UNIVERSITY OF CALIFORNIA, MERCED

Characterization and Modeling of Error Resilience in HPC Applications

by

Luanzheng Guo

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Electrical Engineering and Computer Science

Committee in charge:

Professor Dong Li, Chair
Professor Mukesh Singhal
Professor Florin Rusu
Dr. Ignacio Laguna

Summer 2020

The dissertation of Luanzheng Guo is approved:

Professor Mukesh Singhal Date

Professor Florin Rusu Date

Dr. Ignacio Laguna Date

Professor Dong Li (Chair) Date

University of California, Merced

Summer 2020

To the ones I love.

Contents

List of Figures

List of Tables

Acknowledgments

Foremost, I want to express my sincere gratitude to my Ph.D. advisor Professor Dong Li. First, I want to thank him for bringing me to the University of California-Merced. Professor Dong Li has been a great advisor in the past years. Without his consistent guidance and generous help, I cannot imagine how I could work out these projects, cultivate these publications, and prepare these presentations all by myself. Professor Dong Li has offered me the freedom to explore new research ideas and encouraged me to become an independent researcher. I am always inspired by his continuing motivation, persistence, dedication, diligence, and enthusiasm for science and his humility. I still remember these long and short free rides Professor Dong Li shared with me in these early days and these unforgettable conversations we had in the 50th Celebration of the Turing Award.

I would also like to thank my dissertation committee—Professor Mukesh Singhal, Professor Florin Rusu, and Dr. Ignacio Laguna for serving in my dissertation committee, for their time and effort on reviewing my dissertation and attending my defense, and for their continuous help, constructive comments, and immense knowledge that help significantly advance the quality of this dissertation.

I want further to thank my mentor at Lawrence Livermore National Laboratory, Dr. Ignacio Laguna for always being nice to me. I have been a student summer intern at the Center for Applied Scientific Computing, Lawrence Livermore National Laboratory for four times since 2016. I have learned a lot from Dr. Ignacio Laguna during the past years. I have learned not only how to choose an interesting but essential research problem, how to conduct excellent research, and how to write a good research paper in computer science, but also learned from his patience, kindness, humility, intelligence, and faith in science. Dr. Ignacio Laguna is an example to me, where I can always draw strength to continue my research when feeling frustrated. I also thank Dr. Ignacio Laguna for his generous help in my job hunting and for writing countless recommendation letters.

Furthermore, I would like to thank other colleagues at the lab. My sincere thanks go to Dr. Martin Schulz, my host at the lab and a co-author of my first paper. I have learned good writing styles through his amending to my manuscripts. I appreciate Dr. Martin Schulzs patience and generous help in the past years, for writing recommendation letters for me, for introducing me to his friends, and for his encouragement. I also want to thank Dr. Kathryn Mohror for being my host at the lab, and for passing my resume around for my job hunting. My sincere appreciation goes to Dr. Giorgis Georgakoudis, who is a co-author of my most recent paper, and has helped me a lot for reviewing my code, and for providing insightful comments to my proposals and manuscripts. I am always inspired by his extensive knowledge and constructive considerations. I am very grateful to Dr. Kento Sato for being a good friend who is always nice and offers me many good opportunities, and for sharing research ideas and insights with me. I want to thank Dr. Murali Emani for sharing research insights with me and passing my resume around. I thank Dr. Naoya Maruyama for his help and support on my application to the Livermore Graduate Scholar Program and his help with my job hunting. I thank Dr. Stephanie Brink for her help and support on my application to the Livermore Graduate Scholar Program. I also want to thank my sincere friends and roommates—Dr. Stephen Herbein, Dr. Michael Wyatt, Dr. Dylan Chapp, Dr. William Killian, Dr. Johannes Brust,

and Dr. Qunwei Li for these memorable days having fun together and biking to work. Also, I want to thank my officemates at Lawrence Livermore National Laboratory. They are Dr. Teng Wang, Dr, Lai Wei, Dr. Yue Zhu, Dr. Furong Sun, Dr. Jiyuan Zhang, Dr. Cuiyu He, Dr. Zhimin Li, Dr. Christopher Wright, Mr. Duong Hoang, Mr. Ayush Patwari, Ms. Hui Guo, and Mr. Mano Rm.

I want to thank my labmates and friends at UC Merced. Many thanks for these exciting moments and frustrating days we get together. They are Mr. Yingchao Huang, Mr. Himanshu Pillai, Ms. Hanlin He, Mr. Wei Liu, Mr. Kai Wu, Mr. Jing Liang, Ms. Ying Ding, Ms. Wenqian Dong, Ms. Jie Ren, Mr. Letian Kang, Mr. Jiawen Liu, Mr. Jie Liu, Mr. Zhen Xie, Mr. Xin He, Mr. Andrés Torres García, Mr. Jun Hyung Shin, Dr. Tom Kim, Mr. Shattik Rubaiyat Muhammad, Dr. Mina Naghshnejad, Dr. Yijun Li, Mr. Zhixun He, Ms. Belinda Braunstein, Mr. Xin Zhang, Dr. Maryam Shadloo, and Ms. Mahshid Montazer, and many others. I am so grateful to have them in my life in the past five years.

Moreover, I want to thank my mentors, colleagues, and friends I made in conferences for their friendship, inspiration, encouragement, and memorable moments. They are Dr. Christine Harvey, Dr. Jay Lofstead, Dr. Joel Fuentes, Professor Dorian Arnold, Professor Michela Taufer, Mrs. Jenett Tillotson, Dr. Xin Liang, Dr. Sihuan Li, Mr. Tony Liu, Professor Jack Dongarra, Dr. James Rome, Dr. Christian Engelmann, Dr. Hal Finkle, Mr. Mike Lee, Barbara Horner-Miller, Dr. Sean Peisert, Dr. Min Si, Dr. Zhengji Zhao, Professor Sunita Chandrasekaran, Dr. Hongzhang Shan, Dr. Guido Juckeland, Professor Jon Calhoun, Professor Suzanne McIntosh, Dr. Dana Bruson, Dr. Patrick Widener, Professor Ewa Deelman, Dr. Patrick McCormick, Mr. Kevin Walsh, Professor Vladimir Getov, Mr. Eugene Miya, Ms. Tiffany Trader, Dr. Dana Freiburger, Ms. Anna Loup, and many others.

Special thanks go to my advisors during my Master's study who inspired and enlightened my interests in doing research. They are Professor Jun Chu, Professor Chunhong Pan, Professor Shiming Xiang, Professor Guimei Zhang, Professor Jiexian Zeng, Professor Lingfeng Wang, Professor Huaiyu Wu, Professor Gaofeng Meng, Professor Jun Bai, Professor Yin Wang, and Professor Bin Fan. It is my pleasure to begin my journey in academia with them.

Last but not least, I want to express my gratitude to my family for their constant source of inspiration. They are my parents Qiying Lu and Yaohua Guo, my adopted parents Patricia Hachten and Brad Hachten, my paternal grandparents Qiuxiang Xu and Maohuan Guo, and my maternal grandparents Xuemei Huang and Keli Lu. Special thanks to my partner, Qijun Zhang, for the support and love in the past years.

Curriculum Vitae

Education

2015-2020	Ph. D. in Electrical Engineering and Computer Science, University of California-Merced, USA
2011-2014	M. S. in Computer Science, Nanchang Hangkong University, China
2007-2011	B. S. in Computer Engineering, Nanchang Hangkong University, China

Publications

Conference Papers:

Giorgis Georgakoudis, *Luanzheng Guo*, and Ignacio Laguna. Practical MPI Resilience: A Performance and Correctness Evaluation. ISC HPC Conference (ISC), Frankfurt 2020

Luanzheng Guo and Dong Li. MOARD: Modeling Application Resilience to Transient Faults on Data Objects. The 33rd IEEE International Parallel and Distributed Processing Symposium (IPDPS), Rio, Brazil 2019

Luanzheng Guo, Dong Li, Ignacio Laguna, and Martin Schulz. FlipTracker: Understanding Natural Error Resilience in HPC Applications. The 30th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC) 2018

Papers In-submission:

Luanzheng Guo, Dong Li, and Ignacio Laguna. PARIS: Predicting Application Resilience Using Machine Learning. Submitted to Journal of Parallel and Distributed Computing 2020

Luanzheng Guo, Dong Li, and Ignacio Laguna. Cross-Architecture Resilience Characterization: Predicting GPU Resilience Using CPU Code. Submitted to XXX20

Luanzheng Guo, Giorgis Georgakoudis, Ignacio Laguna, Dong Li. MATCH: An MPI Fault Tolerance Benchmark Suite. Submitted to IISWC'20

Ph.D. Forum Papers:

Luanzheng Guo, Dong Li. Characterization and Modeling of Error Resilience in HPC Applications. The 31st ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Denver, CO 2019

Luanzheng Guo, Dong Li. Characterization and Modeling of Error Resilience in HPC Applications. The 33rd IEEE International Parallel and Distributed Processing Symposium (IPDPS), Rio, Brazil 2019

Posters:

Lawrence Livermore National Laboratory Summer Poster Symposium Posters:

Understanding Use of ULFM in MPI Resilience, 2019

Which Fault Injection Tool Should We Use for GPU Programs? 2018

Understanding the Resilience of Fundamental Data Types, 2017

Understanding Resilience Patterns of Algorithms via Application-Level Fault Injection, 2016

Luanzheng Guo, Jing Liang, and Dong Li. Understanding Ineffectiveness of Application-Level Fault Injection. The 28th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC) 2016

Honors

2020	Trusted CI Fellow by the NSF Cybersecurity Center of Excellence
2019	UC Merced GSA Travel Award for SC19
2019	IEEE CS TCHPC Travel Award for SC19
2019	UC Merced Graduate Dean's Dissertation Fellowship
2019	UC Merced Graduate Fellowship Incentive Program Award
2019	IEEE TCPP Travel Award for IPDPS19
2018	Research highlighted by HPCwire in its Whats new in HPC research?
2018	SC18 Outstanding Student Volunteer
2018	SC18 Outstanding Lightning Talk
2018	Outstanding reviewer award by Elsevier
2017	ACM SIGHPC Travel Award to attend the 50th celebration of the Turing Award
2016	Best poster finalist in SC16
2015-2017	Bobcat Fellowship at UC Merced

Abstract

Characterization and Modeling of Error Resilience in HPC Applications
by
Luanzheng Guo
Doctor of Philosophy
in
Electrical Engineering and Computer Science
University of California, Merced
Professor Dong Li, Chair

HPC systems are widely used in industrial, economical, and scientific applications, and many of these applications are safety- and time-critical. We must ensure that the application execution is reliable, and the scientific simulation outcome is trustworthy. As HPC systems continue to increase computational power and size, next-generation HPC systems are expected to incur a higher failure rate than contemporary systems. How to ensure scientific computing integrity in the presence of an increasing number of system faults is one of the grand challenges (also known as the resilience challenge) for large-scale HPC systems.

This dissertation focuses on characterizing, modeling, developing, and advancing resilience strategies and tools in HPC systems to allow scientific applications to survive system failures better. In particular, in this dissertation we systematically characterize HPC applications to find reasons accounting for nature error resilience of HPC applications by tracking error propagation and also by capturing application properties according to their significance to application error resilience using machine learning. We further model application error resilience at different granularities, including individual data objects, small computation kernels, and the whole application. Also, we develop an error resilience benchmark suite to comprehensively evaluate and comparatively study different error resilience designs in the presence of MPI process or node failures. With the knowledge learned from characterization and modeling of application error resilience, we propose a collection of new methodologies and tools that can guide HPC practitioners to find the most effective and efficient error resilience designs, provide helps to advance effectiveness and efficiency of the existing error resilience designs, and build inspiration foundations to future error resilience designs aiming at higher effectiveness and efficiency of HPC systems.

Chapter 1

Introduction

The continued growth of large-scale high-performance computing (HPC) systems is fueled by two trends: continued integration of additional functionality onto system nodes, and the increased number of nodes (and components) in the systems. As a result, these large-scale systems are jeopardized by potentially increasing faults in hardware and software [143, 132, 140, 77, 142, 139]. Ensuring scientific computing integrity and correctness of application execution in the presence of faults remains one of the grand challenges (also known as the resilience challenge) for large-scale HPC systems [31, 32].

We focus on transient faults and the cascading failures caused by transient faults [62, 49]. Transient faults [14] due to high energy particle strikes, wear-out, and other factors are expected to become a critical contributor to in-field system failures of high-performance computing (HPC). As illustrated in Figure 1.1, transient faults can lead to not only interruptions, but also silent data corruption (SDC), which can impact scientific results without users realizing it. Furthermore, transient faults are considered a critical contributor to system process/node failures according to a recent investigation [62]. As the number of transient faults grows, it becomes increasingly necessary to develop more efficient and effective fault tolerance mechanisms to protect application execution from

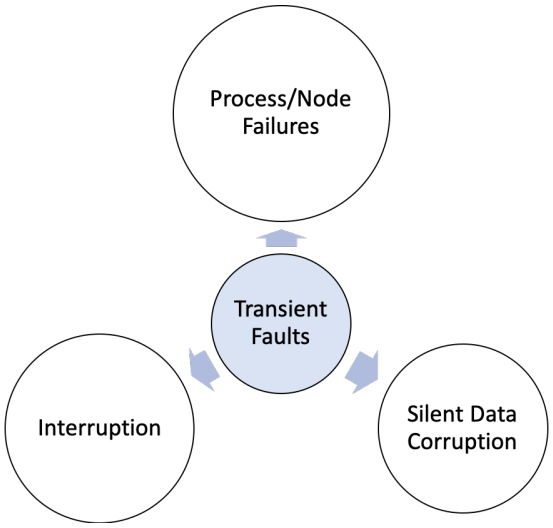


Figure 1.1: System failures caused by transient faults.

transient faults and system failures caused by transient faults.

In this dissertation, we propose a composition of techniques, algorithms, and tools to characterize and model error resilience in HPC applications by analytical and machine learning models from a variety of aspects, and at different granularities.

1.1 Research Problems and Challenges

We face a range of challenges on characterization and modeling of application error resilience in HPC applications. We discuss the research problems and challenges below.

1.1.1 Characterization of Error Resilience in HPC Applications

Previous work on fault tolerance, which typically focused on individual applications, demonstrates that a number of applications can transparently tolerate transient faults before they affect the application's numerical output. Examples of such applications are algebraic multi-grid solvers (AMG) [34], Conjugate Gradient (CG) solvers [129], GMRES iterative solvers [55], Monte Carlo simulations [11], and machine learning algorithms, such as clustering [94] and deep-learning neural networks [9, 123].

While previous work attributes this natural resilience at a high-level to either the probabilistic or iterative nature of the application, the community still lacks the fundamental understanding of the program constructs that result in such natural error resilience. Fundamentally, we do not have clear answers to questions, such as: Are there any common computation patterns (i.e., combinations or sequences of computations) that lead to natural error resilience? If so, how can these patterns be found? How can future application design benefit from patterns exhibiting natural resilience?

Finding answers to these questions is critical for error detection and recovery to avoid overprotecting regions of code that are naturally resilient.

1.1.2 Modeling Error Resilience in HPC Applications

Analytical modeling: Understanding Resilience on Data objects

If transient faults manifest in architecturally visible states (e.g., registers and the memory) and those states hold values of a data object, then we have transient faults on the data object. Transient faults on a data object impact application outcome correctness. Understanding application resilience to transient faults on data objects is critical to ensure computing integrity in future large-scale systems.

Furthermore, many common application-level fault tolerance mechanisms focus on data objects. Understanding application resilience to transient faults on data objects can be helpful to direct those mechanisms. Application-level checkpointing is an example of such an application-level fault tolerance mechanism. By periodically saving the correct values of some data objects into persistent storage, application-level checkpoint makes application resumable when a failure happens. Some algorithm-based fault tolerance methods [39, 45] are other examples. They can detect and locate errors in specific data objects. However, those application-level fault tolerance mechanisms can be expensive (e.g., 35% performance overhead in [53]). If data corruptions of a data object are easily tolerable by the application, then we do not need to apply those mechanisms to

protect the data object, which will improve performance and energy efficiency. Hence, understanding application resilience to transient faults on data objects is useful to direct those application-level fault tolerance mechanisms.

However, we do not have a method or a tool to quantify application resilience to transient faults on data objects. The current common practice to understand application resilience to transient faults in HPC is application-level random fault injection [28, 96, 97, 34, 156, 153, 40, 28, 150, 61, 88, 79]. Although random fault injection is useful, it makes it difficult to study application resilience to transient faults on data objects because of the following two reasons.

First, random fault injection loses application semantics (data semantics). Random fault injection randomly selects instructions and triggers random bit flip in input or output operands of the instructions. Typically, random fault injection performs a large amount of random fault injection tests, and then calculates that among all fault injection tests, how many of them succeed (i.e., having correct application outcomes). However, random fault injection gives us little knowledge of *how* and *where* errors are tolerated [69]. Understanding "how" and "where" is necessary to identify why the application is vulnerable to the value corruption of some data objects, and provides feedback on how to apply application-level fault tolerance mechanisms effectively and efficiently.

How can we model and measure application resilience to transient faults on data objects without doing fault injection? How can we model the application resilience to data objects at individual operations? How can we model the application resilience to data objects during the error propagation? Can we track the error propagation on data objects to the end of the execution?

Machine learning modeling: Predicting Application Resilience

While random fault injection works in practice and is widely used in resilience studies, a key problem of this approach is that it is highly time consuming, and as a result, it is usually applied to limited scenarios, for example, on applications that run for a short period of time and/or single-threaded codes. To illustrate the problem, consider an application that runs for 6 hours—a typical execution time for a large-scale scientific simulation. Using statistical analysis (e.g., using [95]), the number of random fault injections to obtain a low margin of error (e.g., 1%-3%) is in the order of thousands of injections. Thus, the total fault injection campaign could last several days. For multi-threaded or multi-process applications, this time is much higher since random faults must be injected in different threads or processes.

To address the limitations of FI, researchers have built error-propagation analytical models [98], which are faster than FI in estimating application resilience. However, they lack accuracy as they estimate application resilience to errors based on the analysis of possible errors in individual instructions. The analysis inaccuracy at individual instructions is accumulated, causing low accuracy to estimate the whole application resilience. Furthermore, these models do not consider the effects of resilience computation patterns (e.g., dead corrupted locations and repeated addition [65]). Studying those patterns demands analyzing multiple instructions together, while most existing analytical models analyze instructions in isolation. In summary, the community lacks a fundamental approach that enables fast and accurate evaluation of application resilience.

Can we propose an approach to predict application resilience that can solve above problems efficiently and effectively? Is machine learning a solution to these problems?

If yes, what characteristics can we use as features to make the prediction? What machine learning model can fit into our case?

1.1.3 Modeling Fault Tolerance to Process/Node Failures

Next-generation HPC systems are expected to incur a much higher failure rate than contemporary systems. For example, the Sequoia supercomputer located in Lawrence Livermore National Laboratory (LLNL) reported a mean time between node failures to be 19.2 hours in 2013 [51]. After that, in 2014 the Blue Waters supercomputer reported a mean time between node failures to be 6.7 hours [49]. Most recently, the Taurus system located in TU Dresden reported a mean time between node failures to be 3.65 hours [62].

This trend raises concerns in the HPC community for MPI applications running on tens of thousands of processes and nodes to fail when facing an increasing number of process and node failures. An MPI application execution can fail on node failures because of a variety of reasons, such as transient faults and Byzantine faults [62]. These underneath faults may not directly make the application execution fail, whereas they can cause a process or node failure to the node where the application is running. The process or node failure can further cause the entire MPI application to fail.

These crucial facts lead to an increasing importance of and challenges for developing efficient and effective fault tolerance designs for scaling HPC systems. There are numerous fault tolerance techniques proposed to protect MPI application execution from system failures. Checkpointing [84, 71, 23, 8, 89], commonly used in HPC applications, is one type of fault tolerance technique that saves application execution states periodically. Checkpointing helps MPI applications to quickly restore application states from the latest checkpoints. The other type of MPI fault tolerance technique focuses on restoring MPI states in the occurrence of MPI process and node failures. Restarting is a baseline solution for restoring MPI states, which immediately restarting an application after execution collapses due to a failure. Later, researchers realize the inefficiency of restarting an application, and propose MPI process recovery mechanisms to restore MPI states in real-time. User-Level Fault Mitigation (ULFM) [19] and Reinit [91, 35, 60] are the two pioneer MPI process recovery frameworks in this effort. ULFM provides extended MPI interfaces to programmers to detect failures and restore MPI states, which enables the execution to continue with the same number of processes or only with the survivor processes. Reinit also supports real-time MPI recovery, but transparently implement detecting failures and fixing MPI states to the MPI runtime.

Although there has been a large bibliography [19, 35, 91, 20, 102, 120, 73, 84, 71, 23, 89] discussing the programming model and prototypes of those MPI recovery approaches, no study has presented an in-depth performance evaluation of them–most previous works either focus on individual aspects of each approach or perform limited scale experiments. Can we design an extensive evaluation framework to fairly compare the two leading MPI recovery approaches? Can we understand the fault tolerance behavior difference in ULFM and Reinit recovery?

Furthermore, there is not a standard paradigm to follow for developing efficient MPI fault tolerance. The traditional practices [19, 35, 91, 20, 102, 120, 73] in MPI fault tolerance either focus on only checkpointing mechanisms or only MPI recovery techniques. Later, researchers realize the efficiency of combining the two aspects to achieve higher

efficiency of MPI fault tolerance. For example, FENIX [57] and CRAFT [134] both design and develop a checkpointing interface that supports data recovery for ULFM shrinking and non-shrinking process recovery. However, they request developers to explicitly manage and redistribute the restored data among survivor processes in case of a non-shrinking recovery. This can easily cause load imbalance problems. Also, they only evaluate their frameworks on two applications, and do not compare their fault tolerance frameworks to other fault tolerance designs. For example, using Reinit for process recovery, and testing different checkpointing interfaces. In conclusion, there is not a structured way in existing works that either benchmark the design and implementation of MPI fault tolerance, or comprehensively compare the performance efficiency of different combinations or configurations of fault tolerance designs. Can we develop a comprehensive evaluation framework which enables an effective comparison of distinct MPI fault tolerance configurations?

1.2 Research Objectives

The dissertation seeks to characterize representative fault tolerance frameworks to research and identify fundamentally new ways to design and build effective and efficient fault tolerance theorems, mechanisms, and tools for HPC by leveraging domain-specific characteristics at both the system- and application-level.

The objectives of this research are multi-folds: (1) to design a code structure model that enables separation of applications into code regions, which enables a divide-and-conquer approach, to have a framework that allows us to do a fine-grained analysis of error propagation and resilience properties, and to propose a methodology and develop an analytical framework that can help reason the natural resilience of code segments; (2) to propose an effective measurement for error resilience on data objects by counting error masking events, which avoids non-deterministic measurement by random fault injection, and to develop a hierarchical error propagation model in order to efficiently model error masking events on data objects; (3) to develop an effective machine learning model that can accurately estimate application error resilience, to avoid the inefficiency by doing fault injection, and to characterize the implicit relationship between program properties and application error resilience; (4) to characterize and compare the efficiency and effectiveness of existing MPI recovery interfaces, and to advance the efficiency and effectiveness of existing MPI recovery with lessons learned from the characterization and comparison; (5) to come up with an MPI benchmark suite aiming at fault tolerance, where a comparison framework is developed, which allows us to effectively compare distinct MPI fault tolerance techniques under the framework.

