is algebraic multigrid (AMG) [89], which is often used as a preconditioner for a Krylov method such as conjugate gradient (CG) or generalized minimum residual (GMRES).

[1] As floating-point values increase in magnitude, flipping a sign bit causes a larger deviation in value. This can lead to extra iterations.

AMG constructs a hierarchy of successively coarser problems that are used to iteratively refine the error in the solution. Figure 2.2 outlines a *solve* cycle of AMG, where an initial guess $\mathbf{x}^0$ is refined using two compatible operations: relaxation, such as weighted Jacobi, and coarse-grid correction, which is a subspace projection method. Relaxation is responsible for reducing high energy error, while coarse grid correction targets *algebraically* smooth error, or error that is invariant to relaxation. The cycle proceeds by successively coarsening the error equations, $A\mathbf{e} = \mathbf{r}$, until a coarse level problem is effectively solved in a few steps of relaxation or is efficiently processed with a direct method. Errors on coarse levels are then interpolated to correct solutions. Figure 2.2 represents the well-known V-cycle, and is a common approach to traverse the AMG hierarchy in parallel.

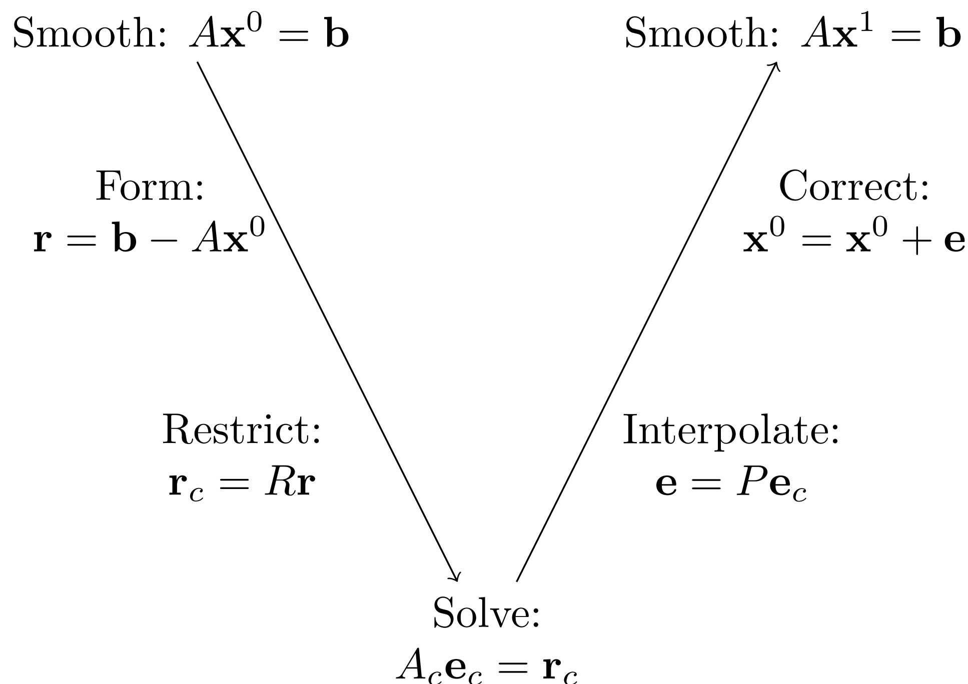


Figure 2.2: AMG V-cycle process used to solve $A\mathbf{x} = \mathbf{b}$.

2.3 Transient Error Detection

Before presenting the SDC detectors, we first discuss our fault model and assumptions. For this study, we assume that memories are sufficiently protected with ECC and Chipkill [32]; as such, we do not model faults in memory. Further, we assume that faults occur during instruction execution of the solve phase of AMG and results in a single-bit perturbation in the result register. In addition, during the solve phase, the operators A, P, and R on each level as well as the right hand side $\mathbf{b}$ are never written and only read. As

a result, these structures are not checkpointed for SDC recovery. They are susceptible to errant stores that corrupt the matrix; however, in practice this is a very rare occurrence (<0.01% of runs). If it is required to protect these data structures from such writes, one approach is to use the system call `mprotect` to force the associated memory pages to be read-only. In the case of a write to a protected page, a segmentation fault is raised triggering recovery. This function is used in some asynchronous checkpoint-restart approaches [83] to exploit copy on write capabilities when scheduling pages to be checkpointed.

2.3.1 Multi-level Recovery Scheme

Faults in AMG are classified as the following: a fault

- decreases the convergence time;
- increases convergence time;
- leads to convergence to wrong solution; or
- results in an unexpected program termination (crash).

In the case of divergence due to an unexpected termination — e.g. segmentation fault — this is a clear indication that an error has occurred. The other possible results from faults lead to silent data corruption (SDC), but the algorithm continues to function. SDC may also lead to longer runtimes or to converging to the wrong solution. This chapter relies on mathematical theory and heuristics of AMG to detect these errors as discussed below.

Checkpoint-restart is the standard method to recover from failure by allowing the application to crash and restart from a checkpoint or beginning state. This chapter utilizes this idea to recovery from detected soft errors. By observing the traversal of the AMG hierarchy, each level is considered an *implicit* checkpoint or containment domain [29], since the data for that level is computed via the previous level in the hierarchy. Upon detection, AMG assumes the previously executed level is correct and restarts execution on the previous level. However, if the same detector flags an error in the same location, or if the error is deemed severe, then AMG restarts the V-cycle. The code executed by the multi-level recovery scheme is found in Algorithm 2.1. After a fault is detected, its severity is measured. Provided the fault is minor, recovery proceeds from the previous level in the AMG hierarchy. If a SDC is flagged in the same location again or the residual check is triggered, then recovery rolls back to the previous *explicit* checkpoint and restarts the solve on the finest level.

In order for the restart routine to successfully restart on the appropriate level, global state information comprised of the direction, level, and iteration of the solver is logged as AMG traverses the hierarchy. The `C`-language function `sigsetjmp` creates restart points for the level and V-cycle recovery. Recovery proceeds

Algorithm 2.1: Multi-level Restart

```
1  if retry_same_level or residual_check_failed then
2  |   restoreFromCheckpoint(x^k)                       {in-memory, fine level}
3  |   restartLevel(level_finest)
4  if down_pass then
5  |   if level ≠ level_finest then
6  |   |   restartLevel(level − 1)
7  |   if iteration > 0 then
8  |   |   up_pass ← TRUE
9  |   |   down_pass ← FALSE
10 |   |   restartLevel(level + 1)
11 |   else
12 |   |   restartLevel(level_finest)
13 if up_pass then
14 |   if level ≠ level_coarsest then
15 |   |   restartLevel(lev + 1)
16 |   else
17 |   |   up_pass← FALSE
18 |   |   down_pass← TRUE
19 |   |   restartLevel(level − 1)
```

with a function call to `siglongjmp`. These functions, commonly used for exception handling in `C`, store and restore the register state of the program at the point of the call respectively — i.e. a label and a non-local `goto`. In addition, the solution vector periodically creates an in-memory checkpoint at the end of the V-cycle to limit the amount of roll back required. These minor augmentations are designed to be generic, allowing them to be added to any sequential or parallel AMG implementation with minimal effort.

2.3.2 Soft Error Detectors

Faults that do not lead to an unexpected termination become SDC and go unnoticed by the standard AMG algorithm. This subsection discusses simple augmentations to AMG that allow the detection of SDC. These augmentations vary in their overhead, applicability to other codes, coverage, and recovery cost, as detailed below.

Residual Check

The typical stopping criterion for AMG is verifying that the relative residual is less than a given tolerance. If SDC occurs during a V-cycle, its effect is present in the residual calculated for that V-cycle and every V-cycle thereafter until the corruption is removed. This is illustrated in the motivating example, Figure 2.1. Unlike other iterative linear solvers, AMG solvers do not guarantee a reduction in the residual or relative

residual monotonically from V-cycle to V-cycle. Heuristics show that for a large class of problems, the residual is expected to decrease *nearly* monotonically; therefore, this work devises a low cost SDC check that examines the newly calculated residual and compares it to the previous residual. Due to the non-monotonically decreasing nature of the residual, a region of plausibility for the new residual is established. This work considers a single order-of-magnitude difference as indication of SDC; while this is subjective, it is important to note that the detection and recovery scheme is attempting to save the solver from a total restart (see Section 2.5). In addition, the scaling factor in the residual check can be modified depending on the type of problem being solved to enhance its ability to detect SDC and to prevent infinite loops. This SDC detector is not AMG specific and extends to other iterative solvers.

Energy Check

AMG does not guarantee that the residual decreases monotonically at the end of every V-cycle. However, it does guarantee a decrease in the A-norm of the error every V-cycle. Although it is not generally possible to measure the error directly, it is possible to use AMG's sense of energy, termed the *energetic stability*, to check for SDCs.

$$E = \langle A\mathbf{x}, \mathbf{x}\rangle - 2\langle \mathbf{x}, \mathbf{b}\rangle \tag{2.1}$$

The energetic stability is valid on each level: energy at level i in the down-pass of the V-cycle is greater than the energy calculated at level i in the down-pass in the next V-cycle. The so-called *energy check* helps ensure that the results calculated on a given level are without egregious errors before execution continues to the next level. This allows recovery to proceed by the multi-level restart algorithm shown in Algorithm 2.1.

In its current form, (2.1) is costly as it requires one sparse matrix-vector multiply (SpMV) and two inner products. The latter is a form of synchronization and limits scalability at large core counts. On the down-pass of a V-cycle and not on the coarsest level, the residual is formed. This operation provides us with $A\mathbf{x}$ at no cost. On the up-pass, the computation is more expensive, but by rewriting (2.1) in the following form

$$E = \langle \mathbf{r}, \mathbf{b}\rangle - 2\langle \mathbf{x} - \mathbf{b}, \mathbf{b}\rangle, \tag{2.2}$$

reduces scalability issues with the inner products are lowered by issuing a single collective reduce for both inner products.

2.3.3 Segmentation Fault Recovery

Unlike the results of the energy or residual check, where there is a global view of the result of each check once the operation completes, segmentation faults are local to the process on which they occur. Once a segmentation fault signal has been detected by the runtime environment, the all parallel processes in application need to be terminated. One approach to segmentation fault recovery is to elevate a local segmentation fault to a global failure inside the application, thus allowing recovery via Algorithm 2.1. However, such a scheme causes high overheads and increases the amount of work redone. A parallel application is considered as a decomposition into two phases: communication and computation. AMG is composed of basic linear algebra operations such as matrix-vector and vector operations. All of these operations have an idempotent form at the expense of a temporary vector. For example, $\mathbf{y} \leftarrow A * \mathbf{x}$ produces the same result independent of the number of times the operation is executed. Due to there idempotent nature, recovery from a segmentation fault is straightforward: restart the operation by the use of the same `C` functions required for Algorithm 2.1, but use new recovery points for the idempotent operations. The error is transient and does not manifest itself as a segmentation fault again. If the segmentation fault occurs during a non-idempotent operation such as communication, recovery by restart using Algorithm 2.1 is possible. Yet, important state information, may be corrupted, for example by redundantly sending/receiving messages. Message logging is able to eliminate redundant messages, but still faces possible corruption in the communication library. As the results in Section 2.4 show, segmentation faults in non-idempotent regions represent a small portion of segmentation faults (less than 3%) for the results in Table 2.2 and Table 2.3; therefore, the extra overhead is not added to the recovery scheme.

2.4 Experimental Results

To better analyze, the impact of soft errors on AMG, the open-source LLVM fault injector FlipIt (see Chapter 4) injects single bit-flip errors into a random dynamic LLVM instruction during the solve phase of the AMG solver in HYPRE[2]. In the following (except Section 2.4.1), 1,000 fault injection trials are preformed on Blue Waters, a Cray supercomputer managed by the National Center for Supercomputing Applications and supported by the National Science Foundation and the University of Illinois[3]. Each compute node we utilize has 2 AMD 6276 Interlagos CPUs and 64 GM of RAM. The problem solved in all trials is a 2D Laplacian with zero on the boundaries using 16 processes and $16,384$ unknowns per process. Because of its rapid convergence and lightweight hierarchy, this problem highlights the overhead of our detectors and recovery

[2]`https://computation-rnd.llnl.gov/linear_solvers/software.php`
[3]https://bluewaters.ncsa.illinois.edu/

scheme with more severity than a more difficult problem where the solver itself is more computationally heavy. Moreover, due to the rapid convergence for this model problem, the numerical errors are quickly attenuated by the solve phase. When a fault is injected it is classified by FlipIt as one of three main types: *Pointer* refers to all calculations directly related to the use of pointers (loads, stores, and address calculation), *Control* refers to all calculations of branching and control flow (comparisons for branches and modification of loop control variables), *Arithmetic* refers to pure mathematical operations (adds, multiplies).

2.4.1 Overhead

By their nature, transient faults are infrequent events, which implies that any resiliency scheme should limit the overhead introduced in a fault free case. To determine the practicality of the SDC detectors, it is critical to assess both their ability to detect as well their cost (overhead). Figure 2.3 details the overhead with respect to solve phase execution time of each SDC detector. The multi-level recovery scheme is enabled for every configuration in which a detector is active. Since the overhead is measured during a fault free case, the multi-level restart code is never executed; therefore, the time required to restart the AMG solve is not reported in the timings. Furthermore, the energy check is enabled on all levels along with an in-memory checkpoint of the solution vector at the end of every V-Cycle to test the worst case scenario for overhead.

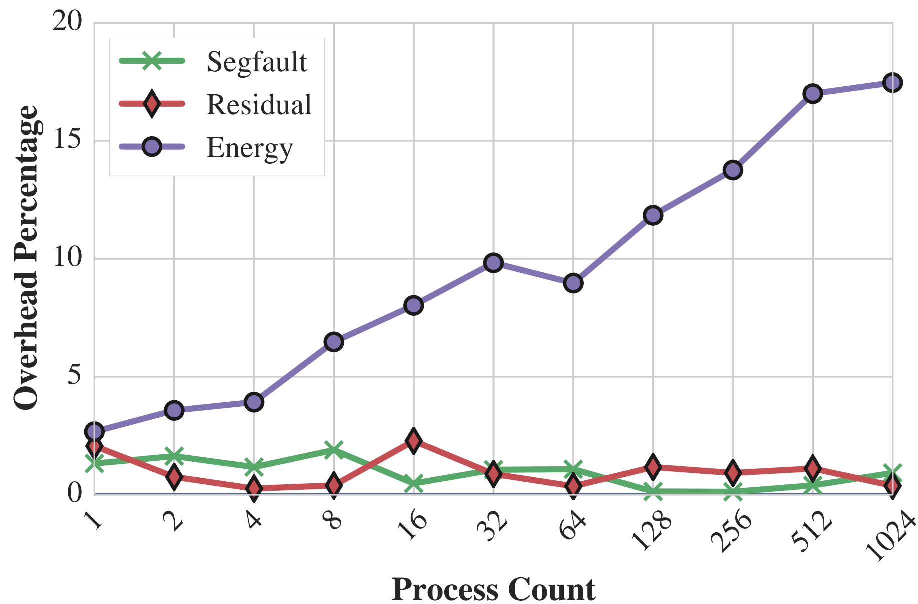


Figure 2.3: Overhead in solve time of fault detectors compared to the original AMG code. Each process has $16,384$ unknowns.

Segmentation Fault (*Segfault*) recovery and the residual check (*Residual*) yield a lower overhead than an energy check (*Energy*). In fact, their runtimes are similar to the original code, differing by less than 1% on average. In contrast, the energy check is more expensive. This is offset by the fact that the energy check is more powerful, enabling detection of SDCs at each level. Performing the energy check on each level limits the amount of work that needs to be redone. The segmentation fault recovery and the residual check have a low combined overhead of less than 1.9% on average; therefore, these two detectors are combined to form the *Low Cost* resiliency configuration. Even with all detectors enabled in a worst-case scenario, the observed maximal overhead is less than 20%. In practice, the frequency of a checkpoint, and the location of an active energy check would be modified to meet a strict resiliency budget.

To better characterize the overhead of the energy check, Figure 2.4 shows the percentage of level time consumed by the energy check. Moving from the finest level (0) to the coarsest level (10) of the V-cycle the problem size decreases, but each level becomes more dense. As the density of the level increases, the energy check has a greater influence on level time. Although there is an increase in percentage of level time, the energy check is cheaper on the coarsest levels than on the finest level. However, for AMG hierarchies that have coarse levels that consume more execution time than fine levels, protecting only the coarse levels may yield high overheads. The energy check is relatively more expensive on level 1 because although the number of unknowns in the problem is less on level 1, the number of non-zero entries is roughly the same leading to a denser matrix used in the check.

2.4.2 Fault Characteristics

In order to devise effective resiliency schemes, it is important to determine where faults are injected into the AMG solve phase. As in Section 2.4.1, the worst-case scenarios for both the residual and energy check are explored. Faults are injected randomly throughout the solve phase with each dynamic instruction having a uniform probability of $1e{-8}$. Seeding the fault injector differently for each trial yields injections in all iterations of the solve.

Operation	No Detectors	Low Cost	All
Relaxation	50.1%	47.4%	42.5%
SpMV	47.0%	49.1%	48.2%
Inner product	1.1%	1.7%	6.7%
Other	1.8%	1.8%	2.6%

Table 2.1: Percentage of injections in AMG components.

Table 2.1 highlights that most of the faults are injected into *Relaxation* and the *SpMV* routines, which

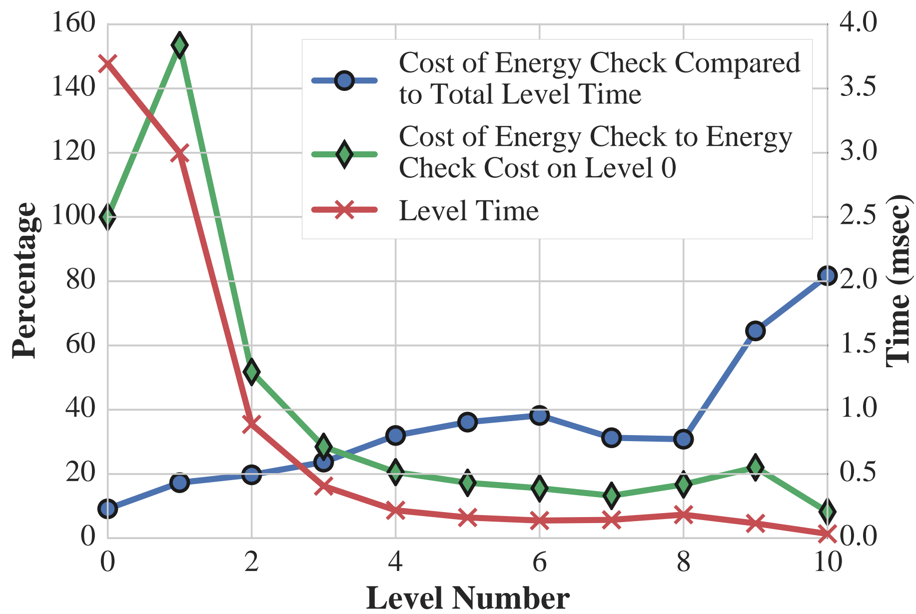


Figure 2.4: Characterization of cost of the energy check compared to total time on each level and cost relative to the energy check on level 0. Total level time is the sum of original level time and time for the energy check on that level.

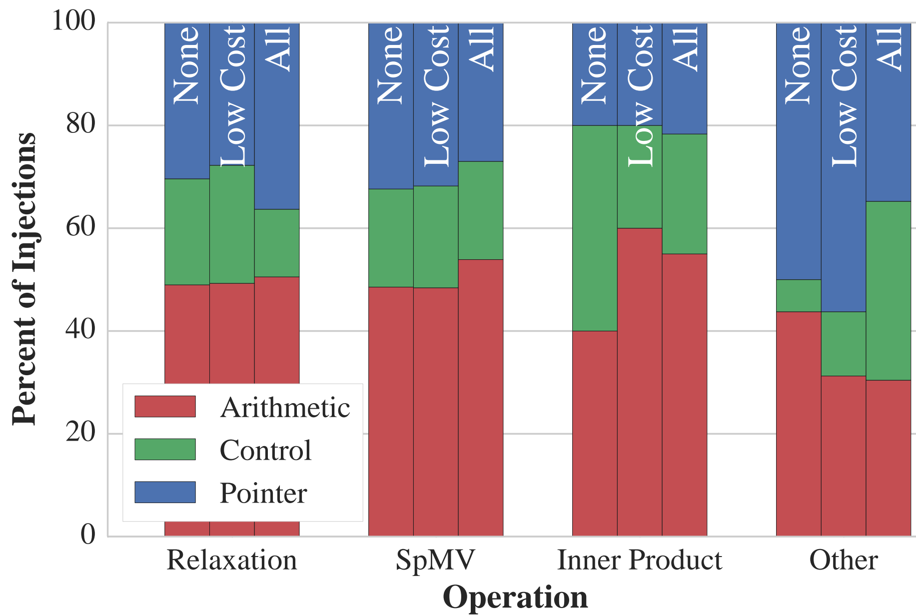


Figure 2.5: Breakdown of faults injected into AMG components based on the type of instruction executed.

includes A for the residual, P, and R. This is expected since most of the solve time is spent in these routines. The classification *Other* comprises all other functions used during the HYPRE solve phase — e.g. cycling code, vector copy, and scaling routines. For configurations where the energy check is enabled, there is an increase in the frequency of injections into the inner product routine. This is a result of the energy check requiring an inner-product.

Figure 2.5 shows what type of instruction where fault injection occurs. A significant number of faults are injected into *Pointer* and *Arithmetic* instructions. The high degree of *Pointer* injections is due to using sparse matrix data structures which uses indirection to access the data elements. A corrupted pointer often leads to a segmentation fault, and corruption of *Arithmetic* computation can produce results as shown in Figure 2.1, thus motivating the need for an efficient detection and recovery scheme.

Adding SDC detectors increases resilience, but suffers from occasionally from false positives. The comparisons for the residual and energy check represent a small portion of the dynamic instructions during the solve phase and are unlikely to experience a SDC. The computation to form the quantities for the comparisons are more likely candidates. Regardless of where SDC occurs, a false positive in the residual or energy check is handled as if it is a true positive.

2.4.3 Convergence Analysis

The following convergence study uses the same initial guess and right hand side to limit variability in the results. Moreover, since an injected fault may lead to an increase in the number of iterations, the maximum number of iterations is set at 20. This allows four more iterations to converge than the fault-free solve.

Results	No Detectors	Low Cost	All
Converged	73.5%	98.6%	99.0%
Did not converge	1.0%	0.4%	0.0%
Crashed	25.5%	1.0%	1.0%

Table 2.2: Percentage of trials that converge with a single injection.

Table 2.2 shows that a single injection has a large impact on convergence, with 25.5% of all trials segmentation faulting before convergence in the unmodified version of AMG. A high rate of segmentation faults is expected since one-third of injections are into instructions classified as *Pointer*. In configurations where segmentation fault recovery is enabled, the segmentation fault recovery successfully recovers allowing the solve to continue. With the aid of the SDC detectors, 12% of injections are caught. If uncaught, these injections lead to extra iterations.

With multiple injections, Table 2.3 shows a large deterioration in convergence for the unprotected AMG

Results	No Detectors	Low Cost	All
Converged	0.0%	86.2%	84.8%
Did not converge	0.0%	2.8%	0.2%
Crashed	100.0%	11.0%	15.0%

Table 2.3: Percentage of trials that converge with multiple injections. Average of 14 injections per trial for *Low Cost* and *All*. Average of 4 injections per trial for *No Detectors*.

implementation with all trials crashing due to segmentation faults. However, the protected versions of AMG remains convergent even in the presence of a high number of faults. Enabling the energy check increases sensitivity to SDC allowing increased detection, but the check also incurs a cost. The energy check flags SDC at a rate that is 1.5× that of the residual check, but recovering from the SDC is more expensive than letting the AMG naturally iterate through the error. Some SDC causes a small increase in the energy from the previous iteration that triggers the energy check, and recovery proceeds with the multi-level restart algorithm. However, the error due to the SDC does not impact the number of iterations to converge. In addition, the energy check also increases the parallel communication in routines where idempotent segmentation fault recovery is not possible. Consequently, as the number of faults encountered per solve increases, the energy check becomes less useful.

Figure 2.6, looks at how convergence is affected by the number of injections in both the *Low Cost* and *All* configurations. In Figure 2.6, the average number of injections in each trial increases until the probability of convergence drops below 0.5. The probability of convergence is computed by dividing the number of trials that converge by the number of trials.

Even with an average of 37 faults injected during each trial, the resilient versions of AMG converges in over 69% of trials for *Low Cost* and in over 62% of trials for *All*. Again the energy check is able to detect more SDC, even those that do not significantly affect convergence, but has a lower probability of convergence. After this point in the testing, the probability of convergence decreases dramatically. This is because the solver is making little progress due to high amounts of recovery and because the segmentation faults are emerging in non-idempotent routines. Indeed, 54% of trials of the *Low Cost* configuration and 75% of the *All* trials terminate due to segmentation faults in non-idempotent regions. This suggests that if the code is restructured with more idempotent regions, convergence rates would improve. For the remaining faults, when a SDC is detected recovery proceeds via the multi-level recovery scheme, but during the recovery process another fault requiring another restart occurs, allowing almost no forward progress.

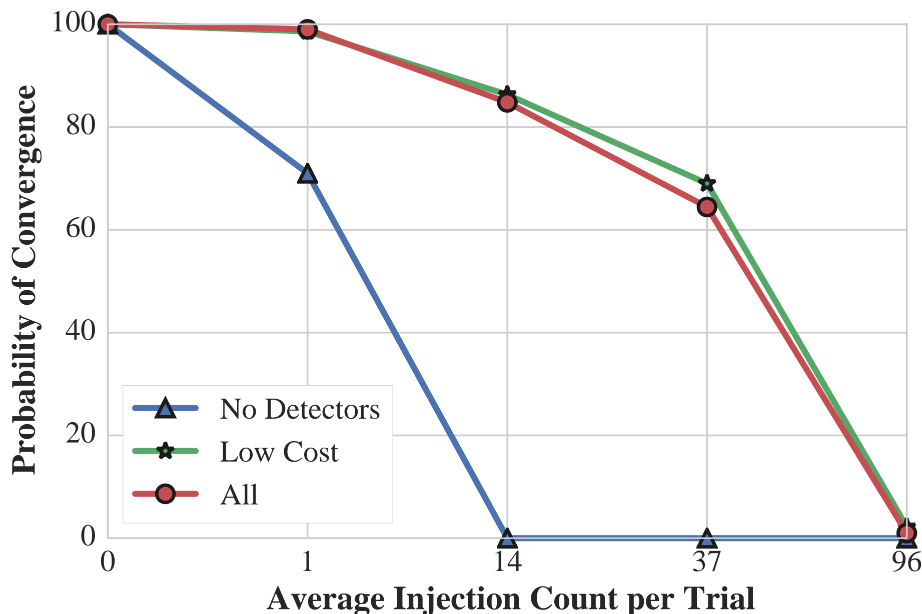


Figure 2.6: Convergence of AMG with multi-level recovery scheme.

2.5 Performance Model

This section constructs a performance model in order to highlight the scalability of the approach and the impact on computational complexity in the scheme. The performance model models the algorithmic based detectors and multi-level recovery scheme in order to understand the impact on the convergence properties of the AMG solver and to see the potential benefits of the methodology. Section 2.4.3 presents results from fault injection experiments on solves of AMG with various error rates and detectors enabled.

2.5.1 Basic Model

AMG is divided into two phases: setup and solve. These phases do not overlap leading to the basic performance model for the runtime of the solution to $A\mathbf{x} = \mathbf{b}$:

$$T_{\text{total}} = T_{\text{setup}} + T_{\text{solve}} \tag{2.3}$$

During the solve phase of AMG, there are several choices that influence the performance of the solver such as depth of the cycle, type of cycle, smoother, etc. In response, our performance model below (see (2.4)) is designed to abstract many of these choices yielding a practical tool, while at the same time providing enough detail to draw firm conclusions. T_{cycle} is the time for a single cycle (iteration) in AMG, and n_{cycles} is

the number of cycles until convergence. In our model, c_i and t_i are the number of visits to level i per cycle and time to visit level i, respectively. Since the fault model only considers faults in the solve phase, let β represent the percent increase in iteration count due to reduced convergence as result of a SDC. β is more accurately modeled by T_{repeat}, using r_i as the number of times a level is repeated due to a SDC. To account for resiliency in the model, the term o_i is added to t_i to account for the overhead of the added resiliency measures on level i. Thus, the time for the solve is modelled as

$$\begin{aligned} T_{\text{solve}} &= n_{\text{cycle}} T_{\text{cycle}} \left(1 + \beta\right) \\ &= n_{\text{cycle}} T_{\text{cycle}} + T_{\text{repeat}} \\ &= n_{\text{cycle}} \left(\sum_{i=0}^{n_{\text{level}}} c_i \left(t_i + o_i\right) \right) + \sum_{i=0}^{n_{\text{level}}} r_i \left(t_i + o_i\right) \end{aligned} \tag{2.4}$$

The number of cycles (iterations) required for AMG to converge to d digits of accuracy is given by $\frac{d}{-\log_{10}(\rho)}$, where ρ is the convergence factor, the spectral radius of the iteration matrix used during the relaxation step. If faults occur during the solve phase, convergence *potentially* deteriorates to $\hat{\rho} = \alpha\rho$, where α is the factor by which the convergence such that $\rho < \hat{\rho} < 1$. As convergence deteriorates, additional cycles are required to achieve convergence to the same accuracy. In some situations, the increase is dramatic as outlined in Figure 2.1. The proposed detection and recovery scheme limits the number of extra cycles required by maintaining the average convergence factor of the fault free problem in most cases; if the average convergence factor is only slightly affected, then a corresponding number of additional iterations are needed.

2.5.2 Segmentation Fault Recovery

To implement the segmentation fault recovery outlined in Section 2.3.3, calls are made to the functions `sigsetjmp` and `siglongjmp`. These calls, along with subsequent retry of the regions after a segmentation fault, add time not reflected in our model (2.4). Thus, the resiliency overhead is defined as

$$o_i = T_{\text{seg}}, \tag{2.5}$$

where T_{seg} is the overhead of segmentation fault recovery. Recovery from a segmentation fault restart the local idempotent operation. This operation time is less than the level time. Our model provides an upper bound on the restart time by defining r_i as the sum of the maximum number of segmentation faults experienced on a single rank on level i during each cycle.

2.5.3 Residual and Energy Check

To add the residual and energy checks to our modelling of resiliency overhead in (2.6), an extension is made to (2.5), incorporating $T_{\text{checkpoint}}$, which is the time to checkpoint the solution vector on the fine level. In addition, T_{residual} is the overhead of the residual calculation (only on level 0), and $T_{\text{energyDown}_i}$ and T_{energyUp_i} are the overheads of doing the energy check on level i of the down and up-pass respectively. This yields

$$o_i = \begin{cases} T_{\text{seg}} + T_{\text{energyDown}_i} + T_{\text{energyUp}_i} + T_{\text{checkpoint}} + T_{\text{residual}} & i = 0 \\ T_{\text{seg}} + T_{\text{energyDown}_i} + T_{\text{energyUp}_i} & i \neq 0 \end{cases} \tag{2.6}$$

2.5.4 Comparison of Time to Solution

Faults that occur during the solve phase that require extra iterations to converge increase the average convergence factor. As the average convergence factor approaches 1.0, the number of iterations required to converge grows exponentially. Ultimately, the same *asymptotic* convergence factor is likely achieved, yet the *effective* convergence factor for the run increases. Figure 2.7, show the relative overhead is equal to β. Each trend is associated with a different value for β, which represents the percent increase in iteration count over the fault free case. Specifically $\beta = 1.0$ corresponds to taking twice as many iterations to solve the problem when compared to the fault free case leading to a relative overhead of 1. The overhead due to a SDC is proportional to the amount of duplicated work. For example, if a problem's average convergence factor changes from 0.2 to 0.3, this is equivalent to $\beta = 0.3$ or a 1.3× increase in the number of iterations of the original problem. This compounds as the average convergence factor approaches 1.0, where the number of iterations required grows exponentially. The proposed detection and recovery scheme keeps β small which limits extra computation. Looking at the speedup of time to solution with a single soft error occurs, Figure 2.8 shows that for problems that convergence rapidity (small convergence factors) the overhead of resiliency impedes time to solution. However, for problems that take more iterations to converge (large convergence factors) using the protection mechanisms of *Low Cost* and *All* improve time to solution. Figure 2.8 assumes that the soft error causes a deterioration in the convergence factor by 10%. The configuration *Low Cost* requires the recovery of the entire V-cycle (increase of 1 iteration over the fault free case), and the configuration *All* requires the recovery of a single level (modeled as the most expensive level retry).

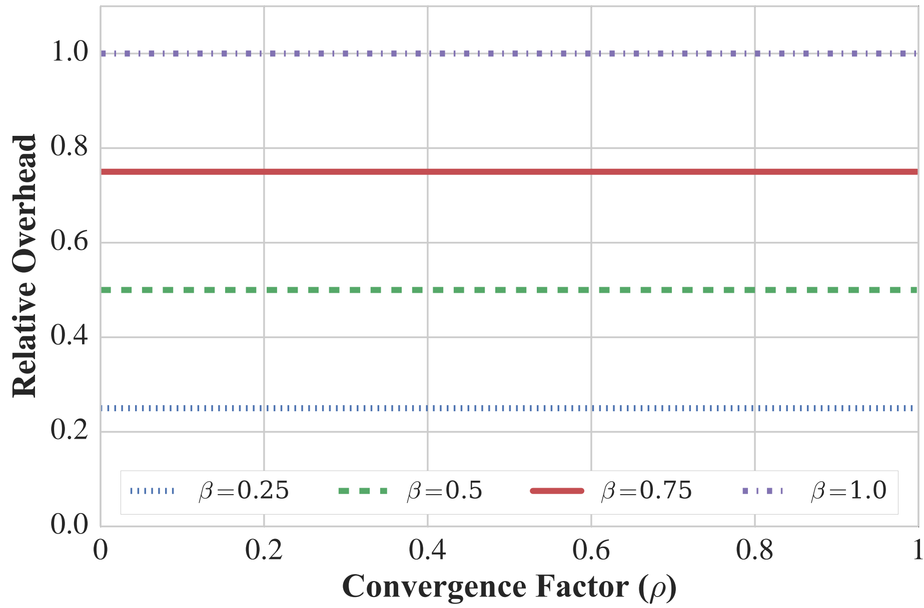


Figure 2.7: Relative overhead in converging to a fixed tolerance for various values of β.

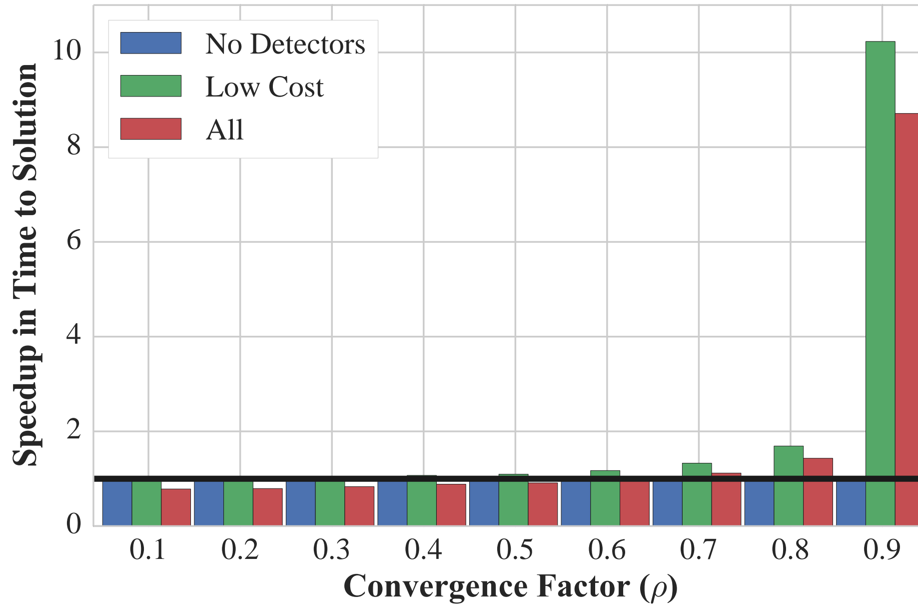


Figure 2.8: Speedup in time to solution relative to non-protected AMG (*No Detectors*).

2.6 Conclusions

This chapter presents an algorithmic specific detection and recovery scheme for the algebraic multigrid linear solver. Through the use of fault injection experiments, this chapter shows the detection and recovery scheme

maintains a high probability of convergence with fault rates where an unprotected AMG fails to converge at all. Finally, a performance model analyzes when the detection and recovery scheme improves time to solution. In the next chapter, the propagation of soft errors in HPC applications and kernels is investigated.

Chapter 3

Error Propagation

Portions of this chapter are taken from the publications "Understanding the Propagation of Error Due to a Silent Data Corruption in a Sparse Matrix Vector Multiply" [22] and "Towards a More Complete Understanding of SDC Propagation" [21]

3.1 Introduction

Global checkpoint-restart is the standard fault tolerance protocol used by HPC applications to recover from fail-stop failures. The limited bandwidth of persistent storage is the main performance bottleneck of checkpointing methods. Current versions mitigate storage bandwidth issues by leveraging the memory hierarchy [79, 11] and use compression techniques [91].

Another way to limit the impact of storage bandwidth is to avoid the need for global checkpointing and global restart. Checkpoint-restart schemes often coordinate checkpointing and global restart: at checkpoint time, all processors are synchronized and the application state is saved, while at restart time, the entire application state is restored. This results in bursts of I/O that slow down checkpoint and recovery. Various schemes have been proposed to support uncoordinated checkpointing and localized restarts [107, 42, 43]. Similarly, application specific recovery reconstructs lost data from the remaining correct data [1].

Most recovery schemes assume that corruption is limited to a subset of values — e.g. values in the memory of one compute node. This condition is easy to satisfy with fail-stop failures, but may not be valid in the case of SDC detection with high latency. One solution is to verify all state outside of a corrupted subset [29]. However, such checks are expensive. An alternative is to determine the propagation of corruption when a detection occurs and to restrict recovery to the potentially impacted state. In this case, some corruption can be ignored as it is attenuated by the algorithm; in other scenarios, corrupted values may permanently influence the solution.

Prior work has explored how deviations from bit-flips in floating-point computation impact on convergence properties [36, 35]. Other work has used corruption propagation in the development of low-level

instruction based detection schemes that check invariants or create SDC detectors inside the compiler [46, 84] or utilize code replication to detect corruption in instructions that are likely lead to and propagate corrupted state [59, 37, 56], and understanding long latency crashes [104, 64]. This chapter combines the latency of programmatic systems of soft errors — e.g. segmentation faults, detection, control flow divergence — with corruption propagation in state variables to discuss the impact of detection latency on recovery options. In the context of tracking corruption propagation inside applications, [4] looks at state corruption propagation in MPI codes by tracking number of incorrect memory addresses, but does not relate corruption back to application level data structure nor explore compiler optimizations and the impact of local problem size on corruption propagation. [65] quantifies corruption in different GPU and host memories in GPGPU programs. This chapter's focus is on MPI applications and tracks corruption in state variables and across MPI processes.

A detailed view of corruption propagation offers a measure of an application's ability to withstand soft errors. Moreover, it also helps application developers identify where to place detectors and to identify locations for data recovery. This chapter makes the following contributions:

- the impact of detection latency on recovery options;
- the data structures critical to corruption and the risk of obtaining invalid results;
- the influence of compiler optimizations on state corruption propagation; and
- the analysis of problem size and inter-node corruption propagation.

The remainder of this chapter is structured as follows. Section 3.2 discusses corruption propagation for sparse matrix-vector multiplication and motivates the need to investigate state corruption propagation in application codes. Error propagation results are presented in Section 3.3 along with a discussion on detection and recovery options.

3.2 Motivation

Sparse matrix-vector multiplies (SpMV) are at the core of many HPC applications. Mathematically a SpMV is a series of inner-products between the rows of the matrix and the input vector. The inner-products can be further broken down into a series of multiplications and additions. Breaking the SpMV down into its constituent operations allows for modeling of corruption propagation.

In Figure 3.1, one element of the input vector is corrupted (red square). Walking though each inner product the erroneous element is used in three inner products. The results of these inner-products are potentially corrupted as shown by the red squares in the result vector. If this SpMV is used as part of

an iterative method or solver, the corruption will propagate further as the output vector is used as the input vector for the next SpMV. From this, it can be generalized that any corruption in element i of the input vector propagates to element j of the output vector if row j in column i is non-zero. Thus, dense matrices propagate corruption much faster than sparse matrices. Figure 3.2 shows the percent of loads/stores corrupted during a SpMV. Corruption propagates faster for a matrix that is 75% sparse compared to a matrix that is 99% sparse. The matrices are the first two levels of an AMG hierarchy for a 2D rotated anisotropic diffusion problem. With each successive SpMV, propagation occurs though iterative method style reuse of the previous result vector. With more iterations, masking starts to occur leading to a reduction in the percent of loads and stores corrupted. This masking is due to convergence of iterative methods schemes. In this case, the repeated matrix-vector multiplies causes the vector to converge to the dominant eigenvector for the matrix.

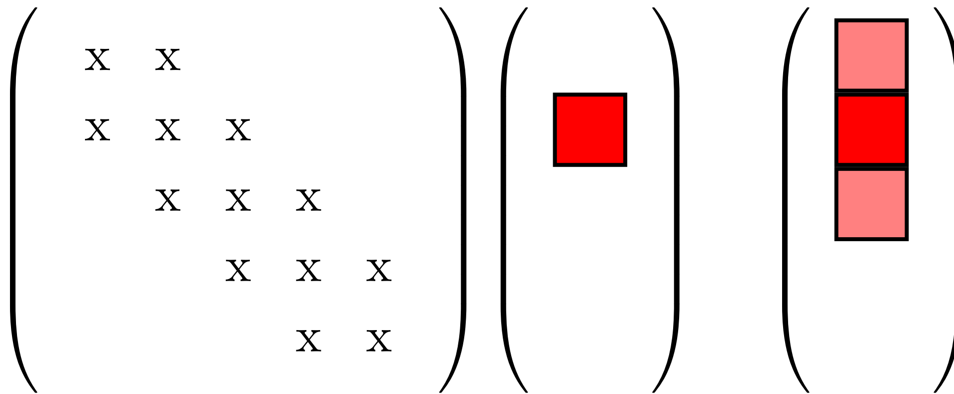


Figure 3.1: Propagation of SDC via a sparse matrix-vector multiply.

Understanding propagation at this level is relatively straight forward, but comes with an important caveat. Corruption is only considered at the level of the algorithm and not how it is expressed in code executed by a computer. To fully measure error propagation for the SpMV, the effect of corruptions of loads, stores, address calculations, branching, looping, etc. needs to be considered. This type of analysis is difficult to perform because it relies on a specific implementation and the manner in which the code is executed on the machine. This motivates the need to track propagation at the execution level. The corruption propagation tool (see Chapter 5) used in this chapter allows for tracking error propagation at the level of application variables and with load/store granularity.

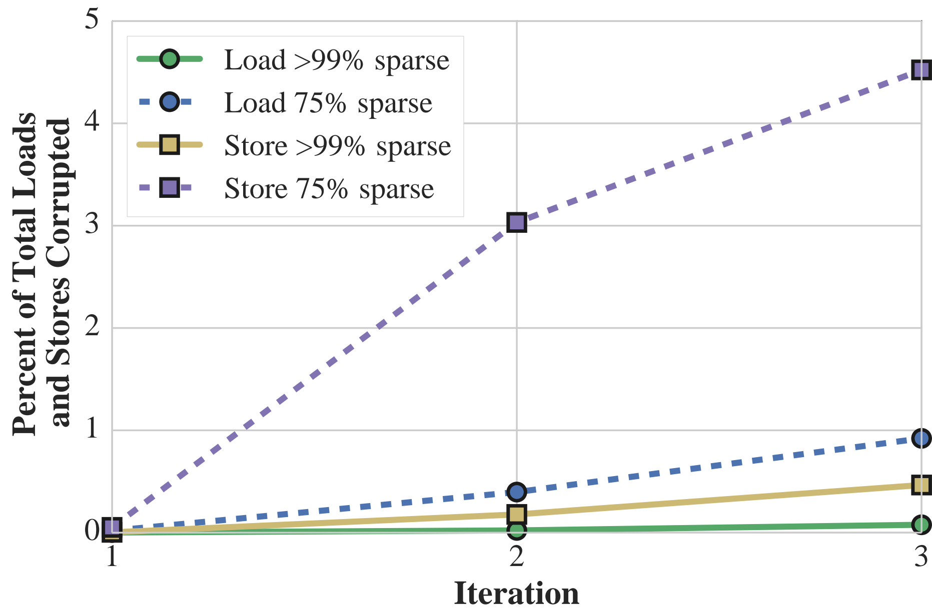


Figure 3.2: Propagation of SDC by iterative method style reuse.

3.3 Experimental Results

To determine if propagation occurs, application source code is compiled with a tool (see Chapter 5) that considers two levels of propagation:

macro tracks deviations in the state variables through the simulation; and

micro tracks deviations in the loads, stores, and other low-level operations.

3.3.1 Testing Methodology

System

Results are collected on Blue Waters. Version 3.5.2 of `clang` and LLVM compile and instrument the source code.

Fault Injection

This chapter does not inject faults into the initialization of the applications. Instead, faults are injected during the main computation using FlipIt (see Chapter 4). The selected applications use MPI, and in the tests, MPI process rank 3 is selected to experience a single bit-flip error during the execution of a unique, random dynamic LLVM instruction. Faults are injected in a single MPI process, and all MPI processes

track propagation via the methods outlined in Chapter 5. Injected instructions are classified based on FlipIt's classification system (see Table 4.2) with *Control-Branch* and *Control-Loop* being combined into one category, *Control*.

Each application is run 1500 times with a different random fault each time. The kernels `WAXPBY`[1] and SpMV are run 1500 times per optimization level each with a different random fault.

Applications

Jacobi: Jacobi performs Jacobi relaxation on a unit square with fixed boundaries is using a 5-point stencil and 1-D row partitioning of parallel processes. This test uses 4 MPI processes with 4096 grid points per process. Jacobi relaxation does not guarantee a reduction in the residual, however unexpected large jumps often indicate the presence of state corruption. At the end of each iteration, a detector flags any increase in the residual by an order of magnitude as SDC. The stopping tolerance used for Jacobi is $1e{-3}$ and the floating-point comparison tolerance when tracking propagation is set to $1e{-5}$. Finally, macro-level propagation results are logged at the end of each iteration.

CoMD: CoMD[2] is a molecular dynamics mini-app created and maintained by the Exascale Co-Design Center for Materials in Extreme Environments (ExMatEx). CoMD is parallelized with MPI and uses a link-cell structure to determine the interaction regions for the atoms. This test uses 16 MPI processes to simulate the motion and interaction of 32,000 atoms over 500 time-steps, where forces between atoms are computed using the Embedded-Atom Method (EAM). SDC is detected by ensuring that the total energy is within five standard deviations of the ensemble mean. Macro-level propagation results are logged at the end of each iteration. Deviations smaller than $1e{-10}$ are considered insignificant to accuracy of CoMD.

HPCCG: HPCCG[3] is a conjugate gradient (CG) benchmark from the Mantevo Suite that simulates a 3D chimney domain using a 27-point finite difference matrix. This test is run with 16 MPI processes and a local block size of $nx = ny = nz = 13$. As with Jacobi, the CG algorithm does not guarantee a reduction in the residual. Here, any increase in the residual by an order of magnitude is flagged as SDC. Macro-level propagation results are logged at the end of each iteration. The convergence tolerance used for HPCCG is $1e{-7}$; when tracking corruption propagation, deviations smaller than $1e{-10}$ are ignored.

[1] The `WAXPBY` kernel is defined as the scaled vector addition operation $\mathbf{w} = a * \mathbf{x} + b * \mathbf{y}$.
[2] https://github.com/exmatex/CoMD
[3] https://mantevo.org/packages.php

3.3.2 Micro-level Propagation

Latency of Detection

Each injected fault starts as a single-bit error in the result register of an instruction. As the program executes, the single-bit error propagates to other registers and memory locations. Depending on the type of instruction the fault is injected into, the latency-to-detection often varies. The analysis in this chapter looks at three failure symptoms: segmentation fault, detection, and control flow divergence. Table 3.1 shows the breakdown of each symptom based on instruction type across the tests in Section 3.3.1.

Segmentation faults occur in 30–35% of all injected faults in each application. From Table 3.1, the majority (over 50%) of segmentation faults are triggered by *Pointer* instructions. Figure 3.3 shows the segmentation fault latency for each application in number of LLVM instructions executed after an injection. A significant number (90%) of segmentation faults occur within 4 LLVM instructions. Of the runs with a segmentation fault within 4 LLVM instructions, 61% percent are classified as *Pointer*, 32% percent are classified as *Arith-Fix*, 0% percent are classified as *Arith-FP*, and 13% percent are classified as *Control*. Looking at the remaining segmentation faults, 16% percent are classified as *Pointer*, 8% percent are classified as *Arith-Fix*, 0% percent are classified as *Arith-FP*, and 76% percent are classified as *Control*. Short latencies are attributed to corruption of address calculation — e.g. corrupting an address or offset before a load or a store. Longer latencies are the result of corruption in the loop induction variables. The instructions used to check a loop conditional increase the segmentation fault latency slightly before the induction variable is used as part of a load/store during the subsequent iteration. Segmentation fault latencies for these applications are consistent with those reported for the Linux kernel [40] and other benchmarks [64].

Instruction Type	Segmentation Fault	Detection	Control Flow Divergence
Arith-FP	0%	49%	20%
Arith-Fix	24%	24%	26%
Pointer	56%	27%	16%
Control	20%	0%	38%

Table 3.1: Breakdown of failure symptom by instruction type.

Figure 3.4 shows the bit locations that generated a segmentation fault. The results highlight that most segmentation faults occur due to a bit-flip near the most-significant bit (MSB). Bit-flips in bits near the least-significant bit (LSB) lead to incorrect indexing and using incorrect data. Flipping bits 0–2 result in unaligned data access for double precision arrays and can cause significant deviation in the load/store, but rarely generates a segmentation fault. As the local problem size grows, the amount of allocated memory also increases. As a result, the number of segmentation faults decreases for low order bits by as they are

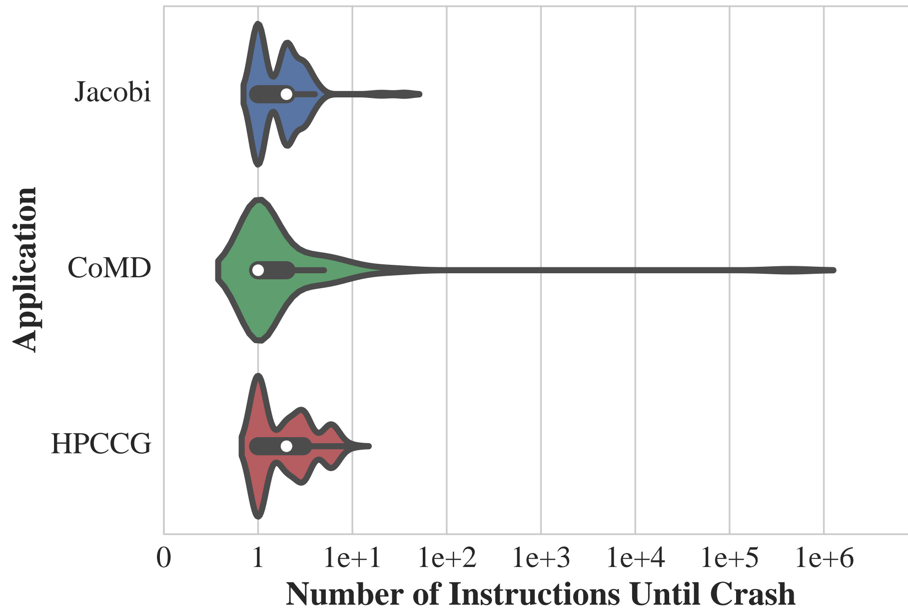


Figure 3.3: Latency (number of LLVM instructions) until a segmentation fault.

replaced with loads/stores on incorrect addresses.

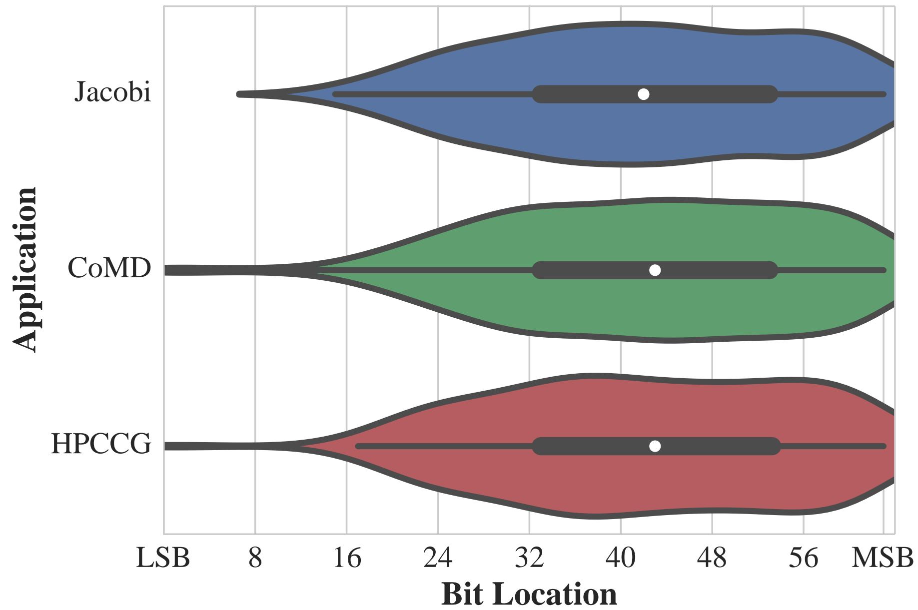


Figure 3.4: Bit positions where injected fault resulted in a segmentation fault.

Segmentation faults are an excellent detector that allow little corruption propagation in most cases. Since

segmentation faults occur in only 30–35% of the tests most faults allow for propagation and SDC. Typical HPC SDC detectors check for errors at a coarse granularity (thousands or millions of instructions) by looking for latent errors that have corrupted the state of application level variables to cause a noticeable deviation. Overall the lightweight SDC detectors added to applications (Section 3.3.1) detect data corruption in 11% of Jacobi runs, 1% of HPCCG runs, and 2% of CoMD runs. Breaking down which faults are detected by instruction type, Table 3.1, shows most detected faults occur in *Arith-FP* instructions. Figure 3.5 shows the latency in number of LLVM instructions executed before the detection occurs. Compared to detection via segmentation faults, detection latency of specially designed detectors is much larger.

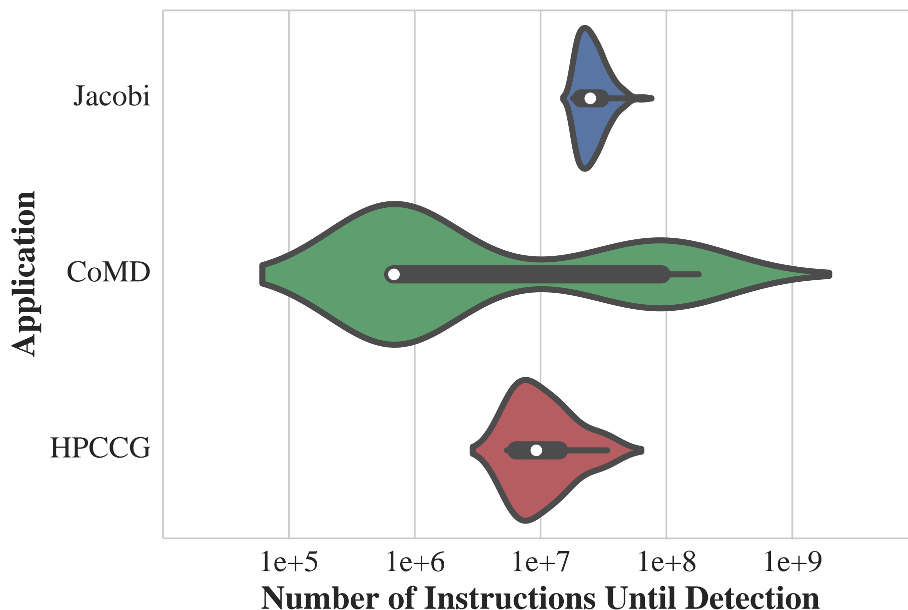


Figure 3.5: Latency (number of LLVM instructions) until SDC detection.

Each test checks for SDC at the end of each iteration. Converting detection latency from LLVM instructions executed to iterations executed shows all runs with detection for CoMD within three iterations of injection and HPCCG within one iteration of injection. Detected SDC in Jacobi has a longer maximal latency at 595 iterations, but a median of 12 iterations. Faults that result in an order of magnitude differences in floating-point values are detected during the subsequent iteration. Without SDC detectors 100% of HPCCG runs and 56% of Jacobi require extra iterations to converge to the correct solution. All CoMD runs with detection appear as outliers when forming an ensemble distribution at the final time-step if no SDC detectors are present.

If recovery from a SDC with a long detection latency uses frequent in-memory checkpoint, then a domino

effect [7] is possible until a checkpoint is found that is free of corruption. At detection time, if the extent of corruption is known in terms of state variables and processes, then forward-recovery schemes offer the best solution to avoid the domino effect of checkpoint-restart (backward-recovery).

Abnormal Behavior

HPC applications commonly iterate over and compute on vectors of data. Faults in *Pointer* and *Arith-Fix* instructions often lead to corruption in pointers used in load/store operations, and can lead to segmentation faults. However, corruption of loop control variables can lead to divergence in the control flow graph. Control flow divergence in loops (see Figure 3.6) results in an early loop exit; consequently, elements of a vector may not be updated or computed. Control flow divergence occurs in 3% of Jacobi runs, 6% of HPCCG runs, and 20% of CoMD runs. Unlike HPCCG and Jacobi where data distribution is static, CoMD has atoms that migrate between processes that dynamically modify the data distribution. The extreme latencies for control flow divergence in CoMD are due to corruption of the atom positions that modifies the control flow of the atom exchange routine as atoms migrate incorrectly (inter-process corruption propagation). In the remaining cases, most control flow diverges within the injected loop structure, leaving some vector entries unmodified. To ensure correct execution of these loops, instruction duplication techniques such as IPAS [59] and FlipBack [81] or low-cost invariant checks [46], offer the ability to ensure correct control flow with minor overheads.

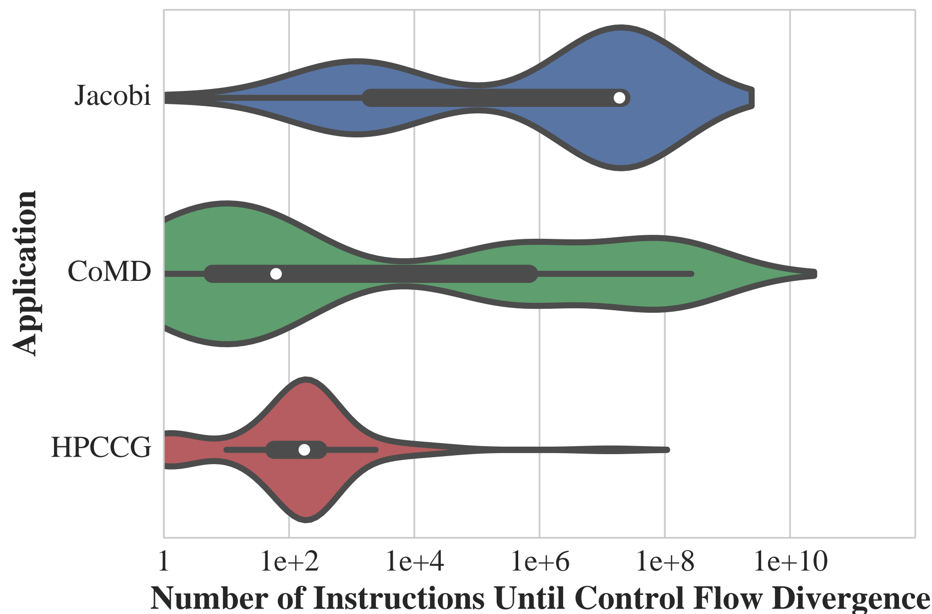


Figure 3.6: Latency (number of LLVM instructions) until control flow diverges.

An optimization to lower the overhead of algorithmic based SDC detection schemes is to assume static data — e.g. the matrix in a linear solver, is sufficiently protected that it does not need explicit checks for consistency. Instead, SDC detectors are placed on dynamically changing data because it is more vulnerable to becoming corrupted. With a fault model that only allows for corruption in *Arith-FP* values, this is a valid assumption, however once corruption occur in *Arith-Fix* and *Pointer* type instructions, static data can be written by errant store instructions. Although very rare (less than 0.01% of runs) it does have the ability to make a convergent algorithm non-convergent or converge to a different solution. To mitigate corruption of static data, checksums can be employed to ensure consistency or pages containing static data can be marked *read-only* after initialization to ensure no errant stores corrupt the data.

Effect of Compiler Optimizations

The ratios of the different instruction types impact the probability of different failure symptoms. Table 3.2 shows the percentage of dynamic LLVM instructions for each application classified as a given instruction type. Across all applications, instructions classified as *Arith-FP* comprises the majority dynamic instructions followed by instruction types *Pointer* and *Arith-Fix* that are used to compute addresses. Finally, *Control* flow makes up the smallest classification percentage.

Instruction Type	HPCCG	Jacobi	CoMD
Arith-FP	35%	41%	35%
Arith-Fix	22%	28%	16%
Pointer	22%	21%	34%
Control	21%	10%	15%

Table 3.2: Dynamic LLVM instruction type percentage.

Instruction mix depends on the data structures, compiler, and optimization level. Two key operations in the HPCCG mini-app and other linear solvers are the sparse matrix-vector multiply (SpMV) and scaled vector addition (`WAXPBY`). The impact of compiler optimizations on these small kernels helps identify the impact on the full mini-app and production application.

`WAXPBY`: A `WAXPBY` operation, Algorithm 3.1, scales two input vectors **x** and **y** and adds them. Compiling without any optimizations (`-O0`) produces verbose and explicit code as every load and store references memory. Register allocating variables `i` and `N` along with hoisting loads outside the loop with `-O1` reduces address calculation and loads/stores that often lead to segmentation faults (see Section 3.3.2). Furthermore, optimization levels `-O2` and `-O3` unroll and vectorize the loop, further reducing the need for control flow instructions. For this kernel, `-O2` and `-O3` produces identical code. To support higher level optimizations such as loop unrolling and vectorization, extra instructions are added to ensure correctness for all sizes of `N`.

Table 3.4 summarizes the impact of compiler optimizations. Vectorization increases the number of integer operations which causes an increase in runs that experience a segmentation fault. Loop unrolling removes branching instructions and instructions that update the loop induction variable which removes locations where corruption of loop induction variables are possible lowering control flow divergence.

Algorithm 3.1: Scaled Vector Addition (`WAXPBY`).

```
1 for i ← 0 to N do
2 └ w[i] ← a * x[i] + b * y[i]
```

Optimizations	*Arith-Fix*	*Pointer*	*Control*	*Arith-FP*
O0	31%(32772)	23%(24579)	15%(16385)	31%(32770)
O1	8%(4097)	23%(12288)	23%(12289)	46%(24576)
O2/O3	17%(5128)	40%(12291)	3%(1031)	40%(12292)

Table 3.3: Dynamic LLVM instruction percentage for `WAXPBY` kernel.

Symptom	O0	O1	O2/O3
Segmentation Fault	29%(435)	27%(405)	40%(600)
Control Flow Divergence	10%(150)	4%(60)	2%(30)
No Symptom	61%(915)	69%(1035)	58%(870)

Table 3.4: `WAXPBY` kernel failure symptom percentage as a function of compiler optimization.

SpMV: The SpMV kernel, Algorithm 3.2, is more complicated than that of `WAXPBY` both mathematically and in machine code. Because the SpMV uses a sparse matrix representation (compressed sparse row in Algorithm 3.2), the level of indirection needed to access entries of the matrix increases. Each increase in indirection involves a pointer dereference; therefore, with more address manipulations and loads/stores, the number of dynamic instructions of *Arith-Fix* and *Pointer* are higher than with `WAXPBY`, as shown in Table 3.5. This also implies that common symptoms of these types of operations will be more prevalent.

Algorithm 3.2: Sparse Matrix-Vector Multiplication.

```
1 for i ← 0 < num_row do
2 │ tmp ← 0
3 │ for jj ← A_i[i] ¡ A_i[i+1] do
4 │ └ tmp ← tmp + A_data[jj] * x_data[A_j[jj]]
5 └ y_data[i] ← tmp
```

With the baseline optimization level `-O0`, the code is explicit and verbose. Higher levels of optimizations retain the base addresses of the arrays `A_i`, `A_j`, `A_data` in register temporaries along with loop induction variables and hoists loads with `-O1`. Optimization levels `-O2` and `-O3` do not vectorize this kernel. Instead,

Optimizations	*Arith-Fix*	*Pointer*	*Control*	*Arith-FP*
O0	38%(1304455)	40%(1363261)	6%(207609)	16%(562824)
O1	22%(316414)	24%(336417)	21%(306413)	33%(464020)
O2	24%(326414)	23%(306417)	19%(257609)	34%(454816)

Table 3.5: Dynamic LLVM instruction classification percentage for SpMV kernel.

the inner most loop is unrolled removing comparisons and branching instructions along with hoisting the load of `num_row` to outside the loops.

Table 3.6 shows that without vectorization, the rate of segmentation faults remains around 30% of executions, which is consistent with `WAXPBY` and the applications. Loop unrolling fails to reduce divergence in control flow compared to `WAXPBY`. Control flow for the inner most loop is more complex than with `WAXPBY` resulting in more locations in which a fault can occur that influences control flow.

Symptom	O0	O1	O2/O3
Segmentation Fault	25%(375)	32%(486)	29%(436)
Control Flow Divergence	12%(180)	13%(201)	14%(207)
No Symptom	63%(945)	55%(813)	57%(857)

Table 3.6: SpMV kernel failure symptom percentage as a function of compiler optimization.

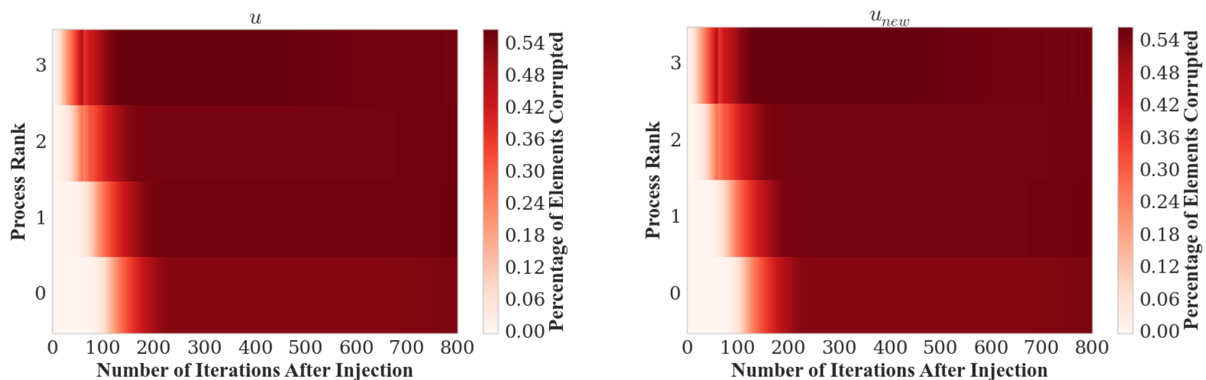
3.3.3 Macro-level Propagation

After a fault occurs, an erroneous value is present in the application state. Over time, as this value is used/reused, the corruption propagates to infect larger portions of the application state. Most HPC SDC detectors ensure correctness of application level variables. Knowing which application variables are corrupted and the extent of corruption can assist in placement of detection and recovery schemes.