The dissertation consists of five works. The relationship between the five works is addressed in Figure 1.2. First, we characterize error resilience of HPC applications to transient faults, in which we develop a code structure framework to help understand application natural error resilience. Furthermore, we seek to model error resilience to transient faults from different perspectives, where we model error resilience on data objects, and predict application error resilience using machine learning models. Lastly, we attempt to study application fault tolerance to process/node failures, where we comprehensively evaluate the start-of-the-art MPI global-restart recovery methods, and develop an MPI fault tolerance benchmark suite.

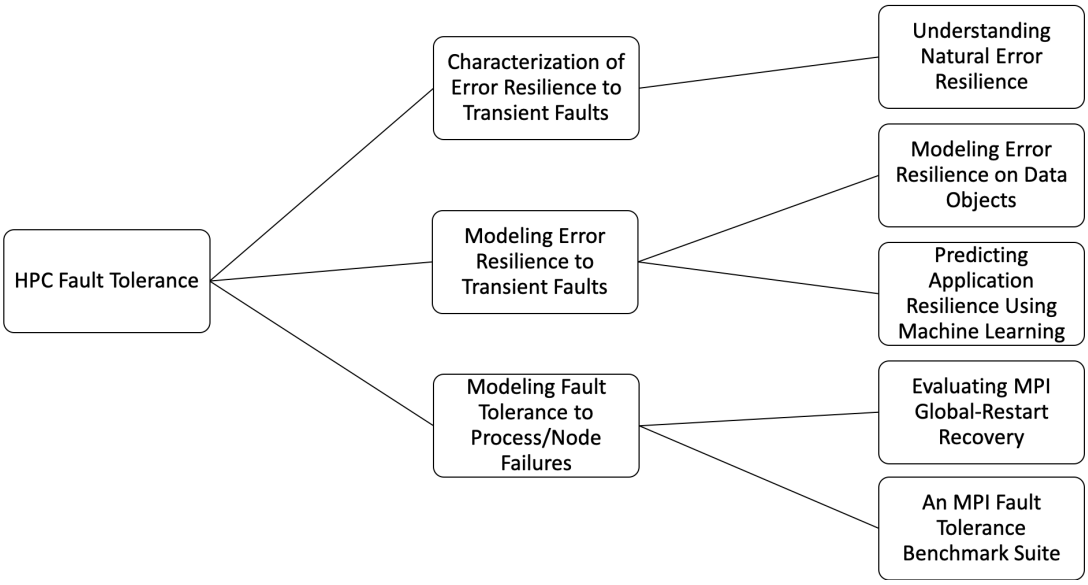


Figure 1.2: The overview diagram of my dissertation works.

1.3 Research Contributions

We summarize the main contributions of this dissertation in this section.

Natural Error Resilience: We present `FlipTracker`, a framework designed to extract resilience computation patterns using fine-grained tracking of error propagation and resilience properties, and we use it to present a set of computation patterns that are responsible for making representative HPC applications naturally resilient to errors. This not only enables a deeper understanding of resilience properties of these codes, but also can guide future application designs towards patterns with natural resilience. The technical details are presented in Chapter 3.

Error Resilience on Data Objects: We introduce a method and a tool (called "MOARD") to model and quantify application resilience to transient faults on data objects. Our method is based on systematically quantifying error masking events caused by application-inherent semantics and program constructs. We use MOARD to study how and why errors in data objects can be tolerated by the application. We perform a comprehensive evaluation and a case study to demonstrate tangible benefits of using MOARD to direct a fault tolerance mechanism to protect data objects. We describe the solution details in Chapter 4.

Prediction of Application Error Resilience: We present `PARIS`, a machine-learning method to predict application resilience that avoids the time-consuming process of random FI and provides higher prediction accuracy than analytical models. PARIS captures the implicit relationship between application characteristics and application resilience, which is difficult to capture using most analytical models. We overcome many technical challenges for feature construction, extraction, and selection to use machine learning in our prediction approach. Our evaluation on 16 HPC benchmarks shows that PARIS achieves high prediction accuracy. PARIS is up to 450x faster than random FI (49x on average). Compared to the state-of-the-art analytical model, PARIS is at least 65% better in terms of accuracy and has comparable execution time on average. We describe

the design details in Chapter 5.

A New Design of MPI Recovery: We present Reinit^{++}, a new design and implementation of the Reinit approach for global-restart recovery, which avoids application re-deployment. We extensively evaluate Reinit^{++} contrasted with the leading MPI fault-tolerance approach of ULFM, implementing global-restart recovery, and the typical practice of restarting an application to derive new insight on performance. Experimentation with three different HPC proxy applications made resilient to withstand process and node failures shows that Reinit^{++} recovers much faster than restarting, up to 6×, or ULFM, up to 3×, and that it scales excellently as the number of MPI processes grows. The implementation details are discussed in Chapter 6.

An MPI Fault Tolerance Benchmark Suite: MPI has been ubiquitously deployed in flagship HPC systems aiming to accelerate distributed scientific applications running on tens of hundreds of processes and compute nodes. Maintaining the correctness and integrity of MPI application execution is critical, especially for these safety-critical scientific applications. Therefore, a collection of effective MPI fault tolerance techniques have been proposed to enable MPI application execution to efficiently resume the application execution and states in the occurrence of system failures. However, there is not a structured way to study and compare different MPI fault tolerance designs, and to guide the selection and development of efficient MPI fault tolerance techniques in distinct scenarios. To solve this problem, we design, develop, and evaluate a benchmark suite `MATCH` to characterize, research, and comprehensively compare different combinations and configurations of MPI fault tolerance designs. Our investigation derives useful findings: (1) Reinit recovery achieves better performance efficiency than ULFM recovery; (2) Reinit recovery is independent of the scaling size and the input problem size, however ULFM recovery is not; (3) using Reinit recovery with FTI checkpointing is a highly efficient fault tolerance design. We give the technical and design details in Chapter 7.

1.4 Organization of the Dissertation

We present the organization of the dissertation below. Chapter 2 describes the technical terms and terminologies used in the dissertation, and discusses the literature related to works in the dissertation. Chapter 3 aims at the characterization of error resilience in HPC applications. Chapter 4 and Chapter 5 model application error resilience in HPC analytically and using machine learning. Chapter 6 and Chapter 7 investigate fault tolerance techniques for the MPI process and node failures. Chapter 8 is the conclusion of the dissertation and thoughts for future work.

Chapter 3 is developed from [65]; Chapter 4 includes content from [63]. Chapter 5 is revised from [64]. Chapter 6 is based on [60]. Chapter 7 derives from a manuscript in submission.

Chapter 2

Background and Literature Survey

2.1 Background

This section introduces the terminologies and technical terms used in the dissertation.

2.1.1 Transient Fault Model

We consider transient faults in computation units of processors. For example, transient faults in the Arithmetic Logic Unit (ALU) and the address computation for loads and stores. We do not consider transient faults in memory components, such as caches, because these components are usually protected by Error Correcting Code (ECC) or parity at the architecture level. Similar assumptions are made in existing work [153, 98].

Furthermore, we consider single bit-flip model, not multiple bit-flip model. Because single bit-flip model is the de-facto fault model commonly adopted by existing work to emulate errors propagated to applications [153, 98, 93]. Despite transient faults can manifest as single and multiple bit-flips in applications, existing studies have demonstrated that multi-bit errors can have a similar impact on the application as single-bit errors [98]. Therefore, we use single bit-flip model in this chapter.

Fault Injection. We use PINFI [153] to perform fault injections into programs. PINFI triggers a single bit-flip into the destination register or memory location of a randomly chosen instruction to emulate the effect of transient faults. The registers or memory locations are chosen as the injection targets by PINFI, because any error in the computation/data paths of the processor shows up in the results of the executed instruction. PINFI's fault model is the same as ours. Comparing with other common fault injection tools (e.g., LLFI [148] and REFINE [61]), PINFI is very accurate and user-friendly. In our study, the number of fault injections is determined by using a statistical approach [95] with the confidence level of 99% and the margin of error 1%.

Fault Manifestation Model. We run fault injection campaigns to measure the application resilience. A fault injection campaign contains many fault injections. In each fault injection, a single-bit error is injected into an input/output operand of an instruction. We classify the outcome, or *manifestation*, of programs corrupted by bit flips into three classes: success, SDC, and interruption:

- **Success**: the execution outcome is the same as outcome of fault-free runs. The execution outcome can also be different from outcome of fault-free runs, but the

execution passes the result verification phase of the application.

- **SDC**: the program outcome is different from the outcome of the fault-free execution, and the execution does not pass the result verification phase of the application.

- **Interruption**: the execution does not reach the end of execution, i.e., it is interrupted in the middle of the execution, because of an exception, crash, or hang.

Rates. To quantify the application resilience in a fault injection campaign, we measure the rate of each of the three classes of manifestations. In particular, we use the formula:

$$\#\text{Manifestations}/N \tag{2.1}$$

where #Manifestations is the number of times a given class of manifestation occurs, and N is the number of fault injections in a fault injection campaign. We consider the rates of success, SDC and interruption as metrics to quantify application resilience. The rates are real numbers between 0.0 and 1.0. Since they are mutually exclusive, the addition of them for a given application is 1.0.

Error Masking. Error masking can happen at the application level and hardware level. The application-level error masking happens because of application inherent semantics and program constructs. The hardware-level error masking happens because a fault does not corrupt the precise semantics of hardware [111].

The key of our error tolerance modeling is the application-level error masking. We particularly study *error masking that happens to individual data objects*. We consider that when an error happens in a data object (*other data objects remain correct before the error happens*) how the error impacts the application outcome correctness. A data object can be an array or other data structures with many data elements. Other than data objects, we do not consider the corruption of other application components (e.g., computing logic). *Hence, we do not aim to model the error tolerance of all application components but focus on data objects*. In addition, we focus on errors happening in data objects and directly consumed by the application. Latent errors in data objects (i.e., the errors not consumed by the application) are not considered because they do not matter to the application outcome correctness.

2.1.2 MPI Failure Recovery Model

MPI failure recovery has multiple modes including *global*, *local*, *backward*, *forward*, *shrinking*, and *non-shrinking*.

Global: The application execution must roll back to a global state to fix a failure.

Local: The application can continue the execution by repairing the failed components such as a failed code block locally without starting over the execution.

Backward: The application execution must go back to a previous state in order to survive a failure.

Forward: The failure can be fixed with the current application state, and the execution can continue.

Non-shrinking: The application manages to bring all failed processes back to resume execution.

Shrinking: The application execution is able to continue with the remaining survivor processes.

The global, backward, non-shrinking recovery best fit into the Bulk Synchronous Parallel (BSP) paradigm of HPC applications. This dissertation focuses on global, backward, non-shrinking recovery.

There are two leading MPI failure recovery frameworks–ULFM and Reinit.

ULFM. User-level Fault Mitigation (ULFM) [19] is an MPI failure recovery framework providing shrinking recovery and non-shrinking recovery. ULFM develops new MPI operations to add fault tolerance functionalities at the application level. These functionalities include fault detection, communicator repairing, and failure recovery. In particular, ULFM leverages the MPI error handler to notify process failures. Once a failure is detected and notified, ULFM uses an operation MPI_Comm_revoke() to revoke processes in the communicator. This operation interrupts communication pending on the communicator at all processes. ULFM then reduces the failed processes using an operation MPI_Comm_shrink(), which also creates a new communicator with survivor processes. ULFM then makes an agreement among processes of the new communicator. The shrinking recovery is done using the above steps. The other recovery mode is non-shrinking recovery. For non-shrinking recovery, ULFM further uses the MPI_Comm_spawn() operation to spawn new processes and create a new communicator. ULFM then uses the MPI_Intercomm_merge() operator to merge the communicator of survivor processes and the communicator of spawned processes, and create a new communicator. We provide an example implementation of ULFM non-shrinking recovery in the Appendix. Please see it for more details.

Reinit. Reinit [35, 92, 60] is an alternative recovery framework designed particularly for global backward non-shrinking recovery. Reinit implements the recovery process into the MPI runtime, which is transparent from users. Therefore, the programming effort of using Reinit is much less than using ULFM. Programmers only need to set a global restarting point; the remaining recovery is done by Reinit. Also, Reinit is much more efficient than ULFM because of running at the MPI runtime level [60].

2.2 Related Work

Resilience Computation Patterns. A limited number of previous studies reveal the existence of resilience patterns [97, 43]; these efforts, however, lack a systematic method to identify these patterns. In [97], Li et al. identify conditional statement and truncation for error masking in GPU programs. In [43], Cook and Zilles identify shift, conditional statement and truncation. Those research efforts manually examine fault tolerance cases, while our work is different in several aspects. First, we introduce a novel *framework and methodology to systematically* identify patterns. For complex applications, manual identification of those patterns is unfeasible. Second, we identify more complex patterns (e.g., DCL and repeated additions). Those new patterns require multiple instructions to take effect. Finding those patterns must be based on a complete picture on error propagation. The existing work identifies patterns based on the analysis of individual instructions without sufficient considerations of interactions between instructions, hence lacking a complete picture to identify patterns.

Error Detector Placement. Existing research uses compiler static and/or dynamic instruction analysis to enable application-level fault tolerance by detecting code vulnerabilities. For example, Pattabiraman et al. use static analysis [119] and a data-dependence analysis [118] to determine the placement of error detectors in applications. Their work determines the critical variables that are likely to propagate errors based on metrics, such as highest dynamic fan-out. Different from us, their work cannot locate resilience patterns.

Visualization. Recently, techniques that allow visualization of corrupted application data across loop iterations and MPI processes have been developed. For example, Calhoun et al. [29] replicate instructions to track and visualize how errors propagate within the application. However, their approach can be expensive when analyzing complex applications. Our approach, based on the abstract code structure model, can accelerate tracking error propagation.

Resilience Metrics. Architectural vulnerability factor (AVF) is a hardware-oriented metric to quantify the probability of an error in a hardware component resulting in incorrect application outcomes. It was first introduced in [18, 111] and then attracted a series of follow-up work. This includes statistical modeling techniques to accelerate AVF estimate [54], online AVF estimation [100], Yu et al. [158] introduce a metric, DVF. DVF captures the effects of application and hardware on error tolerance of data objects. In contrast to AVF and DVF, aDVF is a highly application-oriented metric.

Using Machine Learning to Address Resilience Problems. Recent research starts to use ML to address resilience problems [93, 44, 11, 114, 80, 108, 151]. Mitra et al. [108] build a regression model to predict anomaly output of an application, given a certain combination of input parameters to the application. Laguna et al. [93] train an ML classifier IPAS. IPAS learns which instructions can have a high likelihood of leading to a silent output corruption. IPAS duplicates those instructions to mitigate the effect of silent output corruption. Vishnu et al. [151] use attributes including system and application states to predict whether a multi-bit error will lead to corrupted output. Desh [44] predict node failures by training a recurrent neural network model using system logs. Nie et al. [114] use system characteristics as features to predict the occurrence of GPU errors. PRISM [80] predicts resilience for GPU applications using application properties. However, different from `PARIS`, PRISM focuses on GPU applications, and PRISM does not consider instructions execution order and resilience weights for feature design.

Error Propagation Analysis. Application level error propagation has been widely studied. Li et al. [97] implement a fault injection tool to study error propagation in GPU applications. They also propose `Trident` [98], a three-level error propagation model to predict SDC probabilities of programs. Calhoun et al. [29] study how corruption states change across instructions because of error propagation at the instruction and application variable levels. Ashraf et al. [11] propose an error propagation model to study error propagation for MPI applications. Our work does not focus on error propagation, but includes an N-gram based technique to embed the instruction execution order information into the feature vector to consider the effect of error propagation.

Random Fault Injection. This is the most common method to study application resilience [40, 88, 47, 117, 83, 103]. Typically, application-level fault injection has to be performed many times to ensure statistical significance. Some research prunes unnecessary fault injections to reduce fault injection efforts. Hari et al. [69] and Kaliorakis

et al. [79] explore fault equivalence for selective fault injection by grouping instructions that have the similar effects on program execution at the same static instruction. They further reduce fault injection positions by leveraging the equivalence of intermediate states in execution and instruction-level approximate computing [130, 150]. Although they use instruction grouping, their method is different from ours. They group static instructions at the program level, while we group dynamic instructions based on their functionality and our instruction grouping is independent of the program. Nie et al. [115] prune fault injection sites by only analyzing a subset of threads and a subset of registers that are representative for GPGPU applications. Our work tries to address the inefficiency of using fault injection to study application resilience by circumventing performing fault injections. But the above existing work is complementary to our work for model training.

Data Recovery. Checkpointing [68, 128, 4, 144, 152, 30, 3, 87] is the commonly used approach to restart an MPI application when a failure occurs. Programmers need to have a good sense of the application algorithm and the code structure before they can pinpoint which date objects for checkpointing. On the other hand, writing checkpoints to the file system typically brings at least 20% percent performance overhead. There are many works trying to make checkpointing easier-to-use and to improve checkpointing efficiency.

Hargrove et al. [68] develop a system-level checkpointing library–the Berkeley Lab Checkpoint/Restart (BLCR) library–to run checkpointing at system-level using the Linux kernel. Furthermore, Adam et al. [4], SCR [109], and FTI [15] propose multi-level checkpointing aiming to significantly advance checkpointing efficiency. CRAFT [134] provide a fault tolerance framework that integrates checkpointing to ULFM shrinking and non-shrinking recovery. In this work, we choose FTI for checkpointing for data recovery because the high efficiency and well documenting of FTI. We attempt to integrate and evaluate more checkpointing mechanisms in addition to FTI in future work. Furthermore, different than existing works, we also provide a data dependency analytics tool to aid programmers to identify data objects for checkpointing.

MPI Recovery. ULFM [19, 20] is the leading MPI recovery framework that is in progress with the MPI Fault Tolerance Working Group. ULFM provides new MPI interfaces to remove failed processes and add new processes to communicators, and to agree between processes. ULFM requests programmers to implement shrinking- or non-shrinking recovery using these interfaces. ULFM provides flexibility to programmers, but there is a great effort of learning before programmers can correctly use ULFM interfaces to implement ULFM recovery. A large number of works [102, 120, 73, 84, 71, 23, 89] have explored and extended the applicability of ULFM. Teranishi et al. [147] replace failed processes with spare processes to accelerate ULFM process recovery. Bosilca et al. [21, 22] and Katti et al. [85] propose a series of efficient fault detection mechanisms for ULFM. Fenix [57] provides a user-friendly abstraction layer on top of ULFM. Fenix reduces the effort to implement ULFM recovery, but it does not solve the scalability problems of ULFM reported by previous works [147, 58], also demonstrated in our evaluation.

Reinit [90, 60] is a more efficient solution for global recovery. Reinit hides the process recovery from programmers by implementing it to MPI runtime. Reinit provides only one interface to programmers which sets up the global restart point, protects the target function, and returns the state of spawned and survivor processes. The early

versions [35, 92, 90, 145] of Reinit have limited usage because these versions are not compatible with common job schedulers. Most recently, Georgakoudis et al. [60] fix the design and reimplement Reinit into the OpenMPI runtime.

MPI Fault Tolerance Benchmarking. There have been many benchmark suites [26, 104, 5] developed for performance modeling of programming models using MPI. For example, SKaMPI [124] is an early benchmark suite that evaluates different implementations of MPI. Bureddy et al. [27] develop a benchmark suite to evaluate point-to-point, multi-pair, and collective MPI communication on GPU clusters. Dosanjh et al. [52] propose the first micro benchmark suite to study the multi-threading Remote Memory Access performance in MPI. However, there is not an MPI benchmark suite that focuses on fault tolerance and evaluates fault tolerance designs in MPI. This dissertation proposes a benchmark suite `MATCH` for benchmarking MPI fault tolerance.

Chapter 3

Understanding Natural Error Resilience in HPC Applications

3.1 Introduction

In this chapter, we characterize application natural resilience using common HPC programs and identify six common resilience computation patterns. Examples of such patterns are *dead corrupted variables*, where sets of corrupted temporal variables are not used afterwards, and *repeated additions*, a pattern that amortizes the effect of incorrect data values.

To capture and extract these patterns, however, a new method is required. While some methods exist to inject faults and statistically quantify their manifestation, such as *random fault injection* [34, 28, 96, 97, 138], and to use *program analysis* [69, 130, 119, 118, 29] to track errors on individual instructions, these methods miss the fine-grained information on error propagation as well as the context needed to explain, at a fine granularity, how errors propagate and consequently how natural resilient computations occur. In other words, these approaches do not provide the needed reasoning about how multiple computations work together to make an error disappear or to diminish its impact.

To address the above problems, we design `FlipTracker`, a framework to analytically track error propagation and to provide fine-grained understanding of the propagation and tolerance of errors in HPC applications, and then apply it to a series of representative HPC applications to extract the patterns that provide natural resilience.

Our framework has three key features. *First*, we introduce an application model that partitions the application into code regions. Such a model allows us to build a high-level picture on how an error propagates across code regions, or is tolerated with the combination of multiple code regions. *Second*, using data dependency analysis, we identify the input and output variables of each code region, which allows us to perform isolated fault injections at the entry of code regions to study their resilience in an isolated fashion. Further, it allows us to quickly track how the corrupted values change across code regions as caused by their resilience computation patterns. *Third*, we track how the number of live, yet corrupted locations change within code regions, an approach that reveals resilience patterns that cannot be easily found by traditional high-level fault propagation approaches.

We present two use cases to demonstrate how resilience computation patterns can be used to (1) improve application resilience during programming and (2) predict the degree of application resilience.

In summary, the contributions of this chapter are (1) an abstract code structure model that enables us to reason about the natural resilience properties of code segments; (2) the design of a framework that enables fine-grained and comprehensive analysis of error propagation to capture application natural resilience; (3) an implementation of the framework, `FlipTracker`, using the LLVM compiler and a study of a set of representative HPC programs on which `FlipTracker` is demonstrated; (4) an analysis and formal definition of six resilience computation patterns that we discover in these programs; (5) two use cases that demonstrate the usage of resilience computation patterns.

3.2 Design of `FlipTracker`

In this section, we introduce our method to identify resilience computation patterns.

`FlipTracker` takes as input an HPC program, creates a dynamic execution trace generated using LLVM instrumentation, and then uses our novel analysis techniques to provide a fine-grained representation of error propagation and error tolerance. This analysis allows us to easily identify the resilience computation patterns that may exist in the program, possibly in different code regions of the program.

Our method is based on a top-level characterization of HPC applications, which we then use to track error propagation and tolerance at a low level. In particular, we model an application as a chain of code regions, which work together to produce the final result of the application. Each of these code regions can have *input*, *output*, and *internal* variables. Errors can propagate at any point in time to any of these variables.

Based on the above application model, we build a dynamic data dependency graph (DDDG) from an instruction trace collected at runtime that allows us to check the value variation of corrupted variables across code region instances (i.e., the top level). Using the DDDG, we then build a table, which we call the *alive corrupted locations* (ACL) table, that keeps track of the corrupted locations for each dynamic instruction. This table allows us to examine the variation of the number of alive, corrupted variables to identify fault tolerance at the instruction level (i.e., the bottom level). In the next sections we give more details of each of these steps (see Figure 4.1).

3.2.1 Application Code Region Model

We characterize HPC applications as sets of iterative structures or loops. In an HPC application, a main computation loop usually dominates the application execution time. Within this main loop, there are a number of inner loops that are typically used to update large data objects (e.g., a mesh structure in computational fluid dynamics), and iterative computations are performed to compute properties of these objects, such as energy of particles. Figure 3.1 shows an example of such loop program abstractions corresponding to CG [12].

Code Regions. Since HPC applications are typically composed of combinations of loops, we model an application as a chain of *code regions* delineated by loop structures (Step (a) in Figure 4.1). A code region can be either a loop or any block of code

```
1 static void conj_grad() { //called from the main loop
2     ...
3     for () { //a first level inner loop
4         for () { //a second level inner loop
5             for () {...} //a third level inner loop
6         }
7     }
8     for () {...} //a first level inner loop
9 }
```

Figure 3.1: An example HPC application (CG) with iterative structures.

between two neighboring loops. An application can have multi-level nested loops. We allow the user to decide at which loop level, code regions are defined. Note that code regions defined at different loop levels *only affect* the analysis time (not the analysis correctness) to identify resilient code regions and patterns. Code regions defined at the level of innermost loop tend to be small and easy for fine-grained instruction level analysis. However, we can have many of such small code regions, which increases our exploration space. On the other hand, code regions defined at the level of outermost loop tend to be large and we have a smaller exploration space of code regions, but it would be time-consuming for fine-grained instruction level analysis. In our work, we define each of the first-level inner loops as a code region.

Code Region Variables. Given a code region, we classify the variables within the code region as *input* variables, *output* variables, and *internal* variables. Input variables are those that are declared outside of the code region and referenced in the code region. Output variables are those that are written in the code region and read after the code region. Other variables that the code region writes to or reads from are internal variables. A code region can have many dynamic instances, each of which corresponds to one invocation of the code region at runtime. The values of input, output, and internal variables can vary across multiple instances of a code region.

Rationale Behind the Model. Our loop-based model follows the natural way in which HPC programs are coded and analyzed; HPC programs are composed of a handful of high-level loops where the program spends most of its time. Our loop-based model also enables a divide-and-conquer approach, where we can identify application subcomponents that may or may not have resilience patterns. For example, in the error propagation analysis, if the input variables of a code region are not corrupted, one can infer that the region is not impacted by an error and we can skip propagation analysis on it.

3.2.2 Tracing Code Region Data

The DDDG allows us to identify input, output, and internal variables of a code region. We construct a DDDG for each code region from a dynamic instruction trace of the application using an algorithm inspired by the construction of a program dependence graph [56], except that our graph is dynamic rather than static: vertices are the values of variables obtained from registers or memory; edges are operations transforming input values into output values of variables. Using the DDDG as a code region representation, we identify the input and output variables of the code region: root nodes represent inputs

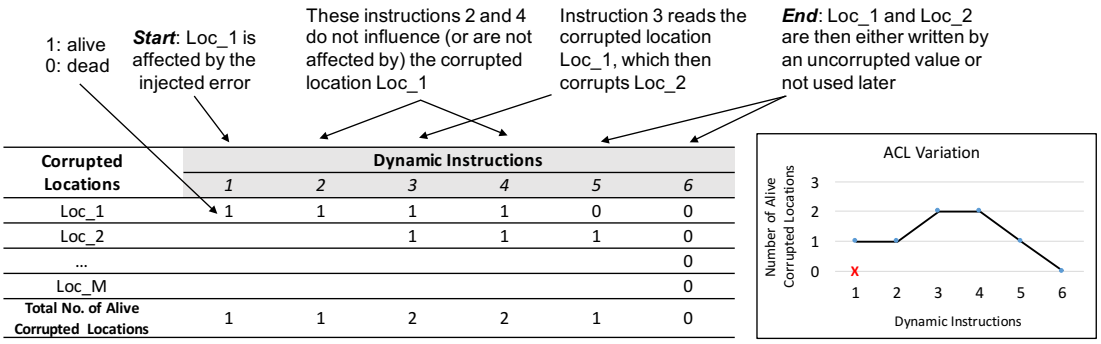


Figure 3.2: An example of the ACL table.

and leaf nodes represent outputs. Other nodes are internals.

Within the corresponding DDDG of each code region, we inject an error into either the input, output, or internal variables (Steps (b)–(c) in Figure 4.1). A DDDG allows us to compare data propagations in regions with and without fault occurrence, which allows us to detect control flow divergence by comparing operations. Further, the values of variables are embedded in the DDDG, which helps us to track how specific variables change their values across operations; such value change reveals whether, how, and where fault tolerance occurs.

3.2.3 Analyzing Corrupted Variables

We identify variables that, once corrupted, return to their non-corrupted state and in which dynamic instruction. This is key in identifying resilience computation patterns since we need to identify the point in time where the error is tolerated and its location in the code region (Step (d) in Figure 4.1).

Using the DDDG, our analysis of corrupted variables gives us a low-level representation in terms of *instructions* of how data propagates in the code region. Since program abstractions, such as variables, are not explicitly represented at this level, we need a different way of tracking variable values. We introduce a method that tracks *alive corrupted locations*, discussed as follows. In the following discussion, since a variable value can be either in a register location or in a memory location, we use the term *location* to cover both options.

Alive Corrupted Locations. Traversing through the collected instruction trace, we use the DDDG to build and dynamically update a table of the *alive corrupted locations*, or ACL. Generally speaking, the ACL table stores the number of alive, corrupted locations after each dynamic instruction. We call a location "alive" if the value in that location will be referenced again in the remainder of the computation.

Each row of the table shows whether a specific location is alive or not after each dynamic instruction, as instructions are encountered in the trace. Each column of the table shows, for a specific corrupted location, whether it is alive or not *after* a dynamic instruction. Based on the column information, we can determine the total number of alive, corrupted locations after each traced instruction.

Figure 3.2 gives an example of the ACL table. Each table element has a value of 1 or 0, which indicates whether a corrupted location after a specific dynamic instruction

is alive or not. We use the first row as an example to explain the table. The location Loc_1 is corrupted by a fault after instruction 1. Loc_1 then becomes an alive, corrupted location. Next, Loc_1 remains alive until instruction 5 where the location is updated and the fault in the location is overwritten by a clean value. The number of alive, corrupted locations are counted after each dynamic instruction, shown in the last row of the table.

3.2.4 Identifying Resilience Patterns from Code Regions

As we traverse the instruction trace, the DDDG and ACL table contain the necessary information to detect resilient code regions. Resilience patterns are extracted from them.

When the DDDG is used to identify resilient code regions, we compare the values of input and output locations in a DDDG between faulty and fault-free runs. An input location can be corrupted *directly*—an error was directly injected into the location—or *indirectly*—an error was injected in a previous code region, but the error propagates to the input location of the code region in question. Given a code region, there are two possible cases when fault tolerance occurs:

- **Case 1:** the value of any input location in the code region's DDDG in a faulty run is incorrect (with respect to the DDDG from a matching fault-free run), i.e., there is at least one corrupted input location; however, the values of all output locations are correct.
- **Case 2:** at least one of the input locations and one of the output locations in a faulty run are incorrect (with respect to the DDDG from a matching fault-free run), but the error magnitude in at least one corrupted input or output location becomes smaller after the code region instance. The error magnitude is defined as

$$error\_magnitude = \frac{|value_{correct} - value_{incorrect}|}{|value_{correct}|}. \tag{3.1}$$

In Case 1, it is reasonable to infer that the code region in question has natural fault tolerance—the corruption of the input location is directly masked within the code region, and does not impact the output correctness.

In Case 2, the error still exists, i.e., there is some amount of error in the code region locations; however, the impact of the error, measured by its magnitude in the input or output locations, becomes smaller, as a function of the code region. This means that the target code region may result in an application outcome that is numerically different from that of the fault-free executions. However, when such a different outcome passes the application verification and is acceptable as a valid result, we say that Case 2 has fault tolerance.

When the ACL is used to identify resilient code regions, the algorithm to detect resilience patterns given an ACL is as follows. We identify first if in any column, an alive corrupted location becomes dead for a given instruction i, where $i < N$ and N is the last instruction before the application outputs its result. If this occurs, we mark i as a potential member of resilience computation patterns. In Figure 3.2, the instruction 5 consuming the location Loc_1 is a potential member of resilience computation patterns. Once all of such instructions are found, we identify their source code locations (file and line of code) and provide them to the user for further analysis.

3.3 Implementation

We implement `FlipTracker` as a two-step process: first we use a parallel tracer built on top of LLVM (in particular, LLVM-3.4) to extract the instruction traces, and then use these traces to dynamically generate and update the DDDGs and the matching ACL tables. We do this for both fault-free runs as well as faulty runs.

3.3.1 Parallel Tracing

`FlipTracker` uses an LLVM instrumentation tool, LLVM-Tracer [136], to generate a dynamic instruction trace. In this trace we store metadata for each instruction, such as the the instruction type, names of registers, and operand values. In our case, *instructions* refer to LLVM instructions, which are generated at the intermediate representation (IR) of the program and instrumented by LLVM-Tracer. This approach does not support MPI programs out-of-the-box, which we need to support our HPC workloads. Thus we extend LLVM-Tracer to instrument Message Passing Interface (MPI) programs, so that traces are saved into a file for each MPI process.

Since trace generation is a per-process task, no synchronization is required to generate and save per-process traces into different files. Note also that, in our study, LLVM-Tracer only instruments program instructions—instructions from the MPI runtime are not instrumented as we expect that most errors arise from application computations. This however, is not a limitation per se—our approach can easily be directed to also instrument instructions in any parallel runtime. Furthermore, our current implementation can identify errors that propagate through MPI communications and then happen in computation, even though we do not instrument MPI runtime.

Trace Splitting. Traces for an HPC program can be quite large for processing. Although there is a number of approaches that handle the problem of large traces (e.g., trace compression [78, 116]), we take a simple approach that splits a trace into smaller pieces. Each of small pieces corresponds to an instance of a code region, which reduces the scope for each analysis and further allows us to parallelize the analysis.

3.3.2 DDDG Generation and Usage

Once the trace is generated, `FlipTracker` takes the dynamic trace as input, and generates a DDDG by examining the data dependency of the operands in each operation. Our technique is based on the work of Holewinski et al. [72], who proposed a methodology to generate DDDG from a dynamic trace. The generated DDDG is then used to identify the input, internal, and output locations for the code region instance using Graphviz [59]. The DDDG is also used to determine corrupted locations by dynamically building the ACL table.

ACL Table Generation. The algorithm to generate an ACL table is motivated by dynamic taint analysis in the security research [113, 7, 160], which focuses on computations affected by contaminated sources. The difference between taint analysis and our approach is that we exclude tainted locations that are never used as well as those that are overwritten by an uncorrupted value from the untainted location set. In other words, we only consider alive corrupted locations in application execution. We use a DDDG to acquire the dynamic data dependence to track the error propagation, and, simultane-

ously, we count the number of alive corrupted locations after each dynamic instruction in the input trace.

3.3.3 Fault Injection and Statistical Significance

We implement a fault injection framework based on FlipIt [28], which allows us to inject a bit flip in the user-specified population of instructions and operands. Injections are performed randomly into input and internal locations of code region instances. Our fault injection uses a uniformly distributed fault model, similar to [61, 95]. Given an input or output location for a code region instance, we calculate the number of fault injection sites by analyzing the dynamic LLVM instruction trace. Then, we follow the statistical approach in [95] to calculate the number of fault injection tests for a target at 95% confidence level and 3% margin of error.

3.4 Evaluation

Table 3.1: Resilience computation patterns in code regions of the HPC programs. DCL, RA, DO represent dead corrupted locations, repeated additions and data overwriting, respectively.

Program	**Code region**	**Line No.**	**#instr in an iteration**	**Pattern Found?**	**DCL**	**RA**	**CS**	**Shifting**	**Trunc**	**DO**
CG	cg_a	434-439	21017	NO						
	cg_b	440-453	14002	YES		√				√
	cg_c	454-460	31755757	YES			√			√
	cg_d	461-574	1196022	NO						
	cg_e	575-584	18202	NO						
MG	mg_a	425-429	606145	YES			√			√
	mg_b	430-437	719	YES	√	√				
	mg_c	438-456	1019509	YES	√					√
	mg_d	457-462	3313305	YES	√		√			√
$KMEANS$	k_a	131-142	1647	NO						
	k_b	144-153	62	NO						
	k_c	156-187	2185944	YES			√			√
	k_d	190-194	36	YES	√					√
IS	is_a	435-472	792630	NO						
	is_b	473-478	983040	YES				√	√	
	is_c	500-638	741367	YES	√					√
$LULESH$	l_a	2652-2693	297376	YES	√				√	√

We apply `FlipTracker` to representative HPC programs to study their resilience properties and ultimately to extract naturally resilient patterns that other programs can use.

3.4.1 Experimental Setup

We use ten representative HPC programs in our experiments, including eight HPC benchmarks (CG, MG, IS, LU, BT, SP, DC, and FT from the NAS Parallel Benchmarks in C [12, 133] with input Class S), an HPC proxy application (LULESH [82] with input "-s 3"), and a benchmark from the machine learning domain (KMEANS from the

Rodinia benchmark suite [36] with input “100.txt"). We run experiments on an HPC cluster having 3,018 nodes. Each node is equipped of two Intel Xeon E5-2695 CPUs, and has 36 cores and 128 GB shared memory.

Trace Partitioning and Code Region Selection. HPC programs can have several static loop structures, and depending on program input, each static loop can generate several dynamic instances. To keep the number of loop instances manageable for analysis, we focus on high-level loop structures. Particularly, we define a code region as a section of the program that is either (a) a first-level inner loop (if there is any inner loop), or (b) a code block between two neighbor inner loops.

We list the code regions that we analyzed and their corresponding line numbers and the number of instructions within one iteration of the main loop in Table 3.1.

3.4.2 Parallel Tracing Overhead

We measure the overhead of trace gathering for MPI programs to study the feasibility of our approach. Figure 3.3 shows that our approach incurs modest overhead: 45% on average when using 64 processes on 8 nodes, comparing to an uninstrumented baseline. It is therefore feasible to gather traces at small/medium scales. For large scales, one can selectively collect traces for individual functions or use techniques such as [41]. We leave the challenge of efficiently gathering traces at very large scale for future work.

Since the resilience computation patterns that we are interested in occur in the computation code regions of the program (not in the communication part), we focus on the single process where the fault is injected.

Nondeterminism. MPI nondeterminism can bring difficulty to match code regions between faulty and fault-free runs. While in many MPI programs, nondeterminism can be controlled by eliminating application sources of nondeterminism, such as calls to `rand()` and/or `time()`, in other programs this is difficult because of nondeterminism introduced by MPI point-to-point communication patterns. To address these applications, we rely on record-and-replay tools [131, 157], on which a fault-free run is recorded and it is then replayed in all subsequent faulty executions.

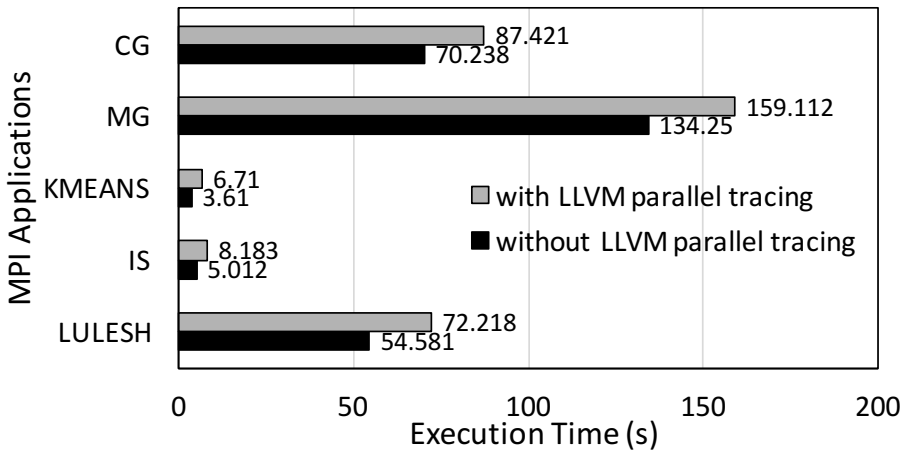
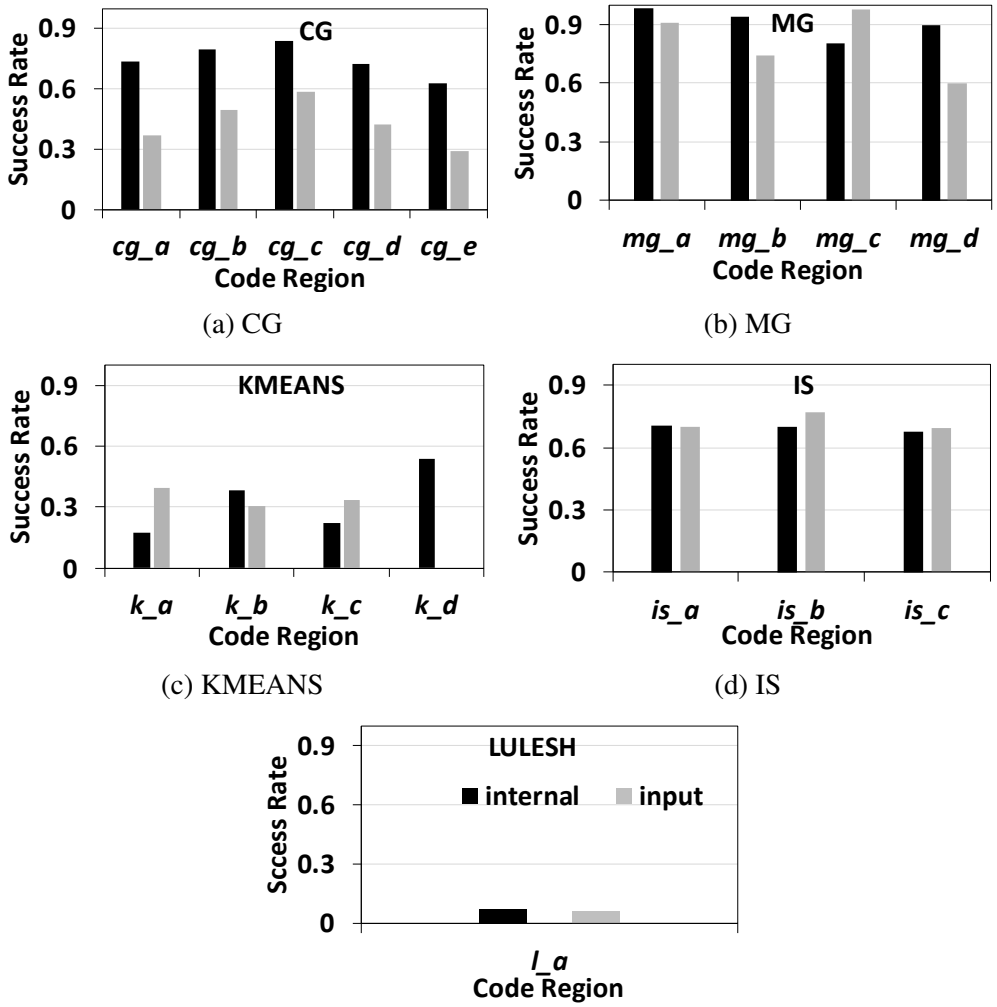


Figure 3.3: LLVM parallel tracing performance (64 processes on 8 nodes)

3.4.3 Code Region Fault Injection Results

We inject faults in input or internal locations of code regions and measure success rate. We perform experiments in two dimensions: (a) across code regions in a given iteration (See "per-code-region" results); (b) in a given code region across all iterations (See "per-iteration" results).

Per-Code-Region Results. Since different code regions could have different numbers of instances, to be consistent, we perform the analysis on the first instance of each code region, i.e., in the iteration 0 of the main loop (see Figure 3.4).



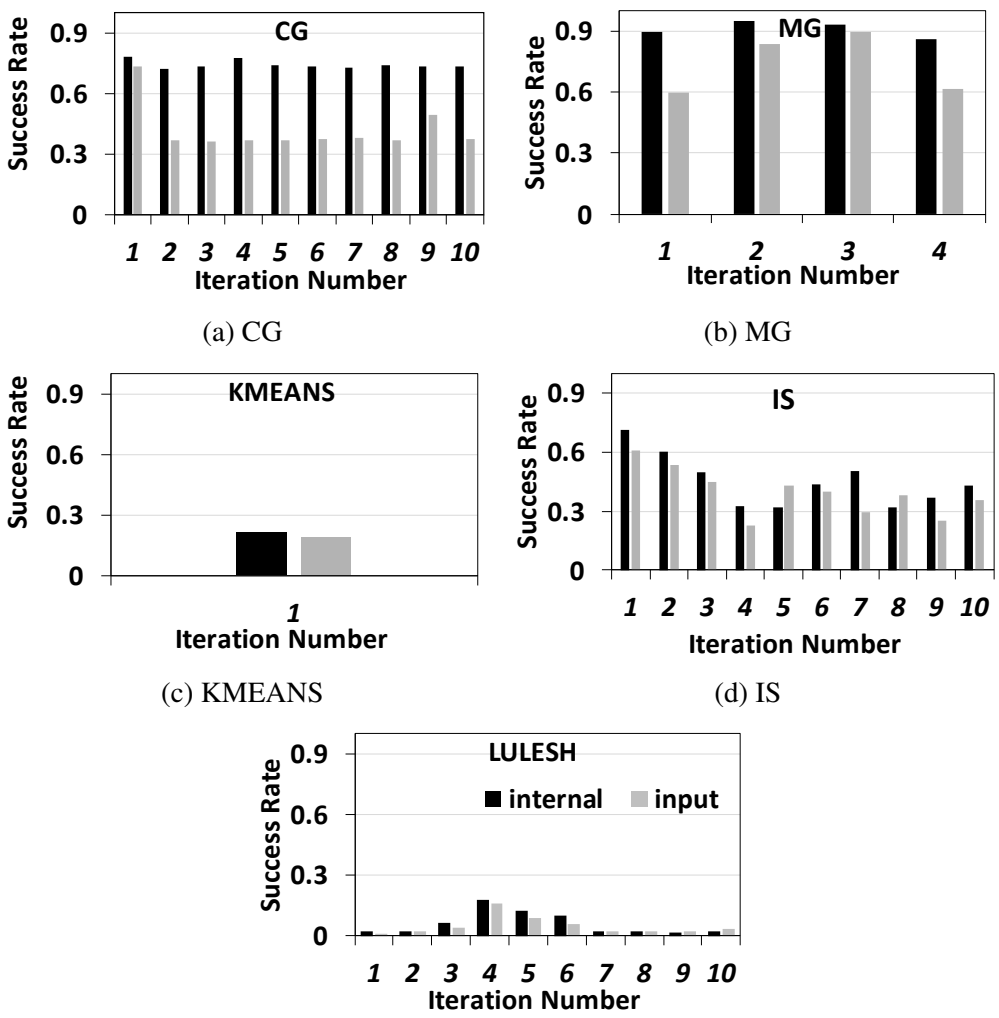
(a) CG (b) MG

(c) KMEANS (d) IS

(e) LULESH

Figure 3.4: Fault injection results for code region instances at iteration 0.