Jacobi

Each iteration of Jacobi refines a solution u to improve the solution accuracy resulting in an updated solution in a separate vector u_{new}. These two vectors represent the two key data structures and are the focus when measuring corruption propagation. Figure 3.7 shows the average percentage of elements that are corrupted across all MPI processes (y-axis) due to an injected fault (intensity of color) in the iterations following the injection iteration (x-axis) for the solution variable u and u_{new}. All runs are aggregated to align the iteration where injection occurs. This allows corruption percentages in all remaining iterations to be averaged over all runs. All faults are injected on rank 3, and as time evolves corruption propagation occurs inside this process indicated by the increase in the intensity of color on the horizontal row for rank 3. Figure 3.8 shows

a detailed view of propagation measured as percentage of elements corrupted in both u and u_{new} with 90% confidence inside rank 3. Over time, corruption propagates inside rank 3 and reaches the region of the array that is communicated through a halo-exchange resulting in corruption of rank 2. This process continues until all processes are corrupted or the corruption is attenuated. Because Jacobi converges to a solution, over time the corruption in the variables is removed. When tracking propagation in u, comparisons are made between the memory of u from the *gold* and *faulty* applications. As Jacobi continues to iterate, error does not appear to reduce because as error due to the fault is removed from the *faulty* u it is being compared to an ever more accurate u from the *gold* application. Only with extra iterations on *faulty* (beyond what is run for *gold*) do the two solution vectors converge. The same logic holds true for the variable u_{new}.



(a) Average percentage of corrupted elements of iterative solution u for Jacobi.

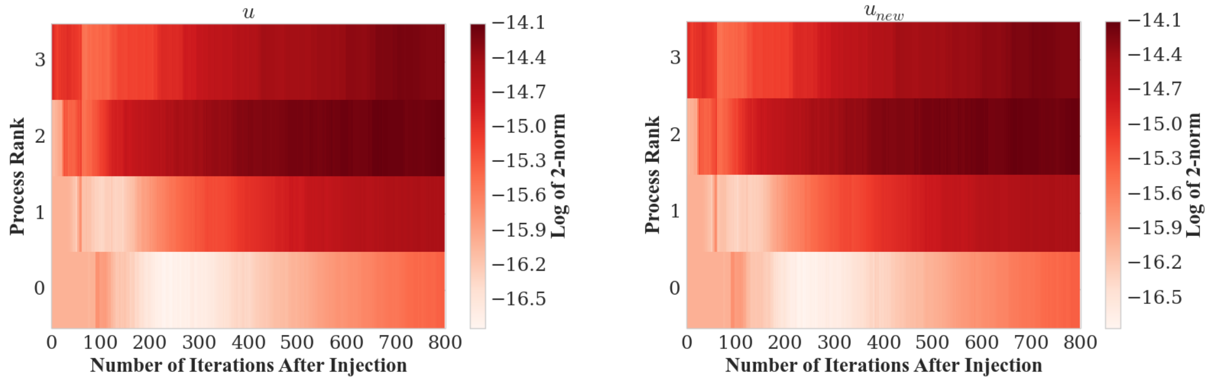
(b) Average percentage of corrupted elements of iterative solution u_{new} for Jacobi.

Figure 3.7: Average percentage in main variables of Jacobi.

Figure 3.9 and Figure 3.10 averages the log of the 2-norm between the key variables in the *gold* and *faulty* executions. In both Figures, the logarithm of 0 is replaced with -16. Figure 3.9 shows an increase in the 2-norm first appearing on the faulty rank 3, and over time the corruption propagates to all other processes. As the iteration count increases there becomes a larger deviation between the *gold* and *faulty* executions. This is due to the corruption propagating to more vector entries and the inability to remove corruption sufficiently fast to converge back to the *gold* variables. Figure 3.10 shows a 90% confidence interval on the log of the 2-norm for rank 3. Throughout all iterations the average log of the 2-norm is expected to be small indicating that the corruption in most executions does not have much effect on the accuracy of the solutions.

Figure 3.11 and Figure 3.12 shows the averages of the log of the max-norm between the key variables in the *gold* and *faulty* executions. As with plots of the 2-norm, the logarithm of 0 is replaced with -16. Figure 3.9 shows an increase in the max norm appears first on the faulty rank 3, and as time-progresses the corruption propagates to all other processes. As the iteration count increases there becomes a larger deviation

Figure 3.8: Average corruption in selected variables on rank 3 for Jacobi within 90% confidence.



(a) Log of average 2-norm of the error in iterative solution u for Jacobi.

(b) Log of average 2-norm of the error in iterative solution u_{new} for Jacobi.

Figure 3.9: Log of average 2-norm of the error in main variables of Jacobi.

between the *gold* and *faulty* executions. This is due to the corruption propagating and accumulating in more vector entries and the inability to remove corruption sufficiently fast to converge back to the *gold* variables. Figure 3.10 shows a 90% confidence interval on the log of the max norm for rank 3. As propagation and accumulation occurs at each time-step the average max norm increases in magnitude. However, because Jacobi is an iterative refinement algorithm any error due to corruption of data can be removed with extra iterations.

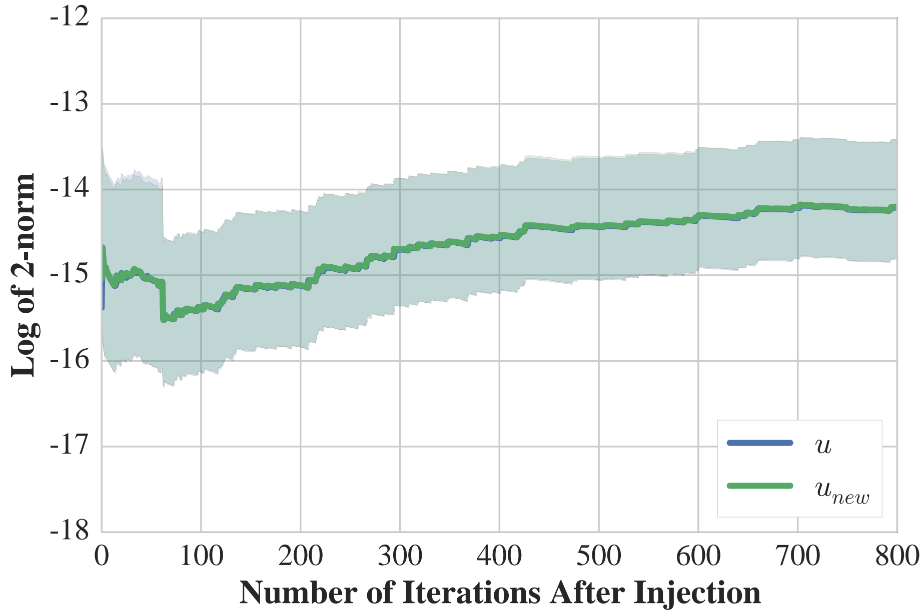
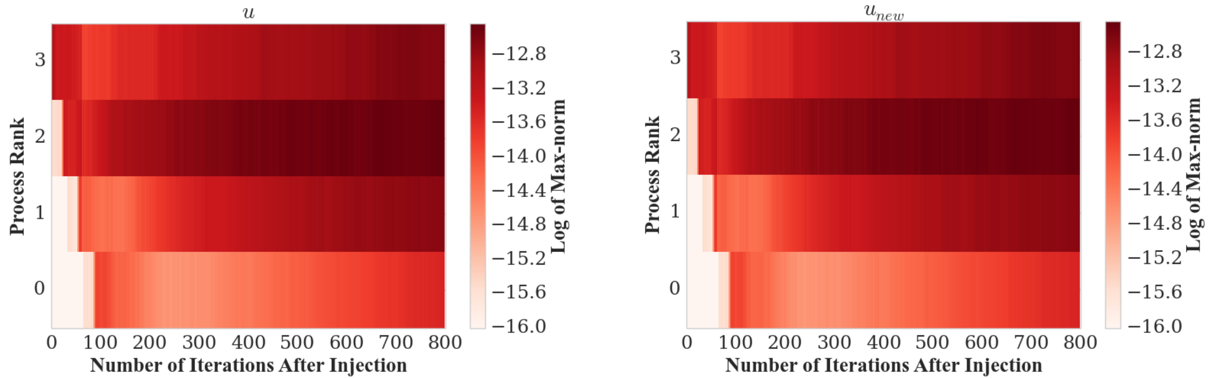


Figure 3.10: Average 2-norm of the error in selected variables on rank 3 for Jacobi within 90% confidence.



(a) Log of average max-norm of the error in iterative solution u for Jacobi.

(b) Log of average max-norm of the error in iterative solution u_{new} for Jacobi.

Figure 3.11: Log of average max-norm of the error in main variables of Jacobi.

After a fault, corruption appears immediately on rank 3. As corruption propagates inside rank 3, it corrupts values sent to process 2 in a halo exchange. The speed of corruption propagation depends on the stencil size. This problem uses a 5-point stencil, and the average worst case propagation latency occurs when an element interior to a local domain is corrupted. On rank 3, this requires $m/2$ iterations were m is the local block size. In general, it takes $m/4$ iterations for ranks without a boundary edge. SDC detected in Jacobi is within 595 iterations (median 12 iterations) of injection. For any reasonable local block size, once

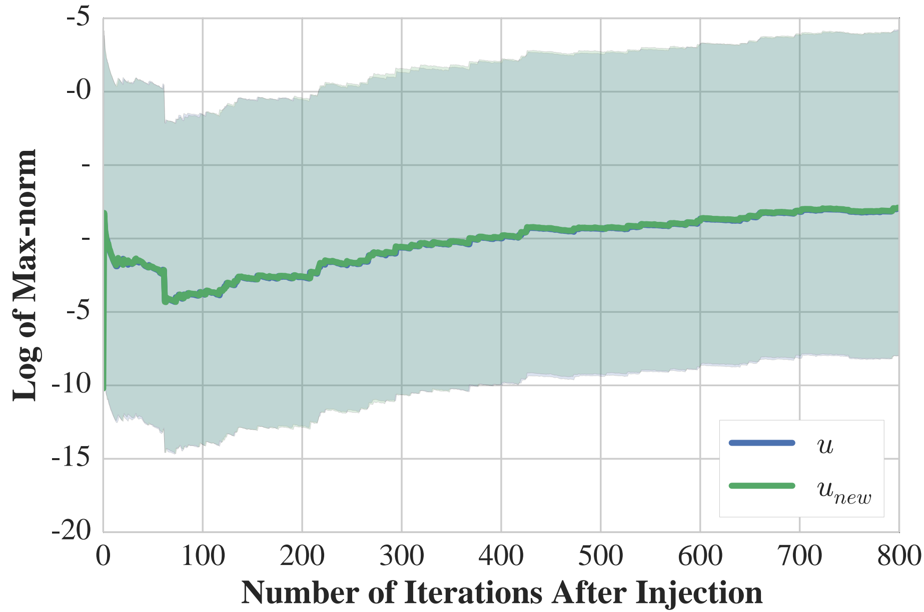


Figure 3.12: Average max-norm of the error in selected variables on rank 3 for Jacobi within 90% confidence.

SDC is detected it can be confined to a process and its immediate neighbors; allowing a customized local recovery scheme to be applied.

HPCCG

Conjugate Gradient (CG) (see Algorithm 3.3) is a popular solver for systems of linear equations. This algorithm relies on four key variables: the iterative solution $\mathbf{x}^k$, search directions $\mathbf{p}^k$ that are used to update $\mathbf{x}^k$, residual vector $\mathbf{r}^k$, and the matrix-vector product $A * \mathbf{p^k}$.

Algorithm 3.3: Conjugate Gradient Method.

1. $\mathbf{r}^0 = \mathbf{b} - A * \mathbf{x}^0$
2. $\mathbf{p}^0 = \mathbf{r}^0$
3. $k = 0$
4. **while** $\| \mathbf{r}^k \|_2 < tol$ **do**
5. $\quad \alpha^k = \frac{(\mathbf{r}^k)^T \mathbf{r}^k}{(\mathbf{p}^k)^T * A * \mathbf{p}^k}$
6. $\quad \mathbf{x}^{k+1} = \mathbf{x}^k + \alpha^k \mathbf{p}^k$
7. $\quad \mathbf{r}^{k+1} = \mathbf{r}^k - \alpha^k * A * \mathbf{p}^k$
8. $\quad \beta^k = \frac{(\mathbf{r}^{k+1})^T \mathbf{r}^{k+1}}{(\mathbf{r}^k)^T \mathbf{r}^k}$
9. $\quad \mathbf{p}^{k+1} = \mathbf{r}^{k+1} + \beta^k \mathbf{p}^k$
10. $\quad k = k + 1$

As with Jacobi, propagation results for HPCCG are aggregated to align with the iteration in which

injection occurs. Figure 3.14 shows corruption propagation in the form of the average percentage of elements corrupted (color) per variable across all MPI processes (y-axis) for subsequent iterations after injection (x-axis). As corruption propagates locally inside each variable, the horizontal color for the row grows darker, while regions of low corruption have a lighter color.

Initially, the average percentage of elements corrupted in each vector is small. However, as HPCCG continues to iterate, corruption begins to propagate both internally and externally of the corrupted process, rank 3. The most severe corruption is confined to rank 3 and the neighboring processes across the majority of the iterations. Dependencies between the four variables in Figure 3.14 are due a corruption in the variables $\mathbf{r}^k$, $\mathbf{p}^k$, or $A * \mathbf{p}^k$, which leads to corruption in the solution vector $\mathbf{x}^k$. Furthermore, the SpMV propagates corruption as shown in the corresponding corruption in $\mathbf{p}^k$ and $A * \mathbf{p}^k$ on every processes of each iteration. Data dependencies in updating other vectors further propagates corruption in $A * \mathbf{p}^k$ to all other variables. Ensuring the correctness of $\mathbf{p}^k$ limits corruption propagation from the SpMV in $A * \mathbf{p}^k$ and subsequently corruption propagating to $\mathbf{r}^k$ and $\mathbf{x}^k$.

To see corruption propagation between variables more closely, Figure 3.13 shows the average percentage of elements from each variable corrupted on rank 3. As with Jacobi, the iterative solution $\mathbf{x}^k$ increases in error initially, but over time does not appear to remove error due to soft error corruption. This error is reduced at each iteration, however the iteration does not converge back to *gold*'s $\mathbf{x}^k$. Errors in $\mathbf{r}^k$ closely follow those in $\mathbf{p}^k$ as both are used in updating the other through a `WAXPBY`. As corruption in $\mathbf{p}^k$ grows and subsides, $A * \mathbf{p}^k$ reflects and propagates the corruption accordingly.

The search direction $\mathbf{p}^k$ is central to corruption propagation as it is used in updating the other variables. Ensuring that $\mathbf{p}^k$ is computed correctly helps ensure that the other variables are computed correctly. Because CG uses inner-products, if masking does not occur, then corruption in input vectors propagates to all processes in one iteration. Therefore, some form of corruption is resident in all variables within 3 iterations for runs that did not produce a segmentation fault. Local recovery is still possible, though it is complicated by the presence of corruption on multiple processes.

Figure 3.15 shows the average log of the 2-norm on each process between the corresponding selected variables for the *gold* and *faulty*. Across all variables the largest deviations are confined to rank 3 and its neighbor processes. For all variables, except the solution vector $\mathbf{x}^k$, there is a strong decease in the magnitude of the norm as HPCCG continues to iterate. Figure 3.13 shows that this variable maintains a higher percentage of elements corrupted leading to little reduction in the average log of the 2-norm on rank 3. This is observed in Figure 3.16 that plots the log of the 2-norm along with a 90% confidence interval for rank 3.

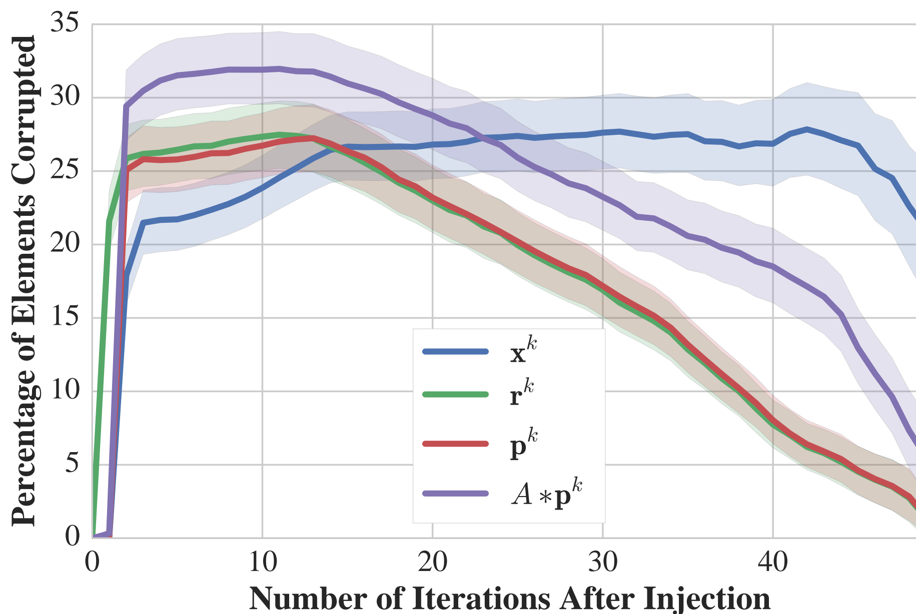
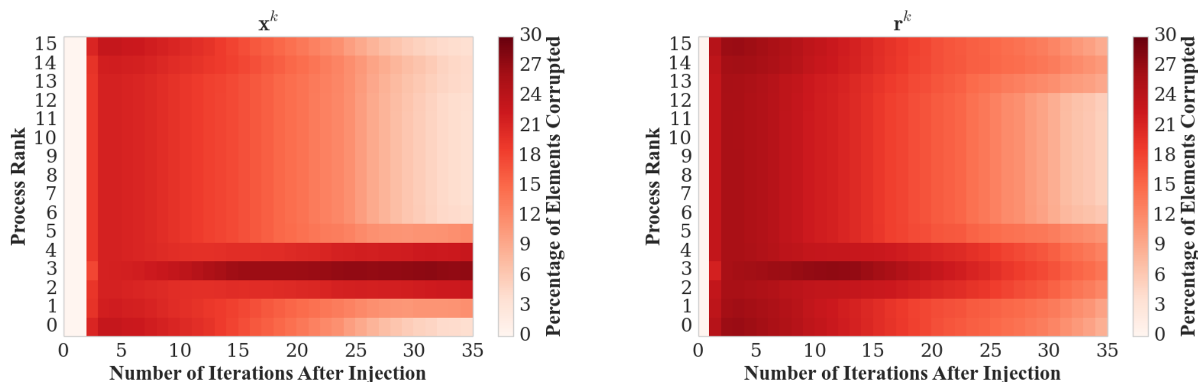


Figure 3.13: Average corruption in selected variables on rank 3 for HPCCG within 90% confidence.

Figure 3.17 shows the average log of the max-norm on each process between the corresponding selected variables for the *gold* and *faulty*. As with the 2-norm, across all variables the largest deviations are confined to rank 3 and its neighbor processes. Due to a larger number of vector components corrupted, Figure 3.13, the solution vector $\mathbf{x}^k$ on rank 3 does not show a strong decrease in the 2-norm as HPCCG iterates. Figure 3.18 highlights this in more detail with for rank 3.

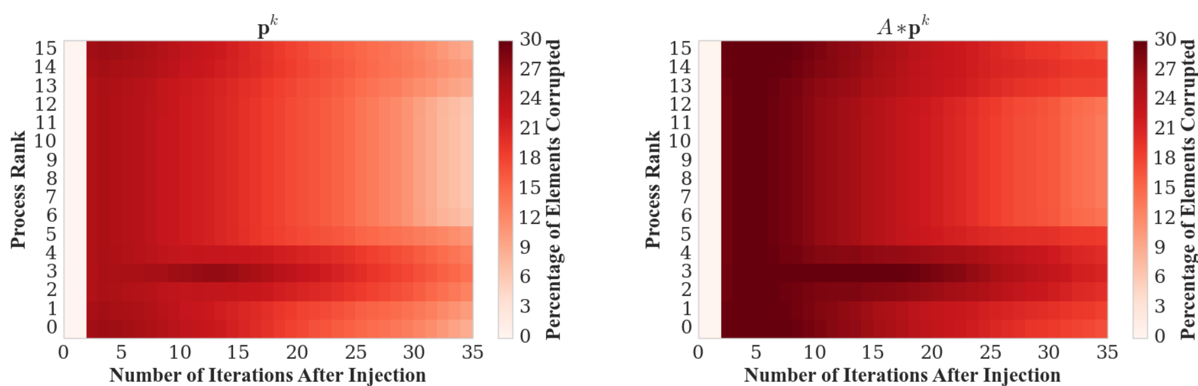
CoMD

Molecular dynamics codes such as CoMD do not converge to the same solution with each execution of the program as with HPCCG and Jacobi. Instead, a single run is combined with many other runs to form a statistical ensemble to analyze the distribution of key properties such as energy. This implies that small deviations can be masked if they do not modify the distribution. The key variables in CoMD for propagation analysis are: atom positions, atom momenta, atom forces, and atom energies. CoMD also differs from the other applications in how it stores its data. Because atoms migrate between processes, arrays are over allocated leaving space for other atoms from remote processes. This slack space is not contiguous within the arrays because it is allocated per local cell and not collectively for the entire local domain. The unused regions of the atoms complicate tracking propagation. Without modifying the data layout of CoMD this chapter factors out the unused atom storage by using the number of atoms per process instead of the memory



(a) Average percentage of corrupted vector elements for iterative solution $\mathbf{x}^k$.

(b) Average percentage of corrupted vector elements for residual $\mathbf{r}^k$.



(c) Average percentage of corrupted vector elements for search directions $\mathbf{p}^k$.

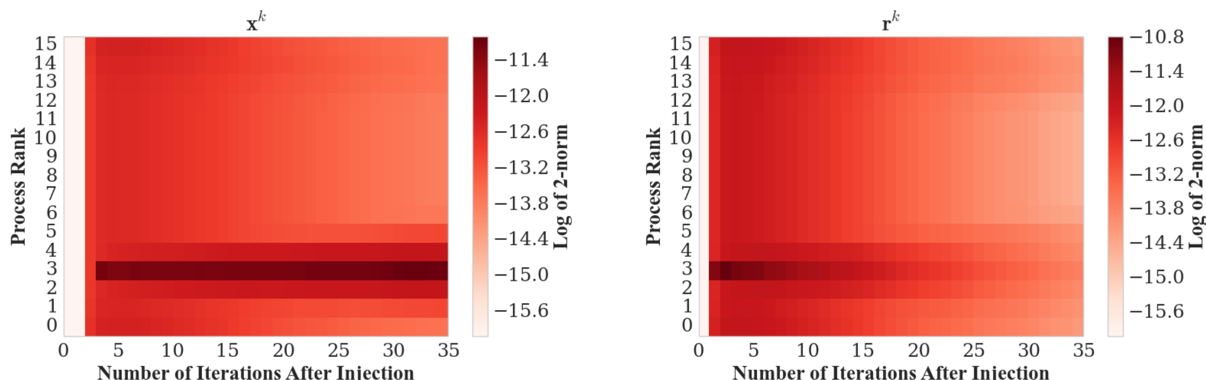
(d) Average percentage of corrupted vector elements for result of the SpMV of $A * \mathbf{p}^k$.

Figure 3.14: Average percentage of corrupted vector elements for variables in the mini-app HPCCG in iterations after injection.

allocation size when computing corruption statistics.

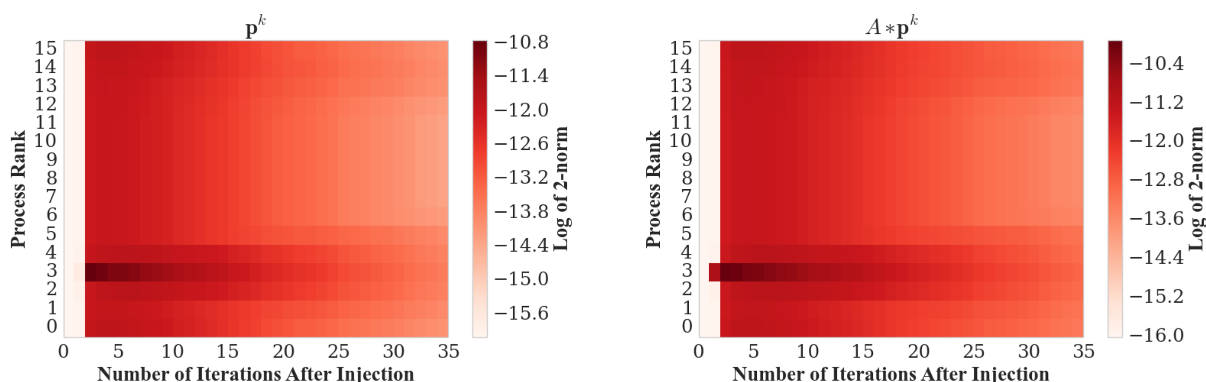
Figure 3.19 shows the average percentage of corruption in the key variables of CoMD. Unlike HPCCG, where the SpMV rapidly propagates corruption between processes, the propagation in CoMD resembles Jacobi with corruption slowly propagating beyond neighboring domain regions. Although slow, the corruption propagation increases as time evolves which leads to an increased likelihood of the run becoming an outlier for large numbers of iterations. Over the iterations after an injection, 3% of the runs are classified as outliers when looking at the energy distribution at the final iteration. All of these outliers are caught by the SDC check. A more detailed view of the propagation on rank 3 is found in Figure 3.20.

The communication pattern in CoMD consists of point-to-point messages as atoms migrate from process to process. A single atom contributes to the force calculation of neighboring atoms. This region of influence is small and needs many iterations for the corruption to spread beyond the initially corrupted region of



(a) Average 2-norm of corrupted vector elements for iterative solution $\mathbf{x}^k$.

(b) Average 2-norm of corrupted vector elements for residual $\mathbf{r}^k$.



(c) Average 2-norm of corrupted vector elements for search directions $\mathbf{p}^k$.

(d) Average 2-norm of corrupted vector elements for result of the SpMV of $A * \mathbf{p}^k$.

Figure 3.15: Average 2-norm of corrupted vector elements for variables in the mini-app HPCCG in iterations after injection.

influence. This accounts for slow rate of propagation inside CoMD. Although corruption can propagate to neighbor processes, it requires tens to hundreds of iterations before corruption of an atom propagates to all processes. Corruption that impacts the simulation's energy distribution are detected within two iterations which allows for little inter-process propagation. Containment domains can be established around the process triggering the SDC detector and nearest neighbors to allow for partial recovery.

Figure 3.21 shows the average log of the 2-norm on all processes as time evolves between the selected variables of *gold* and *faulty*. Although several iterations are required before corruption appears in Figure 3.19, in Figure 3.21 corruption is present on all processes within a few iterations. The corruption present is small enough in magnitude that it is less than the significant deviation tolerance of $1e{-}10$ and does not appear in Figure 3.19. The processes with the largest magnitude of the corruption a neighboring processes of rank 3. Over time the magnitude does decrease slightly, first on rank 3 and then its neighboring processes in the

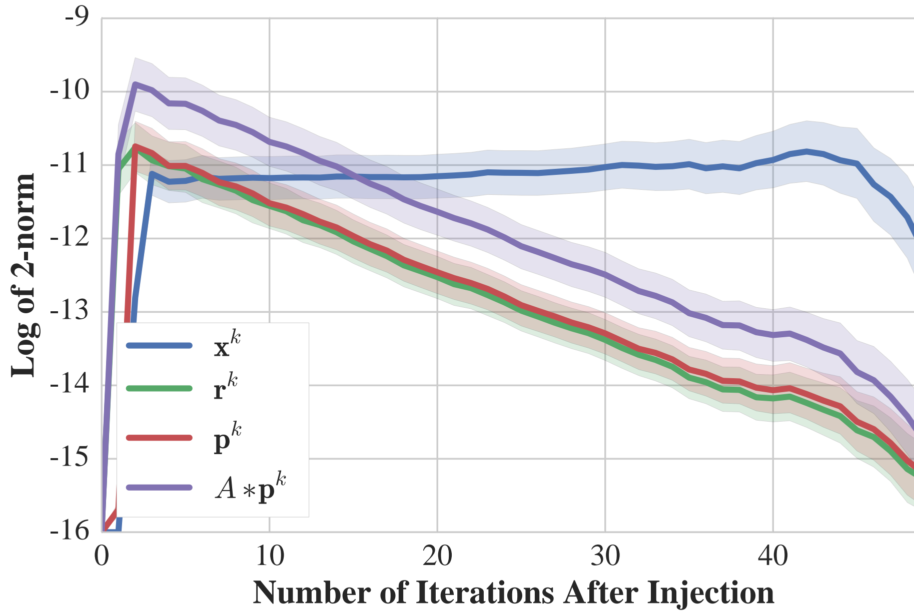


Figure 3.16: Average 2-norm of the error in selected variables on rank 3 for HPCCG within 90% confidence.

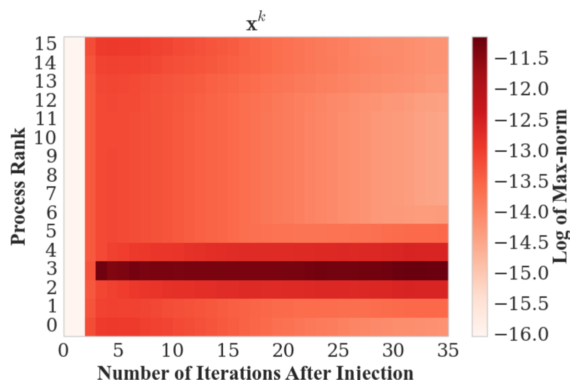
domain. Figure 3.22 shows the 2-norm in more detail for rank 3.

Figure 3.23 shows the average log of the max-norm on all processes as time evolves between the selected variables of *gold* and *faulty*. As with the 2-norm, the max-norm shows that corruption is present on all processes shortly after the fault occurs. Rank 3 shows the largest deviations followed by neighboring processes in the domain. Over time the magnitude does decrease slightly; first on rank 3 and then its neighboring processes in the domain. Figure 3.24 shows the max-norm in more detail for rank 3.

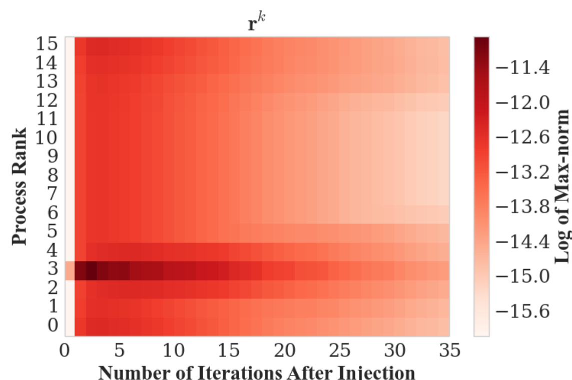
Impact of Local Problem Size

To explore how problem size impacts corruption propagation. The preceding application experiments are re-run with the same number of MPI processes, but with more local work per process. Jacobi sees a 2.25× increase in the number of grid points per process at 36864. The HPCCG local problem is $nx = ny = nz = 16$, a 1.9× increase. Whereas CoMD is run with 62500 atoms, resulting in a 1.9× increase in problem size.

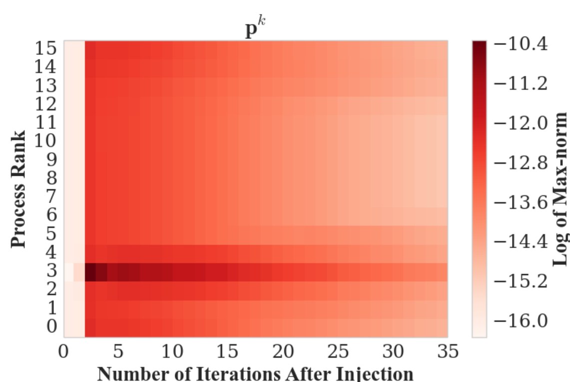
For Jacobi increasing the local problem size causes an increase in the average number of iterations to corrupt other processes by roughly 2×. Corruption propagation patterns and failure symptoms resemble previous results. Increasing the local problem size for HPCCG yields no increase in the speed of propagation to other ranks or a discernible change in failure symptoms. Increasing the problem size for CoMD causes a corresponding increase in the simulation domain size. The corresponding increase in the domain size keeps



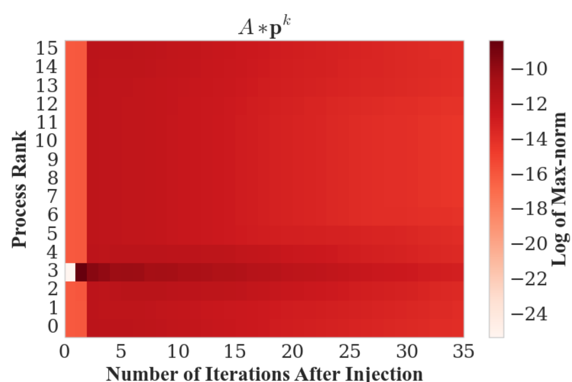
(a) Average max-norm of corrupted vector elements for iterative solution $\mathbf{x}^k$.