In KMEANS we find that, for faults on internal locations the code region k_d is more resilient than others because many memory *free* operations free temporal corrupted locations, while for faults on input locations, many segmentation faults cause almost zero success rate. We find a relatively high success rate in MG—we find cases of repeated addition and dead corrupted location patterns that account for the fault tolerance (Section 3.5 explains these patterns in details). In IS we find that a bit-shift operation that occurs on input locations masks faults in the is_b code region, which increases



(a) CG

(b) MG

(c) KMEANS

(d) IS

(e) LULESH

Figure 3.5: Fault injection results for individual iterations of the main loop.

its success rate. In CG, we find two code regions (b and c) that have higher success rates than others because the error magnitudes in variables (particularly `p[]`) become smaller due to a computation pattern that repeatedly adds values. In LULESH, there is only one code region—faults frequently cause application crashes, which explains the low success rate.

Per-Iteration Results. We focus on a single code region and examine its fault tolerance on several loop iterations. In particular, we treat the main loop of each program as a single code region and each iteration of the main loop as one instance of the code region. Figure 3.5 shows the results. We find that the success rates of different iterations can be similar. MG (internal locations) and CG exemplify this conclusion. The success rates over multiple iterations can also be very different, e.g., in IS and LULESH. After examining the DDDGs, we find that control flow differences between the iterations of the main loop are the main reason accounting for this difference.

3.5 Resilience Computation Patterns

We present a formal description of the resilience computation patterns. Table 3.1 summarizes them in applications.

Pattern 1: Dead Corrupted Locations (DCL)
In this pattern, the values of several corrupted input locations are *aggregated* into fewer output locations, with aggregations being a combination of multiple operations (e.g., additions and multiplications). While the errors in the corrupted input locations can propagate to one (or a few) locations, many of these corrupted input locations are not used anymore (they become dead locations) and the total number of corrupted locations decreases.

We frequently find Pattern 1 in LULESH. Figure 3.7 shows the code excerpt extracted from LULESH that accounts for the decrease of the number of alive corrupted locations within the routine *LagrangeNodal* (see ① and ② in Figure 3.6). The array $hourgram[][]$ is a temporal corrupted location that is dead after the sample code snippet. The error has propagated to its elements before the example code. Although the error propagates from $hourgram$ to temporal variables $hxx[]$, which are then aggregated into $hgfz[]$, the number of alive, corrupted variables decreases since the corrupted elements of $hourgram[][]$ become dead after this code. We also find this pattern in the MG code.

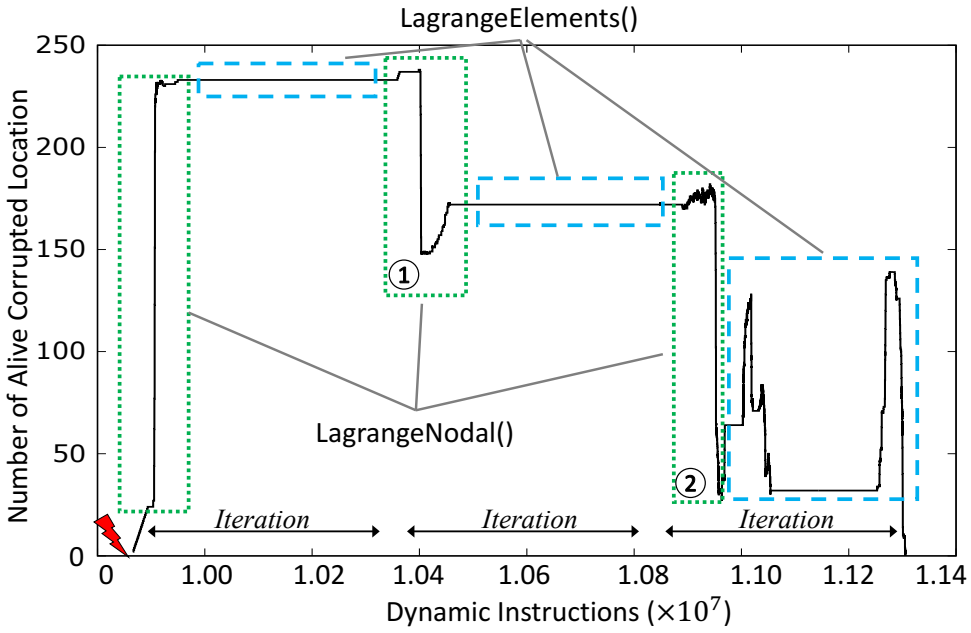


Figure 3.6: A real case of ACL table. It shows the number of ACL-s in LULESH after a fault is injected into the last third iteration of the main loop.

Pattern 2: Repeated Additions
In this pattern, the value of a corrupted location is repeatedly added by other correct values. Those correct values amortize the effect of the incorrect value. This pattern does not necessarily cause a decrease of alive, corrupted locations (as in Pattern 1), but over time the corrupted value approaches the correct value such that the application execution can be successful.

```
1  for(Index_t i = 0; i < 4; i++){
2    hxx[i] = hourgam[0][i]*xd[0]+hourgam[1][i]*xd[1]+
3             hourgam[2][i]*xd[2]+hourgam[3][i]*xd[3]+
4             hourgam[4][i]*xd[4]+hourgam[5][i]*xd[5]+
5             hourgam[6][i]*xd[6]+hourgam[7][i]*xd[7];
6  }
7  ...
8  for(Index_t i = 0; i < 8; i++) {
9        hgfz[i] = coefficient*
10                 (hourgam[i][0]*hxx[0]+hourgam[i][1]*hxx[1] +
11                  hourgam[i][2]*hxx[2]+hourgam[i][3]*hxx[3]);
12 }
```

Figure 3.7: Example of the Dead Corrupted Locations in LULESH

```
1  for (i3 = 1; i3 < n3 1; i3++) {
2      for (i2 = 1; i2 < n2 1; i2++) {
3        ...
4        for (i1 = 1; i1 < n1 1; i1++) {
5          u[i3][i2][i1] = u[i3][i2][i1]
6                +c[0]*r[i3][i2][i1]
7                +c[1]*(r[i3][i2][i1 1]+r[i3][i2][i1+1]
8                                      +r1[i1])
9                +c[2]*(r2[i1]+r1[i1 1]+r1[i1+1]);
10 } } }
```

Figure 3.8: Example of the Repeated Additions pattern in MG

Table 3.2: The repeated additions pattern takes effect in MG

	original value	**corrupted value**	**error magnitude**
itr_1	0	0.000000059604645	∞
itr_2	-0.004373951680278	-0.004373951059397	6.20880999391282E-10
itr_3	-0.004816104396391	-0.004816104262613	1.33777999962448E-10
itr_4	-0.004664456032917	-0.004664455968072	6.48450000292899E-11

We observe Pattern 2 in the iterative solvers MG and CG. Figure 3.8 shows a code excerpt covering this pattern in MG. Here, we inject a fault in an element of the array u and then the array element $u[i3][i2][i1]$ is added with new data values (Lines 6-9). This code is repeatedly executed in the main computation routine ($mg3P$). As a result, the array element $u[i3][i2][i1]$ is repeatedly added along with new data values.

We examine the value of the array element ($u[10][10][10]$) where a single bit-flip happens on the 40th bit in the first invocation of the function $mg3P$. This function is iteratively called four times. We examine error magnitude (as defined in Equation 3.1, recalling that error magnitude is the relative error of a faulty value). Table 3.2 shows that the error magnitude becomes increasingly smaller as $mg3P$ is repeatedly called, reducing the effect of data corruption. Note that although the error magnitude at the second invocation of $mg3P$ is very small, it is still not acceptable for the verification phase of MG. However, as the corrupted value is closer to the correct value at the fourth invocation of $mg3P$, the corrupted value is acceptable by MG and regarded as a correct solution.

```
1 /*Determine the number of keys in each bucket*/
2   for( i=0; i<NUM_KEYS; i++ )
3     bucket_size[key_array[i] >> shift]++;
```

Figure 3.10: Example of the shifting pattern in IS.

```
1 /* find cluster center id with min dist to pt */
2 for (i=0; i<npts; i++) {
3    float dist;
4    dist = euclid_dist_2(pt, pts[i], nfeatures);
5    if (dist < min_dist) {
6       min_dist = dist;
7       index = i;
8    }
9 }
```

Figure 3.9: Example of the Conditional Statement pattern in KMEANS

Pattern 3: Conditional Statements
In this pattern, a conditional statement such as an *if* condition, which tolerates a fault as long as the result of the statement in a faulty case remains the same (true/false) as in a fault-free case, consequently avoiding a control-flow divergence that otherwise could have occurred. The conditional statement can cause a decrease in the number of alive corrupted locations.

Although Pattern 3 is simple, it can become a major reason for fault tolerance in applications. KMEANS exemplifies this case: Figure 3.9 shows a code segment where a condition statement (Line 5) plays a major role to tolerate faults in the array *feature*. In essence, the code tries to find the minimum distance between a target data point and the center data point of each cluster based on the feature values of data points. This conditional statement tolerates errors that happen in the array *feature*, which takes most of the memory footprint of KMEANS. As long as the code segment can find the correct cluster with the minimum distance to the target point, the application outcome remains correct.

Besides the above example, we often find Pattern 3 in the program verification phases of MG and CG, where the final computation result is compared with a threshold to determine the result validity and/or to terminate execution.

Pattern 4: Shifting
In this pattern, bits are lost due to bit shifting operations. If the lost bits are corrupted, fault tolerance occurs and we say that the pattern completely masks (or eliminates) the faulty bit.

We find Pattern 4 in IS—we show an example in Figure 3.10. IS is a benchmark that implements bucket sorting for input integers (called "keys" in the benchmark). The input integers are placed into multiple buckets based on their significant bits. To decide into which bucket a key will be placed, IS applies a shift operation on the key (Line 3 in Figure 3.10). If the data is corrupted in the least significant bits of the key, the shift operations can still correctly place the key into the corresponding bucket, hence tolerating faults in the key.

Pattern 5: Data Truncation
In this pattern, corrupted data is not presented to the user when used as a final result, or corrupted data is truncated.

We find Pattern 5 in LULESH, where in its last execution phase the computation results of a *double* data type are reported in "%12.6e" format (using the `printf` C function). In this format, the mantissa of the computation result is partially cut-off and not fully presented to the user; thus if the cut-off mantissa is corrupted by a fault, the erroneous value will not be seen by the user.

Pattern 6: Data Overwriting
In this pattern, corrupted data is overwritten by a correct value, and the data corruption is consequently eliminated.

We find Pattern 6 in all benchmarks, as it is commonly found in the output of many instructions. This occurs in particular when the value of a corrupted location is overwritten by an instruction that generates a clean uncorrupted value.

Discussion. The effectiveness of some patterns (repeated additions, conditional statement, shifting, and data truncation) depends on the program input. For example, the effectiveness of the shifting pattern is dependent on the number of shifted bits—the more bits are shifted, the more random bit-flip errors can be tolerated. This is different from software design patterns that are general and independent of program input.

3.6 Case Studies

Resilience computation patterns have many potential uses. We give two use cases. Here, whenever we use fault injection, we use 99% confidence level and 1% margin of error to decide the number of fault injection tests based on [95].

Table 3.3: Results after applying resilience patterns to CG.

Resi. Pattern Applied	App. Resi.	Exe time (s)/Average (s)
None	0.59	158.659-159.468 / 159.010
DCL and overwrt.	0.78	158.859-159.457 / 159.167
Truncation	0.614	158.605-159.338 / 158.835
All together	0.782	158.574-159.457 / 158.859

3.6.1 Use Case 1: Resilience-Aware Application Design

We apply resilience patterns to the CG benchmark, aiming to improve its resilience. We successfully apply three patterns: **dead corrupted location** (DCL), **data overwriting**, and **truncation**. The results are shown in Table 3.3, where the first column shows the resilience pattern(s) applied; the second column is the application resilience—the success rate measured by doing fault injection; the third column is the execution time for one run with or without applying resilience pattern(s). We report the average execution time for 20 runs in Table 3.3.

Figure 3.11 and Figure 3.12 show two code excerpts extracted from CG, where dead corrupted location, data overwriting and truncation are applied, respectively. For

the case of dead corrupted location and data overwriting, the original code is shown in Figure 3.11(a) and the new code is shown in Figure 3.11(b) (we include some comments to explain the difference). In particular, we use two temporal arrays v_tmp and iv_tmp to replace two global arrays v and iv. We then copy values in the arrays v_tmp and iv_tmp back to the arrays v and iv after the computation.

To apply DCL and data overwriting, we introduce two temporal arrays at the beginning of $sprnvc()$ to replace two global arrays $v[]$ and $iv[]$ referenced in $sprnvc()$ (see Figure 12). Furthermore, to ensure the program correctness, the updated values of the two temporal arrays are copied back to $v[]$ and $iv[]$ at the end of $sprnvc()$. Because of the copy-back, errors occurring in $v[]$ and $iv[]$ during the execution of $sprnvc()$ can be overwritten. Moreover, errors that might occur in the two temporal arrays become dead (not accumulated as in the global arrays), after the copy-back. Overall, we improve application resilience by 32.2% with less than 0.1% performance loss (caused by a small amount of data movement).

Figure 3.12 shows how we apply the truncation. In particular, we replace 64-bit floating-point multiplications to 32-bit integer multiplications (see Lines 11-12 in Figure 3.12.b). To apply the truncation pattern, we select 10 iterations (340-350th iterations) of a loop within the function $conj\_grad()$, which is used to calculate $p \cdot q$ (see Figure 13). We replace 64-bit floating-point multiplications with 32-bit integer multiplications (particularly lines 508-510 in the source code). After applying the pattern, the precision loss (64 bit vs. 32 bit) does not affect the correctness of the final output. The reason is as follows. As an iterative solver, CG gradually averages out the precision loss across iterations. Furthermore, CG uses a conditional statement that compares the CG output with a threshold to verify the output correctness. Such conditional statement can further tolerate the precision loss. Table 3.3 shows that we improve application resilience by 4.1% with no performance loss. We apply the three patterns together and improve the application resilience by a total of **32.5%** with less than 0.1% performance loss.

3.6.2 Use Case 2: Predicting Application Resilience

The current common practice to quantify the resilience of an application is to use random fault injection. However, random fault injection misses the application context that can explain how errors propagate and consequently are tolerated. In this case study, we are exploring a way alternative to random fault injection to quantify application resilience. Since resilience computation patterns explain application resilience, we may estimate the resilience of an application by counting the number of instances of such patterns in the application. This approach can quantify the contribution of each resilience pattern to application resilience, which demonstrates the effectiveness of resilience patterns.

Model Construction. We build a Bayesian multivariate linear regression model [107] to predict the resilience (i.e., success rate) of an application. The model uses the number of pattern instances for each resilience computation pattern as input, and outputs a single value P_{suc_rate}, the predicted success rate. We model the above idea as follows:

```
1  static void sprnvc(int n, int nz, int nn1, double v[], int iv[]){
2    int nzv, ii, i;
3    double vecelt, vecloc;
4    nzv = 0;
5    while (nzv < nz) {
6      vecelt = randlc(&tran, amult);
7      vecloc = randlc(&tran, amult);
8      i = icnvrt(vecloc, nn1) + 1;
9      if (i > n) continue;
10     logical was_gen = false;
11     for (ii = 0; ii < nzv; ii++) {
12       if (iv[ii] == i) {
13         was_gen = true;
14         break;
15       }
16     }
17     if (was_gen) continue;
18     v[nzv] = vecelt;
19     iv[nzv] = i;
20     nzv = nzv + 1;
21   }
22 }
```

(a)

```
1  static void sprnvc(int n, int nz, int nn1, double v[], int iv[]){
2    int nzv, ii, i;
3    double vecelt, vecloc;
4    double v_tmp[NONZER+1]; //define a temp array
5    int iv_tmp[NONZER+1];  //define a temp array
6    for(i=0;i<=NONZER;i++){
7      v_tmp[i] = v[i];  //initialization
8      iv_tmp[i] = iv[i]; //initialization
9    }
10   nzv = 0;
11   while (nzv < nz) {
12     vecelt = randlc(&tran, amult);
13     vecloc = randlc(&tran, amult);
14     i = icnvrt(vecloc, nn1) + 1;
15     if (i > n) continue;
16     logical was_gen = false;
17     for (ii = 0; ii < nzv; ii++) {
18       if (iv_tmp[ii] == i) { //replace iv with iv_tmp
19         was_gen = true;
20         break;
21       }
22     }
23     if (was_gen) continue;
24     v_tmp[nzv] = vecelt;  //replace v with v_tmp
25     iv_tmp[nzv] = i;  //replace iv with iv_tmp
26     nzv = nzv + 1;
27   }
28   for(i=0;i<=NONZER;i++){
29     v[i] = v_tmp[i];  //copy back
30     iv[i] = iv_tmp[i]; //copy back
31   }
32 }
```

(b)

Figure 3.11: A code excerpt from the function *sprnvc*() in CG for the Use Case 1. (a) shows the original code excerpt before patterns are applied; (b) shows the code excerpt when dead corrupted location and data overwriting are applied.

```
1  static void conj_grad(int colidx[],
2                        ...
3                        double p[],
4                        double q[])
5  {
6    ...
7    // Obtain p.q
8    d = 0.0;
9    for (j = 0; j < lastcol    firstcol + 1; j++) {
10
11     d = d + p[j]*q[j];
12
13   }
14   ...
15 }
```

(a)

```
1  static void conj_grad(int colidx[],
2                        ...
3                        double p[],
4                        double q[])
5  {
6    ...
7    // Obtain p.q
8    d = 0.0;
9    for (j = 0; j < lastcol    firstcol + 1; j++) {
10     if(j<=350&&j>=340){
11       int tmp = p[j]; // truncation
12       int tmp1 = q[j]; // truncation
13       d = d + tmp*tmp1;
14     }else{
15       d = d + p[j]*q[j];
16     }
17   }
18   ...
19 }
```

(b)

Figure 3.12: A code excerpt from the function $conj\_grad()$ in CG for the Use Case 1. (a) shows the original code excerpt before the truncation pattern is applied; (b) shows the code excerpt when the truncation is applied.

$$P_{suc\_rate} = \sum_{i=1}^{\#patterns} \beta_i x_i + \epsilon. \tag{3.2}$$

In Equation 3.2, x_i is the number of pattern instances for a specific pattern i normalized by total number of instructions within the application. We name x_i the pattern rate (e.g., condition rate, shift rate, and truncation rate). We normalize the number of pattern instances to enable a fair comparison between applications with different number of instructions. In total, there are $\#patterns$ patterns ($\#patterns$ is six in our modeling). β_i is the model coefficients and ϵ is the intercept.

Experiments and Model Validation. We perform two experiments. In the *first* experiment, we build the model using all the patterns from the ten benchmark programs (Section 3.4.1) to show that the data fits the model well. This experiment requires

Table 3.4: The quantification of resilience patterns and the prediction accuracy. SR=success rate, OW=overwrite.

App.	Cond. Rate	Shift Rate	Trunc. Rate	Dead Location Rate	Repeat Addition Rate	OW Rate	Measured SR	Pred. SR	Pred. Err. Rate
CG	0.088	2.45E-08	2.185	0.298	2.61E-07	0.999	0.739	0.652	11.8%
MG	0.037	2.74E-03	1.145	0.314	0.000	0.999	0.879	0.810	7.8%
LU	0.022	8.11E-06	0.188	0.319	0.000	0.999	0.575	0.642	11.7%
BT	0.015	0.000	0.074	0.334	0.000	0.999	0.656	0.573	12.7%
IS	0.040	2.86E-02	0.001	0.311	0.000	0.985	0.653	0.712	9.0%
DC	0.139	0.174	0.078	0.302	9.22E-07	0.994	0.578	0.204	64.6%
SP	0.042	0.000	0.428	0.389	4.15E-08	0.999	0.385	0.466	21.0%
FT	0.038	1.99E-03	1.591	0.338	0.000	0.999	0.876	1.000	14.2%
KMEANS	0.079	7.18E-07	2.484	0.375	7.87E-05	0.979	0.843	1.000	18.6%
LULESH	0.048	2.60E-03	0.550	0.378	6.88E-06	0.937	0.926	0.725	21.7%

running the ten benchmarks, collecting the number of pattern instances for each pattern, and performing random fault injection to obtain success rates for each benchmark.

In the *second* experiment, we train the model using data from different combinations of nine of the ten benchmarks, and make a prediction for success rate for the one remaining benchmark. We then validate the model prediction by measuring its accuracy (i.e., relative error) with respect to the success rate that is obtained by doing fault injection. This experiment is to see how accurate the model is in predicting the success rate of an unseen program.

Experimental Results. For the first experiment, we calculate the "$R - square$" value of the model. $R - square$ is used for measuring the fitness of a statistic model. The $R - square$ value in our experiment is 96.4%, which is close to 1. A value close to 1 indicates that the model explains the variability of the prediction result around its mean. The model therefore fits and explains the data very well.

For the second experiment, the prediction results are shown as the prediction error rate in Table 3.4. The average prediction error excluding the prediction error on DC is 14.3%. The prediction error on DC is large (64.6%), because the model does not distinguish error tolerance capabilities of different instances of repeated additions and conditional statement (see the limitation discussed below), thus predictions for DC are affected by this limitation.

Importance of Resilience Patterns: Feature Analysis. We use standardized regression coefficient [25], an indicator that presents the importance of predictors, to un-

derstand which resilience patterns are the most important. We compute the standardized regression coefficients for the model trained in the second experiment.

On average, the *averaged standardized regression coefficients* of Shifting, Truncation, Dead Location, Repeated Addition, Overwriting, and Conditional Statement are 1.48, 1.73, 0.38, 0.25, 0.92, and 1.69, respectively. *We conclude that Truncation (1.73), Shifting (1.48), and Conditional Statement (1.69), that have the largest coefficients*, contribute the most to resilience. On the other hand, patterns such as Repeated Addition and Dead Location have less impact.

Limitation and Future Work. Different instances of a pattern can have different weight into application resilience. For example, considering different cases of shifting where the value is shifted to right/left x times. Depending on the value of x, the error may or may not be masked. While simply counting the number of pattern instances limits the prediction accuracy (one should also take into account the value of locations), this demonstrates a simple but practical use case of the patterns.

3.7 Conclusions

Understanding natural error resilience in HPC applications is important in creating applications that can naturally tolerate errors. However, our knowledge on natural error resilience has been quite limited, mainly because of a lack of systematic methods to identify resilience computation patterns. Our framework, `FlipTracker`, exposes these patterns by enabling fine-grained tracking of error propagation and fault tolerance to enable users to pinpoint resilience computations in HPC programs. By tracking data flows and value variations based on a code region model, we identify and summarize six common resilience patterns, which increase our understanding of how natural resilience occurs. We also present two case studies of practical applications of these resilience patterns.

Chapter 4

Modeling Application Resilience to Transient Faults on Data Objects

4.1 Introduction

In this chapter, we introduce a method to model and quantify application resilience to transient faults on data objects. Our method is based on an observation that, application resilience to transient faults on data objects is mainly because of application-inherent semantics and program constructs. For example, a corrupted bit in a data structure could be overwritten by an assignment operation, hence does not cause an outcome corruption; a corrupted bit of a molecular representation in a Monte Carlo method-based simulation may not matter to the application outcome because of the statistical nature of the simulation. Based on the above observation, the quantification of application resilience to transient faults on data objects is equivalent to quantifying error masking events caused by application-inherent semantics and program constructs, and associating those events with data objects. By analyzing application execution information (e.g., the architecture-independent, LLVM [101] IR trace), we can accurately capture those error masking events, and provide insightful analysis on how and where an error tolerance happens. Furthermore, analyzing application execution information, we can use memory addresses of data objects and track register allocation to associate data values in registers and memory with data objects. Such a method introduces data semantics into the analysis.

Quantifying application resilience to transient faults on data objects must address a couple of research problems. First, we have little knowledge of the characteristics of error masking events. This creates a major obstacle to recognize those events and achieve analytical quantification. Second, we do not have a good metric to make the quantification. Simply counting the number of error masking events cannot provide a meaningful quantification, because the number can be accumulated throughout application execution. The fact that a data object has many error masking events does not necessarily mean that the application is resilient to the value corruption of the data object because those events may be only a small portion of the total operations on data objects. Third, determining the impact of an error occurrence on the correctness of application outcome is challenging. The error can propagate to many data objects. Tracking all of those errors for analysis is prohibitive. In addition, an error may not impact the correctness of

application outcome because of algorithm semantics in the application. However, recognizing algorithm semantics requires detailed application domain knowledge, which is prohibitive for common users.

Based on the method of quantifying error masking events, we systematically model and quantify application resilience to transient faults on data objects, and address the above problems. We first characterize error masking events and classify them into three classes: operation-level error masking, error masking when error propagation, and algorithm-level error masking. We further introduce a metric. The metric quantifies *how often* error masking happens. Based on the metric, the comparison of application resilience to transient faults between different data objects is more meaningful than based on simply counting error masking events. Our classification of error masking events and the proposed metric are fundamental, because they lay a foundation not only for modeling application resilience to transient faults on data objects, but also for other research, such as the placement of error detectors [118] and application checkpoint [110].

Based on our classification and metric, we introduce a model. Given a data object, our model examines operations in the dynamic instruction trace. For each operation that consumes elements of the data object, the model makes the following inference: if an element consumed by the operation has an error, will the application outcome remain correct? The inference procedure of the model includes three practical techniques to recognize the three classes of error masking events: (1) detecting operation-level error masking based on operation semantics, (2) tracking error propagation by limiting propagation length for analysis, and (3) detecting algorithm-level error masking based on *deterministic* fault injection. For (2), limiting propagation length is a technique based on the characterization of error propagation. This technique does not impact our conclusion on error masking while avoiding expensive analysis; for (3), the deterministic fault injection treats the application as a black box without requiring detailed application domain knowledge.

In summary, this chapter makes the following contributions: (1) a systematic method and a metric to analytically model application resilience to transient faults on data objects, which is unprecedented; (2) a comprehensive classification of error masking events, and methods to recognize them; (3) an open-sourced system tool, MOARD [63], to model application resilience to transient faults on data objects. (4) an evaluation of representative, computational algorithms and two scientific applications to reveal how application-level error masking typically happens on data objects; (5) a case study to demonstrate the benefit of using a model-driven approach to direct error tolerance designs.

4.2 Error Tolerance Modeling

We start with a classification of application-level error masking and then introduce a modeling metric.

```
1  void func(double *par_A, double *par_b,
2            double *par_x)
3  {
4      double c = 0;
5
6      //Pre processing par_A
7      par_A[0] = sqrt(initInfo);
8      c = par_A[2]*2;
9      if (c>THR)
10       par_A[4] = (int)c >> bits; //bit shifting
11
12     //Using the algebraic multi grid solver
13     AMG_Solver(par_A, par_b, par_x);
14 }
```

Figure 4.1: The example code to show error masking that happens to a data object, par_A.

4.2.1 General Description

Error masking that happens to data objects has various representations. Listing 4.1 gives a synthetic example to illustrate those representations. In this example, we focus on a data object, par_A, which is an array. We study *error masking that happens to this data object*. We examine every statement in the example code. For each statement, we examine if any element of the data object is involved. If yes, we examine if there is a data corruption in the element, how the data corruption impacts the result correctness of the statement, and how the data corruption propagates to the successor statements which in turn impact the application outcome correctness.

par_A is involved in 4 statements (Lines 7, 8, 10 and 13). The statement at Line 7 has an error masking event: if an error happens at par_A(in particular, the data element $par\_A[0]$, which is consumed by the statement), the error can be overwritten by an assignment operation, no matter which bit is flipped in $par\_A[0]$. The statement at Line 8 has no explicit error masking happen. If an error at $par\_A[2]$ occurs, the error propagates to c by multiplication and assignment operations. If the error propagates to Line 10 (bit shifting), depending on which bit is corrupted at Line 8 and how many bits are shifted at Line 10, the corrupted bit can be thrown away or remain. If the corrupted bit is thrown away, then the error in $par\_A[2]$ propagating from Line 8 to Line 10 is indirectly masked at Line 10 (not directly masked at Line 8).

Line 13 is an invocation of an algebraic multi-grid solver (AMG) taking par_A as input. AMG treats par_A as a multi-dimensional grid and can tolerate certain data corruptions in the grid, because of the algorithm semantics of AMG (particularly, AMG's iterative structure that mitigates error magnitude and tolerates incorrectness of numerical results [34]).

This example reveals many interesting facts. In essence, a program can be regarded as a combination of data objects and operations performed on the data objects. An operation (defined at LLVM instruction level) refers to arithmetic computation, assignment, logical and comparison instructions or an invocation of an algorithm implementation. An operation may inherently come with error masking effects, exemplified at Line 7 (error overwriting); an operation may propagate errors, exemplified at Line 8. Different operations have different error masking effects, and hence impact the application

outcome differently. Based on the above discussion, we classify application-level error masking into three classes.

(1) **Operation-level error masking.** An error that happens to the target data object is masked because of the semantics of the operation. Line 7 in Listing 4.1 is an example.

(2) **Error masking when error propagation.** Some error masking events are implicit and have to be identified beyond a single operation. In particular, a corrupted bit in a data object is not masked in the current operation (e.g., Line 8 in Listing 4.1) but the error propagates to another data object (e.g., the variable c) and masked in another operation (e.g., Line 10). Note that simply relying on isolated operation-level analysis without the error propagation analysis is not sufficient to recognize these error masking events.

(3) **Algorithm-level error masking.** Identification of some error masking events must include algorithm-level information. The identification of these events is beyond the first two classes. Examples of such events include the multigrid solver [34] and certain sorting algorithm [138]. The algorithm-level error masking can tolerate errors that happen to many variables. For example, the multigrid solver can tolerate low-significant bit-flip errors in multiple iterations [34]. The essence of algorithm-level error masking is typically due to algorithm specific definition on execution fidelity and specific program constructs that mitigate error magnitude during application execution [126]. Limited analysis at individual operations or error propagation is not sufficient to build up a big picture to capture the algorithm-level fault tolerance.

Our modeling is analytical and relies on the quantification of the above error masking events on data objects. We create a metric to quantify those events.

4.2.2 aDVF: A New Metric

To quantify application resilience to transient faults on a data object, the key is to quantify how often error masking happens to the data object. We introduce a new metric, *aDVF* (i.e., the application-level Data Vulnerability Factor), to quantify application resilience to transient faults on data objects. aDVF is defined as follows.

For an operation with the participation of the target data object (maybe multiple data elements of the target data object), we reason that if an error happens to a participating data element of the target data object, the application outcome could or could not remain correct in terms of the outcome value and algorithm semantics. If the error does not cause an incorrect application outcome, then an error masking event happens to the target data object. A single operation can operate on multiple data elements of the target data object. For example, an ADD operation can use two elements of the target data object as operands. For a specific operation, aDVF of the target data object is defined as the total number of error masking events divided by the number of data elements of the target data object involved in the operation.

For example, an assignment operation $a[1] = w$ happens to a data object, the array a. This operation involves one data element ($a[1]$) of the target data object a. We calculate aDVF for a in this operation as follows. If an error happens to $a[1]$, we reason that the erroneous $a[1]$ does not impact correctness of the application outcome and the error in $a[1]$ is always masked (no matter which bit of $a[1]$ is flipped). Hence, the number of error masking events for the target data object a in this operation is 1. Also, the total

number of data elements involved in the operation is 1. Hence, the aDVF value for the target data object in this assignment operation is $1/1 = 1$.

Based on the above discussion, the definition of aDVF for a data object X in an operation ($aDVF_{op}^{X}$) is formulated in Equation 4.1, where x_i is a data element of the target data object X involved in the operation and m is the number of data elements involved in the operation; f is a function to count error masking events that can happen to a data element.

$$aDVF_{op}^{X} = \sum_{i=0}^{m-1} f(x_i)/m \tag{4.1}$$

To calculate aDVF for a data object in a code segment, we examine operations in the code segment one by one; For each operation that involves any element of the target data object, we consider that if a transient fault happens to the element, how many error masking events can happen. In general, the definition of aDVF for a data object in a code segment is similar to the above for an operation, except that m is the number of data elements of X involved in all operations of the code segment. [1] According to the above definition, a higher aDVF value for a data object indicates that the application is more resilient to transient faults on the data object; Also, an aDVF value should be in $[0, 1]$.

To further explain it, we use a code segment from LU benchmark in SNU_NPB benchmark suite 1.0.3 (a C-based implementation of the Fortran-based NAS benchmark suite [12]), shown in Listing 4.2.

An example from LU. We calculate aDVF for the array $sum[]$. Statement A has an assignment operation involving one data element ($sum[m]$) and one error masking event (i.e., if an error happens to $sum[m]$, the error is overwritten by the assignment). Considering that there are five iterations in the first loop ($iter_{num1} = 5$), there are five error masking events happening to five data elements of $sum[]$.

Statement B has two operations related to $sum[]$ (i.e., an assignment and an addition). The assignment operation involves one data element ($sum[m]$) and has no error masking because the new value is added to $sum[m]$ (not overwriting it); The addition operation involves one data element ($sum[m]$) and may have one error masking (i.e., certain corruptions in $sum[m]$ can be ignored, if ($v[k][j][i][m] * v[k][j][i][m]$) is significantly larger than $sum[m]$). This error masking is counted as r' ($0 \leq r' \leq 1$), depending on the corrupted bit position in $sum[m]$ and the error propagation result (see Sections 4.2.3 and 7.4 for further discussion). In the loop structure where Statement B is, there are ($r' * iter_{num2}$) error masking events that happen to ($2 * iter_{num2}$) elements of $sum[]$, where "r'" comes from the addition operation [2], and $iter_{num2}$ is the number of iterations in the second loop.

Statement C has two operations related to $sum[]$ (i.e., an assignment and a division) but only the assignment operation has error masking (overwriting). In the loop structure where Statement C is, there are five iterations ($iter_{num3} = 5$). Hence, there are five error

[1] If a data element is referenced multiple times in the code segment, this data element is counted multiple times in m.

[2] The addition operation with the corrupted $sum[m]$ can propagate the error to the assignment. This error propagation effect is included in r'.

```
1  void l2norm(int ldx, int ldy, int ldz, int nx0, \
2   int ny0, int nz0, int ist, int iend, int jst, \
3   int jend, double v[][ldy/2*2+1][ldx/2*2+1][5], \
4      double sum[5])
5  {
6      int i, j, k, m;
7      for(m=0;m<5;m++) //The first loop
8        sum[m]=0.0;   //Statement A
9
10     for(k=1;k<nz0 1;k++)  //The second loop
11       for (j=jst;j<jend;j++)
12         for (i=ist;i<iend;i++)
13           for (m=0;m<5,m++)
14             sum[m]=sum[m]+v[k][j][i][m] \
15                    *v[k][j][i][m]; //Statement B
16
17     for(m=0;m<5;m++){   //The third loop
18       sum[m]=sqrt(sum[m]/((nx0 2)*  \
19              (ny0 2)*(nz0 2))); //Statement C
20     }
21 }
```

Figure 4.2: A code segment from LU.

masking events that happen on five data elements of the target data object. In summary, the aDVF calculation for $sum[]$ is

$$aDVF_{op}^{sum} = \frac{1 * iter_{num1} + r' * iter_{num2} + 1 * iter_{num3}}{1 * iter_{num1} + (1+1) * iter_{num2} + (1+1) * iter_{num3}}, \tag{4.2}$$

where each term in the numerator is the number of error masking events in the first, second, and third loop, respectively; each term in the denominator is the number of target data elements involved in each loop; $iter_{num1} = 5, iter_{num3} = 5$ and $iter_{num2} = (nz0 - 2) * (jend - jst) * (iend - ist) * 5$.

To calculate aDVF for a data object, we must rely on effective identification and counting of error masking events (i.e., the function f). In Sections 4.2.3, 4.2.4 and 4.2.5, we introduce a series of counting methods based on the classification of error masking events.

4.2.3 Operation-Level Analysis

To identify error masking events at the operation level, we analyze all possible operations. In particular, we analyze architecture-independent, LLVM instructions and characterize their error tolerance based on operation semantics. We classify the operation-level error masking as follows.

(1) **Value overwriting**. An operation writes a new value into a data element of the target data object and the error in the data element (no matter where the corrupted bit is in the data element) is masked. For example, the store operation overwrites the error in the store destination. We also include *trunc* and bit-shifting operations into this category because the error could be truncated or shifted away in those operations.

(2) **Logical and comparison operations**. If an error in the target data object does not change the correctness of logical and comparison operations, the error is masked.

Examples of such operations include logical *AND* and the predicate expression in a *switch* statement.

(3) **Value overshadowing**. If the corrupted data value in an operand of an addition or subtraction operation is overshadowed by the other correct operand involved in the operation, then the corrupted data can have an ignorable impact on the correctness of application outcome. For example, the data value "10" in an addition operation ("10e+6 + 10") is corrupted and the addition operation becomes "10e+6 + 11". But such data corruption may not matter to the application outcome because the operand "10e+6" is much larger than the magnitude of the data corruption. We further discuss how the overshadowing effect is determined in Section 7.4.

The above three operation-level error masking impacts the application outcome differently. Error masking based on value overwriting and logic and comparison operations can make the application outcome numerically the same as the error-free case. Error masking based on value overshadowing can make the application outcome numerically different from or the same as the error-free case.

For value overshadowing, if the application outcome is numerically different, the application outcome can still be acceptable because of algorithm semantics; if the application outcome is numerically the same, operations *after* the value overshadowing must help tolerate corrupted bits. For the above two cases, we do not attribute error masking to the algorithm level or error propagation level. Instead, we attribute it to operation-level value overshadowing because value overshadowing initiates error masking. Without value overshadowing, algorithm or error propagation may not mask errors.

The effectiveness of the above error masking heavily relies on the error pattern. *The **error pattern** is defined by how erroneous bits are distributed within a corrupted data element* (e.g., single-bit vs. spatial multiple-bit, least significant bit vs. most significant bit). Depending on where the erroneous bit is, the error in the data object could or could not be masked. Take as an example the bit shift operation (Line 10) in Listing 4.1. Depending on the error pattern, the shift operation can remove or keep the corrupted bit.

To determine the existence of the above (2) and (3) error masking, we must consider error patterns (i.e., the spatial aspect of errors [151]). In the practice of our resilience modeling, given an operation to analyze, we enumerate possible error patterns for the target data object. Then, we derive the existence of error masking for each error pattern without application execution. Suppose there are n error patterns and m ($0 \leq m \leq n$) of which have error masking. Then the number of error masking events is calculated as m/n, which is a statistical quantification of possible error masking. In the example of the bit shift (Line 10 in Listing 4.1), assuming that c is 64-bit and we consider single-bit errors, then there are 64 error patterns. For each error pattern, we decide if the corrupted bit is shifted away. If 10 of the 64 fault patterns have the corrupted bit shifted, then the number of error masking events for the data object c in this shift operation is 10/64.

4.2.4 Error Propagation Analysis

If we analyze a specific error pattern in an operation (named "target error pattern" and "target operation" in the rest of this section) and determine that the error cannot be masked in the target operation, then we use error propagation analysis to capture error masking (i.e., the temporal aspect of errors [151]). Using a dynamic instruction trace

as input, the error propagation analysis tracks whether the errors (including the original one and the new ones because of error propagation) are masked in the successor operations based on the operation-level analysis without application execution. If all of the errors are masked and hence the application outcome remains *numerically the same* as the error-free case, then we claim that the original error in the target operation is masked.

For the error propagation analysis, a big challenge is to track all contaminated data which can quickly increase as the error propagates. Tracking all the contaminated data significantly increases analysis time and memory usage. A solution to this challenge is *deterministic* fault injection. Different from random fault injection, the deterministic fault injection injects an error at the target operation using the target error pattern and then run the application to completion. If the application outcome is *numerically the same* as the error-free case, then the original and the new errors are masked, and the error masking based on error propagation takes effect. If the application outcome is numerically different but still accepted, then the algorithm-level error masking takes effect.

Because of the deterministic fault injection, we do not need to analyze operations one by one to track data flow and error contamination. Hence it is faster. However, the deterministic fault injection can still be time-consuming, if application execution time is long. To improve the efficiency of the error propagation analysis, we optimize the analysis based on the characteristics of error propagation.

Optimization: bounding propagation path. We observe that tracking a limited number of operations (k operations) after the target operation is often sufficient to decide the existence of the propagation-based error masking. Our observation is based on 1000 random fault injection tests on 16 data objects from eight benchmarks (see Table 4.1 for benchmark details). We observe that 87% of the fault injection tests that cannot mask errors within 10 operations ($k = 10$) after fault injection lead to numerically incorrect application outcomes; 100% of the fault injection tests that cannot mask errors within 50 operations ($k = 50$) after fault injection lead to numerically incorrect application outcomes. This fact indicates that errors that are not masked within a limited number of operations have little chance to be masked by further error propagation.

The rationale to support the above observation is as follows. An error in a data object typically propagates to a large amount of data (objects) quickly. After a certain number of operations, it is very unlikely that all errors are able to be masked by further error propagation and making a conclusion of no error masking *by error propagation* is correct in most cases.

Based on the above observation, we only need to track the first k operations after the target operation to determine the existence of the propagation-based error masking. In particular, after analyzing k operations ($k = 50$ in our evaluation), (1) If not all errors due to error propagation are masked at the operation level, we conclude that the errors will not be masked at the operation level by further error propagation. But those errors may be masked by algorithm (if the user wants to do algorithm-level analysis), pending further investigation; (2) If all errors due to error propagation are masked and based on the operation-level analysis we can derive that the application outcome remains numerically correct, then we claim error masking due to error propagation happens.

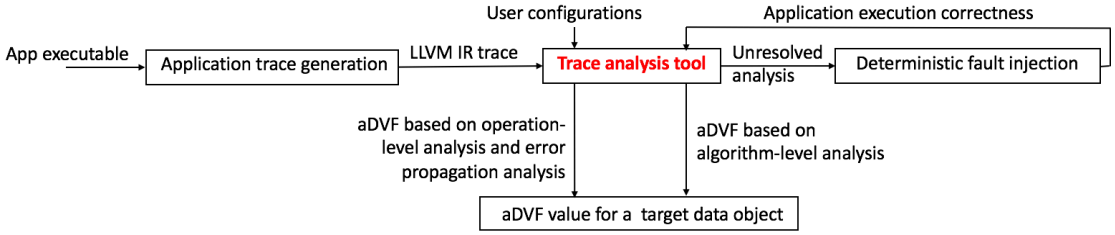


Figure 4.3: MOARD, a tool for modeling application resilience to transient faults on data objects

4.2.5 Algorithm-Level Analysis

Identifying the algorithm-level error masking demands domain and algorithm knowledge. In our modeling, we want to minimize the usage of that knowledge, such that the modeling methodology can be general across different domains. The traditional random fault injection treats the program as a black-box. Hence, using the traditional random fault injection could be an effective tool to identify the algorithm-level error masking. However, to avoid the randomness, we use the deterministic fault injection again.

In particular, when we analyze a specific error pattern in a target operation and decide that the error cannot be masked in the target operation and next k operations, we inject an error using the error pattern in the target operation and run the application to completion. If the application outcome is numerically different from the error-free case but acceptable in terms of algorithm semantics, then algorithm-level error masking takes effect. If the application outcome is numerically the same, then error masking due to error propagation happens, which should be rare based on the above discussion on "bounding propagation path".

Discussion: Although we employ the deterministic fault injection, it cannot replace our modeling because of two reasons. First, the fault injection space without our modeling is typically huge (trillions of fault injection sites [69]), which is prohibitive for implementation. Second, the deterministic fault injection tells us little about how an error is tolerated.

4.3 Implementation

To calculate the aDVF value for a data object, we develop a tool, named *MOARD* (standing for *MO*deling *A*pplication *R*esilience to transient faults on *D*ata data objects). Figure 4.3 shows the tool framework and its algorithm. MOARD has three components: an application trace generator, a trace analysis tool, and a deterministic fault injector.

The **application trace generator** is an LLVM instrumentation pass to generate a dynamic LLVM IR trace. LLVM IR is architecture independent and each instruction in the dynamic IR trace corresponds to one operation. We extend a trace generator [136] to enable trace generation for MPI applications. During the trace analysis, we consider error propagation by MPI communication, but do not consider those cases where errors happen in the communication.

The **trace analysis tool** is the core of MOARD. Using an application trace as input, the tool can calculate the aDVF value of any data object with known memory address range. In particular, the trace analysis tool conducts the operation-level and error prop-

agation analysis. For those unresolved analyses, the trace analysis tool will output a set of fault injection information for the deterministic fault injection. Such information includes dynamic instruction IDs, IDs of the operands that reference the values of the target data object, and the bit locations of the operands that correspond to those error patterns with undetermined error masking. After the fault injection results (i.e., the numerical values of application outcome and whether the outcome is acceptable) are available from the deterministic fault injector, we re-run the trace analysis tool, and use the fault injection results to address the unresolved analyses and update the aDVF calculation.

For the error propagation analysis, we associate data semantics (the data object name) with the data values in registers, such that we can identify the data of the target data object in registers. To associate data semantics with the data in registers, MOARD tracks the register allocation when analyzing the trace, such that we can know at any moment which registers have the data of the target data object.

To **determine the existence of value overshadowing** in an addition or subtraction operation, we use the deterministic fault injection. Particularly, given a target operand in an addition or subtraction operation for value overshadowing analysis, we enumerate all error patterns for deterministic fault injection tests. If the following two conditions are true, then we derive that the value overshadowing happens in the operation:

- Some error patterns result in small magnitudes of the operand (smaller than the magnitude of the other operand in the operation); the application outcome is acceptable.
- The other error patterns result in larger magnitudes of the operand (larger than those in the first condition) but the application outcome is not acceptable.