(b) Average max-norm of corrupted vector elements for residual $\mathbf{r}^k$.



(c) Average max-norm of corrupted vector elements for search directions $\mathbf{p}^k$.



(d) Average max-norm of corrupted vector elements for result of the SpMV of $A * \mathbf{p}^k$.

Figure 3.17: Average max-norm of corrupted vector elements for variables in the mini-app HPCCG in iterations after injection.

the density of the atoms constant. Without increasing the density of the atoms, a similar number of atoms are in the force calculation cut-off region allowing for a similar speed of propagation. In addition, the failure symptoms frequencies for CoMD are comparable with previous results.

3.4 Conclusion

In this chapter, propagation of corruption due to a soft errors is considered at two levels macro/micro. Micro-level propagation shows the latency of symptoms of soft errors, and how compiler optimizations affect this. Macro-level propagation results show the speed and intensity of corruption on the process suffering the fault. The following Chapter 4, discusses the fault injection tool FlipIt.

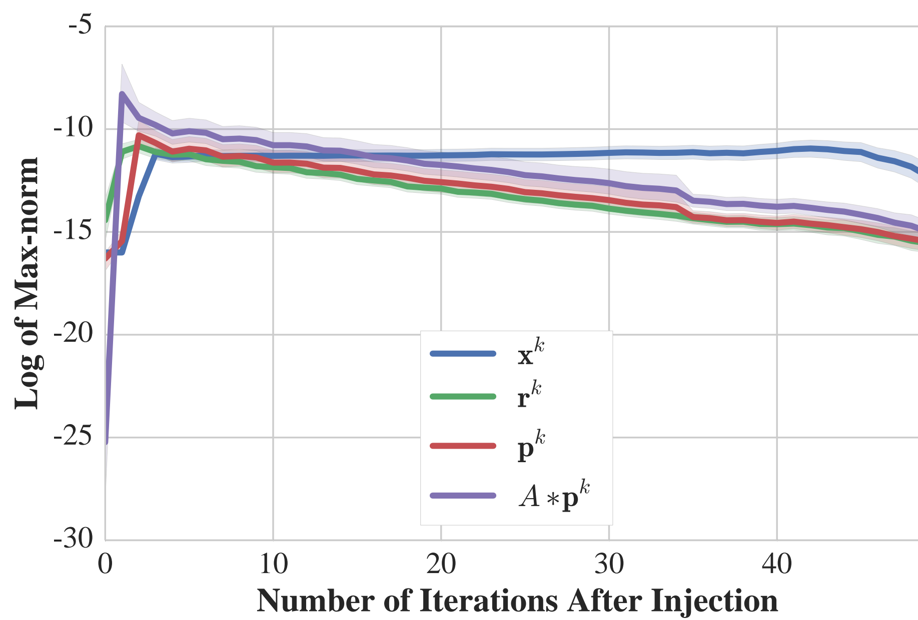
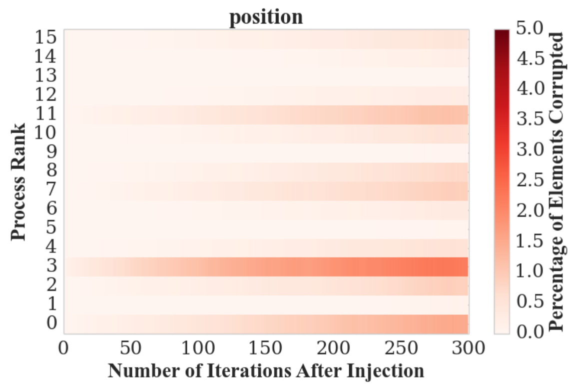
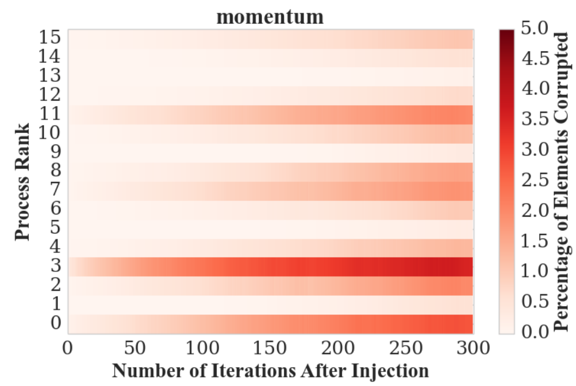


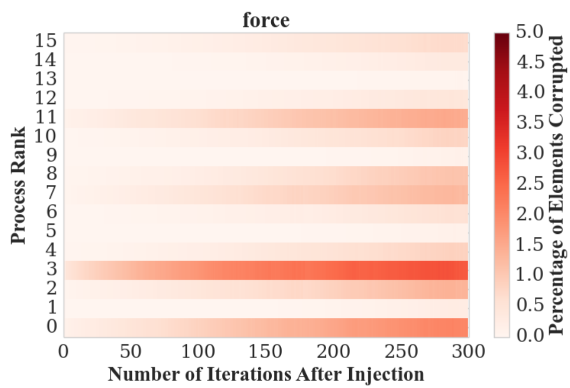
Figure 3.18: Average max norm of the error in selected variables on rank 3 for HPCCG within 90% confidence.



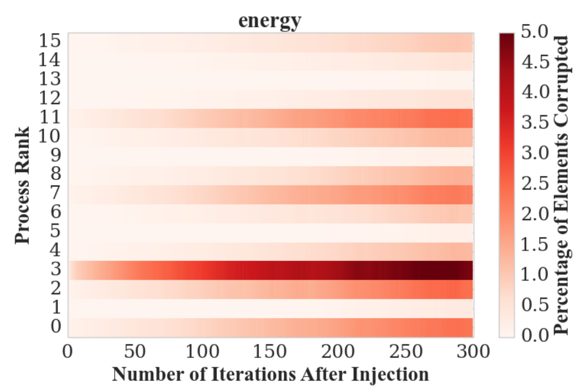
(a) Average percentage of corrupted vector elements for position vector.



(b) Average percentage of corrupted vector elements for momentum vector.



(c) Average percentage of corrupted vector elements for force vector.



(d) Average percentage of corrupted vector elements for energy vector.

Figure 3.19: Average percentage of corrupted vector elements for variables in the mini-app CoMD in iterations after injection.

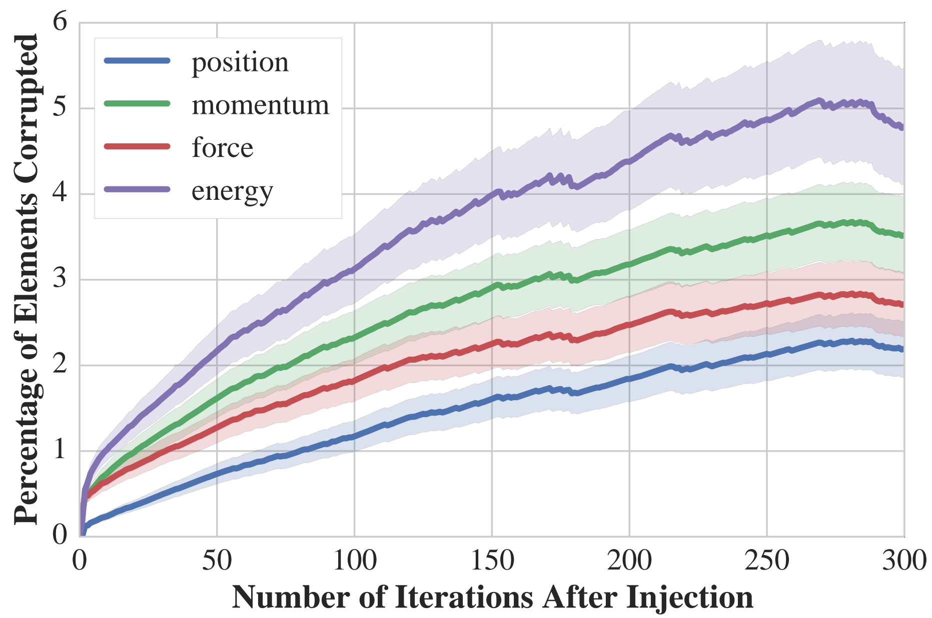
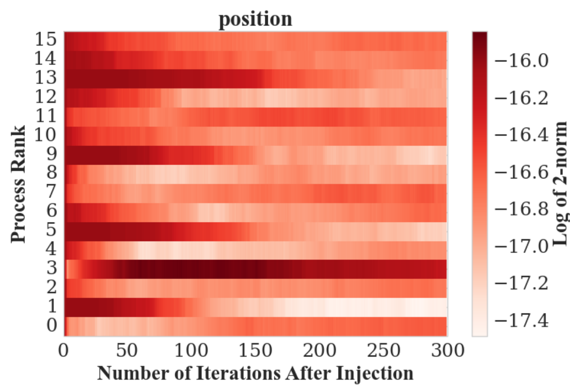
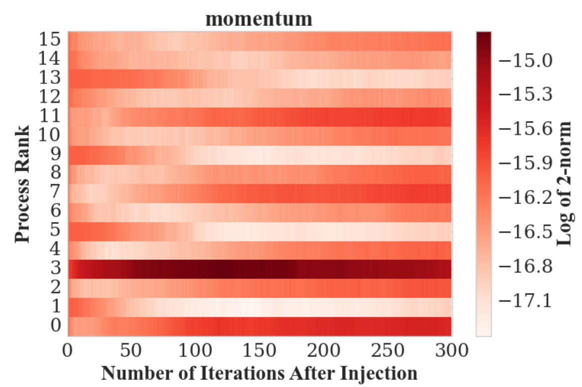


Figure 3.20: Average corruption in selected variables on rank 3 for CoMD.

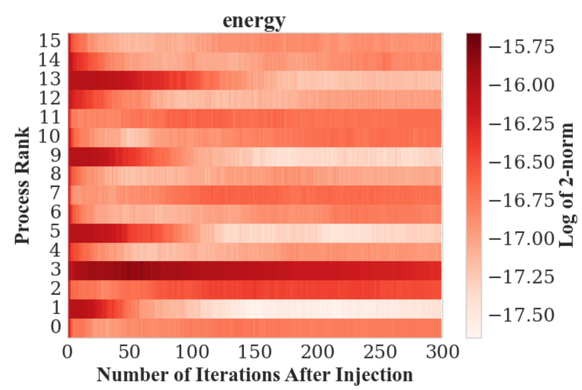


(a) Average log of 2-norm for position vector.



(b) Average log of 2-norm for momentum vector.

(c) Average log of 2-norm for force vector.



(d) Average log of 2-norm for energy vector.

Figure 3.21: Average log of 2-norm for variables in the mini-app CoMD in iterations after injection.

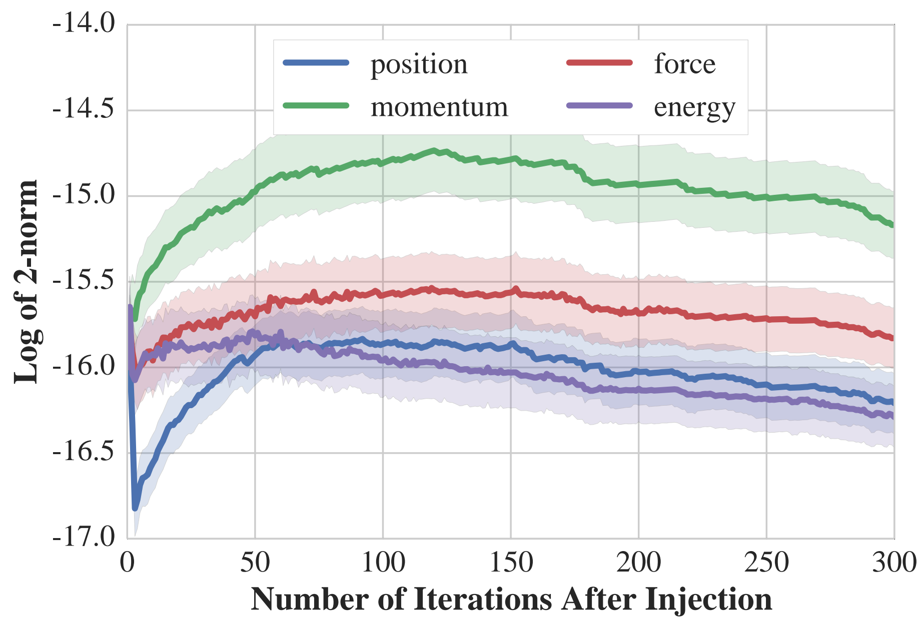
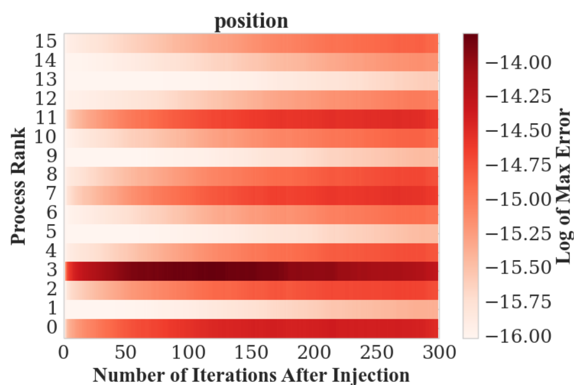
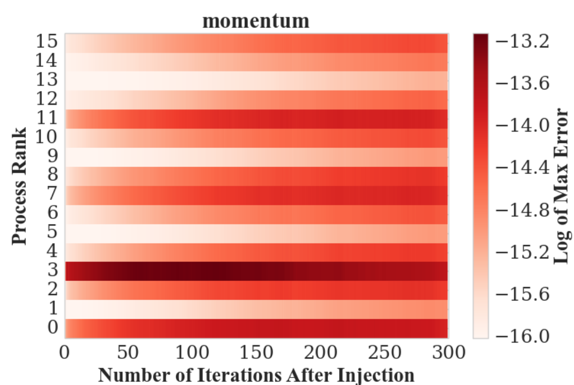


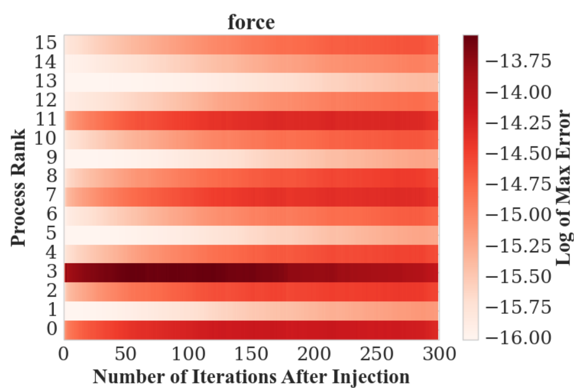
Figure 3.22: Average log of 2-norm in selected variables on rank 3 for CoMD.



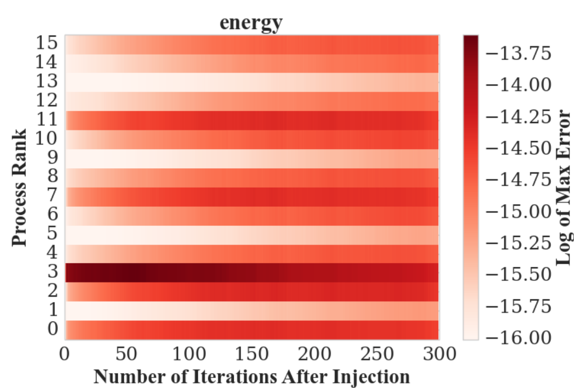
(a) Average log of max-norm for position vector.



(b) Average log of max-norm for momentum vector.



(c) Average log of max-norm for force vector.



(d) Average log of max-norm for energy vector.

Figure 3.23: Average log of max-norm for variables in the mini-app CoMD in iterations after injection.

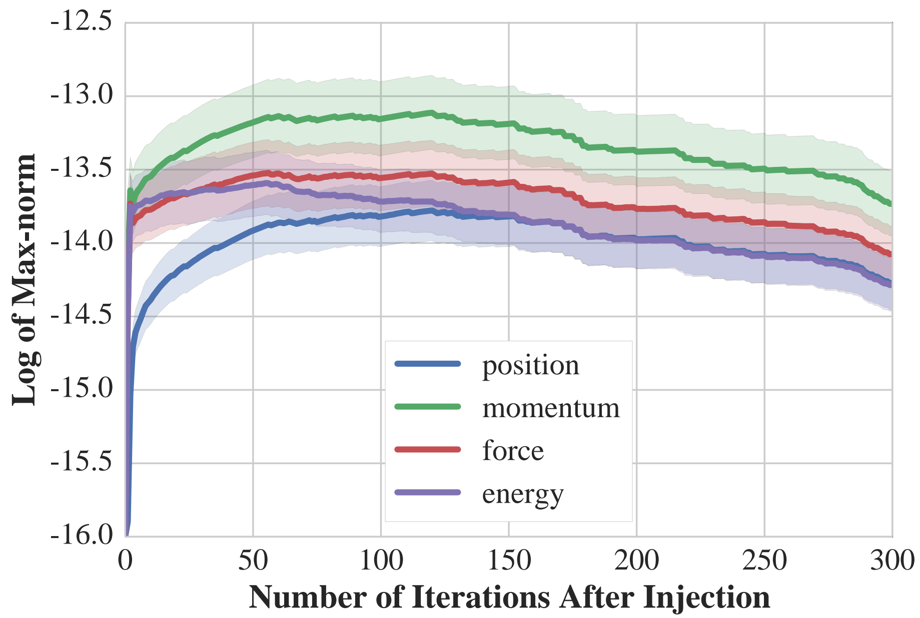


Figure 3.24: Average log of max-norm in selected variables on rank 3 for CoMD.

Chapter 4

Fault Injection

Portions of this chapter are taken from the publication "FlipIt: An LLVM Based Fault Injector for HPC" [18]

4.1 Introduction

To assess the resiliency of an application and to enable efficient fault injection we make the following contributions:

- the development of a robust LLVM based fault injector that targets HPC applications;
- an overview in its utility as a general purpose fault injection framework; and
- a demonstration of its usability, scalability, and robustness on production level HPC code.

The remainder of this chapter is structured as follows. The next section, discusses related background in the area of fault injection. Section 4.3 details the design, use, and adaptation to individual applications. Results from its usability, scalability, and robustness are shown in Section 4.4.

4.2 Background

There are many forms of fault injectors. The closer the fault is injected to physical hardware, the more accurate and realistic the fault injector. At the lowest level, injections come in two forms: real and simulated.

In real injections, the hardware is irradiated with a neutron beam. While this method is highly accurate, it has significant drawbacks mainly its cost and availability to a limited number of researchers, which limits its applicability to HPC.

Simulated injections comprise many techniques at both the hardware and software level. At the hardware level, gate-accurate models are constructed, and fault injection occurs systematically with gate level granularity. With gate-accurate simulations, execution of a full application is possible, but a large scale machine

is prohibitive due to execution overhead. A fault injection framework that operates with low overhead on current hardware is needed to inject faults in HPC applications.

Since DRAMs have a higher level of protection from silent errors than processors thanks to being easier to protect by ECC and Chipkill [32], the focus here is on faults that arise in the processor. Section 4.3 details the design of the fault injector that simulates the presence of soft errors as the manifestation of bit-flips in register values. KULFI [97] is a publicly available LLVM based fault injector. Although not specifically designed for HPC applications, its structure provides a basis for the fault injector developed here.

4.3 Fault Injector

FlipIt[1] is structured as an LLVM [62] compiler pass and is based on KULFI [97]. Notable extensions have been added to increase its robustness and efficacy. Such extensions include:

- support for more data types;
- ability to work with multiple source files simultaneously;
- user customized fault distribution and event logger;
- support for a larger subset of the LLVM instructions; and
- MPI rank aware.

FlipIt chooses to use an LLVM compiler pass to simulate transient errors instead of randomly flipping bits inside the binary in order to provide more control over time and location of an injected fault. FlipIt is structured in two parts: (1) a compiler pass that identifies locations for fault injection and inserts code to perform the injections and (2) a runtime API that defines when and where faults occur.

4.3.1 LLVM Compiler Pass Design

The compiler pass proceeds by iterating over all functions in a source file. On discovery of a function marked for injection, the included instructions are surrounded by instructions that injects a fault based on a probability model. Here, a fault results in a single bit-flip in a source operand or the result of an LLVM instruction. The compiler pass identifies and instruments locations for faults to be injected at runtime. Figure 4.1 illustrates the code transformation for a single `add` instruction.

The result of the `add` instruction is sign extended before being passed to a function `crptInt` that performs the injection. The result of the `crptInt` function is equivalent to the result of the `add` if no fault is injected,

[1] https://github.com/joncalhoun40/FlipIt

```
define i32 @add(i32 %a, i32 %b) #0 {
entry:
    %add = add nsw i32 %a, %b
    ret i32 %add
}
```

(a) Original LLVM IR.

```
define i32 @add(i32 %a, i32 %b) #0 {
entry:
    %add = add nsw i32 %a, %b
    %data = sext i32 %add i64
    %tmp = call i64 @crptInt(i32 788529152,
                    double 0.01, i64 %data)
    %crptAdd = trunc i64 %tmp to i32
    ret i32 %crptAdd
}
```

(b) Transformed LLVM IR.

Figure 4.1: Code transformation by FlipIt to inject faults.

or has a single-bit that differs if a fault is injected. After truncating back to the original data size, all subsequent use of the `add`'s result variable are replaced with the value returned from the injection function. This process continues for every instruction inside the function.

Corrupting Functions

As shown in Figure 4.1, a function call to `crptInt` is used to determine if a fault should be injected and to perform the injection. Algorithm 4.1 provides generic logic for the injection. In every corrupting function, the argument list is the same except for the data type of the value being corrupted. Table 4.1 details the argument list of the corrupt functions from left to right. The first argument is a 32-bit unsigned integer containing three packed values: byte location for injection, bit location for injection, and the unique fault site index. Figure 4.2, shows how the bit fields in the 32-bit unsigned integer are partitioned. The byte field allows for a fixed byte location (0–7) for injection, and the bit field allows for a fixed bit inside that byte to corrupt (0–7). To inject into a random byte and bit location, the value of -1 can be provided. If the byte specified is outside the range of the data type, FlipIt uses the modulus operator to wrap the byte to the correct range. The unique fault site index comprises the lowest 24 bits in the compressed argument. This allows for 16,777,216 unique static instructions to be instrumented for fault injection. The remaining arguments of the corruption function are the probability of injecting a fault in this location and the data to corrupt.

By default, and in Section 4.4, FlipIt assumes that all LLVM instructions have an equal probability of experiencing a fault. However, in practice, each LLVM instruction should have differing probabilities that are a function of the underlining hardware. Therefore, if information is known for a processor *a priori*, then the scaled probabilities should be incorporated into to the configuration file used by the FlipIt compiler pass.

Complex data types — e.g. user defined classes and structures — pointers with multiple levels of indirection, are not considered for injection in KULFI. All data types are represented using a finite number of bits;

Algorithm 4.1: Generic corrupt logic.

Input: site_prob: Probability that this site is faulty.
site_index: Unique index of this fault site.
data: Value eligible for corruption.
Result: Data unmodified(no injection), or data with a single bit-flip(injection).

1 **if not** shouldInject(injector_on, rank_inject, site_prob) **then**
2   **return** data
3 **else**
4   bit_position ← *random bit position in targeted byte*
5   logInjection(site_index, bit_position)
6   data_corrupt ← data ⊕ (`0x1` ≪ bit_position)
7   **return** data_corrupt

Table 4.1: Corrupt function's arguments (left to right).

Argument	Description
1	Compressed argument: Byte location, bit location, and unique fault site index.
2	Probability that this instruction is faulty.
3	Data that can become corrupted by a bit-flip.

therefore, FlipIt allows for injections into these data types by first casting the variable to a 64-bit integer. Next, a call to corrupt the 64-bit integer is inserted and the value returned from this call is cast back to the appropriate data type. As with all corruptions, all subsequent uses are replaced.

FlipIt's instrumentation is limited to code it directly compiles; however, a function targeted for corruption may include function calls. Corrupting these function calls is critical to properly modeling a fault function. There are two cases that must be considered. First, if source code is available for the function it should be compiled and instrumented with FlipIt. Second, if the function is unable to be compiling is not an option — e.g. due to unavailable source (pre-compiled library) — then FlipIt is provided the probability of

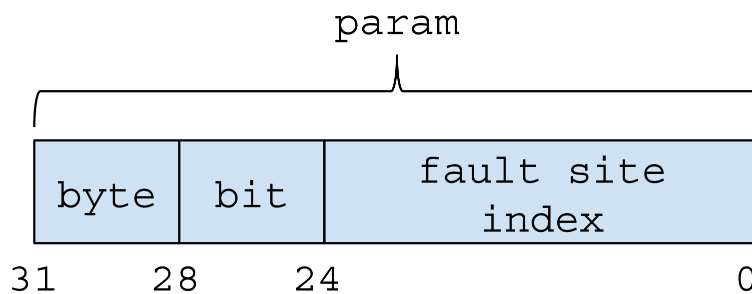


Figure 4.2: Partitioning of compressed argument for FlipIt's corrupting functions.

injecting a fault into return value or argument of the `call` instruction. The probability is read from the compiler passes configuration file as a key-value pair with the function's name. Calculating the probability is left to the user. One option uses the function's runtime to estimate the number of instructions executed. The probability is proportional to the number of instructions inside the function. This second method of injecting into functions ignores side effects of the function. ABFT resiliency schemes focus on data directly pertaining to the algorithm — e.g. pressure variable, a matrix. Modifications of global/external state inside a function does not impact the FlipIt's ability to corrupt the application data that ABFT resiliency schemes operate on; either before or after the function call.

Parallel Design

In order to support HPC applications, our fault injector is aware of the processes' current MPI rank inside `MPI_COMM_WORLD`. This allows fault injections on a subset of ranks specified at runtime via command line arguments or through FlipIt's API. Large scale machines have custom MPI distributions that are tuned for performance. Because a non-negligible portion of time is spent in MPI routines, FlipIt must consider MPI calls as potentially faulty. The source code for the machine optimal MPI is not available. This implies that FlipIt needs to utilize the method outlined previously to when inject faults into MPI routines.

Compiler Log File

As the compiler pass instruments source code, details about the injection locations are logged for use when controlling injections with the runtime API and during analysis of fault injection runs. In particular, FlipIt logs following information when instrumenting code:

- unique fault site number (enumeration of static instructions);
- location in instruction — e.g. result or operand;
- instruction type — e.g. *Arith-FP*, *Arith-Fix*, *Pointer*, *Control-Branch*, *Control-Loop*; and
- source file location and line number.

Definitions of FlipIt's instruction classification types are found in Table 4.2. An example of a compiler log file in ASCII is shown in Figure 4.3. In practice, FlipIt creates binary log files to reduce the file size.

Table 4.2: FlipIt instruction classification types.

Instruction Type	Description
Arith-FP	Floating-point arithmetic — e.g. add, multiply.
Arith-Fix	Fixed-point (integer) arithmetic — e.g. add, multiply.
Pointer	Calculations directly related to the use of pointers — e.g. loads, stores, address calculation.
Control-Branch	Comparisons and branching.
Control-Loop	Operations on loop induction variables.

```
Function Name: waxpby
-----------------------------------------------------------------------------
#1291   FCmp    Arg 0   Control-Branch  /home/jcalhoun/HPCCG-1.0/waxpby.cpp:53
#1292   ICmp    Arg 1   Control-Branch  /home/jcalhoun/HPCCG-1.0/waxpby.cpp:57
#1293   Add     Result  Arith-Fix       /home/jcalhoun/HPCCG-1.0/waxpby.cpp:57
#1294   Add     Result  Arith-Fix       /home/jcalhoun/HPCCG-1.0/waxpby.cpp:57
#1295   Add     Result  Arith-Fix       /home/jcalhoun/HPCCG-1.0/waxpby.cpp:57
#1296   And     Result  Arith-Fix       /home/jcalhoun/HPCCG-1.0/waxpby.cpp:57
#1297   ICmp    Arg 0   Control-Branch  /home/jcalhoun/HPCCG-1.0/waxpby.cpp:57
...
```

Figure 4.3: ASCII version compiler log file.

4.3.2 Usability and Extensibility

Source Modification

FlipIt is designed to require minimal modification to existing codes while at the same time providing a high degree of robustness and flexibility to conduct detailed fault injection campaigns. Figure 4.4 shows the minimum required changes to the mini-app HPCCG in order to use FlipIt. The call to `FLIPIT_Init` should precede all usage of code compiled with our fault injector, and no such code should be executed after the call to `FLIPIT_Finalize`. `FLIPIT_Init` requires the processes MPI rank; therefore, FlipIt ensures that MPI has been initialized by electing the user to insert calls to `FLIPIT_Init`. FlipIt can not do this inside the compiler as it is possible that the file containing `MPI_Init` is never seen by the compiler pass. To provide the user with a more fine grain control over fault injection, the functions in Table 4.3 are provided.

Compiling

The use of FlipIt requires little modification to current building practices. Once functions have been identified by the programmer, the source files containing the functions is recompiled using our compiler pass. Figure 4.4 shows this process with a change in the Makefile.

The compiler script `flipit-cc` handles the more complicated process of instrumenting the code. This process is outlined in Figure 4.5. Step 1 in the compilation process compiles the original source with `clang`[2]

[2]https://clang.llvm.org/

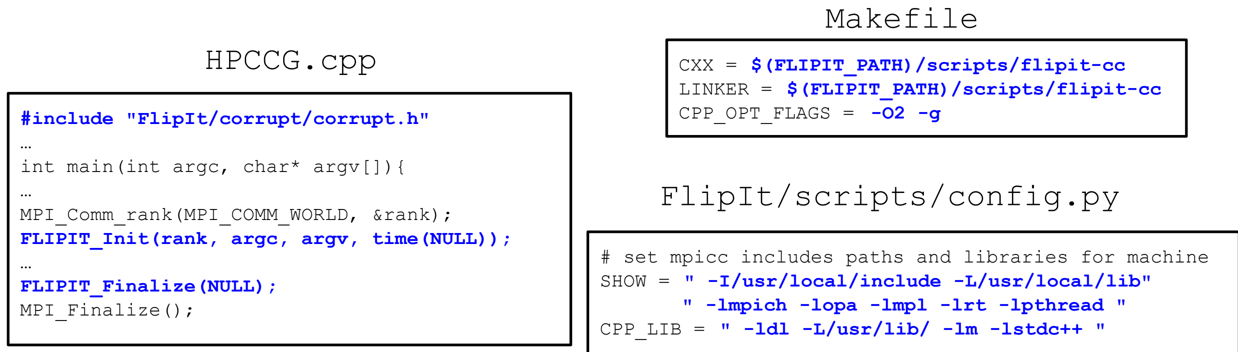


Figure 4.4: Blue text shows required changes to HPCCG source files (`HPCCG.cpp` and `Makefile`) and additional libraries to FlipIt should link against (`FlipIt/scripts/config.py`) to enable fault injection.

Table 4.3: User callable functions.

Function name	Description
`FLIPIT_Init`	Initializes fault injector. Turns on injector.
`FLIPIT_Finalize`	Cleans up injector. Turns off injector.
`FLIPIT_SetInjector`	Zero: turns off injector. Non-zero: turns on injector.
`FLIPIT_SetFaultProbability`	Sets the probability function with a user defined function.
`FLIPIT_SetCustomLogger`	Sets user defined logging function. Called on all injections.
`FLIPIT_SetRankInject`	Zero: rank not faulty. Non-zero: rank can suffer faults.
`FLIPIT_CountdownTimer`	Sets number of instructions to execute before injecting.
`FLIPIT_GetExecutedInstructionCount`	Gets number of fault injection locations traversed.
`FLIPIT_GetInjectionCount`	Gets number of faults injected locally.
`FLIPIT_SetMaxInjections`	Sets max number of faults for process.
`FLIPIT_GetMaxInjections`	Gets maximum number of faults locally.

into LLVM bitcode. LLVM bitcode is an intermediate representation (IR) formate that optimizations and transformations are performed on before generating machine code. It is necessary to compile with `-g` to relate fault sites indexed by the compiler pass to the source code line. If this flag is omitted, the compiler pass relates the injection to the location in the bit-code which does not always have a clear translation back to the source code due to a provided optimization level. Generally it is always possible to map an injection location to a function. The bitcode generated by `clang` is linked with a bitcode header file that defines the interfaces for the corruption functions in Step 2. Step 3 takes the resulting bit code and runs the FlipIt compiler pass generating the log file and the instrumented bitcode. Table 4.4 details all possible arguments to the compiler pass and their default values. Step 4 compiles the bit-code into object code for linking. Once all the object code is generated, the application is linked against the FlipIt runtime static library as the executable is generated.

For MPI applications, `mpicc` is a wrapper around a native compiler. The compiler flags `-show` and `-showme` for MPICH and OpenMPI, respectively, provides the exact compiler command used to com-

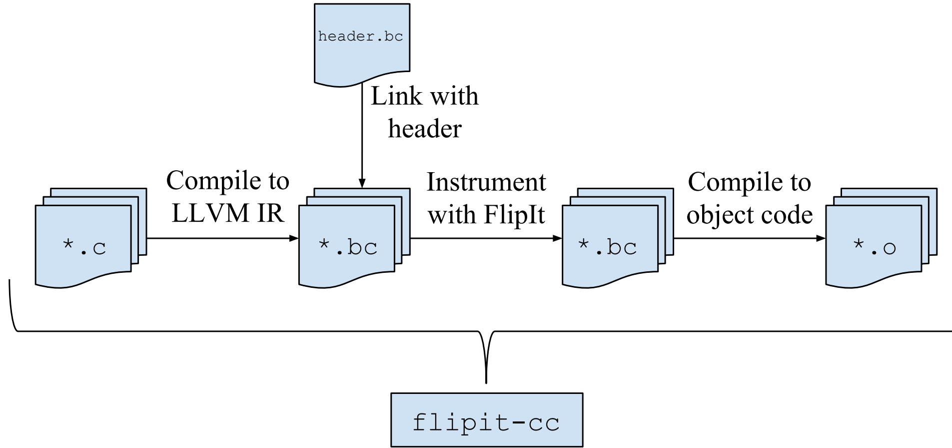


Figure 4.5: Steps in compiling and instrumenting code with `flipit-cc`

Table 4.4: Compiler pass arguments.