The error masking of the value shadowing is quantified as x/y, where x is the number of error patterns in the first condition and y is the number of all error patterns. For example, suppose we have an addition operation ($a + b$, $a = 1000$ and $b = 1$) and b is our target data object. We enumerate error patterns in b (assuming 32 single-bit-flip error patterns). If five patterns result in the values of b as 0, 3, 5, 9 and 17, which are smaller than a and the application outcome is acceptable, and the other 26 patterns result in larger b (larger than 0, 3, 5, 9, and 17) but the application outcome is not acceptable, then the value overshadowing happens (the corrupted b is overshadowed by a), and is quantified as 5/32.

The **deterministic fault injector** is a tool to resolve those error masking analyses undetermined by the trace analysis tool. The input to the deterministic fault injector is a list of fault injection sites generated by the trace analysis tool. Similar to the application trace generation, the deterministic fault injector is also based on the LLVM instrumentation. We use the LLVM instrumentation to count dynamic instructions and trigger bit flips. The application execution will trigger bit flip when a fault injection site is encountered.

To **accelerate the calculation of aDVF**, we leverage the existing work [69, 130] that explores "error equivalence" based on the similarity of intermediate execution states to avoid repeated analysis and fault injections on instructions. During our evaluation, MOARD calculates aDVF for 16 data objects in eight benchmarks within one day on a

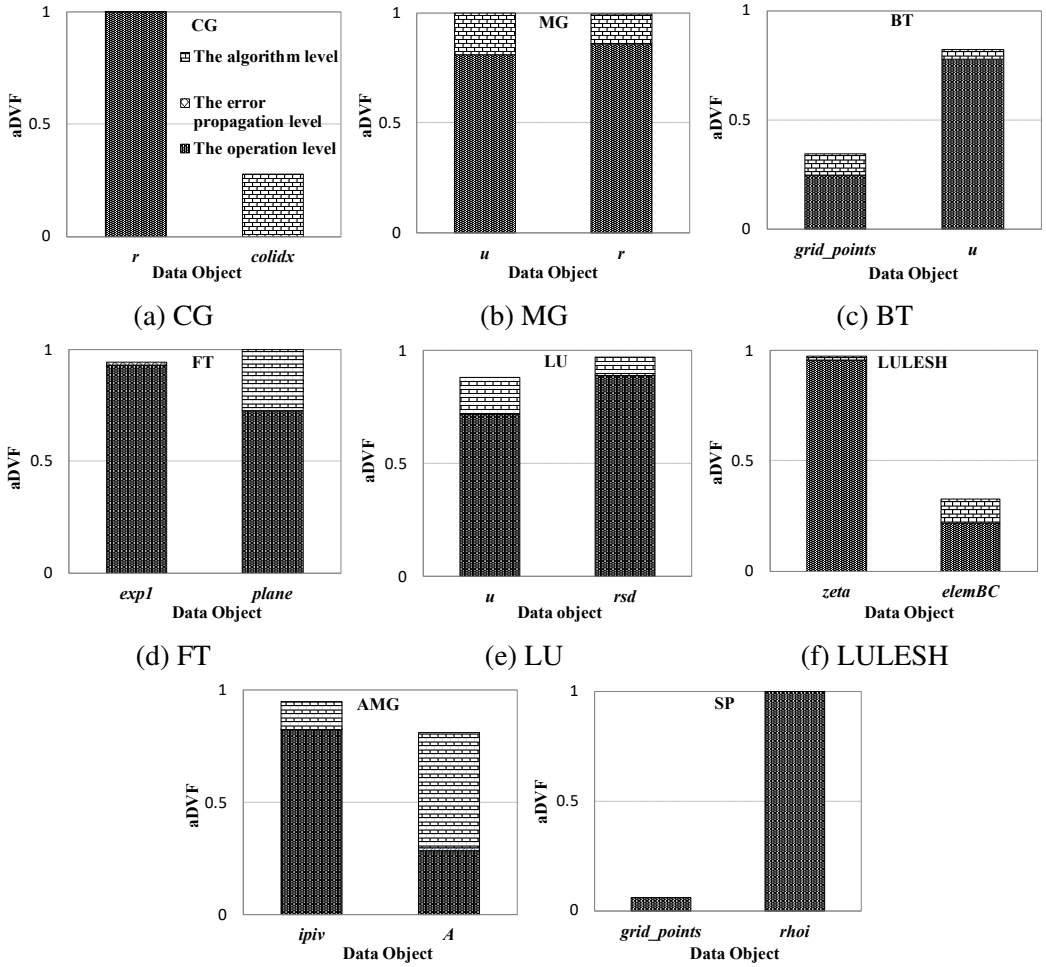
Table 4.1: Benchmarks and applications for the study

Name	**Benchmark description**	**Code segment for evaluation**	**Target data objects**
CG (NPB)	Conjugate Gradient, irregular memory access (input class S)	The routine conj_grad in the main loop	The arrays r and $colidx$
MG (NPB)	Multi-Grid on a sequence of meshes (input class S)	The routine mg3P in the main loop	The arrays u and r
FT (NPB)	Discrete 3D fast Fourier Transform (input class S)	The routine fftXYZ in the main loop	The arrays $plane$ and $exp1$
BT (NPB)	Block Tri-diagonal solver (input class S)	The routine x_solve in the main loop	The arrays $grid\_points$, u
SP (NPB)	Scalar Penta-diagonal solver (input class S)	The routine x_solve in the main loop	The arrays $rhoi$ and $grid\_points$
LU (NPB)	Lower-Upper Gauss-Seidel solver (input class S)	The routine ssor	The arrays u and rsd
LULESH [81]	Unstructured Lagrangian explicit shock hydrodynamics (input 5x5x5)	The routine CalcMonotonicQRegionForElems	The arrays m_elemBC and m_delv_zeta
AMG2013 [70]	An algebraic multigrid solver for linear systems arising from problems on unstructured grids (we use GMRES(10) with AMG preconditioner). We use a compact version from LLNL with input matrix $aniso$.	The routine hypre_GMRESSolve	The arrays $ipiv$ and A

cluster of 256 cores, which is comparable to the execution time of existing fault injection work [69, 130].

4.4 Evaluation

In this section, we use aDVF as an metric to evaluate application resilience to transient faults on data objects with a set of benchmarks. Furthermore, we validate the accuracy of our aDVF calculation. We also compare aDVF calculation with the traditional fault injection to show the power and benefits of aDVF calculation.



(a) CG (b) MG (c) BT

(d) FT (e) LU (f) LULESH

(g) AMG (h) SP

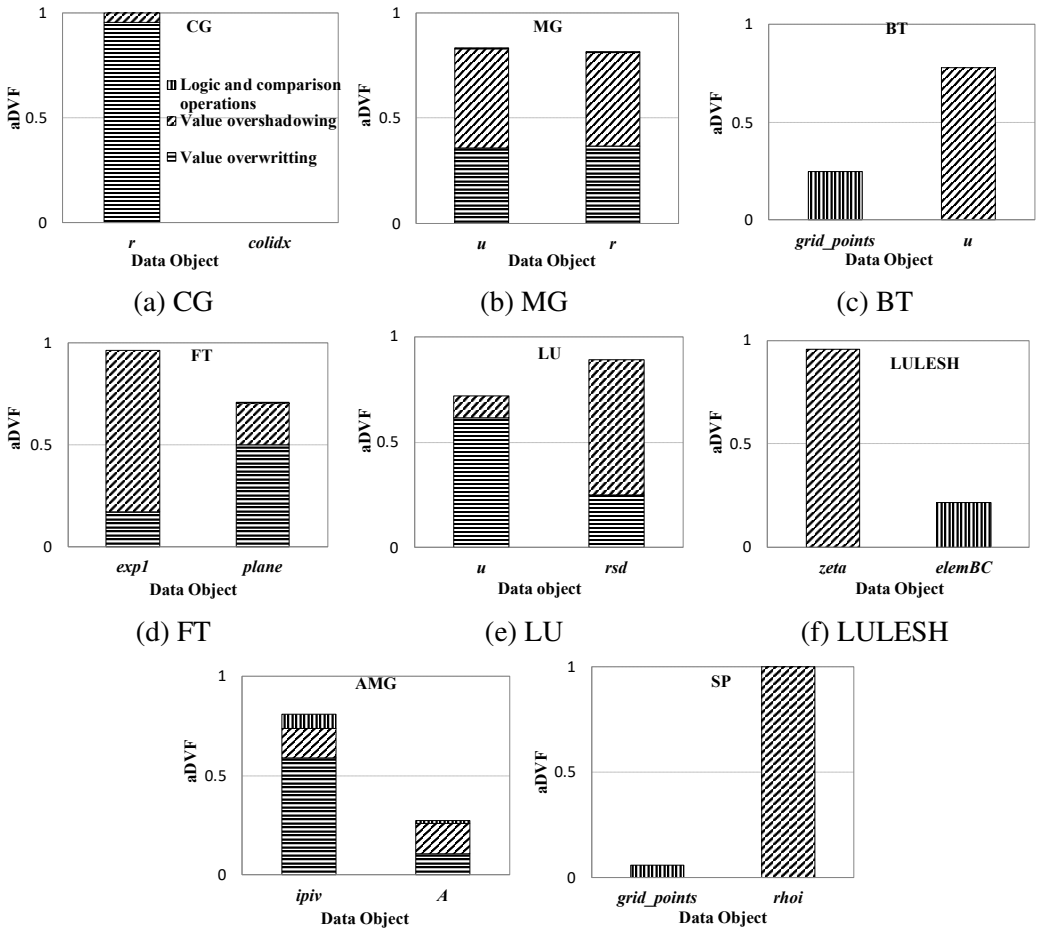
Figure 4.4: The breakdown of aDVF results based on the three level analysis. The x axis is the data object name.

4.4.1 Evaluating Application Resilience to Transient Faults on Data Objects Using aDVF

We study 12 data objects from six benchmarks of the NAS parallel benchmark (NPB) suite and four data objects from two scientific applications. Those data objects are chosen to be representative: they have various data access patterns and participate in different execution phases. Table 4.1 gives details on the benchmarks and applications. The maximum error propagation path for aDVF analysis is 50, for which we do not lose analysis accuracy as we discuss in Section 4.2.4. Similar to [69, 130, 150], we only study single-bit errors because they are the most common errors.

Figure 4.4 shows the aDVF results and breaks them down into the three levels (i.e., the operation level, error propagation level, and algorithm level).

Error masking happens commonly in data objects across benchmarks and applications including those scientific applications (e.g., LULESH and AMG) that are highly sensitive to data correctness. Several data objects (e.g., r in CG, and $exp1$ and $plane$ in FT) have aDVF values close to 1 in Figure 4.4, which indicates that most operations working on these data objects have error masking. Those data objects are double-



(a) CG (b) MG (c) BT

(d) FT (e) LU (f) LULESH

(g) AMG (h) SP

Figure 4.5: The breakdown of aDVF results based on value overwriting, value overshadowing, and logic and comparison operation at the levels of operation and error propagation. The x axis is the data object name. *zeta* and *elemBC* in LULESH are *m_delv_zeta* and *m_elemBC*.

precision floating-point and their error masking mainly comes from value overshadowing and overwriting (Figure 4.5). However, a couple of data objects have much less intensive error masking. For example, the aDVF value of $colidx$ in CG is only 0.28 (Figure 4.4). Further study reveals that $colidx$ is an integer array to store indexes of sparse matrices and there is few operation-level or error propagation-level error masking (Figure 4.5). Its corruption can easily cause segmentation error caught by the deterministic fault injection. $grid\_points$ in SP and BT also have a small aDVF value (0.06 and 0.38 for SP and BT respectively in Figure 4.4). Further study reveals that the array $grid\_points$ defines input problems for SP and BT. An error in $grid\_points$ can easily cause major changes in computation caught by the error propagation analysis.

Evaluation conclusion 1: The above aDVF-based analysis reveals the variation of application resilience to transient faults on data objects and provides insights on whether the corruption on a data object impacts application outcomes, which is useful to direct fault tolerance mechanisms.

We further notice that the data objects *colidx* and *r* in CG have 2.19e+09 and

4.54e+07 error masking events (not shown in Figure 4.4), respectively. Although *colidx* has more error masking events, CG is not more resilient to errors on *colidx* than on *r*. In particular, 75% bit flips that happen in the elements of *colidx* involved in the operations of CG causes incorrect application outcome or segmentation faults, while less than 1% in r. The above observation provides a strong support to introduce the metric, aDVF.

Evaluation conclusion 2: Simply counting the number of error masking events is not sufficient to evaluate application resilience to errors on data objects.

We further look into the results based on the analysis of the three levels. Operation-level error masking is very common. Figure 4.4 shows that there are 12 data objects whose operation-level error masking contributes more than 70% of the aDVF values. For $exp1$ in FT and $rhoi$ in SP, the contribution of the operation-level error masking is close to 99%.

We further notice that the contribution of error masking at the error propagation level to the aDVF result is very limited. For most of the data objects, the contribution is less than 10% (Figure 4.4). For five data objects ($colidx$ in CG, $grid\_points$ and u in BT, and $grid\_points$ and $rhoi$ in SP), there is no such error masking. Note that our analysis at the error propagation level is valid even if we increase the error propagation length. We discuss the impact of error propagation length in Section 4.2.4.

Different from error masking at the error propagation level, the contribution of the algorithm-level error masking to the aDVF result is relatively large. For example, the algorithm-level error masking contributes 19% to the aDVF value for u in MG and 27% for $plane$ in FT (Figure 4.4). The large contribution for u in MG is consistent with the existing work [34]. For FT (particularly 3D FFT), the large contribution of algorithm-level error masking in $plane$ comes from frequent transpose and 1D FFT computations that average out the data corruption. CG, as an iterative solver, is known to have the algorithm-level error masking because of the iterative nature [135]. Interestingly, the algorithm-level error masking in CG contributes most to application resilience to transient faults on $colidx$ which is a vulnerable integer data object (Figure 4.4).

Evaluation conclusion 3: The aDVF analysis gives us deep information on how errors are tolerated. This may be useful for refactoring application (e.g., using different algorithms or different data structures and data types) to improve error tolerance of data objects.

We further break down the aDVF results based on classifications of the value overwriting, logical and comparison operations, and value overshadowing) based on the analysis at the operation and error propagation levels, shown in Figure 4.5. We have the following observation.

The value overshadowing is very common, especially for (double-precision) floating point data objects (e.g., u in BT, $zeta$ in LULESH, and $rhoi$ in SP in Figure 4.5). This finding has an important indication for studying application-level error tolerance. We have the following conclusion: the impact of data corruption can be correlated with the input problem, because different input problems can have different values of the data objects, which in turn have different effects of value overshadowing. Hence, the existing conclusions on application-level fault tolerance [28, 96, 97, 138, 99] with single input problems must be re-examined with different input problems to validate the conclusions of application resilience.

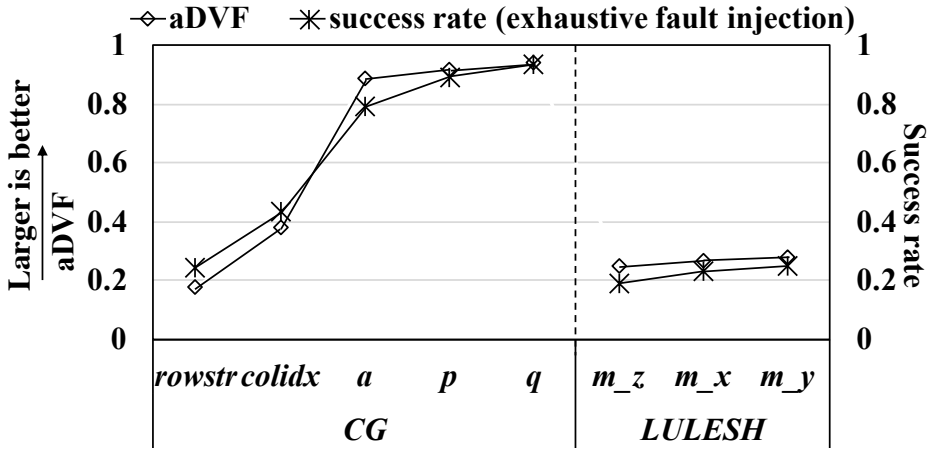


Figure 4.6: Model validation against exhaustive fault injection. The x axis shows the data object name.

4.4.2 Model Validation

In this section, we aim to (1) validate the accuracy of our approach to calculate aDVF, and (2) demonstrate that aDVF correctly quantifies application resilience to transient faults on *data objects*.

We validate our modeling approach by comparing the aDVF result with the result of *exhaustive fault injection* (particularly, the success rate of exhaustive fault injection tests). The exhaustive fault injection is different from the traditional random fault injection. With an exhaustive fault injection campaign, we inject faults into *all* valid fault injection sites. A valid fault injection site is a bit in an instruction operand or output that has a value of the target data object. We use those fault injection sites, because we quantify application resilience to transient faults on *data objects*. The exhaustive fault injection is accurate to quantify application resilience to transient faults on data objects, because of its full coverage of all fault sites. However, the number of valid fault injection sites can be very large (e.g., trillions of sites in CG (Class A)). Hence, although the exhaustive fault injection is accurate and good for model validation in this section, this method is not practical, compared with aDVF.

Note that the aDVF result *cannot be exactly the same* as the exhaustive fault injection result, because the definitions of aDVF and exhaustive fault injection are different. Hence, we validate the modeling accuracy by quantifying application resilience to transient faults for multiple data objects, and then ranking them based on the quantification. Ideally, the rank order of data objects based on the aDVF calculation should be exactly the same as that based on the exhaustive fault injection. A correct order of data objects in terms of application resilience to transient faults is critical to decide which data objects should be protected by fault tolerance mechanisms.

We focus on a function ($conj\_grad()$) from CG and a function (CalcMonotomicQRegionForElems()) from LULESH. We study major data objects in the two functions (those data objects take most of memory footprint). We use single-bit flip in fault injection. The results are shown in Figure 4.6. We notice that the aDVF and exhaustive fault injection rank the data objects in the same order. aDVF correctly reflects application resilience to transient faults on data objects.

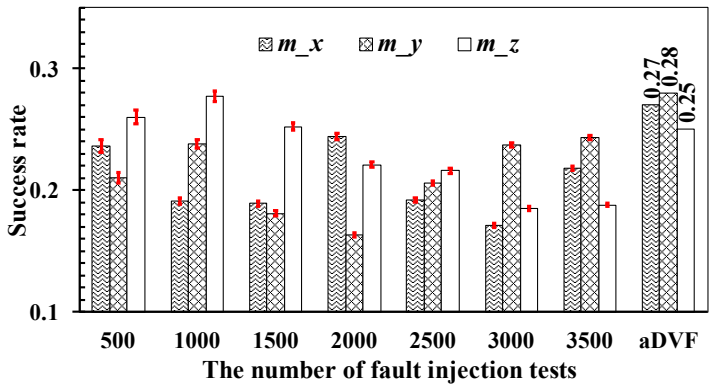


Figure 4.7: The RFI results with the margin of error (the confidence level 95%) and aDVF results. The results are for three data objects (m_x, m_y, and m_z) from $CalcMonotomicQRegionForElems()$ of LULESH.

4.4.3 Comparing aDVF Calculation with the Traditional Random Fault Injection (RFI)

We compare aDVF calculation with RFI. We aim to reveal the limitation of this traditional approach, and demonstrate the predictive power of aDVF, compared to RFI.

RFI

We use the following method for RFI. We use valid fault injection sites, as defined in Section 4.4.2, for RFI. In each fault injection test, we randomly trigger a single-bit flip in a valid fault injection site. The number of fault injection tests is determined by a statistical approach [95] using confidence-level of 95% to ensure statistical significance. We do seven sets of fault injection tests, and the number of fault injection tests in the seven sets ranges from 500 to 3500 with a stride of 500. We use three equal-sized, floating-point arrays (m_x, m_y, and m_z) in the function CalcMonotomicQRegion-ForElems() of LULESH for study.

Figure 4.7 shows the results of RFI (the success rate). *The figure also shows the margin of error* (shown as small red bars in the figure). The results reveal that the results of RFI are sensitive to the number of fault injection tests. For example, for m_z, the success rates of RFI are 0.28 and 0.19 for 1000 and 3000 random fault injection tests, respectively. There is 49% difference between the two results. Furthermore, in terms of application resilience to transient faults on data objects, we cannot rank the three target data objects in a consistent order across the seven test sets. For example, the success rate of RFI for m_x is lower than that for m_z, when the number of fault injection tests is 500, 1000, and 1500. However, the observation is opposite, when the number of fault injection tests is 2000 and 3500. In other words, using RFI, we cannot make any conclusion that LULESH is more resilient to transient faults on a data object than on another data object (even through the margin of error is considered). The reason is three-fold: randomness of RFI, limited confidence level, and inability to capture error masking events.

aDVF

We measure aDVF of the three data objects. Figure 4.7 shows the results (see the last group of bars). We rank the three objects in a determined order (i.e., no inconsistence in the aDVF calculation results, no matter how many times we calculate aDVF). The order is also verified by the accurate, exhaustive fault injection (see Section 4.4.2 for discussion). Having a determined order is important for guiding error tolerant designs (e.g., deciding which data object should be protected by a fault tolerance mechanism).

Evaluation conclusion 4: The calculation of aDVF is deterministic, meaning that we can deterministically rank data objects in terms of application resilience to transient faults on the data objects. Using the traditional RFI, we cannot do so. RFI can be ineffective for guiding error tolerant designs.

4.5 Case Study

In this section, we study a case of using aDVF to help system designers decide whether a specific application-level fault tolerance mechanism is helpful to improve application resilience to transient faults on data objects.

Application-level fault tolerance mechanisms, such as algorithm-based fault tolerance [38, 155, 76], are extensively studied as a means to increase application resilience to transient faults on data objects. However, those mechanisms can come with big performance and energy overheads (e.g., 35% performance loss in [53]). To justify the necessity of using those mechanisms, we must quantify the effectiveness of those mechanisms. With the introduction of aDVF, we can evaluate if application resilience to transient faults on data objects is effectively improved with fault tolerance mechanisms in place.

We focus on a specific application-level fault tolerance mechanism, the algorithm-based fault tolerance (ABFT) for general matrix multiplication ($C = A \times B$) [155]. This ABFT mechanism encodes matrices A, B, and C into a new form with checksums. If an error happens in an element of C, leveraging the checksums, we are able to correct and detect the erroneous element. We apply the aDVF analysis on this ABFT and the matrix C is the target data object. We compare the aDVF values of C with and without ABFT. Figure 4.8 shows the results. The figure shows that ABFT effectively improves error tolerance of C: the aDVF value increases from 0.0172 to 0.82 (the larger is better). The improvement mostly comes from the value overwriting during error propagation. This result is expected because a corrupted element of C is not corrected by ABFT right away. Instead, it will be corrected in a specific verification phase of ABFT during error propagation.

Given the effectiveness of this ABFT, we further explore whether this ABFT can help us improve resilience to transient faults on a data object in an application, Particle Filer (PF) from Rodinia [36], without knowing the application resilience of PF. PF has a critical variable, xe, which is repeatedly used to store vector multiplication results. Given the fact that a vector can be treated as a special matrix, we can apply ABFT to protect xe for those vector multiplications. Using xe as our target data object, we perform the aDVF analysis with and without ABFT. We want to answer a question:

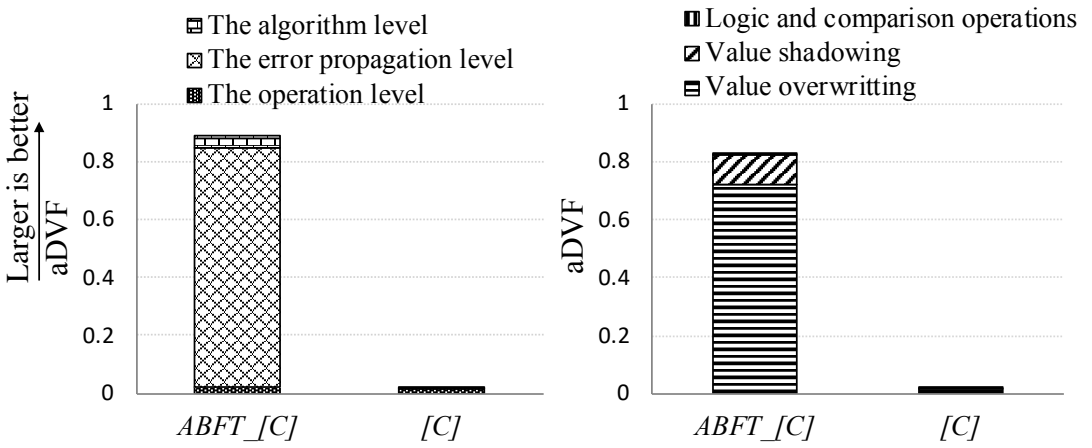


Figure 4.8: Using aDVF analysis to study application resilience to transient faults on C in matrix multiplication (MM). Notation: [C] is MM without applying ABFT on C; ABFT_[C] is MM with ABFT taking effect.

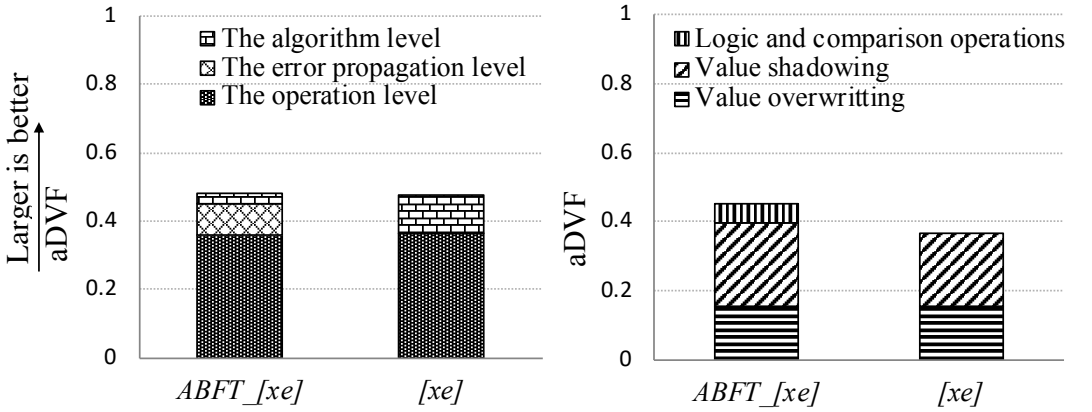


Figure 4.9: Using aDVF analysis to study the effectiveness of ABFT for a data object xe in PF. [xe] has no protection of ABFT; ABFT_[xe] has ABFT taking effect on xe.

Will using ABFT be an effective fault tolerance mechanism for protecting xe in PF?

Figure 4.9 shows the results. The figure reveals that using ABFT does not improve much application resilience to transient faults on the data object xe: there is only little change to the aDVF value (0.48 vs. 0.475). We find two reasons for it: (1) The operation-level error masking accounts for a large part of error masking, no matter whether we use ABFT or not; (2) Most errors corrected by ABFT are also correctable by PF. The second reason is demonstrated by the following fact: with ABFT, the number of error masking events increases at the error propagation level but decreases at the algorithm level. But in total, the number of error masking events at the both levels with ABFT is almost the same as without ABFT. This case study is a clear demonstration of how powerful the aDVF analysis can direct error tolerance designs.

4.6 Discussions

4.6.1 Program Optimization by aDVF

aDVF has many potential usages. We discuss two cases that use aDVF to optimize programs.

Code optimization: Programmers have been working on code optimization to improve performance and energy efficiency. However, the impact of code optimization on application resilience is often ignored. There are cases where optimizing code to improve application resilience is necessary (e.g., [48] and [105]). The code optimization (including common compiler optimization on applications) can change memory access patterns and runtime values of data objects, which in turn impacts error propagation and value shadowing. aDVF and its analysis give programmers a feasible tool to study and compare application resilience (from the perspective of data objects) before and after code optimization. The aDVF analysis is also helpful to pinpoint which part of the application code is vulnerable from the perspective of data objects, and hence demands further optimization.

Algorithm choice: To solve a specific computation problem, we can have multiple algorithm choices. For example, to solve the Poisson's equation on a 2D grid, we could use direct method (Cholesky factorization), Multigrid, or red-black successive over relaxation. Different algorithms have different implications on data distribution, parallelism, and blocking [10]. Which algorithm should be employed depends on users' requirements on performance, energy/power efficiency and resilience. aDVF and its analysis can help users (especially those users working on HPC) make the algorithm choice from the perspective of application resilience. It would be also interesting to integrate the aDVF analysis with programming language and compiler for algorithm choice, such as PetaBricks [10].

4.6.2 Beyond Single-Bit Errors

MOARD and aDVF calculation are general, meaning that they can be used for analyzing single-bit errors and multi-bit errors. In our study and evaluation, we focus on single-bit errors for two reasons: (1) Multi-bit errors rarely occur in HPC systems, and most of the existing studies on application resilience focus on single-bit errors; (2) Existing work reveals that multi-bit errors can have similar effects as single-bit errors on applications [127].

To use MOARD and aDVF for analyzing multi-bit errors, we need to make the following extension. (1) Define multi-bit error patterns. For example, for two-bit errors, the error pattern could be spatially contiguous; it could also be spatially separated (the spatial separation is four bits, for example). (2) Re-implement the function f (defined in Equation 4.1) in MOARD. This indicates that we must re-examine error masking. For the operation-level analysis, the effects of logical and comparison operations and value overshadowing will be different from that for single-bit errors; the effect of value overwriting may be the same as that for single-bit errors. For the error propagation analysis, we can use the same method as for single-bit errors to track error propagation, but the empirical bound of error propagation (i.e., the parameter k in Section 4.2.4) must be reset using fault injection tests. For the algorithm-level analysis, we use the

same fault injection-based method as for single-bit errors, but the injected errors must follow the defined error pattern.

4.6.3 Impact of Input Problems

The aDVF analysis is input dependent. This means that an application with different input problems may have different aDVF values for a data object. Such input dependence is because of multiple reasons. *First*, the effectiveness of operation-level error masking is input dependent. For example, a bit shifting operation for integers, $x >> y$, can tolerate a single bit error in the least significant bit of x if $y = 1$, but can tolerate three single bit errors in the three least significant bits of x if $y = 3$. *Second*, different input problems can result in different control flows, which in turn results in different error propagation. *Third*, different input problems can result in the employment of different algorithms. Different algorithms can result in different algorithm-level error masking.

Because of input dependence nature of the aDVF analysis, we must do the aDVF analysis whenever the application changes its input problem. This is a common limitation for many resilience study, including fault injection, AVF [18, 111], PVF [141], DVF [158] and [151]. However, a static analysis-based method cannot address the limitation because of unresolved branches and data values. Fortunately, MOARD allows a user to easily leverage hardware resource to parallelize the analysis (e.g., deterministic fault injection and trace analysis), making the analysis easy and efficient, even if the user has to repeatedly do the aDVF analysis. Furthermore, leveraging common iterative structures of HPC applications, analyzing a small trace of the application instead of the whole trace is often enough. This makes the repeated aDVF analysis even more feasible. Nevertheless, studying the sensitivity of aDVF analysis to input problems is our future work.

4.7 Conclusions

Understanding application resilience (or error tolerance) in the presence of hardware transient faults on data objects is critical to ensure computing integrity and enable efficient application level fault tolerance mechanisms. The traditional methods (such as random fault injection) cannot help because of losing data semantics and insufficient information on how and where errors are tolerated. This chapter introduces a fundamentally new method to quantify application resilience to transient faults on data objects. In essence, our method measures error masking events at the application level and associates the events with data objects. We perform a comprehensive classification of error masking events and create a series of techniques to recognize them. We develop an open source tool to quantify application resilience from the perspective of data objects. We hope that our method can make the quantification a common practice. Currently, the deployment of fault tolerance mechanisms is often a problem because of a lack of a method to quantify its effectiveness on protecting data objects. Our work provides a tangible solution to address the problem.

Chapter 5

Predicting Application Resilience Using Machine Learning

5.1 Introduction

In this chapter, we present a novel framework called PARIS[1], which avoids the time-consuming process of randomly selecting and executing many injections (as in FI), and provides higher prediction accuracy than analytical models, making it a unique solution to the problem. In essence, PARIS uses a machine learning model to predict application resilience, which provides several advantages. *First*, machine learning models, once trained, can be repeatedly used for any fault manifestations—silent data corruption (SDC), interruptions, and success cases—for new, previously unseen applications. Therefore, PARIS avoids a large amount of repeated fault injection tests, which leads to high efficiency in comparison to FI. *Second*, machine learning models can capture the implicit relationship between application characteristics (e.g., intensity of resilience computation patterns) and application resilience, which is difficult to capture by analytical models.

The most challenging part of using the machine learning approach is to efficiently build effective features that can cause high prediction accuracy. We use the following methods to construct features. First, we count the number of instruction instances within each instruction type as a feature; instruction instances are dynamic execution of instructions. We characterize instructions in such a way because different instruction types show different resilience to errors [29, 75]. To reduce the number of features, we classify instruction types into four representative and discriminative groups in terms of the functionality of instructions. This reduction of features reduces the training complexity and avoids undertraining.

Second, we count resilience computation patterns as features. Guo et al. [65] discover six resilience computation patterns from HPC applications. Those patterns are considered the fundamental reason for application resilience. Four of those patterns are based on individual instructions, and can be included as features using the above instruction type-based approach. The remaining two ("dead locations" and "repeated addition") contain more than one instruction and cannot be captured by examining instructions individually. To efficiently count the two patterns, we introduce optimization

[1] PARIS: **P**redicting **A**pplication **R**e**S**ilience.

techniques to avoid repeatedly scanning the instruction trace and find correlation between instructions.

Third, we introduce instruction execution order information into features to improve modeling accuracy. Execution order information is important to application resilience, because error propagation is highly correlated to the order and type of operations. Inspired by "N-gram" technique [122, 37] in computational linguistics, we embedded the sequence of instruction chunks into features to introduce execution order of instructions. Our evaluation shows that having execution order information decreases prediction error by up to 30%.

Fourth, we introduce *resilience weight* when counting instruction instances. Different instruction instances, even though they have the same instruction type, can have different capabilities to tolerate faults. Resilience weight quantify the resilience difference of those instruction instances. Introducing resilience weight decreases prediction error by 13% on average when predicting the rate of some fault manifestation (particularly, the interruption rate).

Based upon the above features, we use feature selection techniques to sort and further reduce features. We perform ablation study to understand the sensitivity of features to prediction accuracy. We reveal significance of memory-related instructions and data overwriting to application resilience.

In summary, our contributions are three folds. (1) We present PARIS, a machine learning-based approach to predict application resilience. Our method breaks the fundamental tradeoff between evaluation speed and accuracy in the existing common practice to estimate application resilience. (2) We develop a framework and overcome a series of technical challenges for feature construction, extraction and selection. We reveal how to use machine learning to effectively and efficiently model application resilience. (3) We test our model on 16 benchmarks. We find that our approach is up to 450x faster than random FI (49x on average). The model has high prediction accuracy: a prediction error of 8.5% and 22% on average for predicting success rate and interruption rate (excluding two obvious outliers) respectively. We compare PARIS with Trident [98] (the state-of-the-art analytical model): PARIS can predict any fault manifestation rate (SDC, interruptions, and success), while Tridentonly predicts SDC rate; PARIS is at least 63% better than Tridentin terms of accuracy for predicting SDC rate, and has comparable execution time (but faster for 12 out of the 16 benchmarks with 15x speedup on average).

5.2 Overview

Our problem to predict application resilience is naturally a regression problem. More formally, we aim to find a model $f()$, such that given an feature vector v corresponding to an application A, $f(v)$ gives us the rates of SDC, interruption, and success for A. We give a high-level overview of PARIS. Figure 5.1 depicts the workflow of the training process of PARIS. The most challenging part of the training process is to construct features relevant to application resilience that can produce high modeling accuracy.

Features Construction. We use instruction type and number of instruction instances for each type as a feature. A static instruction in a program has an instruction type (opcode), and can be executed many times, each of which is an *instruction instance*. Using the number of instruction instances for each instruction type as a feature

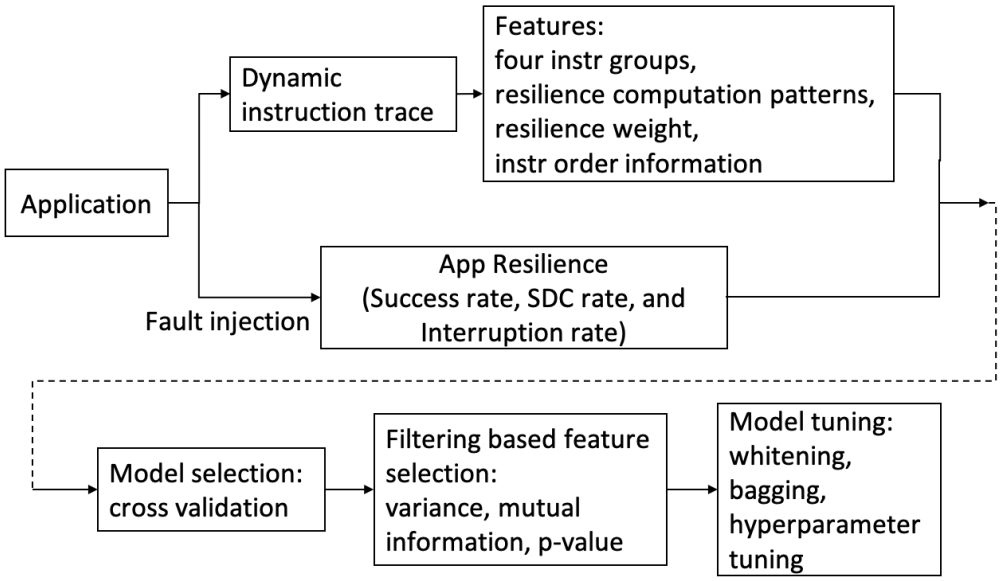


Figure 5.1: Overview of PARIS and the workflow of the training process in our ML method.

will result in too many features, which demands a large training dataset. To reduce the number of features, we group all instruction types (65 in total) into four groups: control flow instructions, floating point instructions, integer instructions, and memory-related instructions. For each instruction group, we count the number of instruction instances as a feature.

Furthermore, we use six resilience computation patterns proposed in [65] as features. Among the six patterns, four of them (conditional statement, shifting, data truncation, and data overwriting) are individual instructions that are not grouped into the four instruction groups, because of the significance of these instructions to application resilience. Two of them (dead corrupted locations and repeated additions) include multiple instructions, where these instructions all together contribute to application resilience.

Counting dead corrupted locations and repeated additions from the dynamic instruction trace as features is challenging, because we must repeatedly search within the trace to find correlation between instructions. To detect dead corrupted locations, we cache intermediate results of trace analysis to avoid repeated trace scanning. To detect repeated additions, we build a data dependency graph for addition instructions. Such graph enables easy detection of repeated additions.

Because different instruction instances can have different capabilities to tolerate errors, even though those instruction instances have the same instruction type (or the same resilience computation pattern), we introduce *resilience weight* when counting instruction instances. The resilience weight gives each instruction instance a weight quantifying the possible number of single-bit errors tolerable by the instance.

Furthermore, we introduce IEO information as a feature. We demonstrate that a small change in IEO can affect the application resilience using an example illustrated in Figure 5.3 and described in Section 5.3.2. However, representing the execution

order information of all instruction instances as a feature is a challenge. We use N-gram [122, 37], a technique commonly used for processing speech data, to capture the order information.

Training and Testing Phases. The modeling process of ML includes training and testing. We use a set of representative applications to train the model—once it is trained, the model is used to predict, or test, the manifestation rates on new applications. We call the applications used for training and testing the training dataset and the testing dataset, respectively.

Prediction Accuracy. To evaluate the trained model, we compare the Mean Absolute Percentage Error (MAPE) [46] of the predicted application resilience against the ground-truth application resilience measured by performing FI. Equation 5.1 gives the definition of MAPE. MAPE is often used for regression model evaluations because it can interpret modeling accuracy in terms of *relative errors* [46]. A low MAPE means a better accuracy. The lowest MAPE is zero.

$$MAPE = \frac{Measured - Predicted}{Measured}. \tag{5.1}$$

5.3 Design

5.3.1 Feature Construction

For feature construction, we have the following requirements: (1) features should be relevant to application resilience; (2) the number of features should be small enough (smaller than the number of applications used for training) to avoid under-determination of the model; (3) we should avoid redundant and irrelevant features since these features can increase prediction error. Following the above requirements, we introduce instructions, resilience computation patterns, resilience weight, and Instruction Execution Order (IEO) as features. We describe why and how we collect these features in following subsections.

Instruction Groups

The primary features are instruction types and number of instruction instances in each type. These features are highly relevant to application resilience. For example, recent studies [106, 98] reveal that floating point instructions are highly related to resilience because the faults in mantissa bits of floating-point numbers can be negligible by the application (especially HPC applications). Load/store instructions also have a significant impact on application resilience, because computations following load/store instructions can take those loaded/stored values.

We use the following method to construct instruction-based features. We use LLVM-Tracer [136], an LLVM pass to compile the application and generate a dynamic LLVM instruction trace. The LLVM instructions are architecture independent, allowing us to build a more general and reusable model. We enumerate all LLVM IR instructions and get 65 instruction types.

We could add all 65 instruction types as features. However, this significantly increases the number of features. With the introduction of IEO as features (See Sec-

Table 5.1: Four groups of instruction types and four resilience computation patterns as features to build our ML model.

Group Name	**Instruction types**
Control Flow Instructions (CFI)	Br, Indirectbr, Select, PHI, Fence, DMAFence, Call
Floating Point Instructions (FPI)	Fadd, Fsub, Fmul, Fdiv, Frem, Cosine, Sine
Integer Instructions (II)	add, sub, mul, Udiv, Sdiv, Urem, Srem
Memory-related Instructions (MI)	Load, Store, DMAStore, DMALoad, Getelementptr, ExtractElement, InsertElement, ExtractValue, InsertValue, FPToUI, FPToSI, UIToFP, SIToFP, PtrToInt, IntToPtr, AddrSpaceCast
Pattern name	**Instruction types**
Conditional Statements	ICmp, FCmp, Switch, And, Or, Xor
Shifting	Shl, LShl, AShl
Data Truncation	Trunc, ZExt, Sext, FPTrunc, BitCast, FPExt
Data Overwriting (DO)	All instructions having at least one output operand

tion 5.3.2 for why and how we introduce IEO into features), the number of features will be more than 195, larger than the number of training samples, which makes the training under-determined.

To address the above problem, we group 65 instruction types into four groups based on the functionality of instructions to reduce the number of features. For example, we group control flow related instructions (e.g., `Br` and `Select`) into a group. Table 5.1 lists the four groups, including control flow instructions, floating point instructions, integer instructions, and memory-related instructions. For each instruction group, we count the number of instruction instances from the dynamic instruction trace, and then normalize the number by the total number of instruction instances. We use the normalized number as a feature to make the feature value independent of the size of the dynamic instruction trace. This enables us to fairly compare application resilience of applications with different trace sizes.

Using Resilience Computation Patterns as Features

Recent work [65] finds six resilience computation patterns (dead corrupted locations, repeated additions, conditional statements, shifting, data truncation, and data overwriting) the fundamental reason for application resilience. A resilience computation pattern is defined as a combination of computations that affect application resilience. The reason we introduce dead corrupted locations and repeated additions as features is that the two patterns are composed of multiple instructions that together contribute to application resilience [65]. The other four patterns (conditional statements, shifting, data truncation, and data overwriting) are individual instructions shown in Table 5.1. We use them separately as features because of their especial significance to application resilience [65].

To count the six patterns as features, we cannot use the method in [65], because it tracks error propagation after fault injection and leverages error masking to discover *unknown* patterns, whereas they do not provide a method to count patterns from the application. We must propose our own method to count resilience patterns from applications to construct features. To efficiently count patterns, we must address below

challenges.

First, counting the number of pattern instances[2] for dead corrupted locations and repeated additions is time-consuming, because we must find correlations between instructions to determine if the location is dead or if addition repeatedly happens to the same variable. Doing so requires repeatedly scanning dynamic instruction trace. We discuss how to efficiently count pattern instances for the two patterns in Section 5.3.1 and Section 5.3.1, respectively.

Second, for the patterns that are represented as individual instructions (see the last four rows in Table 5.1 for these instructions), simply counting the number of pattern instances cannot discriminate resilience capabilities of different pattern instances. For example, the resilience capability of the "shifting" pattern (a pattern involving a *shift* instruction) depends on how many bits are shifted. A *shift* instruction instance shifting three bits can tolerate three single-bit errors, while a *shift* instruction instance shifting one bit can only tolerate one single-bit error. To distinguish fault tolerance capabilities of different instruction instances, we introduce weights (named *resilience weight*) when counting instances of the patterns.

Besides introducing weights for the four patterns, we also introduce weights to instructions of instruction groups whose instances can also have different fault tolerance capabilities. We describe the relevant details in Section 5.3.1.

Extracting the Feature of Dead Corrupted Locations

A combination of operations (e.g., additions and multiplications) aggregate the values of corrupted input locations into fewer output locations. Meanwhile, many of these corrupted input locations are not used anymore (they become dead corrupted locations), which leads the total number of corrupted locations to decrease. A code region with a higher percentage of locations that are dead corrupted locations has higher resilience.

To efficiently detect dead corrupted locations and calculate the percentage of dead corrupted locations, we split the dynamic instruction trace into chunks and pre-process the chunks before detecting dead corrupted locations. A chunk of instructions is the dynamic instruction trace of a first-level inner loop or the code region between two neighbor first-level inner loops. During the trace pre-processing, we analyze instructions in each chunk and save locations of each chunk into an array. To determine if a location in a chunk is dead, we check whether the location is further used in any future chunks by examining the sequence of arrays. If the location is not used in any future chunks, then the location is a dead corrupted location. In essence, the arrays for chunks save instruction analysis results to avoid repeatedly scanning the trace. For each chunk, we compute the percentage of locations that are dead corrupted locations for the chunk. We use the average percentage of dead corrupted locations across all chunks (named "dead corrupted location rate" or DLR) as a feature.