Argument	Type	Req.	Default	Description
config	string	No	FlipIt.config	Path to configuration file.
funcList	string	Yes	—	Functions to corrupt.
prob	double	Yes	1e-8	Default per instruction fault probability.
byte	int	No	-1	Byte to flip bit in (0–7). (-1 random).
bit	int	No	-1	Byte to flip bit in (0–7). (-1 random).
ptr	bool	No	true	Allow bit-flips in *Pointer* instructions.
ctrl	bool	No	true	Allow bit-flips in *Control-Branch* and *Control-Loop* instructions.
arith	bool	No	true	Allow bit-flips in *Arith-FP* and *Arith-Fix* instruction.
srcFile	string	No	UNKNOWN	Name of the original source file being compiled.
stateFile	string	No	.FlipItState	Name of the state file being updated when compiling. Used to provide unique fault site indices.

pile the source file. For successful compilation of MPI applications it is necessary to inform FlipIt of the libraries and include paths that `mpicc` wraps. These libraries and paths are added into the file `FlipIt/scripts/config.py` in the `SHOW` variable. In addition, libraries required for `C++` applications can be provided and any other libraries that may not be explicitly stated in the Makefile.

Programmer Control

The choice to use a static library for the corruption routines is influenced by three key points: 1) the need to compile multiple source files for a single executable; 2) the ability to limit overhead of the fault injector; and 3) to allow for application specific behavior such as the collection of user defined statistics. Straightforward use of KULFI is restricted to one source file, which limits use by requiring the programmer to place all functions of interest into a single source file, or by requiring multiple recompilations to cover all functions of interest, but sacrificing the ability to look at complex function interactions.

FlipIt's approach to fault injection increases the static and dynamic instruction count for the application, which leads to increased execution time. The overhead is mitigated when FlipIt is given more direction about where and when to inject a fault. During compile time, the compiler pass can be directed to corrupt a subset of functions and instructions allow for selective injections via the configuration file, or the redefinition of the `shouldInject` function (see Algorithm 4.2) called by each corruptible function. However, if the location targeted for injection changes, then the application would need to be recompiled. At a high level, FlipIt provides a series of command line arguments listed in Table 4.5 to control which MPI ranks experience an injection and which fault injection locations are valid dynamically at runtime. The command line arguments are sufficient to pinpoint where a fault is injected, but do not easily account for when. To address this, the runtime API functions in Table 4.3 assist in refining the location and time for injection with combination of the functions `FLIPIT_SetInjector`, `FLIPIT_CountdownTimer`, and `FLIPIT_SetFaultProbability`. These functions are used in Section 4.4 to perform a selective injection into the computation of the residual vector.

Algorithm 4.2: Basic `shouldInject` logic.

```
Input: injector_on: Boolean signifying if injector is on.
       rank_inject: Boolean signifying if rank is faulty.
       site_prob: Probability that this site is faulty.
Result: Boolean indicating an injection should occur.
1 P ← probability()
2 if injector_on and rank_inject and site_prob > P then
3 |   return TRUE
4 else
5 |   return FALSE
```

Table 4.5: Command line arguments for fault injector.

Argument	Description
`--numFaulty`	Number of faulty MPI ranks.
`--faulty`	List of faulty MPI ranks.
`--numberFaulty`	Number of faulty MPI ranks.
`--numberFaultyLoc`	Number of active fault locations.
`--faultyLoc`	List of active fault locations.
`--stateFile`	Location of statistic file generated by compiler pass.

Analysis

When the code is compiled, the compiler pass generates a log file that specifies all locations where faults can be injected. As the application is run, information about the faults being injected is logged per rank for later inspection. Two types of data are logged every time a fault is injected. The first kind is information about

the faults themselves — e.g. the fault site numbers, bits flipped, and values from the fault distribution. This information is used in conjunction with the fault site log files to determine in what function and where in this function the fault is injected. The second type of information logged on each fault injection is accomplished through a user-defined function. This user defined function is set using `FLIPIT_SetCustomLogger` (see Table 4.3). Application specific information — e.g. iteration number, application phase — can be logged to obtain a temporal look of the injection during the application.

Even if no faults are injected, statistics are still collected. For every rank, a histogram is generated showing the frequency of each fault site. To determine if the execution path of the application changes due to a fault, the histogram generated in the fault free case is compared with the histogram from an application run with faults. Any discrepancies in these histograms suggest differing paths of execution. Further insights are found by using the injection log and the compiler logs alongside the histograms to determine precisely what occurred due to the fault. To simplify this process the fault analysis tool FaultSight[3] can be used to quickly and efficiently analyze the results of a fault injection campaign.

4.4 Experimental Results

In order to show the scalability and flexibility of FlipIt, sections of Algebraic multigrid (AMG) solver in HYPRE[4] are compiled with FlipIt to enable fault injection during solving of the linear system. AMG uses one iteration of Jacobi relaxation for smoothing while solving a 2D Laplacian with zero on the boundaries to a tolerance of $1e{-}7$. Each process has 16384 degrees of freedom. Profiling HYPRE allows us to determine the call stack inside `HYPRE_BoomerAMGSolve`. All functions in this call stack are instrumented by FlipIt. For the mini-apps, CoMD[5] and HPCCG[6] the entire mini-app is compiled with FlipIt, and 4000 atoms over 500 time-steps and a local block size of $nx = ny = nz = 48$ per process, respectively.

4.4.1 Scalability

To characterize the scalability of FlipIt, weak scaling experiments, Figure 4.6, for the applications are run on Blue Waters with 16 MPI processes per node. Every data point is the average of five runs. In this figure, FlipIt is fully active on all MPI ranks and fault sites; however, no fault is injected due to every location having a fault probability of 0. FlipIt's two methods of determining when to inject a fault are tested: *Probability* calculates a pseudo random value from a distribution at each fault site, and *Countdown*

[3]https://github.com/einarhorn/faultsight
[4]https://computation.llnl.gov/projects/hypre-scalable-linear-solvers-multigrid-methods/software
[5]https://github.com/exmatex/CoMD
[6]https://mantevo.org/applications.php

decrements a counter of the number of dynamic fault sites to iterate over before injecting. In Figure 4.6, the execution time for the instrumented code time has a similar scaling behavior of the original code. FlipIt's instrumentation increases computation but not communication. As a result, the scaling behavior changes by a constant factor. Looking at the two methods determining when to inject a fault, the instrumented code that calculates a pseudo random number, *Probability*, adds additional computation and sees a larger overhead than *Countdown* that decrements a counter.

To quantify the impact on performance, instrumentation overhead is presented in Figure 4.7. FlipIt increases the dynamic instruction count, which produces a corresponding increase computation time; however, as HYPRE utilizes more processes, communication time begins to dominate computation time. As this occurs, FlipIt's overhead diminishes as the problem scales; this is due to masking by inter-node communication. HPCCG shows a similar trend, but it is not as pronounced as HYPRE. CoMD shows no decrease in overhead as the number of processes increase; this is due to its good weak scaling behavior.

4.4.2 Selective Injection

To highlight the flexibility of FlipIt, we inject a fault in the first element of the residual vector just before it is written to memory. The fault occurs during the first cycle on the finest level during the construction of the residual vector. In order to yield an accurate injection, the fault site location must be known. Inspecting the compiler pass logs determines the correct fault site index. This information is passed to FlipIt along with the faulty rank number, 0, via the command line arguments. Because the residual is computed via a SpMV routine that is also used during problem setup, two calls to `FLIPIT_SetInjector` are added, one to turn off the injector after initialization of FlipIt and the other to turn it on just before calculating the residual. Figure 4.8 shows the fault's effect on the relative residual, the stopping criterion for AMG. The name of the trend indicates which bit is flipped in the 64-bit floating-point number.

In this location, a single bit-flip is either masked by the application or increases the number of iterations. Since this fault occurs in the floating-point arithmetic of the problem, it would not be detected and would create silent data corruption (SDC). Indication of an error does not occur until the application's results were analyzed. This high latency suggests that work should be done to design SDC detectors to catch such SDCs early limiting their impact on HPC applications. In order to test the effectiveness of these SDC detectors, a fault injector such as FlipIt is required.

FlipIt allows targeting of different classes of instructions, and depending upon which classes are active, the effects on the application vary. The classifications of *Control-Branch* and *Control-Loop* are combined into *Control*, and the classifications of *Arith-FP* and *Arith-Fix* are combined into *Arithmetic*. Table 4.6

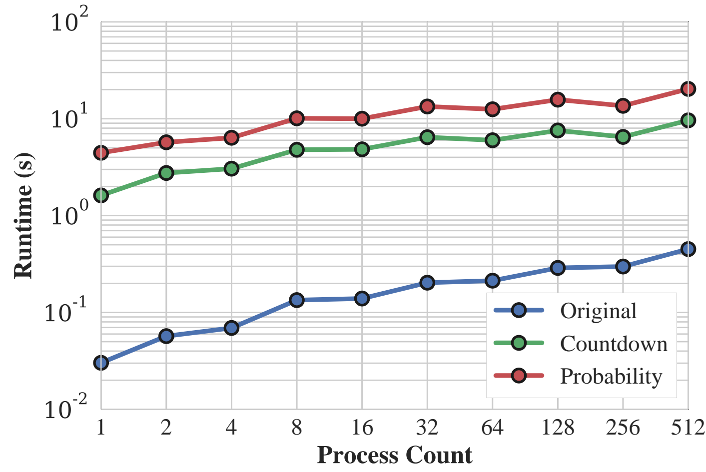
	Pointer	*Control*	*Arithmetic*	All
Crash	41%	29%	21%	29%
More V-Cycles	6%	0%	6%	4%
Same V-Cycles	53%	71%	73%	67%

Table 4.6: Results of injecting into certain types.

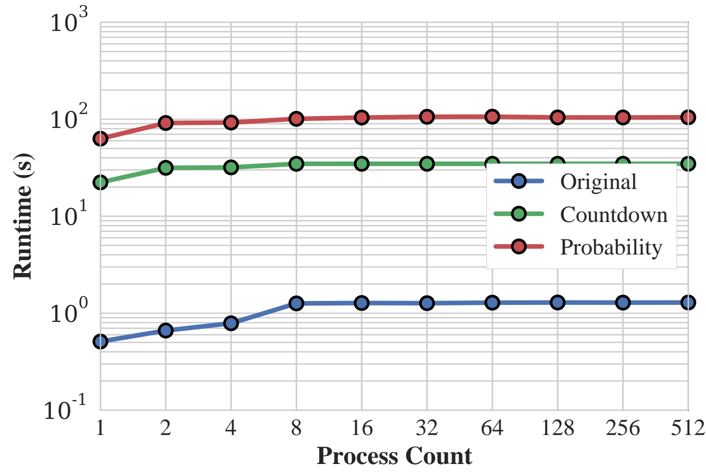
reports the average of 1,000 trials and shows injections into pointers have a corresponding increase in the percent of trials that crash. Likewise, injection into arithmetic operations of AMG or access to the incorrect data with corrupted pointers, increases the percent of trials that require a higher number of iterations required to converge. There is a small increase in the percent of trials that crash with control injections due to taking incorrect paths and incorrect indexing. By the use of these instruction classes, unique injection campaigns can be created allowing the study of an application's susceptibility to certain types of errors and the effectiveness of detection schemes.

4.5 Conclusion

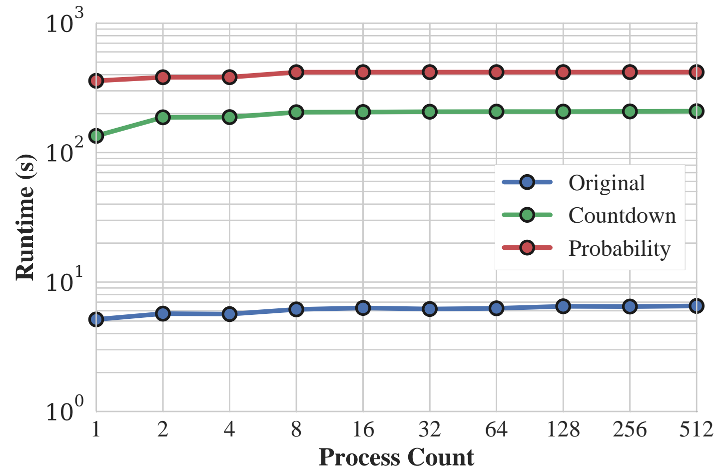
This chapter presents the design of LLVM based fault injection tool FlipIt. In addition, experimental results show the scalability and overhead of FlipIt. The usability of FlipIt is explored by selectively injecting a fault inside the AMG solver in HYPRE. The following chapter, Chapter 5, presents a tool is built on top of FlipIt that tracks corruption propagation in application variables at a high and low level.



(a) Main computation loop runtime for HYPRE.



(b) Main computation loop runtime for HPCCG.



(c) Main computation loop runtime for CoMD.

Figure 4.6: Weak scaling of applications with and without compilation with FlipIt.

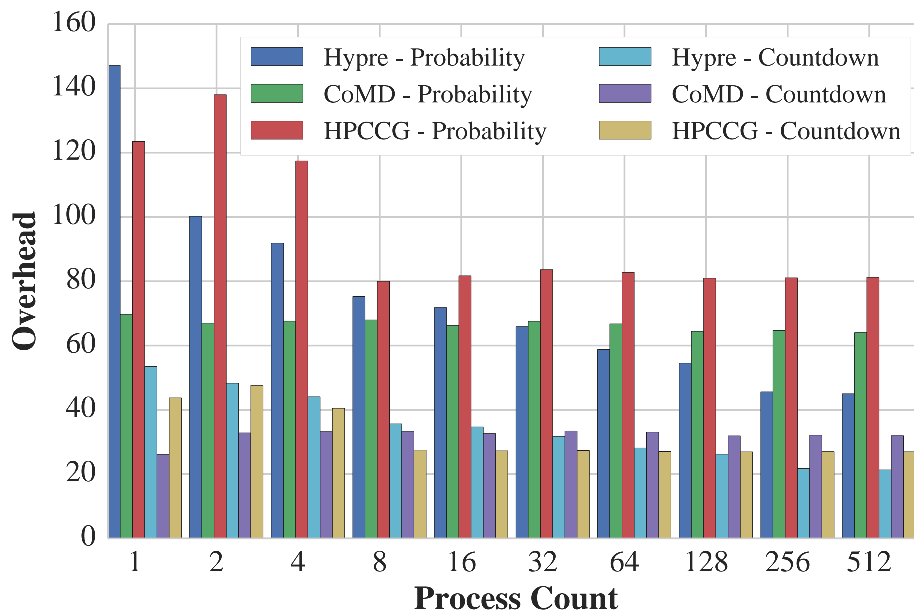


Figure 4.7: Overhead of instrumentation of applications compared to reference code.

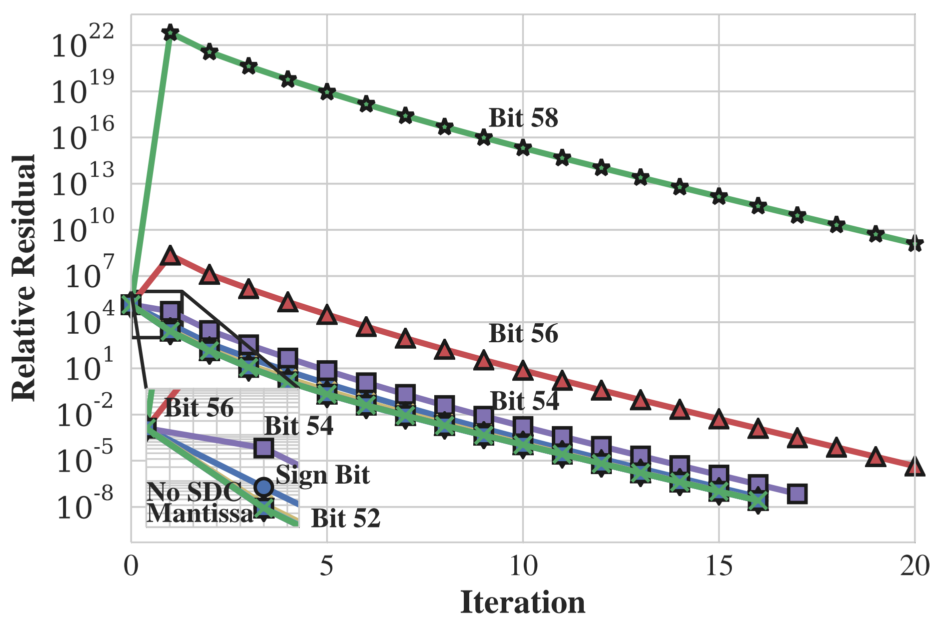


Figure 4.8: Selective injection in residual calculation on rank 0. 16 processes with approximately 16,384 unknowns per process.

Chapter 5

Soft Error Propagation Tracking

Portions of this chapter are taken from the publications "Towards a More Complete Understanding of SDC Propagation" [21]

5.1 Introduction

In this chapter, error resulting from an activated fault is referred to as state corruption or simply as corruption in order to avoid confusion with numerical errors in HPC applications. To assist in the analysis of the propagation of corruption of inside applications, an open source LLVM based tool called SDCProp[1] is developed to track propagation at various levels of granularity.

5.2 Tracking Propagation

Once a fault is activated, corruption is present in the system. The location of the initial corrupted value is critical to identifying corruption propagation. During the execution of the remaining program instructions, the corruption can be masked due to programmatic or algorithmic properties. This can lead to a system detectable event, such as a segmentation fault or can propagate to corrupt more of the program state and become a silent data corruption (SDC). SDC detection schemes detect by investigating application state for significant deviations and abnormalities.

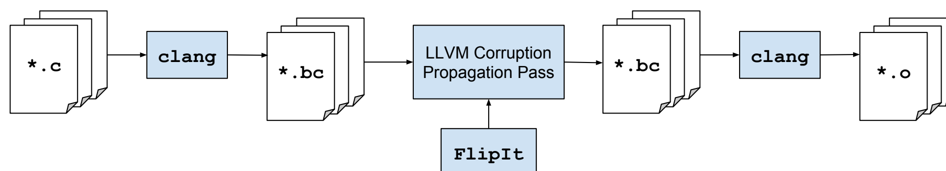


Figure 5.1: Overview of the instrumentation process that transforms source code to be able to track propagation and inject faults.

[1]https://github.com/joncalhoun40/SDCProp

Two levels of corruption propagation are considered:

macro tracks deviations in the state variables through the simulation; and

micro tracks deviations in the loads, stores, and other low-level operations.

In order to track corruption propagation at both the micro and macro level, an LLVM-based [62] instrumentation tool, SDCProp, is developed to emulate lock-step execution of a correctly executed (*gold*) along with a faulty execution (*faulty*). Optimized LLVM IR code is run though a compiler pass to instrument it to track corruption propagation and to insert fault injection code before being compiled to object code (see Figure 5.1). Figure 5.2 illustrates the instrumentation process performed in the compiler pass. Source code is instrumented (Section 5.2.1) after compiler optimizations are performed in order to mitigate the impact of instrumentation on the object code generated. An API call at natural points in the application — e.g. end of an iteration — tracks propagation of application level memory allocations at the macro-level. At the micro-level, the LLVM compiler pass adds instrumentation to track propagation at a load/store granularity and relates loads/stores to application level memory allocations.

After replication, there exists two distinct sets of instructions: one for the *gold* application (green) and one for the *faulty* application (red) (see Figure 5.2a). Instructions for the *gold* application forms a reference set of loads, stores, and correct program behavior that the *faulty* application instructions are compared against. Replicating all instructions creates separate memory images for both the *gold* and *faulty* applications. Thus, these two sets of instructions do not share the same data.

5.2.1 Micro-level Propagation Tracking

At the micro-level, loads and stores are monitored for deviations since these operations directly impact the state variables. Stores *commit* erroneous values to state variables, while loads allow for reuse and propagation of corrupted data. Examining corruption propagation at the micro-level allows a fine grain view of corruption propagation inside the application. This is useful in assessing the bound on a pointer dereference leading to a segmentation fault, to determine the number of loads and stores that are perturbed, to quantify the deviation when a corruption is masked by the application, and to examine the impact of compiler optimizations at a fine-grain level.

Because instructions from *gold* and *faulty* are interleaved, corresponding operations — i.e. loads and stores — are next to each other allowing for straightforward comparison of both the values and the addresses. To simplify the comparison, all loads and stores in *faulty* are replaced with an instrumentation call (see Figure 5.2c). In addition to determining if a loaded or stored value deviates from the expected *gold*

```
a = ld a_ptr        a_f = ld a_ptr_f
b = add a, 2        b_f = add a_f, 2
c = cmp b, 0        c_f = cmp b_f, 0
br c, if, else      br c_f, if, else
```

(a) Code is duplicated forming two sets *gold* (green code) and *faulty* (red code).

```
a = ld a_ptr
a_f = ld a_ptr_f

b = add a, 2
b_f = add a_f, 2

c = cmp b, 0
c_f = cmp b_f, 0
br c, if, else
br c_f, if, else
```

(b) *gold* and *faulty* code is interleaved to emulate lock-step execution.

```
a = ld a_ptr
a_f = chkLd(a_ptr_f, a_ptr, a)

b = add a, 2
b_f = add a_f, 2

c = cmp b, 0
c_f = cmp b_f, 0
br c, if, else
br c_f, if, else
```

(c) All loads and stores in *faulty* are replaced with instrumentation calls to track corruption propagation at a load/store level of accuracy.

```
a = ld a_ptr
a_f = chkLd(a_ptr_f, a_ptr, a)

b = add a, 2
b_f = add a_f, 2

c = cmp b, 0
c_f = cmp b_f, 0
checkDiverged(c, c_f)
br c, if, else
```

(d) *faulty* branches are replaced with an instrumentation call to check for control flow divergence.

```
a = ld a_ptr
a_f = chkLd(a_ptr_f, a_ptr, a)
a_f_crpt = crpt(a_f)
b = add a, 2
b_f = add a_f_crpt, 2
b_f_crpt = crpt(b_f)
c = cmp b, 0
c_f = cmp b_f_crpt, 0
checkDiverged(c, c_f)
br c, if, else
```

(e) *faulty* instructions are run though the fault injector FlipIt (see Chapter 4) to instrument them for fault injection.

Figure 5.2: Overview of the code transformation for micro-level propagation tracking.

value, the deviation can be calculated (with additional computational cost) along with information on the associated allocation. The latter requires the base address and size of all memory allocations to be logged by

instrumenting memory allocation calls and by inserting the base address and size into a table. Information that relates the *gold* and *faulty* allocations is also logged since it is used to assess macro-level propagation.

Algorithms 5.1 and 5.2 detail the logic of the instrumentation calls for loads and stores, respectively. The function call to `checkInMemory` determines whether an address is contained in a logged memory allocation of the *faulty* application. The information collected inside the function `logDeviation` of Algorithm 5.3 accumulates information into the same variables for both loads and stores; however, in practice a vector of statistics exists for loads and stores. If a store is outside an allocated memory region of the *faulty* application, then the address and value are added into an outside store hash table to allow subsequent loads to read the incorrectly written value and to prevent overwriting of data used by the *gold* application. This is illustrated in Figure 5.3. Conversely, if a load from *faulty* occurs outside an allocated memory region, the load proceeds only if the address does not exist in the outside store hash table. If the address is found in the outside store hash table, the *faulty* address is dereferenced in an effort to generate a segmentation fault, but execution proceeds with the value from the store hash table. This process is illustrated in Figure 5.4.

Algorithm 5.1: Logic for load instrumentation function call.

Input: fptr: Address from *faulty* application.
gptr: Address from *gold* application.
gvalue: Value from the *gold* load.
Result: Value contained at the *faulty* memory address.

```
1  Function chkLd(fptr, gptr, gvalue)
2      in_memory ← checkInMemory(fptr)
3      if not in_memory then
4          in_store_table ← checkInStoreTable(fptr);
5          if in_store_table then
6              fvalue ← readStoreTable(fptr)
7              *fptr                          {Attempt to generate segmentation fault}
8          else
9              fvalue ← *fptr                 {Read memory}
10     else
11         fvalue ← *fptr                     {Read memory}
12     logDeviation(fptr, gptr, fvalue, gvalue, in_memory);
13     return fvalue
```

Once a fault is injected, it propagates to registers and memory locations based on data dependencies. Periodically, these data dependencies flow to a comparison used in a branch. Control flow in the lock-step execution relies on the values from the *gold* application (see Figure 5.2d). Comparisons in *faulty* remain, but the branching instruction is replaced with an instrumentation call to log information about this branching deviation. Due to lock-step style execution, micro-level propagation tracking is unable to accurately resolve control flow divergence. Resolving this requires a coarser view of propagation tracking: macro-propagation

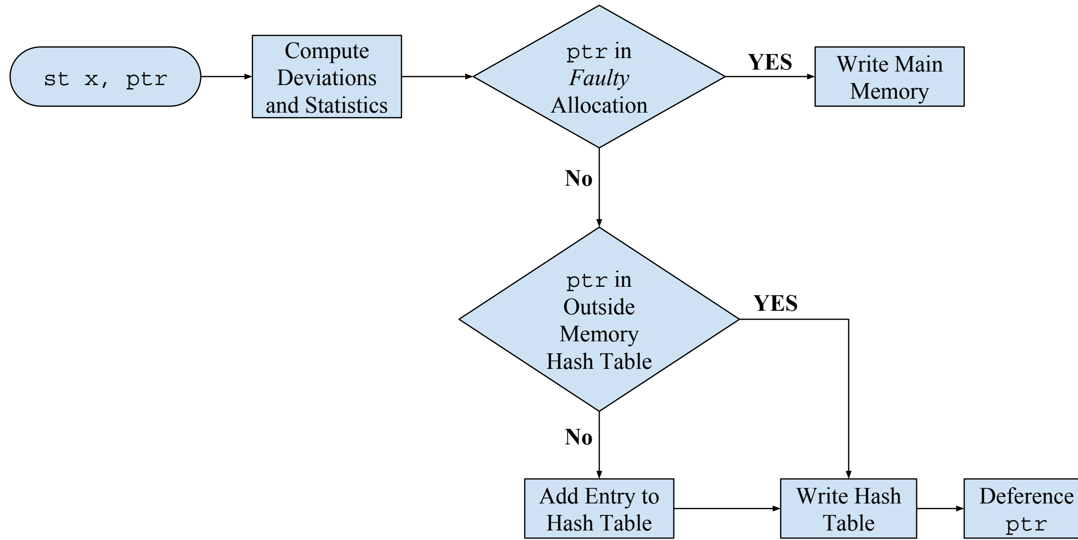


Figure 5.3: Logic of store instrumentation call.

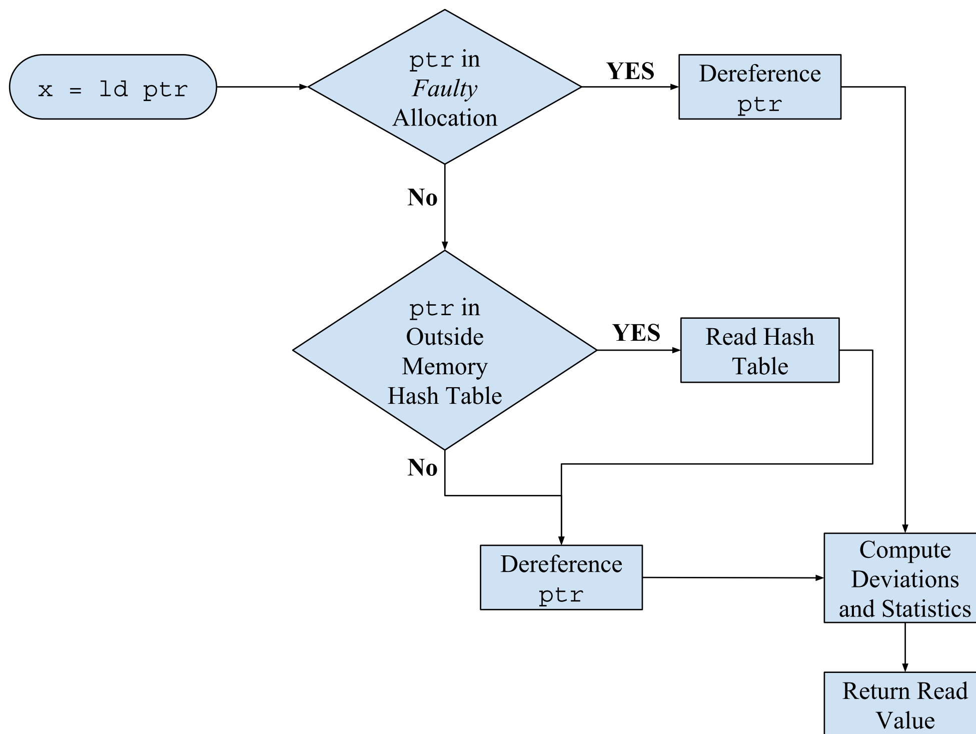


Figure 5.4: Logic of load instrumentation call.

Algorithm 5.2: Logic for store instrumentation function call.

```
   Input: fptr: Address from faulty application.
          fvalue: Value from the faulty application to store.
          gptr: Address from gold application.
          gvalue: Value from the gold application stored.
1  Function chkSt(fptr, fvalue, gptr, gvalue)
2      in_memory ← checkInMemory(fptr)
3      logDeviation(fptr, gptr, fvalue, gvalue, in_memory)
4      if not in_memory then
5          writeStoreTable(fptr, fvalue)        {Write to outside store hash table}
6          *fptr                                {Attempt to generate segmentation fault}
7      else
8          *fptr ← fvalue                       {Write memory}
9      return
```

Algorithm 5.3: Logic for logging information on deviation of load or store at the micro-level.

```
   Input: fptr: Address from faulty application.
          gptr: Address from gold application.
          fvalue: Value from the faulty application.
          gvalue: Value from the gold application.
          in_memory: Boolean. faulty address in memory and not outside store hash table.
1  Function logDeviation(fptr, gptr, fvalue, gvalue, in_memory)
2      dev ← abs(fvalue - gvalue)
3      if dev > tol then
4          if not in_memory then
5              num_access_outside ← num_access_outside + 1
6          max_deviation ← max(max_deviation, dev)
7          num_deviated_access ← num_deviated_access + 1
8      return
```

tracking.

To resolve control flow divergence and to handle regions of code not open to instrumentation — e.g. `MPI_Send`, `MPI_Recv` — function calls are duplicated. For functions that modify global state and yield a different result when called by both the *gold* and *faulty* code — e.g. `rand`, `MPI_Init` — the return value is duplicated allowing both the *gold* and *faulty* codes to proceed with the intended value/action.

For functions open to instrumentation, the LLVM pass creates a new version of that function with an extended interface. The function's argument list is duplicated providing arguments for the *gold* and *faulty* code. In addition, the return type is modified to return a structure containing two elements: the return value for both the *gold* and the *faulty* code. Code contained inside the original function is duplicated and combined as shown in Figure 5.2. *faulty* code depends on and consumes the *faulty* arguments, and the *gold* code depends on and consumes the *gold* arguments.

After the *gold* and *faulty* codes have been merged and all instrumentation is complete, *faulty* instructions

are passed to FlipIt (see Chapter 4) our LLVM based fault injector, to instrument the faulty instructions for fault injection (see Figure 5.2e). For instructions that have been replaced with instrumentation calls, FlipIt is instructed to only inject faults in arguments belonging to the *faulty* code. Arguments coming from *gold* will not suffer fault injection in instrumentation calls. Instructions selected for fault injection are classified into the following categories based on its use in code: floating-point arithmetic (*Arith-FP*), fixed-point integer arithmetic (*Arith-Fix*), pointer and address calculation (*Pointer*), and branching, comparisons, and loop induction variables (*Control*). Latency, in number of instructions, counts dynamically executed LLVM instructions.

5.2.2 Macro-level Propagation Tracking

Macro-level propagation tracking targets the deviation of high-level state variables, data structures, and propagation across process boundaries. To facilitate tracking propagation of corruption in state variables, the base address of all memory allocations along with the allocation's length are stored in a table. During compilation, the LLVM corruption propagation pass inserts an instrumentation call after the memory allocation in the *gold* and *faulty* code that logs both allocations into the table. Furthermore, this instrumentation call creates a relationship between the two allocations allowing for comparison of indices when computing propagation statistics. Algorithm 5.4 details the logic used when comparing two memory allocations. By default, only the percent of elements corrupted, 2-norm, and max-norm between the *gold* and *faulty* memory allocations is logged for post run analysis.

Algorithm 5.4: Logic for comparing two memory allocations and generating propagation statistics.

```
1 for i ← 0 to num_allocs do
2     gbase ← goldAllocAddr[i]                          {Base address of gold allocation}
3     fbase ← faultyAllocAddr[i]                        {Base address of faulty allocation}
4     gsize ← goldAllocSize[i]                          {Number of bytes in gold allocation}
5     fsize ← faultyAllocSize[i]                        {Number of bytes in faulty allocation}
6     numElemDiff ← calcNumDiffElem(gbase, fbase, gsize, fsize, tol)
7     norm2 ← calc2norm(gbase, fbase, gsize, fsize)
8     maxNorm ← calcMaxNorm(gbase, fbase, gsize, fsize)
9     saveResults(fbase, numElemDiff, norm2, maxNorm)   {Log for post run analysis}
```

Comparing the state of all allocated variables is expensive; therefore, comparison points are placed at natural termination points — e.g. end of an iteration. These locations correspond to locations where SDC checks are often placed and checkpointing occurs. Initially, corruption is confined to the process in which the fault occurred. As the application progresses, inter-process communication allows corruption to propagate beyond the process suffering the fault. Understanding how fast this occurs and which processes are likely to

have corrupted data allows for only a subset of processes to recover by checkpoint-restart or a more tailored algorithmic solution.

Unlike micro-level propagation tracking, macro-level propagation tracking is able to resolve divergence in the control flow graph by executing functions to completion once for the *gold* arguments and lastly for the *faulty* arguments. Control flow may diverge inside the function, but after the function returns, the *gold* and *faulty* resolves the control flow divergence. Thus, comparing the states of the *gold* and *faulty* memory allocations is safe and meaningful.

5.2.3 Weak Scaling Performance

Instrumenting code to track propagation at the both the micro and macro-level causes an increase in the number of dynamic instructions — i.e. duplicating the static instructions and the instrumentation calls. Chapter 3 tracks corruption propagation in three applications: Jacobi, CoMD, and HPCCG. In the experiments, a fixed number of processes are used in the parallel executions. Figure 5.5 shows the overhead of instrumentation compared to an unmodified execution. Actual runtimes are shown in Figure 5.6. Each process of Jacobi has 4096 local grid points, CoMD has 2000 atoms, HPCCG has a local problem size of $nx = ny = nz = 13$. For HPCCG and Jacobi, as the process count scales the overhead of instrument decreases; however, for CoMD the overhead is more constant averaging 75× for all process sizes.

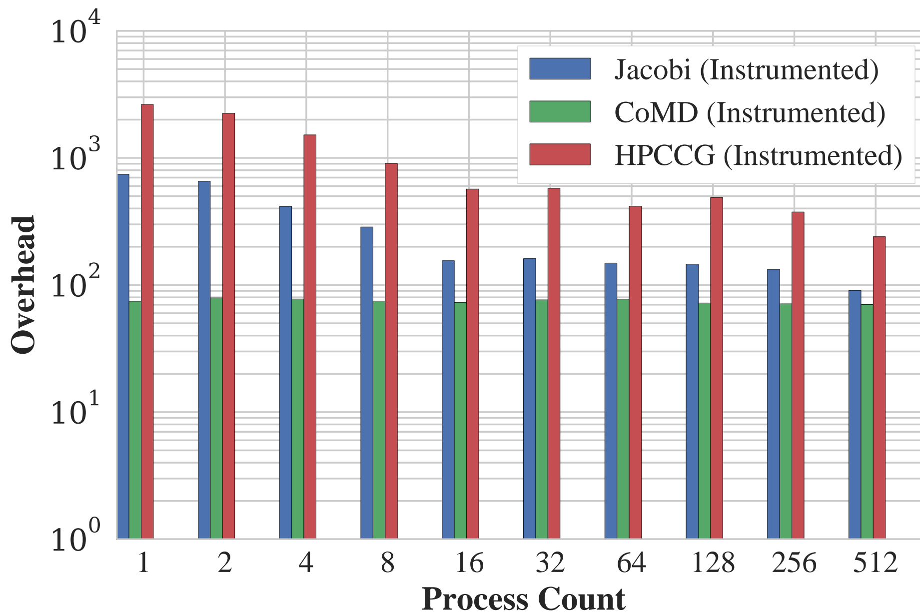
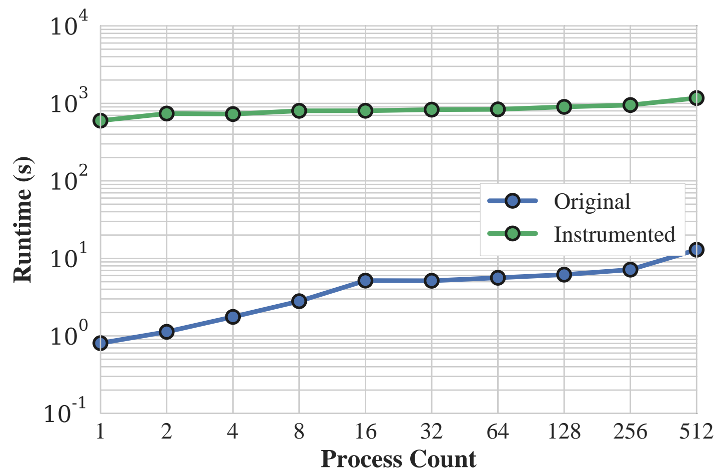


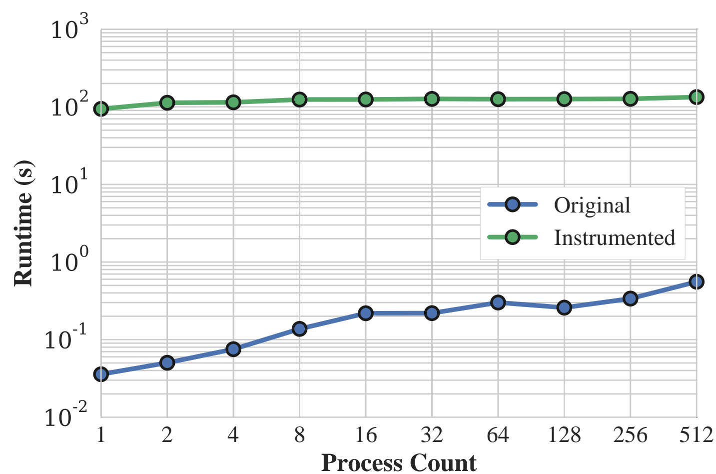
Figure 5.5: Overhead of weak scaling applications instrumented to track corruption propagation.

5.3 Conclusion

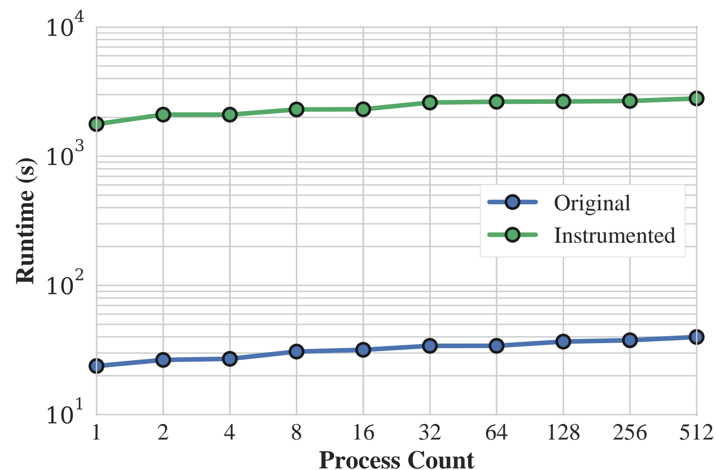
This chapter presents the design of SDCProp, the corruption tracking tool used in Chapter 3. This tool tracks propagation at two levels micro (instruction level) and macro (application variable level). In addition, the scalability and overhead of the tool is explored. The next chapter, Chapter 6, investigates how lossy compression can be effectively used to in HPC checkpoint-restart.



(a) Main computation loop runtime for Jacobi.



(b) Main computation loop runtime for HPCCG.



(c) Main computation loop runtime for CoMD.

Figure 5.6: Weak scaling of applications with and without instrumentation for corruption propagation tracking.

Chapter 6

Lossy Compression

Portions of this chapter are taken from the publication "Exploring the Feasibility of Lossy Compresion for PDE Simulations" [17]

6.1 Introduction

High-performance computing (HPC) applications rely on checkpoint restart to extend execution time beyond the requested allocation time and to tolerate failures in software and in hardware during the simulation run. While the performance and memory of HPC systems continue to increase in size, one potential bottleneck in continued use of checkpoint restart is that file system bandwidth has exhibited only slow growth in current machines. For example, the current 10 petaflop machines Blue Waters at the University of Illinois and Titan at Oak Ridge National Laboratory, and the upcoming 100 petaflop machines Aurora[1] at Argonne National Laboratory and Summit[2] at Oak Ridge National Laboratory, all incorporate 1 TB/s file systems. Since the memory capacity in the 100 petaflop systems are approximately 5–9 times larger than in current 10 petaflop systems, scalable applications are expected to see an increase in checkpoint restart times with a similar factor. However, as HPC systems continue to increase in size and complexity, new design constraints [13, 96] such as the high cost of data movement and comparatively free cost of computation allow for new optimizations.

Exascale systems are expected to put further pressure on the checkpoint system as the mean time between failure (MTBF) is expected to decrease, thus demanding more frequent checkpointing [23]. This aspect has motivated so-called multi-level checkpoint restart [79, 11] schemes, where the memory hierarchy is leveraged to reduce the time to checkpoint and to recover when a failure occurs. The addition of burst buffers [69] to HPC systems offers another approach to reduce time to checkpoint. Yet, multi-level checkpointing and burst buffers do not target a reduction in checkpoint size, which is another major avenue for checkpoint time reduction. Compression of the state variables can be used to reduce the checkpoint size. The relative cost of compression decreases as computation becomes cheaper relative to the cost of data movement. For example,

[1]http://aurora.alcf.anl.gov/
[2]https://www.olcf.ornl.gov/summit/

for processors on HPC systems, a single IEEE-754 fused multiply-add consumes three orders of magnitude less power than a DRAM memory operation [90]. In addition, the cost of data movement increases as the memory hierarchy is traversed from registers to file systems.

Standard compression techniques are often ill suited for floating-point data [99] which has prompted the development of floating-point specific compressors. These are often lossless algorithms [68, 16] and, in most cases, achieve around a 50% reduction in checkpoint size. By switching to lossy compression schemes [33, 60, 67], higher compression ratios can be achieved. Previous works [80, 92, 61] have shown success in restarting from a lossy checkpoint, but have not focused on the relationship between the compression error and the truncation error of the numerical approximations used in the simulation. Establishing this relation is fundamental to guaranteeing correctness for users of numerical simulations.

Various break-even points for system and application level checkpointing have been analyzed for lossless compressors on HPC machines [50]. For lossy compression, the performance benefits are realized if the overall checkpoint time is reduced. Thus, the time to compress plays a key role in analysis. Lossy compression often exhibits improved compression and decompression times in comparison to lossless schemes [60, 33].

This chapter investigates the feasibility of using lossy compression for checkpointing PDE simulations, by interpreting the error introduced through lossy compression as numerical error and by relating it to spatial discretization error and approximation properties in the numerical methods used in simulation. The chapter highlights two important PDE model problems (advection and diffusion) and two production level HPC applications (PlasComCM[3] and Nek5000[4]) that the error introduced by lossy compression at restart does not increase beyond the discretization error, even in the scenario of multiple restarts in the same execution. This underscores the idea that the compression error exhibits little influence on the quality of the final result of the simulation.

Specifically, the contributions are

- expression of lossy compression error as numerical error;
- the relationship between error due to lossy compression with physical properties in the problem;
- a discussion of boundary conditions and the attenuation of compression error;
- the feasibility of lossy compression for checkpoint-restart on kernels and real applications; and
- a performance model that distinguishes the trade-off between efficiency and accuracy in lossy compression.

[3] http://xpacc.illinois.edu/
[4] https://nek5000.mcs.anl.gov/

The rest of this chapter is outlined as follows. Section 6.2 discusses background in the area of compression and HPC checkpoint restart. Section 6.3 outlines the relationship between error introduced through lossy compression and the approximation properties of the discretization. Section 6.4 uses approximation bounds to investigate propagation and reduction of error through several model PDE problems. Section 6.5 gives an overview of two production level applications, PlasComCM and Nek5000, and presents results from using a compression tolerance that is consistent with the numerical properties of the simulation. Section 6.6 uses a performance model to explore efficiency of lossy compression.

6.2 Background

A critical element of lossy compression is the *level* of lossy compression — i.e., compression factor (factor by which the data set size is reduced). Compression factors are dependant on the magnitude of error that the compressor introduces. High compression factors leads to larger errors, but the magnitude of acceptable compression error is problem dependent. A natural approach to determine the acceptability of a compression error is to compare the lossy state with that of an uncompressed checkpoint and verify it passes simulation validation metrics. This trial and error method can be effective, however a more direct route is to measure the compression in comparison to the approximation properties in the application. This chapter explores the applicability of lossy checkpointing from the numerical perspective and provides guidance on the setting of the lossy checkpointing compression error tolerance.

Control of compression error is an important trait of a compressor. Indeed, only a strict control of the compression error allows for the association between lossy compression and the properties of the numerical methods. Lossy compressors for floating point data sets, such as ISABELA [60], NUMARCK [26], ZFP [67], FPZIP [68], SZ [33], allow for varying degrees of control. In this work, SZ is used since the compressor adheres to preset relative and absolute bounds on the tolerance on a per element basis and provides the highest compression factors. In particular, SZ-1.3 is used, although other compressors fit within the scope of this work.

6.3 Linking Compression Error to Numerical Error

This section describes the lossy compressor SZ [33] that is used for all experiments. In addition, it introduces our methodology of selecting lossy compression error tolerances based on numerical truncation error.

6.3.1 Lossy Compressor SZ

To use SZ, the user sets the error bound by selecting the tolerance, ϵ. It is chosen so that the difference between the original value and decompressed value is bounded relatively and/or absolutely by ϵ.

SZ compresses data by predicting value $i+1$ from values at i, $i-1$, and $i-2$ using curve fitting. If the value predicted is within the prescribed tolerance, then the curve fitting function predicting $i+1$ is encoded as a codeword. If the value at $i+1$ is inaccurate using a curve fitting, then an escaped codeword is encoded. The value of $i+1$ is then added to a list of other hard-to-compress data values. This list is compressed by inspecting the binary representations of the data for commonality. SZ has compression routines for single and double precision arrays. This chapter uses the double precision compression routines during all experiments.

6.3.2 Method

Selecting the correct level of lossy compression is often based on trial and error. The quality of a decompressed checkpoint is ultimately application dependent and assessing the quality requires deep knowledge of the underlying physics. To this end, *a priori* bounds on the numerical discretization scheme are useful in quantifying error in specific variables of the simulation. In addition, numerical stability of the numerical method plays an important role.

This chapter exploits the fact that many HPC applications approximate the solution to PDEs or ODEs. The level of accuracy given by the truncation error gives us an upper bound on compression error tolerances. A compression tolerance less than the truncation error produces results that are within a factor of the discretization error, but not bit-reproducible to the uncompressed solution. A compression tolerance greater than truncation error will add more error into the state variables, potentially impacting simulation results.

To understand truncation error consider an approximation of $u(x+h)$ by Taylor Series expansion, a method used in deriving and analyzing numerical methods to solve PDEs and ODEs:

$$u(x+h) = u(x) + u'(x)h + \frac{u''(x)h^2}{2} + \mathcal{O}(h^3) \tag{6.1}$$

This Taylor Series truncates $u(x+h)$ to second order accuracy. Therefore, any error that is less than $\mathcal{O}(h^2)$ results in a numerically equivalent approximation to $u(x+h)$ according to the accuracy specified when truncating the Taylor Series.

The approach used in this chapter is to select a level of lossy compression that results in an error that is less than the spatial discretization error of the problem. Given a spatial mesh size, h_x in 1D, the order of

accuracy of the method is specified as $\mathcal{O}(h_x^p)$, where p is the order of discretization accuracy. A simulation that uses a second-order discretization with $h_x = 0.1$, suggests an error of approximately $e \approx ch_x^2 = c \cdot 0.01$, for some constant c. This implies that a numerical approximation, u^h, and a perturbed numerical approximation $\widetilde{u}^h = u^h + \epsilon$ are close if $\epsilon < \mathcal{O}(h_x^2)$. This motivates the approach taken in this chapter, where the compression tolerance ϵ is selected such that it is less than the simulation's truncation error e^h. For problems that use adaptive mesh refinement, the compression tolerance is selected dynamically just before the application is checkpointed based on the current finest grid resolution, and is the study of future work.

Figure 6.1, details the selection of lossy compression error tolerances. The numerical accuracy and spatial discretization size are used to calculate a bound on the error and to produce a compatible compression error tolerance. Optionally, the error tolerance can be further refined by leveraging application-specific knowledge about the problem such as convergence properties, physical domain, and boundary conditions. After restart from a lossy compressed checkpoint, results will not be bit-reproducible to those from the standard approach, but are within the simulation's numerical accuracy.

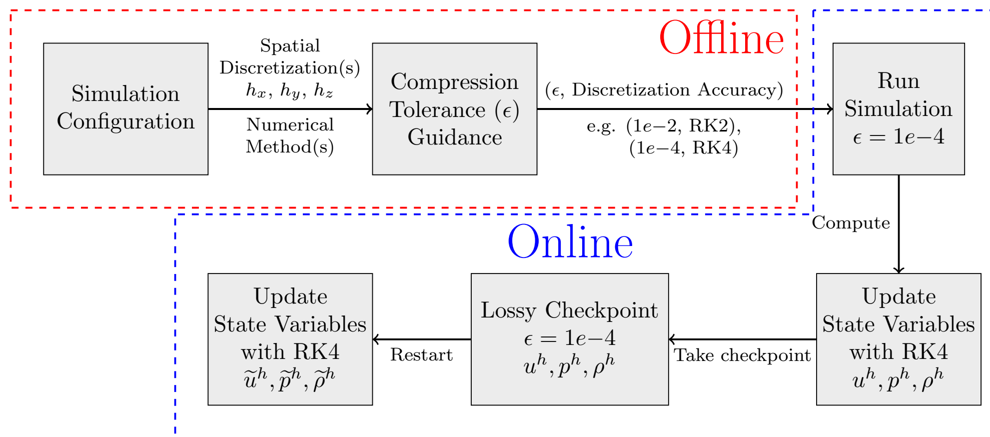


Figure 6.1: Overview of method to select the lossy compression error tolerance.

Our method can also be used to guide the selection of tolerances to allow more error in return for higher compression factors. For example, if ϵ is the compression tolerance for a fourth-order accurate method, it is straightforward to identify a compression tolerance for a second-order method. This new tolerance allows for higher compression factors, but larger errors in the data. This new tolerance is interpreted as a low-order approximation. In fact, similar approaches are used in HPC applications such as Nek5000 to reduce checkpoint size. On restart, Nek5000 bootstraps itself with a low-order method, requiring data from fewer time-steps until it is possible to use a high-order method.

Algorithm 6.1, outlines a generic time-stepping code that utilizes lossy compression during checkpoint restart. The user computes an *a priori* bound on the simulation's truncation error e^h by using the smallest

spatial mesh size and the numerical method's order of accuracy as outlined above. Next the user selects a compression error tolerance, ϵ, less than the truncation error, e^h. The simulation starts and runs as normal until a checkpoint is taken. The state variables u, p, and ρ represent the physical properties velocity, pressure, and density, respectively. These physical properties are fundamental to computational fluid dynamics codes such as PlasComCM and Nek5000. As the simulation progresses the state variables u, p, and ρ are lossy compressed before the checkpoint is written. After the checkpoint is established, computation proceeds as normal until the final time-step. If the simulation requires a restart before reaching the final time-step, then the lossy checkpoint is read and computation proceeds using the decompressed versions of the state variables.

Algorithm 6.1: Outline of generic time-stepping code that using lossy compressed checkpoints.

```
Input: e^h: User estimate of truncation error.
       ϵ: User selected compression error tolerance less than e^h.
       checkpoint_interval: Number of iterations between checkpoints.
1  Function main()
2      if restarting == FALSE then
3          u ← initVelocity()
4          p ← initPressure()
5          ρ ← initDensity()
6      else
7          u ← readDecompressVelocity()
8          p ← readDecompressPressure()
9          ρ ← readDecompressDensity()
10     for t ← 0 to T do
11         u ← updateVelocity()
12         p ← updatePressure()
13         ρ ← updateDensity()
14         if t mod checkpoint_interval then
15             lossyCompressCheckpoint(u, p, ρ, ϵ)
```

6.4 Understanding Compression Error Behavior on 1D Model Problems

To motivate the approach outlined in Section 6.3, two model PDEs are presented: 1D heat and 1D advection equations. These two model PDEs are selected to highlight the impact of physical properties on the selection of lossy compression error tolerances. The behavior of these model problems are reflected in more complex HPC applications, as observed in Section 6.5.

6.4.1 1D Heat

The 1D heat equation finds the temperature solution $u(x, t)$ for all positions x and time t along a rod with fixed length L is given by

$$u_t = u_{xx} + q(x), \quad 0 < x < 1, \tag{6.2}$$

$$u(0, t) = 20, u(L, t) = 50, \quad t > 0 \tag{6.3}$$

$$u(x, 0) = f(x), \quad 0 < x < L, \tag{6.4}$$

where $q(x)$ is a time independent external heat source forcing term along the length of the rod. For this example, Backward Euler with second-order finite differences is used to discretize (6.2) yielding

$$\frac{u_j^{n+1} - u_j^n}{h_t} - \frac{u_{j+1}^{n+1} - 2u_j^{n+1} + u_{j-1}^{n+1}}{h_x^2} = q(x). \tag{6.5}$$

This results in a tri-diagonal linear system that is solved using a direct method at each time-step to simulate the PDE. The numerical accuracy of (6.5) is $\mathcal{O}(h_t; h_x^2)$, first-order accurate in time and second-order accurate in space. Thus, errors in the solution that are much smaller than $\mathcal{O}(h_t; h_x^2)$ will not impact the overall accuracy of the numerical approximation. Since restarting from a lossy compressed checkpoint adds an error into the simulation, this lossy compression error tolerance is selected so that the error is below the truncation error of the numerical scheme. For this problem, $h_x = 0.02$ implies an accuracy of $e^h = 0.02^2 = 4e{-4}$ and a compression tolerance of $\epsilon = 1e{-4}$. Figure 6.2 shows error in the solution of (6.5) with and external heat source:

$$q(x) = \begin{cases} 100 & : x = L/2 \\ 0 & : x \neq L/2 \end{cases}$$

Figure 6.2 shows the error $e = u^h - \widetilde{u}^h$ in restarting the simulation at time $t = 0.6$ and $t = 1.1$ from lossy compressed checkpoints at the preceding time steps of $t = 0.5$ and $t = 1.0$, respectively. Each tick on the y-axis section represents a single time-step in the simulation. The state of the second restart is affected by two lossy compressed checkpoints. A blank (white) region in the figure denotes no error in the compressed numerical solution, $\widetilde{u}^h$, compared to an uncompressed numerical solution u^h. After restart, Figure 6.2 shows that error is distributed throughout the entire domain and is attenuated after a few time-steps. This is expected as this PDE models heat conduction, resulting in smooth solutions and propagation of error through the domain as time advances. In addition, with the selection of a constant temperature

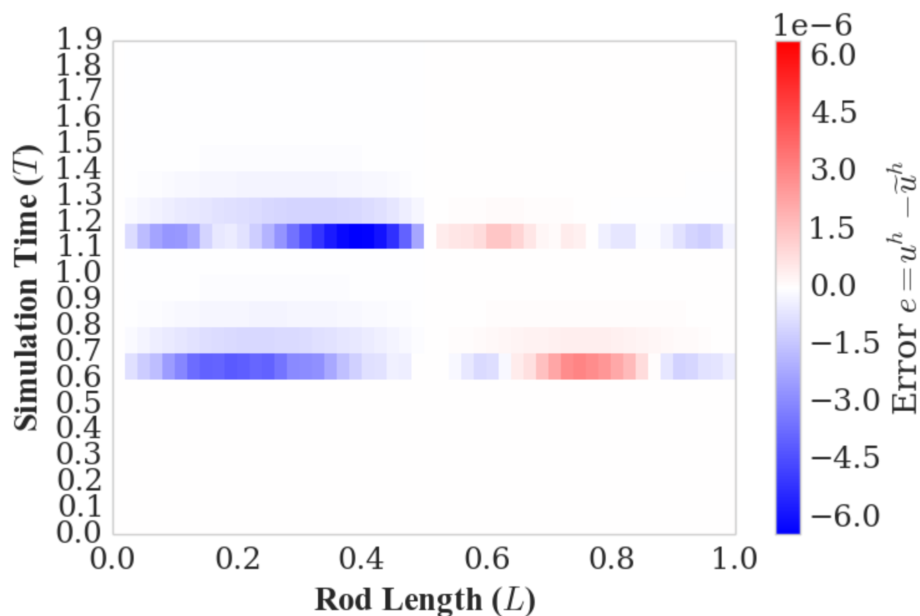


Figure 6.2: Error at each grid-point in a 1D heat equation between numerical solution, u^h, and a numerical solution restarted from lossy checkpoints, $\widetilde{u}^h$, at time $t = 0.6, 1.1$.

boundary conditions the solution will converge to a steady-state solution. This property further accelerates the aforementioned removal of error in the heat equation.

Figure 6.3, underscores the limited impact of lossy compression at a tolerance of $\epsilon = 1e{-}4$, which does not contribute to error above the truncation error of $e^h = 4e{-}4$. Thus, the numerical solution that depends on two restarts from lossy compressed checkpoints, $\widetilde{u}^h$, is equivalent to the original numerical solution, u^h. Although the solution is compressed to a tolerance $\epsilon = 1e{-}4$, the resulting error in $\widetilde{u}^h$ is small due to the smoothness of the solution between any two consecutive points in the domain being easily predicted by the lossy compressor.

6.4.2 1D Advection

A notable aspect of (6.2) is the attenuation of error as time evolves. The advection equation (6.6), does not have such a property. Instead, solutions (and error) propagate following a characteristic in the direction of flow. Any corruption in the solution remains in the wave until the boundary of the domain. In contrast, application of periodic boundaries does not allow error to escape the domain. As a result, error can accumulate with each restart from a lossy compressed checkpoint. To illustrate this point, consider a 1D advection

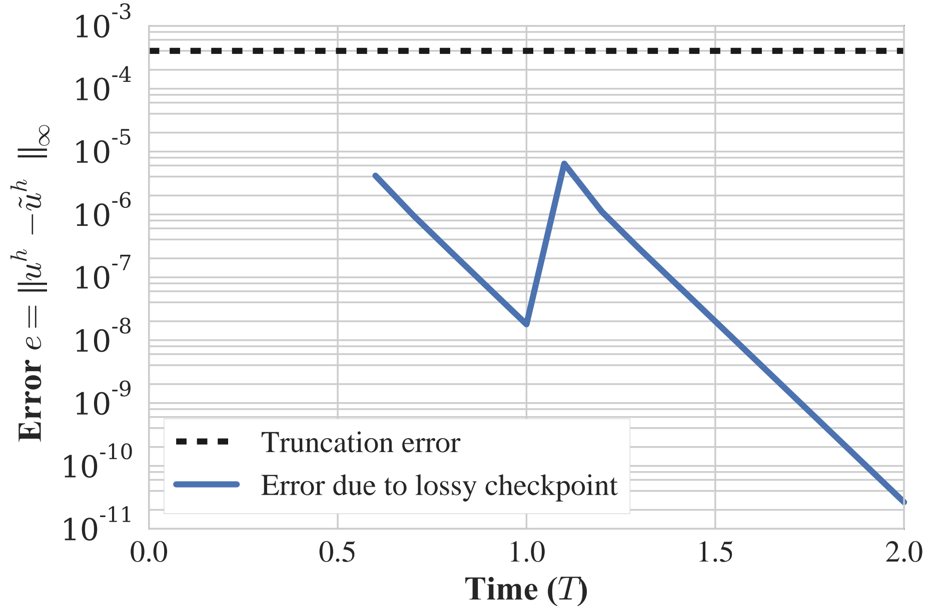


Figure 6.3: Maximum absolute error in the compressed numerical solution, $\widetilde{u}^h$ due to restarting from a lossy compressed checkpoint for the 1D heat equation (6.2).

equation with periodic boundaries:

$$\begin{aligned} u_t &= -u_x, 0 < x < 2\pi \\ u(0,t) &= u(L,t), \quad t > 0 \\ u(x,0) &= f(x), \quad 0 < x < L \end{aligned} \tag{6.6}$$

The equations in (6.6) are solved using Lax-Wendroff, which has a spatial truncation error of $\mathcal{O}(h_x^2)$, and with an initial condition of $u(x,0) = f(x) = \sin(3x)$. In this example, the lossy compression error tolerance is set at $\epsilon = 1e{-}5$ since the spatial discretization of $h_x = 0.006$ suggests accuracy up to $0.006^2 = 3.6e{-}5 = e^h$. The restarts are employed at times $t = 1.25$, 2.5, and 3.75. In Figure 6.4, the error history shows that with each successive restart, the error persists (error lines do not reduce in magnitude between checkpoints), increases (error lines become darker with the number of checkpoints), and is propagated though the domain in the direction of advection (error at each point shifts to the right as time increases). If periodic boundary conditions are not used, then error still exist between the two solutions due to the restart, but only until the component of the error leaves the domain.

Periodic boundary conditions allow for the accumulation of error shown in Figure 6.5. Figure 6.5, shows that the maximum error in the compressed numerical solution $\widetilde{u}^h$ at each time-step is always less than

truncation error for this simulation, even when restarting multiple times. The compression error tolerance is chosen to guarantee this by staying below the truncation error of the method. This is done heuristically in a small training run by observing the max-norm over the first few time-steps after restarting and selecting a tolerance, ϵ, that yields a max-norm roughly an order of magnitude less than the truncation error, e^h, to allow for unforeseen accumulation. This can be repeated in quick secession to gauge the impact of multiple restarts.

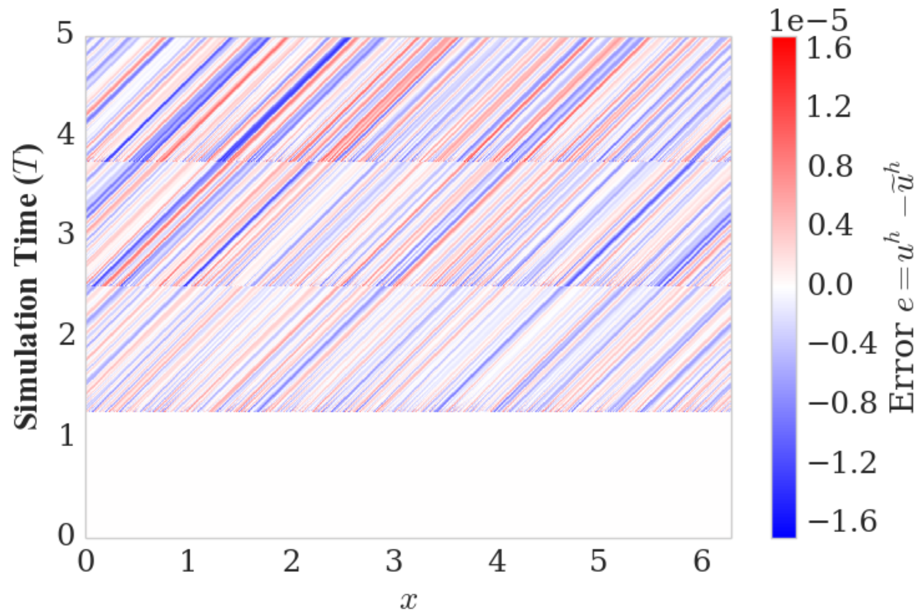


Figure 6.4: Error at each grid-point in a 1D advection equation between a normal numerical solution, u^h, and a numerical solution restarted from lossy checkpoints, $\widetilde{u}^h$, at times $t = 1.25$, 2.5, and 3.75.

6.5 Lossy Compressing Production Applications

This section applies the approach of selecting lossy compression error tolerances from Section 6.3 to two production level HPC applications: PlasComCM and Nek5000. Results are collected on Blue Waters, a Cray machine managed by the National Center for Supercomputing Applications. This chapter uses the XE6 nodes on the machine, which are equipped with 64GB of memory and two AMD Interlagos CPUs per node. The checkpointing routines of PlasComCM and Nek5000 are modified to lossy compress all state variables just before they are written to disk. Each state variable is compressed with SZ-1.3 [33] using relative and absolute error bounds. In production runs of both applications, checkpoints are taken every

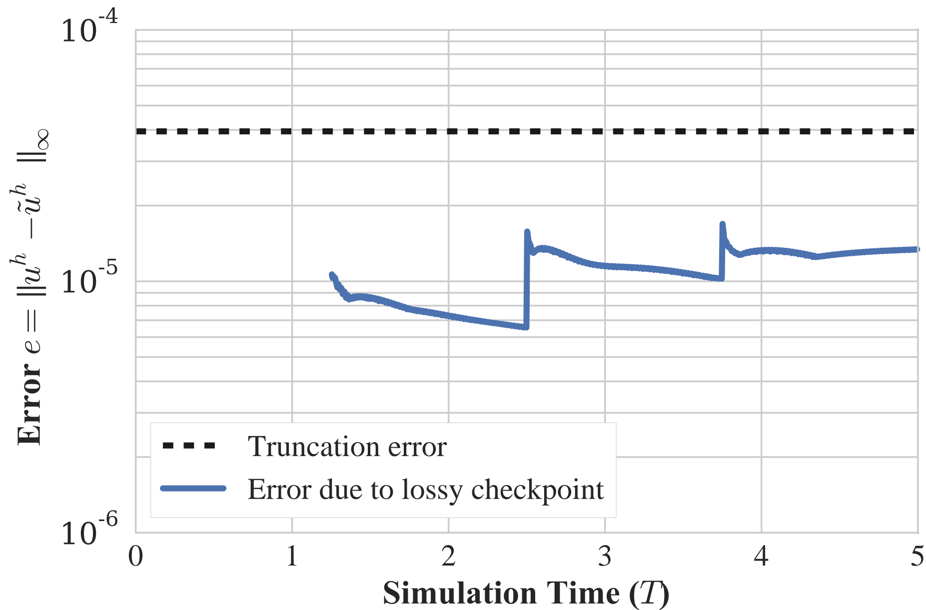


Figure 6.5: Maximum absolute error in a compressed numerical solution, $\widetilde{u}^h$ due to restarting from lossy compressed checkpoints for the 1D advection equation (6.6). Restarts from a lossy checkpoint occur at times $t = 1.25$, 2.5, and 3.75.

1,000–3,000 time-steps which equates to around 1–10% of simulation time for the data sets used during testing. Simulations with higher execution times exhibit a larger portion of time spent in checkpointing.

6.5.1 PlasComCM

The first example uses PlasComCM[5], a multi-physics plasma combustion code. The core functionality of the code targets the incompressible Navier-Stokes equations. Production runs checkpoint 1MB – 1GB per process. Here, the 2D homogeneous Euler equations are used as a model problem. Inside the domain, there is a fixed cylinder object obstructing the flow (see Figure 6.7). An overset mesh, Figure 6.6, is placed around the cylinder (white circle inside mesh) to help resolve the obstructed flow. This problem is run with 4 processes each checkpointing about 1MB per process. The momentum solution of this problem after evolving for 60,020 time-steps is shown in Figure 6.7.

The checkpoint file in PlasComCM consists of four state variables: density, x-momenta, y-momenta, and energy. Figure 6.8 shows a large difference in the compression factor depending on the selected compression error tolerance ranging from 104× for $\epsilon = 1e{-1}$ to 2.2× for $\epsilon = 1e{-10}$. Compression factors for small error tolerances are consistent with reported lossless compression factors [99]. As the compression error tolerances

[5] `https://bitbucket.org/xpacc-dev/plascomcm`

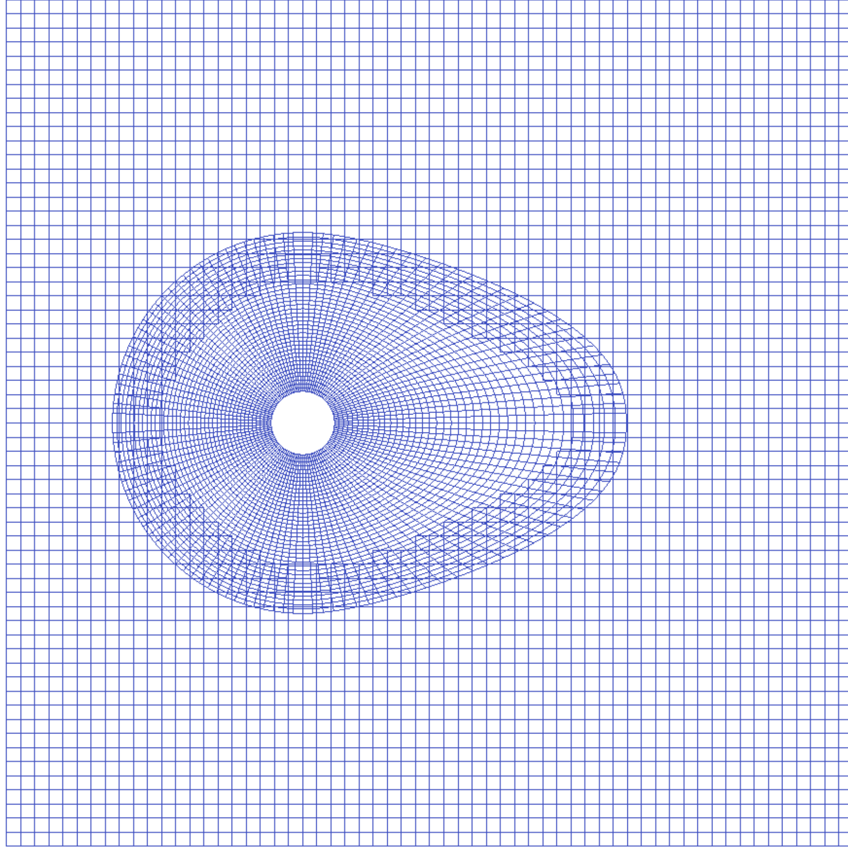


Figure 6.6: Overset mesh in PlasComCM adds a curvilinear mesh around cylindrical object (white circle).

decrease there is a corresponding increase in compression time as shown in Figure 6.9.

The 2D flow problem is solved using fourth-order Runge-Kutta with $h_x = 0.065$ and $h_y = 0.065$. Applying the approach from Section 6.3, accuracy up to $e^h = 1.8e{-5}$ is assumed in both x and y. Thus, a compression error tolerance of $\epsilon < 1.8e{-5}$ is required such that error added due to compression into this does not impact simulation results. The tolerance $\epsilon = 1e{-6}$ is selected to allow for accumulation of error due to multiple restarts. This tolerance yields an average compression factor of $7\times$.

To study the propagation of error in the simulation, 75,000 time-steps are used to reach a fully developed flow. To simulate either a fail-stop failure or exceeding a time allocation at time-step 15,000, 30,000, and 45,000, the simulation restarts from a lossy compressed checkpoint. Figure 6.11 plots the max-norm between the numerical solution u^h and the compressed numerical solution $\widetilde{u}^h$. With each checkpoint there is an increase in error in the system, but the error from lossy compression and advancing the simulation from

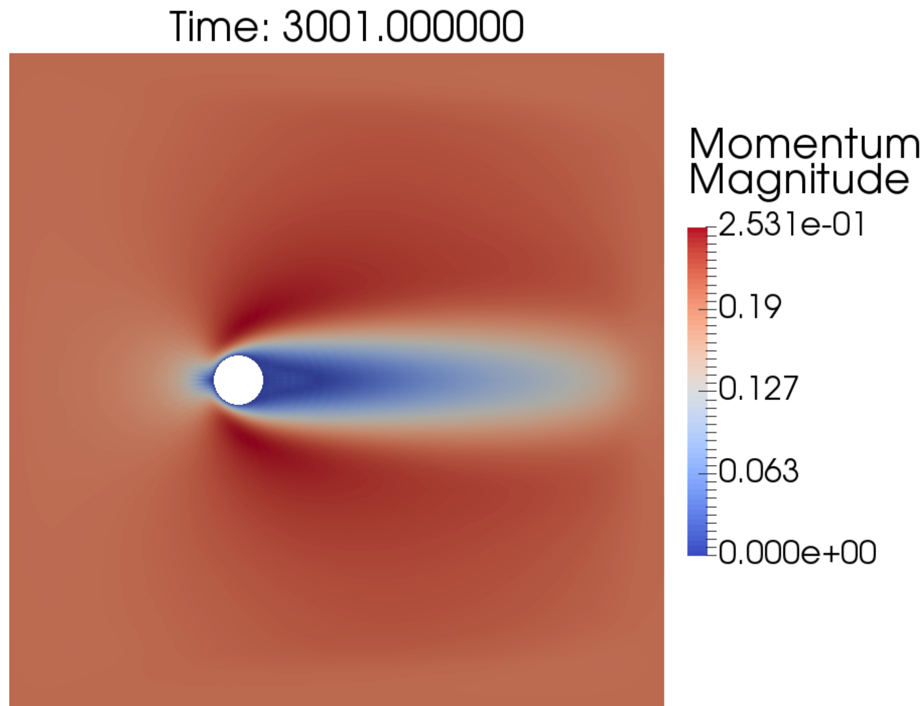


Figure 6.7: Momentum magnitude after 60,020 time-steps for PlasComCM. The fixed cylinder object (white circle) obstructs the flow causing momentum to slow around the object in the direction of flow (to the right).

lossy state is always less than truncation error.

Between the first and second restart (time-steps 15,000–30,000), there is a reduction in the error of all state variables. This is due to the selection of non-periodic boundary conditions, which allow the error to flow out of the domain. However, between the second and third restart (time-steps 30,000–45,000) the magnitude of error increases due to concentration of error in regions of the domain with low momentum — i.e., to the left of the cylindrical object. After the third restart, the simulation evolves another 30,000 time-steps in order to allow the error to accumulate. Over these time-steps the error remains near the compression tolerance of $\epsilon = 1e{-6}$, and there is a slow reduction as error moves from regions of low momentum to regions of higher momentum before exiting the domain.

The restart frequency used in this problem corresponds to a system MTBF of approximately 18 minutes. This MTBF is a much lower in comparison to current systems and expected exascale systems, but does highlight the nature of error propagation/accumulation for this problem. If the frequency of restarting from

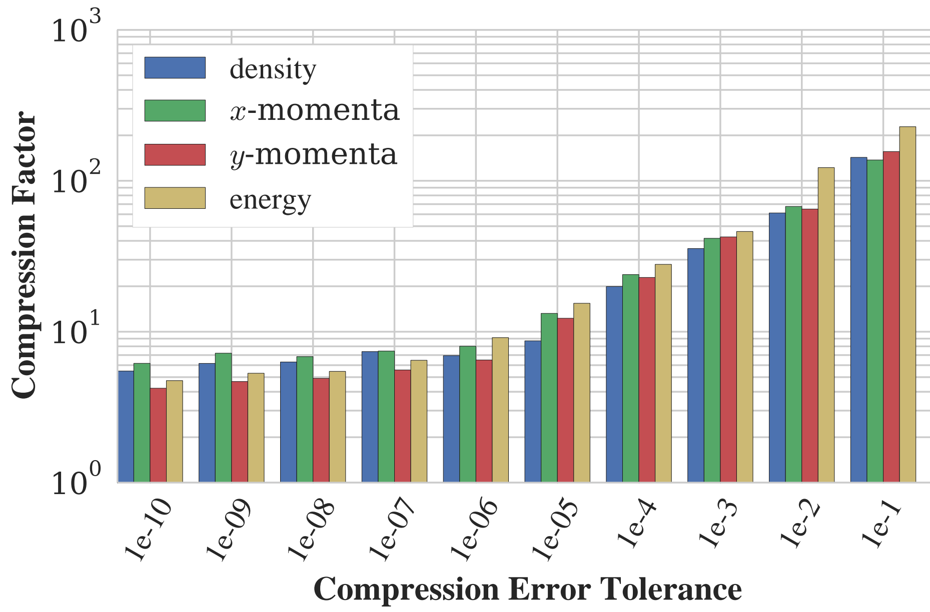


Figure 6.8: Compression factors for PlasComCM.

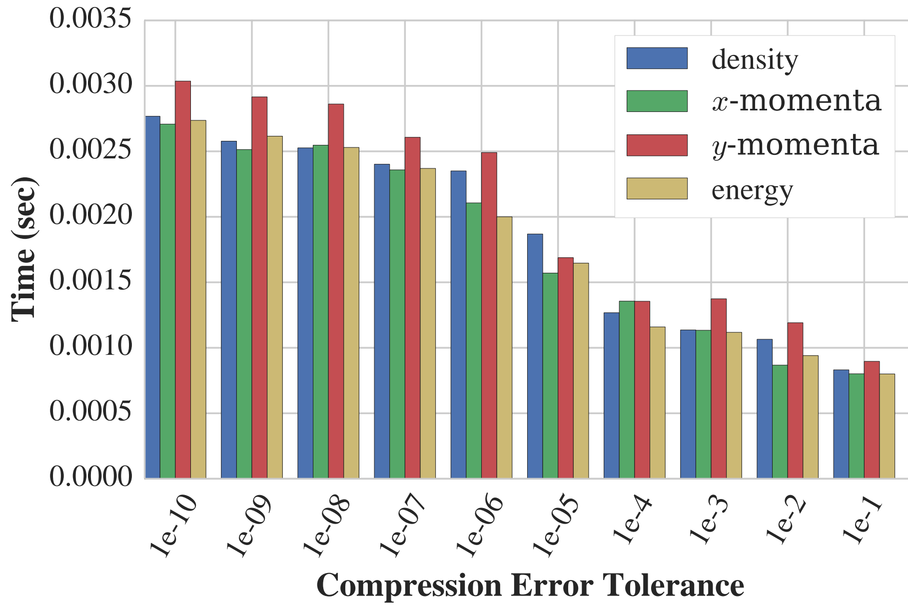
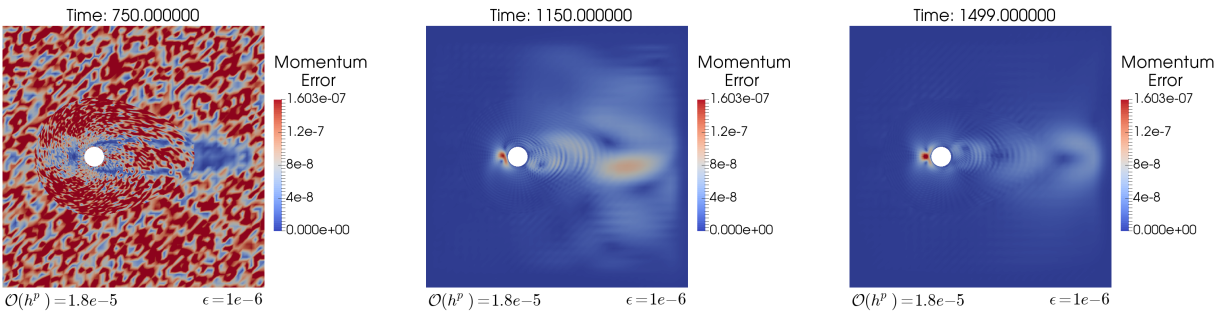


Figure 6.9: Compression time for PlasComCM.

lossy checkpoints is high, then error accumulation above the level of truncation error can occur. In this case, a smaller compression tolerance ϵ should be used minimizing the risk of exceeding the level of truncation error.



(a) Time-step 15000: Distribution of compression error due to lossy restart. Overset grid is visible in error.

(b) Time-step 23000: Error propagates though domain in direction of flow (to the right).

(c) Time-step 29980: Error accumulates in regions of low momentum, Figure 6.7

Figure 6.10: Propagation of momentum error in PlasComCM.

Visualizing the error, Figure 6.10, in the simulation highlights two key properties about this problem's handling of error. First, the domain has non-periodic boundary conditions. This allows flow that is slightly perturbed to leave the simulation as time evolves reducing the amount of error. Any error that accumulates above or below the cylindrical object is removed from the domain as its path out of the domain is unobstructed. Second, in the same momentum error plots, Figure 6.10, a large spike in error is observed near the leftmost part of the fixed cylindrical object. Accumulation of error at this point is due to the near-zero momentum (cf. Figure 6.7). With almost no momentum to move this error though the domain, it becomes concentrated and slowly increases in magnitude as time evolves. These domain specific properties of error propagation suggests that lossy compression routines can improve by adjusting compression error tolerances to the physical properties of the domain that can accumulate or propagate error.

6.5.2 Nek5000

Nek5000[6] is a spectral element code developed at Argonne National Laboratory designed to simulate a variety of problem types such as heat transfer, unsteady incompressible Navier-Stokes flow, and incompressible magnetohydrodynamics. Production runs checkpoint 0.1MB – 2MB per process.

The test problem considered in this example is a 3D Navier-Stokes simulation of a channel flow with periodic boundary conditions. Figure 6.12 shows the solution of velocity at 30,000 time-steps. The simulation restarts twice during execution at time-step 1,500 and time-step 11,500 (MTBF of 50 min.). This test problem is run with 16 processes each checkpointing about 1 MB of data.

Compression factors for Nek5000, shown in Figure 6.13, are as high as 280× for the x-component of

[6]https://github.com/Nek5000/Nek5000

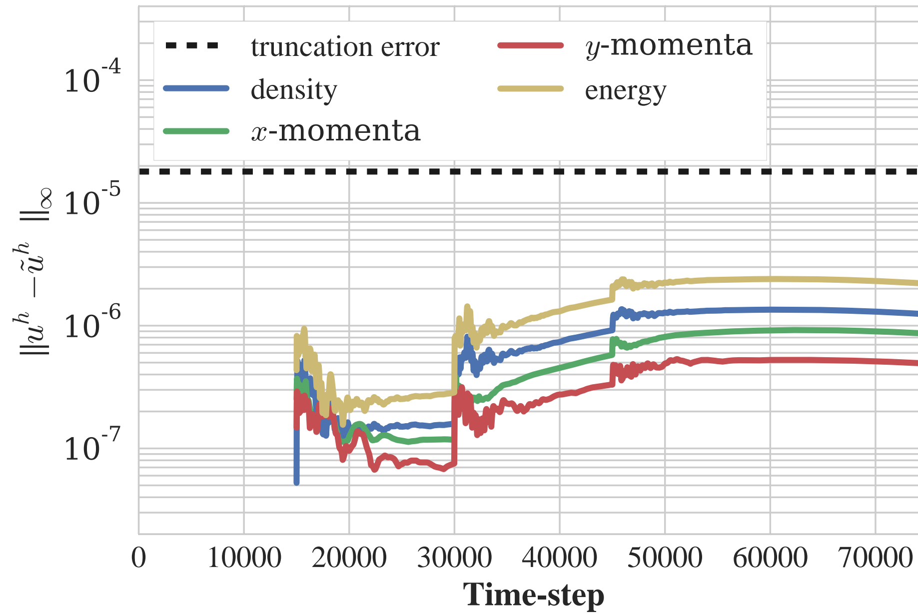


Figure 6.11: Max-norm between numerical solution u^h and compressed numerical solution $\tilde{u}^h$ for PlasComCM.

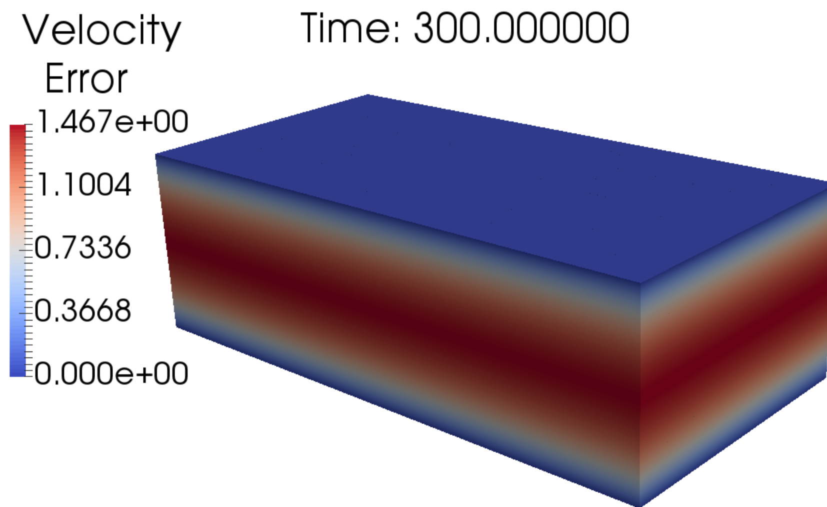


Figure 6.12: Magnitude of velocity after 30,000 time-steps for Nek5000.

velocity with compression tolerance $\epsilon = 1e{-}1$. As the compression error tolerance decreases, compression factors approach $2\times$ for all variables. The discrepancy in compression factors between the x-component of velocity and the other variables is due to the ability of SZ to more accurately predict the values. SZ adopts curve-fitting to predict future values in the array. If a section of data is easily represented with a curve-fitting predictor, then compression factors are high. If the data is not easily predicted by curve-fitting, SZ utilizes binary analysis on the hard to compress regions. However, binary analysis is less effective at reducing data size in comparison to curve-fitting. This also suggests that compressors more directly designed for physical properties may yield improved compression factors.

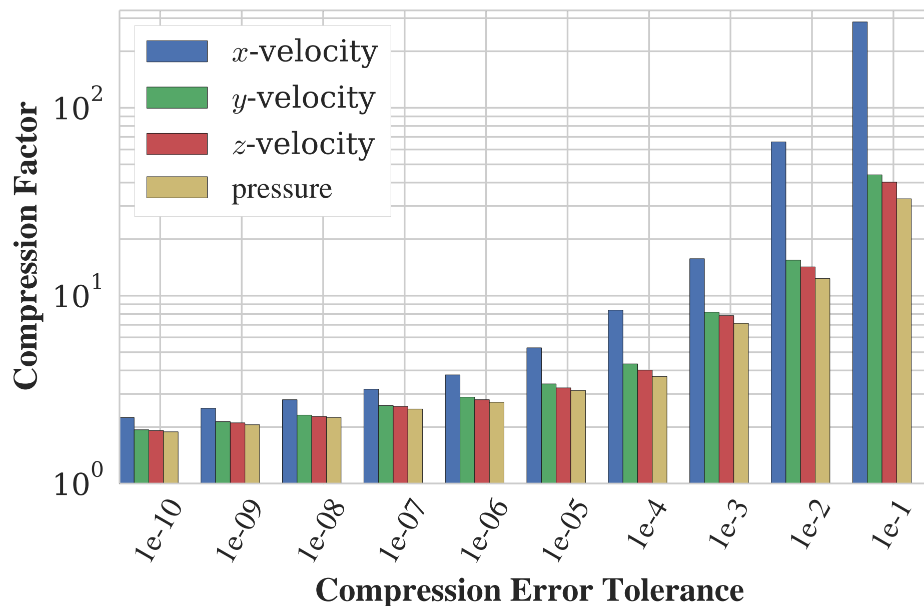


Figure 6.13: Smaller compression tolerances lead to higher compression factors for Nek5000.

Compression time of the Nek5000 state variables, shown in Figure 6.14, fall within two regimes that mirror the compression factor results. Variables that are easier to compress and exhibit large compression factors take less time than difficult-to-compress variables. Furthermore, selection of the lossy compressor can greatly influence the time to compress and compression factor. This highlights a need for further study into development of specialized compressors that are tailored to certain state variables in PDEs.

The 3D channel flow problem solves two linear systems at each time-step: one for pressure and one for velocity. The linear systems for pressure and velocity are solved using GMRES to a tolerance of $e^h = 1e{-}5$. The accuracy of the linear system solve suggests compression tolerances $\epsilon < 1e{-}5$ are required, due to periodic boundary conditions that do not permit error to escape the domain the compression tolerance is

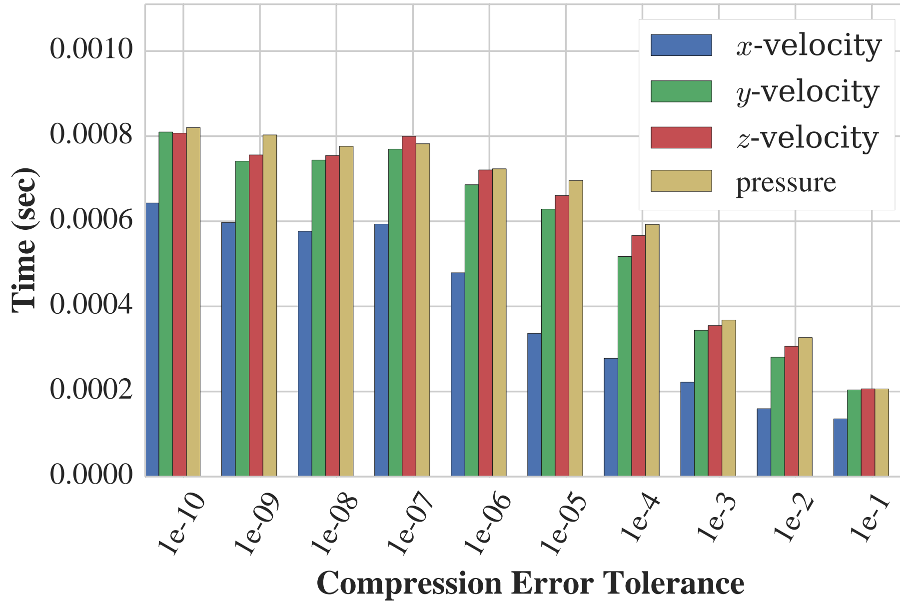


Figure 6.14: Time to compress for Nek5000 increases as the error tolerance decreases. Easy to compress data compresses faster, Figure 6.13.

set at $\epsilon = 1e{-}7$, allowing for an increase in error due to multiple restarts.

Figure 6.15, shows the maximum error in the pressure along with the directional components of velocity between the numerical solution u^h and a compressed numerical solution $\widetilde{u}^h$. After restarting from a lossy checkpoint, there is an initial spike in error in the time-steps immediately following the restart, but it remains below discretization error. As time evolves, the error reduces as it migrates though the domain and approaches a steady state (due to boundary conditions). After the second restart, at time-step 11,500, there is another spike in error due to restarting from lossy state. As time evolves the error propagates though the domain, but remains less than discretization error.

Figure 6.16 shows the spatial location of error for the velocity solution in Figure 6.12 at time-step 30,000. Regions of large error correspond to regions of high velocity. The initial distribution of error after a lossy restart, Figure 6.16a, illustrates processor boundaries and data distribution. Even though ghost regions on other processors are needed to advance the simulation, SZ's compression algorithm only considers local data when compressing. This results in faster compression times, but results in compression artifacts around processor boundaries. Compression artifacts are magnified when the per process distribution of data is not spatially contiguous. Values that appear to be neighbors locally due to how data is stored may in fact be distant globally leading to compression artifacts and lower compression rates.

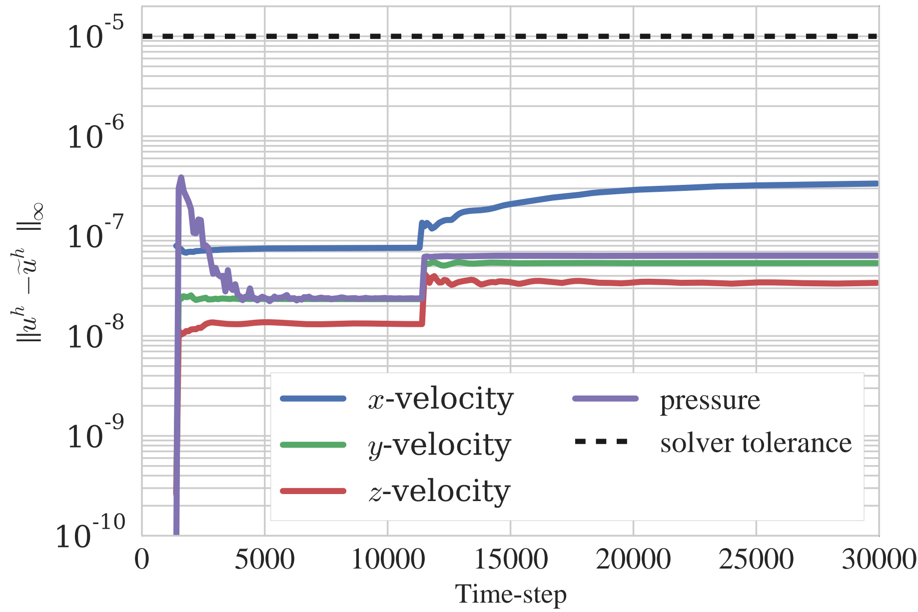
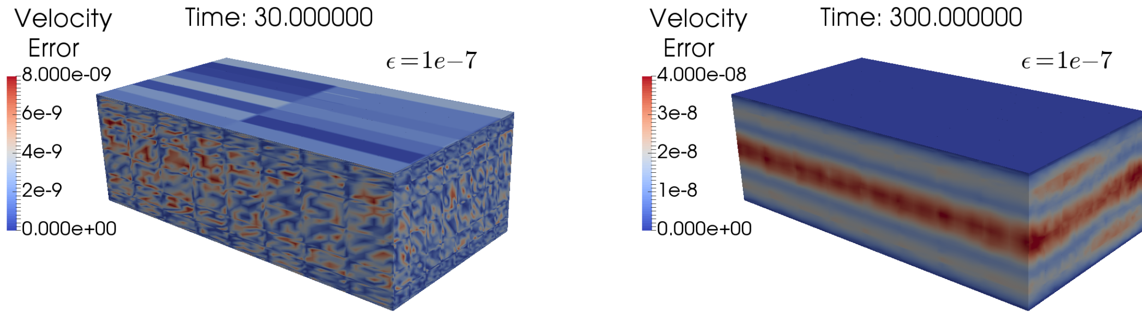


Figure 6.15: Max-norm between numerical solution u^h and compressed numerical solution $\tilde{u}^h$ for pressure and velocity for Nek5000. There is no error before time-step 1,500 as the simulation is not yet restarted.



(a) Time-step 1,500: Distribution of compression error due to lossy restart. Processor boundaries are visible.

(b) Time-step 30,000: Error in velocity magnitude for Nek5000 from Figure 6.12.

Figure 6.16: Propagation of velocity error in Nek5000.

6.6 Modeling Time to Checkpoint

The previous sections show that leveraging the truncation error of HPC applications is an effective method of selecting a lossy compression error tolerance when compressing checkpoint files. Ultimately, however, lossy compression is only used for checkpointing if it results in a more efficient checkpoint. To explore several trade-offs for lossy compression, the time to checkpoint is modeled on a machine similar to Blue Waters. A

simple model for time to checkpoint, T, is given by

$$T = c_b \cdot V + \frac{w_b \cdot V}{f}, \quad (6.7)$$

where c_b is the bandwidth of the compressor, V is the number of bytes being written per process, w_b is the file system bandwidth, and f is the compression factor of the compressor. For the case without lossy compression, $c_b = 0$ and $f = 1$. For lossy compression to be useful, the time to checkpoint using lossy compression, T_{lossy}, needs to be less than the time to checkpoint without compression, T_{reg}: $T_{lossy} < T_{reg}$. This simplifies to

$$f > \frac{w_b}{w_b - c_b}, \quad (6.8)$$

which is used to estimate the minimum compression factor for lossy compression to be viable. This model does not take into consideration the complex nature of the parallel I/O system. Despite this limitation however, the model is useful to estimate when compression improves performance. A more refined I/O model [52] that considered the parallel data exchanges in collective I/O reads/writes is parametrized for Blue Waters.

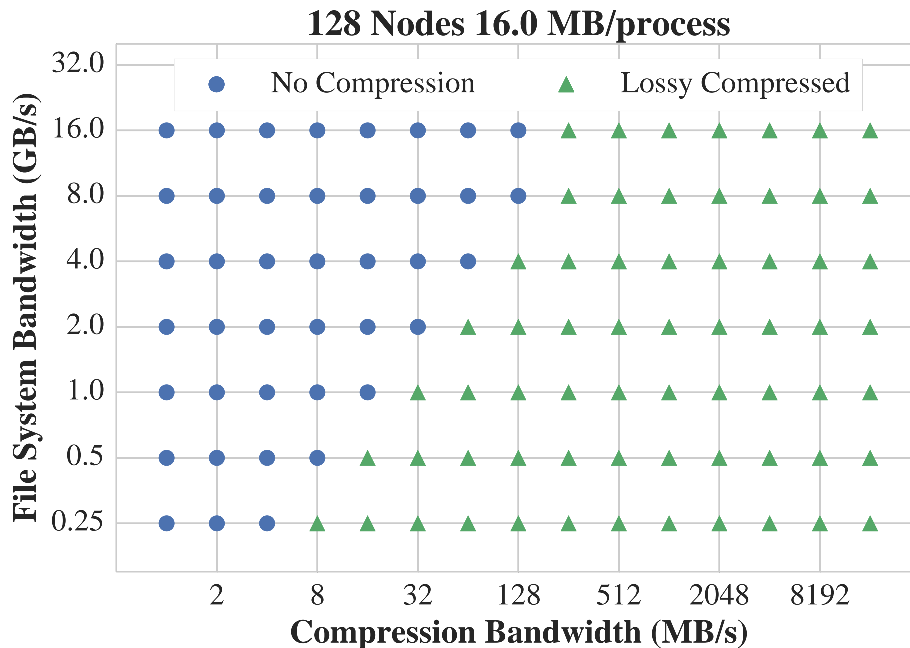


Figure 6.17: Fastest I/O configuration with various file system and compression bandwidths.

Figure 6.17 explores a combination of compression bandwidths and file system bandwidth to see if using lossy compression can improve time to checkpoint for a parallel job consisting of 2048 processes on 128

nodes. Each process checkpoints 16 MB of data. From Figure 6.17, lossy compression becomes increasingly attractive as compression bandwidth increases relative to I/O bandwidth. Practice shows effective I/O bandwidth is much less than peak [73], while compression can reach memory bandwidth. For example, a PlasComCM runs of this size achieves an average aggergate file system bandwidth of around 2 GB/s on Blue Waters. A compression bandwidth of 64 MB/s is enough to improve the time to checkpoint. The measured compression bandwidth for PlasComCM[7] is 78 MB/s. For PlasComCM, lossy compression can improve checkpointing time by 1.8× compared to standard checkpointing. Using Amdahl's law with an I/O fraction of 10%, the improvement in overall execution time is 4.7%. Because many HPC applications achieve a low I/O bandwidth, even modest compression bandwidths make lossy compression an attractive approach to improve the time to checkpoint.

6.7 Discussion

The previous sections explore the feasibility of setting lossy compressor error tolerances based on the numerical accuracy of the simulation. Results show that a compression error relative to the numerical error results in minimal impact on the simulation. Yet, there are several directions of development that would improve lossy checkpointing and increase its utility in practice.

Error Accumualtion: Error propagation after a restart and the frequency of restarts complicate the process of selecting a compression tolerance. Further analysis on the propagation of error due to lossy compression and quantifying the compression error is needed. In addition, the impact of multiple restarts should be explored.

Physics Based Compression: Simulations often rely on the conservation of key quantities — e.g. mass, energy — require symmetries in the domain, or compute statistics on state variables. Development of compressors that are designed specifically for certain physical properties could ensure key properties are satisfied.

Adaptive Compression: This chapter uses one global compression error tolerance for all checkpoints and variables. Using different tolerances spatially and temporally for all state variables will allow new optimizations that reduce checkpoint size and mitigate the effect of error accumulation.

Lower Precision Computation: This work considers computation on double precision data. If computation is performed on lower precision, single or half precision, lossy compression can still be employed,

[7]Data compressed to $\epsilon = 1e{-6}$ with a single threaded unoptimized SZ-1.3 using relative and absolute error bounds.

but requires more from the lossy compressor in terms of compression factor and/or compression time. Using single or half precision reduces the data set size by $2\times$ or $4\times$ compared to double precision. For lossy compressors to reduce data set size in these scenarios, it will need to achieve compression factors larger than $2\times$ or $4\times$ on the double precision data.

6.8 Conclusion

This chapter investigates how to use numerical truncation error to set the compression error tolerance of lossy compression schemes used during checking. The effectiveness of this approach is shown on two 1D model problems and two production level applications. Modeling checkpointing time highlights system and application properties where lossy compression improves the time to checkpoint. The next chapter presents prior research that is related to the Chapters 2–6 and topics in this dissertation.

Chapter 7

Related Work

7.1 Resilience in Algebraic Multigrid

The resiliency of AMG to SDC has been studied previously [78], where checksums are used to detect SDCs. The approach is limited to dense matrices, but is robust, incorporating checks for matrix-vector multiplications, relaxation, and interpolation.

More recent work [25] reports that AMG is resilient to SDC due to its iterative and multi-level nature. Provided that AMG does not experience a segmentation fault, a SDC with high probability emerges as an error in the solution vector. This error is subsequently removed at the cost of more work. Moreover, the error is reduced on a coarser level, where the error is more pronounced. The recovery scheme addresses a segmentation faults by applying triple modular redundancy (TMR) [74] to key pointers and ranks the three instances when accessed, accepting the pointer with the most credibility. This is used in each access to the protected arrays, therefore yielding resiliency at a large overhead. Regardless, the approach decreases the number of segmentation faults for AMG in a faulty environment; on the other hand the resiliency overhead is present even in the case when AMG is not exposed to faults. In contrast, the recovery scheme presented in Chapter 2 induces less overhead, even when AMG does not suffer a segmentation fault. In addition, the scheme provides checks that alert the AMG solver of the presence of SDC and attempts to avoid the impact of the fault instead of allowing AMG to remove the error through more iterations or a full restart.

Recent works have analyzed the multigrid algorithm and have concluded that it is not fully fault resilient [3, 2]. To protect the solver, they establish theoretically and experimentally that a protected Prolongation operator yields a more robust method. Other work uses a fail-stop fault model [49] to explore forward and backward recovery strategies.

7.2 Error Propagation

Redundancy: Redundancy is a common tool in detection schemes. Triple modular redundancy (TMR) [74] ensures correctness by computing values through triplicating computations and majority voting on results. Redundant threads or processes have been used to detect transient and permanent errors in hardware and software [39, 82]. Redundancy at these levels often yield high overheads, but also high protection from soft errors. The GangES error simulator [93] seeks to limit the time required for fault injection experiments on applications running on micro-architectural simulators by checking for similarities in application state to prior fault injection runs. GangES does not consider parallel applications.

Tools: Static analysis modeling corruption propagation combined with fault injection experiments is extensively used in compiler based protection schemes to identify key instructions to protect. IPAS [59] leverages machine learning on fault injection experiments to learn which instructions are good candidates for replication to prevent program output corruption. However, this approach can be limited by a dependence on training. Compiler analysis with profiling [56] can be used to improve the effectiveness of an instruction replication scheme; however the long term impact of corruption is not considered, such as between processes. SDCTune [72] uses static analysis, heuristics, and fault injection data to construct a probability model for corruption to decide what instructions are likely to generate SDC. Extensions of the Program Vulnerability Factor [37] metric have been used to distinguish between crashes and SDC, thus allowing for more accurate estimates of SDC and crash rates. The modeling of corruption propagation is again limited to sequential programs. CrashFinder [64] explores the effectiveness of combining compiler analysis and fault injection results to identify instructions that generate long latency crashes due to pointer, loop, and state corruption; the propagation of corruption is not investigated. SymPLFIED [85] uses symbolic execution to abstract the state of corrupted values in the program to find hard-to-detect fault injection locations that random fault-injection may not target; their work is limited to sequential integer programs.

Characterization: A study of the numerical impact of a single bit-flip on matrix-vector computations and GMRES is presented in [36, 35], respectively. An analysis of long latency crashes [104], shows corruptions in memory have long fault activation times leading to long latencies for crashes compared to corruption in registers, but does not investigate the extent of corruption propagation at the time of detection and the ability to recover locally. The impact of soft errors on the Linux kernel is studied [40], but the corruption of system state variables to determine how to prevent propagation for long latency crashes is not quantified. Y-branches [103] explores how control flow divergence impacts application correctness and performance. Control flow re-converges if the architectural state exactly matches a golden copy; however, for applications that can remove or mask corruption over-time, this may underestimate the number Y-branches.

Characterizations of corruption in floating-point computation inside HPC applications is presented in [4]. Corruption is tracked based on terms of number of incorrect memory accesses and does not relate this to application level variables. A similar study [65], quantifies corruption in GPU and host memories for GPU benchmarks, but does not look at how corruption propagation in a distributed memory environment.

Recovery: Establishing a bound on the subset of total program state that is corrupted allows for localized recovery. Containment domains [29] surround code regions, using verification to ensure correctness. Identifying the location of code containment remains open. Similarly, transactional semantics [47] has been used for MPI state, but not on application variables that may be corrupted. Esoftcheck [106], uses compiler analysis to remove redundant SDC detectors to maintain high reliability, but does not consider the latency of detection and how it effects propagation. An analytic version of this problem which investigates optimal placement of detectors of different capabilities to verify a checkpoint is corruption free is presented in [8], but considers a fixed recovery time that does not change based on the level of state corruption.

7.3 Fault Injection

Many fault injectors have been created that operate in real time and on real hardware. DOCTOR [44] injects faults into memory, the CPU, and network communications by using time-outs and traps to overwrite memory locations and modify the binary. XCEPTION [24] is an exception handler that injects faults when triggered by accesses to specific memory addresses, and simulates stuck-at-zero, stuck-at-one, and bit-flips. NFTAPE [102] uses a driver based fault injection scheme to inject fault inside the user or kernel space, but requires OS modification.

In compiler based fault injection, hard errors are simulated by adding extra instructions that always inject in the same location. To address the static nature of these injections, the injections are made dynamic by addition of code that corrupts data at runtime based on programmatic and environmental factors.

Fault injection for MPI applications is often focused on *message* injection [39]. Yet other works consider soft errors that manifest in register modifications [25] or that arise in the memory image of the running application [70, 63]. The approach in [63] allows user identification of stack and heap items to target for injection. This is similar to the approach presented in Chapter 4, and allows the user to select functions that are faulty.

P-FSEFI [41] runs parallel applications insider a virtual machine (VM). The hypervisor has control over all code and memory executed by the application, and can create extensive fault injection campaigns in all layers of the software stack. However, not all HPC systems allow deployment of VMs.

Relax [31] and LLFI [71], are LLVM based fault injectors similar to FlipIt; however, Relax is not publicly available and is designed from an old version of LLVM, and LLFI was released after development of FlipIt was underway. Regardless, both injectors do not target HPC applications.

KULFI [97] is a publicly available fault injector similar to Relax and LLFI. Because KULFI was open source at the time development started and is easily modifiable, its structure provides a basis for our fault injector FlipIt. In particular, FlipIt utilizes the framework of their compiler pass, but FlipIt provides a fault injector that is more expansive, extensible, and provides more user control than KULFI.

7.4 Lossy Compression

The applicability of lossy compression for data transfer to and from main memory has been studied for several applications, including LULESH, pF3D, and Miranda [61], where simulation correctness is measured by post-simulation analysis of the physics. In Chapter 6, we utilize the application's spatial discretization properties to construct a bound for the amount of error allowed in the simulation. The benefit is that the accuracy of the associated numerical method is known *a priori* and compression error can be dominated by the discretization truncation error. Another approach uses a two-stage scheme to compress: first lossy, then lossless [80]. The checkpoint size is observed to be around 15% of the original size and restarts are shown to be successful. Another observation is that errors introduced through a particular variable may result in a large propagation of error, leading to the use of lossy compression on only a subset of the checkpointed variables. Wavelet based lossy compression has been used in the climate application NICAM [92], where it is shown that the relative error remains less than 1.5% 1500 time-steps after restart from a lossy compressed checkpoint. One disadvantages is that the error continues to grow linearly with time.

Chapter 8

Conclusions

HPC systems have become instrumental to many fields of science and engineering. However, due to power and budgetary constraints on future systems, the rates of soft errors that cause SDC in HPC applications is expected to increase.

Analyzing how SDC impacts applications allows for creating of algorithmic based detection and recovery schemes. Through the use of application specific detectors the algebraic multigrid (AMG) linear solver is augmented, allowing it to detect SDC that significantly impact convergence. With SDC detected, the proposed multi-level recovery scheme is able to recover and converge with a high probability even when a high number of faults occur during the solve. Going forward, fault consideration is going to become a driving factor in the design of long running and large-scale software. With the additions proposed in the dissertation, AMG has shown it is capable of being used in faulty environments. Moreover, the residual check and the local segmentation fault recovery scheme are general and likely to be valuable in other numerical libraries.

SDCProp, a tool that tracks SDC propagation at a load/store and application level data structure granularity, is created and used to explore how state corruption due to a soft error propagates at a micro (instruction level) and macro (application variable) level for three applications: Jacobi, HPCCG, and CoMD.

At a micro-level, latency of segmentation faults, divergence of control flow, and detection are investigated. In addition, the impact of compiler optimizations is explored on two kernels: `WAXPBY` and SpMV. Results show that the majority of segmentation faults occur shortly after a fault occurs allowing for little propagation. Deviations in control flow predominantly occur as premature loop termination in loops where the fault occurs. Detections have a large enough instruction latency to allow for inter-process propagation.

Macro-level results highlight the speed and intensity of corruption on the processes where the failure occurs and between other processes. Corruption results for critical data structures are discussed along with the ability to define regions of containment for local recovery. Finally, increasing the local problem size increases the latency of inter-process corruption propagation in Jacobi, while it does not influence the latency of CoMD and HPCCG.

Latencies are useful in determining the effectiveness of a detection scheme — i.e. short latencies limit

corruption propagation and can lower recovery costs. However, low cost recovery in a parallel application requires knowledge of the corrupted processes and data structures. Tracking propagation at the macro-level enables discovery of the variables that are most susceptible to corruption. In addition, it identifies the speed of corruption between processes, and indicates whether operations reduce or amplify corruption.

To help assess to the resiliency of HPC applications we create a fault injection framework, FlipIt, to simulate soft errors inside HPC applications. Scalability of FlipIt is shown with weak scaling experiments with HYPRE, HPCCG, and CoMD. FlipIt's overhead diminishes as the application's communication begins to dominate computation. To support various application requirements, FlipIt is designed to be flexible allowing for complex fault injection campaigns and locations. Using FlipIt's features, we inject a fault into HYPRE at a specific location and time to show that a soft error can significantly impact the convergence time of AMG.

Checkpoint restart is an integral part of long running HPC applications. Yet, data movement is becoming a key bottleneck on current and future machines. In response, lossy compression is a method to effectively reduce the volume of data required for a checkpoint, but at the expense of adding error into the simulation. The compression error tolerance is often difficult to determine *a priori*. Chapter 6 investigates the feasibility of using numerical truncation error as a basis for selecting the lossy compression error tolerance. First, by demonstrating this use on a 1D heat and 1D advection equation, it is shown that error introduced through compression can have limited impact on the simulation. In addition, results for two production level HPC applications PlacComCM and Nek5000 highlight its use. Results show that limiting lossy compression error to that of the discretization error allows simulations to restart from a lossy compressed checkpoint without significantly impacting the simulation. Finally, lossy compression is shown to reduce checkpoint time for compression error bounds that respect the truncation error of the application.

References

[1] Emmanuel Agullo, Luc Giraud, Abdou Guermouche, Jean Roman, and Mawussi Zounon. Towards resilient parallel linear krylov solvers: recover-restart strategies. Rapport de recherche RR-8324, INRIA, July 2013.

[2] Mark Ainsworth and Christian Glusa. Is the multigrid method fault tolerant? the multilevel case. *CoRR*, abs/1607.08502, 2016.

[3] Mark Ainsworth and Christian Glusa. *Multigrid at Scale?*, pages 237–253. Springer International Publishing, Cham, 2016.

[4] Rizwan A. Ashraf, Roberto Gioiosa, Gokcen Kestor, Ronald F. DeMara, Chen-Yong Cher, and Pradip Bose. Understanding the propagation of transient errors in HPC applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 72:1–72:12, New York, NY, USA, 2015. ACM.

[5] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[6] Allison H. Baker, Haiying Xu, John M. Dennis, Michael N. Levy, Doug Nychka, Sheri A. Mickelson, Jim Edwards, Mariana Vertenstein, and Al Wegener. A methodology for evaluating the impact of data compression on climate simulation data. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 203–214, New York, NY, USA, 2014. ACM.

[7] Roberto Baldoni, Jean-Michel Hélary, Achour Mostefaoui, and Michel Raynal. On modeling consistent checkpoints and the domino effect in distributed systems. Research Report RR-2569, INRIA, 1995.

[8] Leonardo Bautista-Gomez, Anne Benoit, Aurlien Cavelan, Saurabh K. Raina, Yves Robert, and Hongyang Sun. Which verification for soft error detection? In *HiPC*, pages 2–11. IEEE Computer Society, 2015.

[9] Leonardo Bautista-Gomez and Franck Cappello. Detecting silent data corruption for extreme-scale MPI applications. In *Proceedings of the 22Nd European MPI Users' Group Meeting*, EuroMPI '15, pages 12:1–12:10, New York, NY, USA, 2015. ACM.

[10] Leonardo Bautista-Gomez and Franck Cappello. Exploiting spatial smoothness in HPC applications to detect silent data corruption. In *Proceedings of the 2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conf on Embedded Software and Systems*, HPCC-CSS-ICESS '15, pages 128–133, Washington, DC, USA, 2015. IEEE Computer Society.

[11] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. FTI: high performance fault tolerance interface for hybrid systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 32:1–32:32, New York, NY, USA, 2011. ACM.

[12] Austin R Benson, Sven Schmit, and Robert Schreiber. Silent error detection in numerical time-stepping schemes. *The International Journal of High Performance Computing Applications*, 29(4):403–421, 2015.

[13] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, et al. Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 15, 2008.

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

[15] Shekhar Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, November 2005.

[16] M. Burtscher and P. Ratanaworabhan. High throughput compression of double-precision floating-point data. In *Data Compression Conference, 2007. DCC '07*, pages 293–302, March 2007.

[17] Franck Calhoun, Jon Cappello, Luke Olson, and Marc Snir. Exploring the feasibility of lossy compression for pde simulations. In *In submission to the 2017 IEEE International Conference on Cluster Computing*, CLUSTER '17, Washington, DC, USA, 2017. IEEE Computer Society.

[18] Jon Calhoun, Luke Olson, and Marc Snir. FlipIt: An LLVM based fault injector for HPC. In *Proceedings of the 20th International Euro-Par Conference on Parallel Processing (Euro-Par '14)*, 2014.

[19] Jon Calhoun, Luke Olson, Marc Snir, and William D. Gropp. Towards a more fault resilient multigrid solver. In *Proceedings of the Symposium on High Performance Computing*, HPC '15, pages 1–8, San Diego, CA, USA, 2015. Society for Computer Simulation International.

[20] Jon Calhoun, Luke Olson, Marc Snir, and William D. Gropp. Towards a more fault resilient multigrid solver. In *Proceedings of the Symposium on High Performance Computing*, HPC '15, pages 1–8, San Diego, CA, USA, 2015. Society for Computer Simulation International.

[21] Jon Calhoun, Luke N. Olson, Marc Snir, and William D. Gropp. Towards a more complete understanding of sdc propagation. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '17, New York, NY, USA, 2017. ACM.

[22] Jon Calhoun, Marc Snir, Luke Olson, and Maria Garzaran. Understanding the propagation of error due to a silent data corruption in a sparse matrix vector multiply. In *Proceedings of the 2015 IEEE International Conference on Cluster Computing*, CLUSTER '15, pages 541–542, Washington, DC, USA, 2015. IEEE Computer Society.

[23] Franck Cappello, Al Geist, Bill Gropp, Laxmikant Kale, Bill Kramer, and Marc Snir. Toward exascale resilience. *Int. J. High Perform. Comput. Appl.*, 23(4):374–388, November 2009.

[24] J. Carreira, H. Madeira, and J.G. Silva. Xception: a technique for the experimental evaluation of dependability in modern computers. *IEEE Transactions on Software Engineering 24,*, 24(2):125–36, 1998.

[25] Marc Casas, Bronis R. de Supinski, Greg Bronevetsky, and Martin Schulz. Fault resilience of the algebraic multi-grid solver. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pages 91–100, New York, NY, USA, 2012. ACM.

[26] Zhengzhang Chen, Seung Woo Son, William Hendrix, Ankit Agrawal, Wei-keng Liao, and Alok Choudhary. NUMARCK: machine learning algorithm for resiliency and checkpointing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 733–744, Piscataway, NJ, USA, 2014. IEEE Press.

[27] Zizhong Chen. Algorithm-based recovery for iterative methods without checkpointing. In *Proceedings of the 20th international symposium on High performance distributed computing*, HPDC '11, pages 73–84, New York, NY, USA, 2011. ACM.

[28] Zizhong Chen. Algorithm-based recovery for iterative methods without checkpointing. In *Proceedings of the 20th international symposium on High performance distributed computing*, HPDC '11, pages 73–84, New York, NY, USA, 2011. ACM.

[29] Jinsuk Chung, Ikhwan Lee, Michael Sullivan, Jee Ho Ryoo, Dong Wan Kim, Doe Hyun Yoon, Larry Kaplan, and Mattan Erez. Containment domains: A scalable, efficient, and flexible resilience scheme for exascale systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 58:1–58:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[30] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener. Comput. Syst.*, 22(3):303–312, February 2006.

[31] Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. Relax: an architectural framework for software recovery of hardware faults. In *Proceedings of the 37th International Symposium on Computer Architecture (ISCA)*, 2010.

[32] Timothy J. Dell. A white paper on the benefits of chipkillcorrect ECC for PC server main memory. Technical report, IBM Microelectronics Division, 1997.

[33] Sheng Di and Cappello Franck. Fast error-bounded lossy HPC data compression with SZ. In *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium*, IPDPS '16, page To Appear., Washington, DC, USA, 2016. IEEE Computer Society.

[34] Ronald G. Dreslinski, Michael Wieckowski, David Blaauw, Dennis Sylvester, and Trevor N. Mudge. Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits. *Proceedings of the IEEE*, 98(2):253–266, 2010.

[35] James Elliott, Mark Hoemmen, and Frank Mueller. Evaluating the impact of SDC on the GMRES iterative solver. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 1193–1202, Washington, DC, USA, 2014. IEEE Computer Society.

[36] James Elliott, Frank Mueller, Miroslav Stoyanov, and Clayton Webster. Quantifying the impact of single bit flips on floating point arithmetic. Technical report, Oak Ridge National Laboratory, August 2013.

[37] Bo Fang, Qining Lu, Karthik Pattabiraman, Matei Ripeanu, and Sudhanva Gurumurthi. ePVF: an enhanced program vulnerability factor methodology for cross-layer resilience analysis. *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 00:168–179, 2016.

[38] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. Shoestring: Probabilistic soft error reliability on the cheap. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 385–396, New York, NY, USA, 2010. ACM.

[39] 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.

[40] Weining Gu, Zbigniew Kalbarczyk, Ravishankar K. Iyer, and Zhen-Yu Yang. Characterization of linux kernel behavior under errors. In *DSN*, pages 459–468. IEEE Computer Society, 2003.

[41] Qiang Guan, Nathan BeBardeleben, Panruo Wu, Stephan Eidenbenz, Sean Blanchard, Laura Monroe, Elisabeth Baseman, and Li Tan. Design, use and evaluation of p-fsefi: A parallel soft error fault injection framework for emulating soft errors in parallel applications. In *Proceedings of the 9th EAI International Conference on Simulation Tools and Techniques*, SIMUTOOLS'16, pages 9–17, ICST, Brussels, Belgium, Belgium, 2016. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[42] Amina Guermouche, Thomas Ropars, Elisabeth Brunet, Marc Snir, and Franck Cappello. Uncoordinated checkpointing without domino effect for send-deterministic MPI applications. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 989–1000. IEEE, 2011.

[43] Amina Guermouche, Thomas Ropars, Marc Snir, and Franck Cappello. Hydee: Failure containment without event logging for large scale send-deterministic MPI applications. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 1216–1227. IEEE, 2012.

[44] Seungjae Han, Harold A. Rosenberg, and Kang G. Shin. DOCTOR: an IntegrateD SOftware fault InjeCTiOn EnviRonment, 1995.

[45] Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (BLCR) for linux clusters. *Journal of Physics: Conference Series*, 46(1):494, 2006.

[46] Siva Kumar Sastry Hari, Sarita V. Adve, and Helia Naeimi. Low-cost program-level detectors for reducing silent data corruptions. In *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.

[47] Amin Hassani, Anthony Skjellum, Purushotham V. Bangalore, and Ron Brightwell. Practical resilient cases for fa-mpi, a transactional fault-tolerant mpi. In *Proceedings of the 3rd Workshop on Exascale MPI*, ExaMPI '15, pages 1:1–1:10, New York, NY, USA, 2015. ACM.

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

[49] Markus Huber, Björn Gmeiner, Ulrich Rüde, and Barbara I. Wohlmuth. Resilience for multigrid software at the extreme scale. *CoRR*, abs/1506.06185, 2015.

[50] Dewan Ibtesham, Dorian Arnold, Kurt B. Ferreira, and Patrick G. Bridges. On the viability of checkpoint compression for extreme scale fault tolerance. In *Proceedings of the 2011 International Conference on Parallel Processing - Volume 2*, Euro-Par'11, pages 302–311, Berlin, Heidelberg, 2012. Springer-Verlag.

[51] Dewan Ibtesham, Kurt B. Ferreira, and Dorian C. Arnold. A checkpoint compression study for high-performance computing systems. *IJHPCA*, 29(4):387–402, 2015.

[52] F. Isaila, Prasanna Balaprakash, S. M. Wild, D. Kimpe, R. Latham, R. Ross, and Paul D. Hovland. Collective I/O tuning using analytical and machine-learning models. In *IEEE Cluster 2015*, Chicago, IL, 09/2015 2015. IEEE, IEEE.

[53] Tanzima Zerin Islam, Kathryn Mohror, Saurabh Bagchi, Adam Moody, Bronis R. de Supinski, and Rudolf Eigenmann. McrEngine: a scalable checkpointing system using data-aware aggregation and compression. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 17:1–17:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[54] Luc Jaulmes, Marc Casas, Miquel Moretó, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. Exploiting asynchrony from exact forward recovery for DUE in iterative solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 53:1–53:12, New York, NY, USA, 2015. ACM.

[55] Himanshu Kaul, Mark Anders, Steven Hsu, Amit Agarwal, Ram Krishnamurthy, and Shekhar Borkar. Near-threshold voltage (NTV) design: Opportunities and challenges. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1153–1158, New York, NY, USA, 2012. ACM.

[56] Daya Shanker Khudia, Griffin Wright, and Scott Mahlke. Efficient soft error protection for commodity embedded microprocessors using profile information. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES '12, pages 99–108, New York, NY, USA, 2012. ACM.

[57] Peter M. Kogge, Patrick La Fratta, and Megan Vance. [2010] facing the exascale energy wall. In *Proceedings of the 2010 International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, IWIA '10, pages 51–58, Washington, DC, USA, 2010. IEEE Computer Society.

[58] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. In *Proceedings of 1986 ACM Fall Joint Computer Conference*, ACM '86, pages 1150–1158, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[59] Ignacio Laguna, Martin Schulz, David F. Richards, Jon Calhoun, and Luke Olson. IPAS: intelligent protection against silent output corruption in scientific applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO 2016, pages 227–238, New York, NY, USA, 2016. ACM.

[60] Sriram Lakshminarasimhan, Neil Shah, Stephane Ethier, Scott Klasky, Rob Latham, Rob Ross, and Nagiza F. Samatova. Compressing the incompressible with ISABELA: in-situ reduction of spatio-temporal data. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part I*, Euro-Par'11, pages 366–379, Berlin, Heidelberg, 2011. Springer-Verlag.

[61] Daniel Laney, Steven Langer, Christopher Weber, Peter Lindstrom, and Al Wegener. Assessing the effects of data compression in simulations using physically motivated metrics. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 76:1–76:12, New York, NY, USA, 2013. ACM.

[62] Chris Lattner and Vikram Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[63] Dong Li, Jeffrey S. Vetter, and Weikuan Yu. Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 57:1–57:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[64] Guanpeng Li, Qining Lu, and Karthik Pattabiraman. Fine-grained characterization of faults causing long latency crashes in programs. In *DSN*, pages 450–461. IEEE Computer Society, 2015.

[65] Guanpeng Li, Karthik Pattabiraman, Chen-Yong Cher, and Pradip Bose. Understanding error propagation in GPGPU applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 21:1–21:12, Piscataway, NJ, USA, 2016. IEEE Press.

[66] Man-Lap Li, Pradeep Ramachandran, Swarup K Sahoo, Sarita V Adve, Vikram S Adve, and Yuanyuan Zhou. Swat: An error resilient system. *Proceedings of SELSE*, 2008.

[67] P. Lindstrom. Fixed-rate compressed floating-point arrays. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2674–2683, Dec 2014.

[68] P. Lindstrom and M. Isenburg. Fast and efficient compression of floating-point data. *Visualization and Computer Graphics, IEEE Transactions on*, 12(5):1245–1250, 2006.

[69] Ning Liu, Jason Cope, Philip H. Carns, Christopher D. Carothers, Robert B. Ross, Gary Grider, Adam Crume, and Carlos Maltzahn. On the role of burst buffers in leadership-class storage systems. In *IEEE 28th Symposium on Mass Storage Systems and Technologies, MSST 2012, April 16-20, 2012, Asilomar Conference Grounds, Pacific Grove, CA, USA*, pages 1–11, 2012.

[70] Charng-da Lu and Daniel A. Reed. Assessing fault sensitivity in MPI applications. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC '04, pages 37–, Washington, DC, USA, 2004. IEEE Computer Society.

[71] Qining Lu, Mostafa Farahani, Jiesheng Wei, Anna Thomas, and Karthik Pattabiraman. Llfi: An intermediate code-level fault injection tool for hardware faults. In *Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security*, QRS '15, pages 11–16, Washington, DC, USA, 2015. IEEE Computer Society.

[72] Qining Lu, Karthik Pattabiraman, Meeta S. Gupta, and Jude A. Rivers. SDCTune: a model for predicting the sdc proneness of an application for configurable protection. In *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '14, pages 23:1–23:10, New York, NY, USA, 2014. ACM.

[73] Huong Luu, Marianne Winslett, William Gropp, Robert Ross, Philip Carns, Kevin Harms, Mr Prabhat, Suren Byna, and Yushu Yao. A multiplatform study of I/O behavior on petascale supercomputers. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 33–44, New York, NY, USA, 2015. ACM.

[74] R. E. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM J. Res. Dev.*, 6(2):200–209, April 1962.

[75] Catello Di Martino, Zbigniew Kalbarczyk, Ravishankar K. Iyer, Fabio Baccanico, Joseph Fullop, and William Kramer. Lessons learned from the analysis of system failures at petascale: The case of blue waters. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, pages 610–621, Washington, DC, USA, 2014. IEEE Computer Society.

[76] T. C. May and Murray H. Woods. Alpha-particle-induced soft errors in dynamic memories. *Electron Devices, IEEE Transactions on*, 26(1):2–9, January 1979.

[77] Esteban Meneses, Xiang Ni, and L. V. Kale. A message-logging protocol for multicore systems. In *Proceedings of the 2nd Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS)*, Boston, USA, June 2012.

[78] Amitabh Mishra and Prithviraj Banerjee. An algorithm-based error detection scheme for the multigrid method. *IEEE Trans. Comput.*, 52(9):1089–1099, September 2003.

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

[80] Xiang Ni, Tanzima Islam, Kathryn Mohror, Adam Moody, and Laxmikant V Kale. Lossy compression for checkpointing: Fallible or feasible? In *Poster Session of the 2014 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, Washington, DC, USA, 2014. IEEE Computer Society.

[81] Xiang Ni and Laxmikant V. Kale. FlipBack: automatic targeted protection against silent data corruption. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 29:1–29:12, Piscataway, NJ, USA, 2016. IEEE Press.

[82] Xiang Ni, Esteban Meneses, Nikhil Jain, and Laxmikant V. Kale. ACR: automatic checkpoint/restart for soft and hard error protection. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13. IEEE Computer Society, November 2013.

[83] Bogdan Nicolae and Franck Cappello. AI-Ckpt: leveraging memory access patterns for adaptive asynchronous incremental checkpointing. In *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 155–166, New York, NY, USA, 2013. ACM.

[84] Karthik Pattabiraman, Ravishankar K. Iyer, and Zbigniew T. Kalbarczyk. Automated derivation of application-aware error detectors using static analysis: The trusted illiac approach. *IEEE Transactions on Dependable and Secure Computing*, 8:44–57, 2009.

[85] Karthik Pattabiraman, Nithin M. Nakka, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. SymPLFIED: symbolic program-level fault injection and error detection framework. *IEEE Transactions on Computers*, 62(11):2292–2307, 2013.

[86] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, pages 109–116, New York, NY, USA, 1988. ACM.

[87] F. Petrini, K. Davis, and J. C. Sancho. System-level fault-tolerance in large-scale parallel machines with buffered coscheduling. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 209–, April 2004.

[88] Rolf Riesen, Kurt Ferreira, Dilma Da Silva, Pierre Lemarinier, Dorian Arnold, and Patrick G. Bridges. Alleviating scalability issues of checkpointing protocols. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 18:1–18:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[89] J. W. Ruge and K. Stüben. Algebraic multigrid. In *Multigrid methods*, volume 3 of *Frontiers in Applied Mathematics*, pages 73–130. SIAM, Philadelphia, PA, 1987.

[90] Somayeh Sardashti. *Using Compression for Energy-Optimized Memory Hierarchies*. PhD thesis, University of Wisconsin - Madison, 2015.

[91] Naoto Sasaki, Kento Sato, Toshio Endo, and Satoshi Matsuoka. Exploration of lossy compression for application-level checkpoint/restart. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium*, IPDPS '15, pages 914–922, Washington, DC, USA, 2015. IEEE Computer Society.

[92] Naoto Sasaki, Kento Sato, Toshio Endo, and Satoshi Matsuoka. Exploration of lossy compression for application-level checkpoint/restart. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium*, IPDPS '15, pages 914–922, Washington, DC, USA, 2015. IEEE Computer Society.

[93] Siva Kumar Sastry Hari, Radha Venkatagiri, Sarita V. Adve, and Helia Naeimi. GangES: gang error simulation for hardware resiliency evaluation. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 61–72, Piscataway, NJ, USA, 2014. IEEE Press.

[94] Kento Sato, Todd Gamblin, Adam Moody, Bronis R. de Supinski, Kathryn Mohror, and Naoya Maruyama. Design and modeling of non-blocking checkpoint system. In *Proceedings of the ATIP/A*CRC Workshop on Accelerator Technologies for High-Performance Computing: Does Asia Lead the Way?*, ATIP '12, pages 39:1–39:2, Singapore, Singapore, 2012. A*STAR Computational Resource Centre.

[95] Bianca Schroeder and Garth A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '06, pages 249–258, Washington, DC, USA, 2006. IEEE Computer Society.

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

[97] Vishal Chandra Sharma, Arvind Haran, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. Towards formal approaches to system resilience. In *Proceedings of the 19th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2013.

[98] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, Andrew A Chien, Paul Coteus, Nathan A DeBardeleben, Pedro C Diniz, Christian Engelmann, Mattan Erez, Saverio Fazzari, Al Geist, Rinku Gupta, Fred Johnson, Sriram Krishnamoorthy, Sven Leyffer, Dean Liberty, Subhasish Mitra, Todd Munson, Rob Schreiber, Jon Stearley, and Eric Van Hensbergen. Addressing failures in exascale computing. *International Journal of High Performance Computing Applications*, 28(2):127 – 171, May 2014.

[99] Seung Woo Son, Zhengzhang Chen, William Hendrix, Ankit Agrawal, Wei keng Liao, and Alok Choudhary. Data compression for the exascale computing era - survey. *Supercomputing frontiers and innovations*, 1(2), 2014.

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

[101] Vilas Sridharan and Dean Liberty. A study of DRAM failures in the field. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 76:1–76:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[102] David T. Stott, Benjamin Floering, Daniel Burke, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors. In *In Proceedings of the IEEE International Computer Performance and Dependability Symposium*, pages 91–100, 2000.

[103] Nicholas Wang, Michael Fertig, and Sanjay Patel. Y-branches: When you come to a fork in the road, take it. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, PACT '03, pages 56–, Washington, DC, USA, 2003. IEEE Computer Society.

[104] Keun Soo Yim, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. Quantitative analysis of long-latency failures in system software. In *Proceedings of the 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*, PRDC '09, pages 23–30, Washington, DC, USA, 2009. IEEE Computer Society.

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

[106] Jing Yu, Maria Jesus Garzaran, and Marc Snir. ESoftCheck: removal of non-vital checks for fault tolerance. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 35–46, Washington, DC, USA, 2009. IEEE Computer Society.

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

[108] J. F. Ziegler and W. A. Lanford. Effect of Cosmic Rays on Computer Memories. In *Science*, volume 206, pages 776–788, 1979.

[109] James Ziegler and Helmut Puchner. *SER – History, Trends and Challenges: A guide for designing with Memory ICs.* Cypress, 2004.