Extracting the Feature of Repeated Additions

Repeated additions (RA) refers to the addition operations repeatedly happening to the same variable, such that the corruption in the variable can be amortized. To decide if

[2]A pattern is repeatedly executed in application execution. We name the dynamic execution of a specific pattern the *pattern instance*.

```
1   for(i=0;i<N;i++){
2       …
3       e = a + 4;
4       …
5       y = d + e;
6       …
7       c = z + y;
8       …
9       a = b + c;
10      …
11  }
```

We use a data dependency tree to decide repeated addition at 'a'.

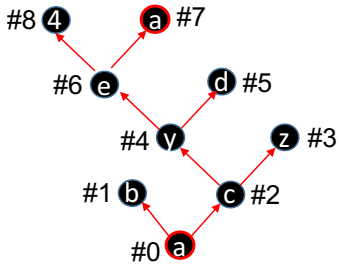


Figure 5.2: An example to detect repeated additions.

an addition instruction is part of repeated additions, we must first decide if the addition instruction is involved in a self addition. The self addition is defined as that a location adds other locations to itself. The pseudo code in Figure 5.2 is an example of self addition.

To detect a self addition, we first build a data dependency graph for addition operations, where nodes are locations; edges between nodes represent data dependency. When given an addition instruction, we examine its output operand and decide if the location (the output operand) is an input operand of a previous addition operation by backward traversing the graph.

Figure 5.2 illustrates what a data dependency graph looks like and how a self addition is found. We have four addition statements (operations) in a *for* loop. The location a appears as the output of the last addition statement ($a = b+c$ in Line 9). To determine if the addition statement is involved in a self addition, we find the node 0 corresponding to a in the data dependency graph. We traverse the graph backward, and find a appears in a previous node, the node 7. The node 7 corresponds to a source operand of a previous addition statement ($e = a + 4$). Doing so, a self addition is detected. A pattern of repeated additions is composed of multiple self additions.

To use repeated additions as a feature, we normalize the number of repeated additions by total number of instruction instances. This makes the feature value independent of the size of the dynamic instruction trace.

Resilience Weight

Given an instruction, all bit locations of its input and output operands are subject to error corruption. The resilience weight ($\mathcal{R}es$) of an instruction is defined below.

$$\mathcal{R}es = \frac{\#bit\ locations\ that\ tolerate\ errors}{\#of all\ bit\ locations} \tag{5.2}$$

Using the *right-shift* instruction as an example. The instruction has three 8-bit operands and in total 24 locations. Assume that an instance of the instruction shifts four least significant bits of an operand. The shifted four bits can tolerate four single-bit errors. Also, the eight bits in the output operand of the instruction can tolerate errors

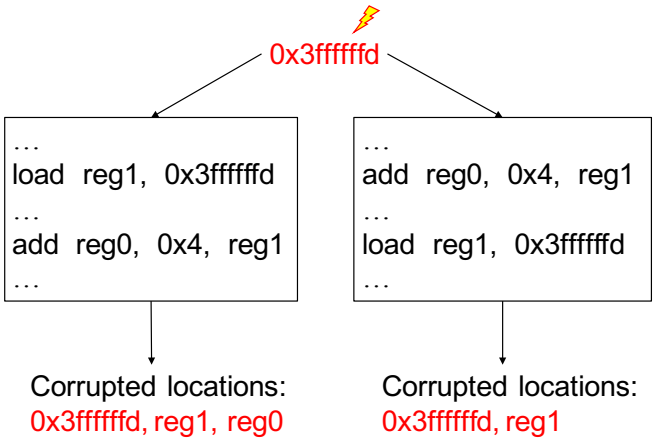


Figure 5.3: An example to show that the instruction execution order matters to error propagation.

because of the result overwriting in the output operand. Hence, in this example, the resilience weight for this instruction instance is (4+8)/24 = 0.5. Consequently, the bit locations that can constantly tolerate errors are bit locations of the output operands, because we expect errors in the output operands to be overwritten. Notably, we use the weight in case counting the number of instruction instances or the number of pattern instances.

Putting All Together. As a result of feature construction, we construct a feature vector of ten features, formulated in Equation 5.3 where "DLR" and "RA" are the dead corrupted locations and repeated additions, respectively. Notations for the equation can be found in Table 5.1.

$$\begin{aligned}\mathcal{F}_{10}^{ave} = [&CFI, FPI, II, MI,\\ &Condition, Shift, Truncation, DO, DLR, RA]\end{aligned} \tag{5.3}$$

We call $\mathcal{F}_{10}^{ave}$ the *foundation feature vector* and consistently call the ten features *foundation features* in the rest of the chapter.

5.3.2 Introducing Instruction Execution Order (IEO)

The foundation features are not good enough to achieve high prediction accuracy. In particular, the foundation features lack IEO information. Capturing the IEO is important because it matters to error propagation.

We use an example shown in Figure 5.3 to depict why IEO matters. In this example, we have a $load$ instruction and an $addition$ instruction. Assume that an error happens in a memory address 0x3fffffffd. If the $load$ instruction happens first, then the erroneous value in the memory address propagates to the locations $reg1$ and $reg0$. But if the $addition$ instruction happens first, then the erroneous value in the memory address only propagates to the location $reg1$. This example is a demonstration of how IEO matters to error propagation.

To introduce IEO into the feature vector, we use the "N-gram" technique [37]. The N-gram is a technique used in computational linguistics. It can work on a sequence of streaming words, and predict the next word using sequences of previous words. N-gram can capture the word order information. Particularly, every n continuous words

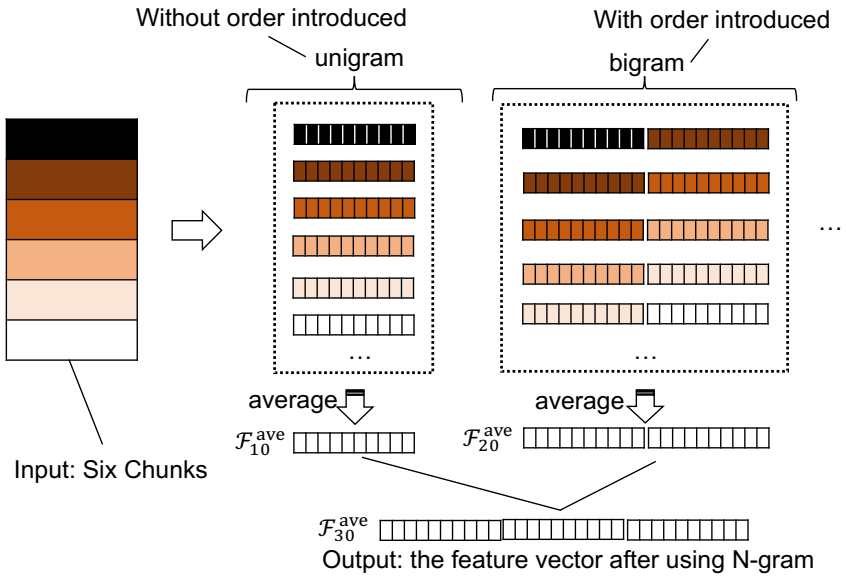


Figure 5.4: Applying the N-gram technique to introduce instruction execution order information.

compose an n-gram ($n = 1, 2, 3, ...$). Figure 5.4 depicts how we build the feature vector with IEO included. Particularly, we partition the dynamic instruction trace into chunks (each chunk is a gram). Each chunk is regarded as a "word", and the sequence of chunks is processed as the sequence of "words". For each chunk, we collect the ten foundation features and build a foundation feature vector of size ten. Then, we build an average foundation feature vector (denoted as $\mathcal{F}_{10}^{ave}$) which is the average of foundation feature vectors of all chunks.

Furthermore, we combine every two neighboring chunks to build a bigram (n=2 for n-gram). Particularly, we concatenate two foundation feature vectors to build a bigram feature vector of size 20. We then build an average bigram feature vector (denoted as $\mathcal{F}_{20}^{ave}$) which is the average of all bigram vectors.

In consequence, we have $\mathcal{F}_{10}^{ave}$ of size 10 and $\mathcal{F}_{20}^{ave}$ of size 20. The final feature vector with IEO information in consideration is the combination of $\mathcal{F}_{10}^{ave}$ and $\mathcal{F}_{20}^{ave}$. The final feature vector has a size of 30, which is denoted as $\mathcal{F}_{30}^{ave}$.

We do not consider trigram (i.e., 3-gram) or higher gram, because existing research [37] demonstrate that there is no need to use higher grams than bigram. In [37], bigram achieves better accuracy than trigram while using trigram or higher grams does not provide better prediction accuracy but dramatically increases feature vector size and complexity of model training.

5.3.3 Feature Selection

Following the requirement of feature construction, we aim to eliminate irrelevant and redundant features and further reduce the feature vector size. We use three filtering-based methods to select features. Compared to other feature selection methods such as wrappers and embedded methods, the filtering-based methods are faster because of their simplicity and low complexity. In addition, the filtering-based methods are independent

of the prediction model [66]. In such a way, the selected features can be used with different prediction models.

We use the following filtering-based methods to select features: the p-value-based method [24], the mutual information-based method [13], and the method of calculating variance [66]. Simply speaking, the p-value is a metric that measures the significance level between a feature and the modeling result (i.e., the success, SDC, or interruption rate). The mutual information measures the mutual dependency between a feature and the modeling result. The variance measures the variance of feature values across different input applications. Using each of the three methods, we can rank features into a sorted list according to the importance of features with respect to application resilience. In total, we have three lists.

Using a voting strategy, we combine the three sorted lists of features into one list for feature selection. This voting strategy and feature selection algorithm are common in ML [161]. In particular, each feature has an index in each of the three lists. For each feature, we add its three indexes to get a global index. We sort the features based on global indexes into a single list.

We then decide how many features we want to use to construct the feature vector for modeling. Based on the sorted features in the single list, we choose the best k (where $k = 2, 3, ..., 30$) features to build a sublist of features. In total, we have 29 sublists. We choose the features in the best sublist (in terms of the prediction accuracy) as the final features.

5.3.4 Model Construction

Model Selection. There are tens of regression models. Each of them has pros and cons, and can fit into different scenarios. We use scikit-learn [121] and test all regressions models in scikit-learn (18 in total). We use cross-validation (CV) to test 18 regression models on the training dataset to select a regression model with the best prediction accuracy. CV partitions the dataset into p folds. q of p folds are used for training, while the remaining $p - q$ folds are used for testing. There are $p/(p - q)$ rounds of training/testing. In each round, different $p - q$ folds are used for testing. We choose the regression model that has the lowest prediction error on average. We use 10-fold cross validation in our study. Based on the CV results, we choose the Gradient Boosting Regression to predict application resilience.

Model Tuning. We use the following techniques to tune the model for better prediction accuracy. (1) Whitening [42]. Whitening is used to normalize features to avoid domination effects of any features for better generalization and to improve the modeling accuracy. (2) Bagging (model averaging) [50], which is often used for reducing variation in training data. We use this technique to eliminate the effect of bad outliers. (3) Hyperparameters tuning. Each regression model has multiple hyperparameters. We use "grid-search" [16] to decide the values of hyperparameters for training.

5.4 Implementation

Dataset Construction. We have multiple requirements for creating training and testing dataset. (1) The training dataset must be large to avoid model underdetermination; (2)

Applications used to generate training and testing dataset must have diverse computation and diverse resilience characteristics; (3) Applications used to generate training and testing dataset must have explicit result verification phases. Having the verification phase allows us to determine the fault manifestations.

We use representative benchmark suites and scientific applications to create the testing dataset, including NAS parallel benchmark suite [12], PARSEC benchmark suite [17], CORAL benchmark suite [1], Rodinia benchmark suite [36], and two scientific applications (Hercules for earthquake simulation [6] and PuReMD for reactive molecular dynamics simulation [146]). From these resources, we choose 16 applications for testing because of their diverse characteristics. The 16 applications are shown in Table 5.2. We call the 16 applications *big benchmarks* in the rest of the chapter.

To train `PARIS`, we use 100 common computation kernels obtained from HackerRank [67]. These kernels are smaller than the big benchmarks, but these kernels all have explicit verification phases. With these kernels, the ranges of modeling output during training are [0.126; 0.982], [0.000; 0.656], and [0.018; 0.874], for the rates of success, SDC, and interruption, respectively; The average values of modeling output during training are 0.502, 0.155, and 0.348 with a variance of 0.033, 0.019, and 0.021 for the rates of success, SDC, and interruption, respectively. The above numbers show that our training is sufficient with these kernels. Also, using the 100 computation kernels is adequate for training because the training is determined when the size of training dataset (100) is larger than the number of features (30).

Trace Generation. We use LLVM-Tracer [136], a tool to generate dynamic LLVM IR traces based on LLVM instrumentation. The trace includes LLVM IR instructions and their operands. We extend LLVM-tracer to generate a subtrace for each chunk of instructions and generate traces for MPI programs.

Whitening. We use the whitening technique [42] to normalize features to avoid domination effects of any features for better generalization and to improve the modeling accuracy.

5.5 Evaluation

We use the trained model to predict the rate of success and interruption (two classes of fault manifestation). We then calculate the SDC rate by subtracting the rates of success and interruption from one (“1”). We do not directly predict the SDC rate, because the value of SDC rates can be zero for small computation kernels, in which any variation when predicting the SDC rate can cause unreasonable MAPE of infinite values when the denominator in the MAPE Equation is zero. Hence, Table 5.3, Figure 5.6 and Figure 5.7 do not have results for SDC.

Using the above approach to predict the SDC rate can cause a negative SDC rate. This is because we predict success and interruption rates independently, and there is a chance that the sum of predicted success and interruption rates is larger than one (“1”). For such cases, we force the value of the SDC rate to be zero. Also, we normalize the three rates by their sum in case the sum of the three rates is larger than one.

We evaluate our model and modeling methods from two perspectives: (1) modeling accuracy; (2) contributions of modeling and optimization techniques to modeling accuracy.

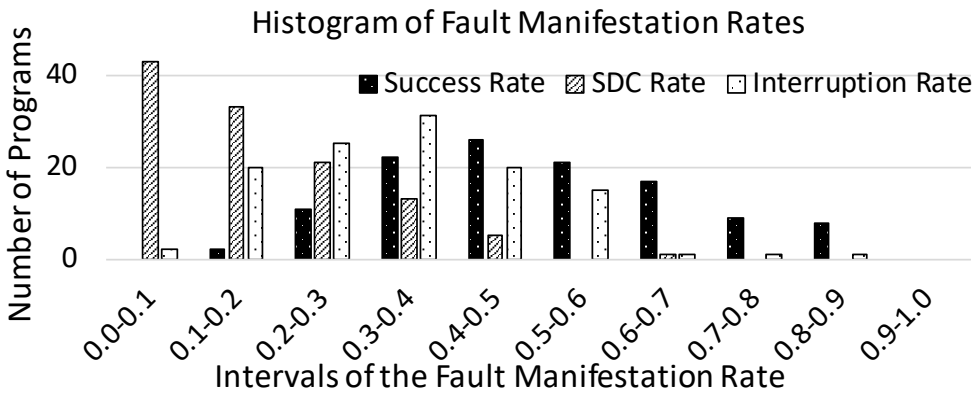


Figure 5.5: Histogram of the three fault manifestation rates.

Artifact Description. We conduct experiments on compute nodes each equipped with Intel(R) Xeon(R) CPU E5-2630 v3 and Ubuntu-14.04.5. Each compute node has Clang-v3.4, OpenMP-v4.0, and scikit-learn installed.

5.5.1 Prediction Accuracy

Table 5.2 shows the prediction results. Using the results of traditional fault injection as ground truth, MAPE for success rate and SDC rate are 8% and 45%, respectively. Our prediction accuracy for success rate is overall good, but our prediction accuracy for SDC rate is relatively low, but better than the state-of-the-art (see the following discussion in "Comparison with the state-of-the-art for predicting SDC rate"). Predicting SDC rate is challenging because SDC rate can be very small or even zero. A small deviation from the ground truth can cause a large prediction error to MAPE.

To support the statement that the SDC rate tends to be small, we study 116 programs from training and testing datasets. We perform random fault injection and count the histogram (shown in Figure 5.5) of the three fault manifestation rates of these programs. Figure 5.5 shows that there are more than 65% of programs whose SDC rates are distributed in the range of $0.0 - 0.2$, while values of success rate and interruption rate are distributed in a greater range. We further find that 40% of the programs have the SDC rate less than 0.1.

Comparison with the State-of-the-Art for Predicting the SDC Rate. We compare `PARIS` with `Trident` [98], a recent work that uses analytical models to estimate the SDC rate. We use `Trident` downloaded from their github website (commit #90b38ab) to estimate the SDC rate for the 16 big benchmarks. The 16 benchmarks include all the benchmarks used in Trident; the number of benchmarks used in Trident is 11. For the 11 benchmarks, we use the same input as in [98]. Table 5.2 shows the prediction error of `Trident` in the fourth last column.

Table 5.2 shows that the MAPE of `PARIS` for SDC rate is 45%, while the MAPE of `Trident` for SDC rate is 680%. We notice that there are two outliers (MG and PuReMD) that make the average prediction error of `Trident` very large. To make the comparison fair, we remove the two outliers. After that, the new MAPE of `Trident`

Table 5.2: The detailed prediction results for 16 big benchmarks. Notation: SR=Success Rate; SDCR=SDC Rate; IR=Interruption Rate; Pred.=Prediction; Meas.=Measured.

Big benchmarks	Suite	Program input	Meas. SR	Pred. SR	Relative Error for SR
IS	NAS	Class S	0.653	0.625	4.23%
Nn	Rodinia	filelist_4 5 30 90	0.980	0.910	7.16%
Myocyte	Rodinia	100 1 0 4	0.741	0.764	3.11%
MG	NAS	Class S	0.781	0.721	7.75%
Kmeans	Rodinia	100	0.843	0.749	11.12%
Libquantum	SPEC	33 5	0.863	0.879	1.85%
Blackscholes	PARSEC	in_4.txt	0.663	0.591	10.81%
Sad	Parboil	reference.bin frame.bin	0.475	0.506	6.53%
Bfs-parboil	Parboil	graph_input.dat	0.960	0.906	5.61%
Hercules	CMU	scan simple_case.e	0.580	0.646	11.36%
PuReMD	Purdue Univ.	geo ffield control	0.420	0.438	4.26%
Lulesh	CORAL	-s 1 -p	0.634	0.441	30.44%
Hotspot	Rodinia	64 64 1 1 temp_64 power_64	0.714	0.752	5.30%
Bfs-rodinia	Rodinia	graph4096.txt	0.655	0.674	2.92%
Nw	Rodinia	2048 10 1	0.664	0.647	2.49%
Pathfinder	Rodinia	1000 10	0.623	0.759	21.89%
MAPE	N/A	N/A	N/A	N/A	**8.55%**

(a) Prediction results for success rate

Big benchmarks	Meas. SDCR	Pred. SDCR	Relative Error for SDCR	**Relative Error for SDCR by `Trident`**
IS	0.083	0.092	11.14%	192.31%
Nn	0.000	0.000	0.00%	93.39%
Myocyte	0.022	0.025	14.67%	826.67%
MG	0.008	0.010	31.14%	5633.33%
Kmeans	0.045	0.098	117.93%	42.64%
Libquantum	0.034	0.000	100.00%	7.60%
Blackscholes	0.122	0.210	72.05%	12.22%
Sad	0.216	0.318	47.36%	34.95%
Bfs-parboil	0.000	0.000	0.00%	3.32%
Hercules	0.182	0.1822	0.11%	128.19%
PuReMD	0.090	0.018	80.00%	3740.00%
Lulesh	0.120	0.255	112.85%	39.01%
Hotspot	0.121	0.124	2.86%	58.97%
Bfs-rodinia	0.124	0.047	62.10%	31.31%
Nw	0.140	0.193	38.34%	20.96%
Pathfinder	0.080	0.052	35.02%	20.81%
MAPE	N/A	N/A	**45%**	**108% (with outliers removed)**

(b) Prediction results for SDC rate

Big benchmarks	Meas. IR	Pred. IR	Relative Error for IR
IS	0.264	0.283	6.97%
Nn	0.02	0.090	350.95%
Myocyte	0.237	0.211	11.07%
MG	0.211	0.269	27.49%
Kmeans	0.112	0.153	36.32%
Libquantum	0.103	0.121	17.51%
Blackscholes	0.215	0.199	7.55%
Sad	0.309	0.176	42.91%
Bfs-parboil	0.040	0.094	134.54%
Hercules	0.238	0.172	27.76%
PuReMD	0.490	0.544	10.93%
Lulesh	0.246	0.304	23.69%
Hotspot	0.165	0.124	25.03%
Bfs-rodinia	0.221	0.279	26.43%
Nw	0.196	0.159	18.94%
Pathfinder	0.279	0.189	32.38%
MAPE	N/A	N/A	**22% (with outliers removed)**

(c) Prediction results for interruption rate

is 108%, which is still worse than the prediction of `PARIS`. We conclude that `PARIS` is better than `Trident` in terms of the prediction accuracy on SDC.

Notably, Li et al. [98] reports Mean Absolute Error (MAE), which is different from MAPE we report. When evaluating the SDC rate, MAE may not be as appropriate as MAPE. A small MAE (e.g., 0.01) can cause a large MAPE. MAPE measures the relative error. When relative variation matters and needs to be considered, MAPE is better than MAE [46].

Even though `PARIS` is better than `Trident` in predicting the SDC rate, `PARIS` shows a high relative error on some benchmarks. For example, the relative prediction error for SDC rate for Kmeans, Libquantum, and Lulesh are 117%, 100%, and 112%, respectively. After examining the prediction results closely, we find that the absolute prediction error for the three benchmarks are 0.053, 0.034, and 0.135, respectively, which are small; the ground truth of the SDC rate for the three benchmarks are 0.045, 0.034, and 0.120, respectively, which are also small and close to zero. Accordingly, although the absolute prediction error for SDC is smaller with `PARIS` comparing to `Trident` (on average 0.041 with `PARIS` vs. 0.063 with `Trident`), the relative prediction error with `PARIS` for SDC can be large but still smaller comparing to `Trident`.

Prediction of the Interruption Rate. The MAPE for predicting the interruption rate is 50%. This prediction error seems relatively high. However, we find two outlier benchmarks, which contribute to the bad prediction accuracy. They are Nn and Bfs_parboil. The MAPE for them are 350% and 134%, respectively. Excluding the two outliers, the new MAPE for predicting interruption rate is 22% which is much acceptable.

After we profile Nn and Bfs_parboil, we find that these codes have a relatively large number of load instructions (19% and 44% of total instructions), which is larger than

Table 5.3: Feature voting scores for each dimension of the feature vector $\mathcal{F}_{30}^{ave}$.

(a) Feature voting scores for predicting the success rate.

Dimension Number	4 24 8 28 17 12 14 22 18 27
Sorted voting score (Smaller is better)	20 22 23 24 25 27 29 29 31 32
Dimension Number	2 3 23 7 20 16 6 21 13 26
Sorted voting score (Smaller is better)	33 39 39 40 43 45 46 48 50 50
Dimension Number	11 1 30 15 5 10 25 29 9 19
Sorted voting score (Smaller is better)	53 54 62 69 70 71 74 74 86 87

(b) Feature voting scores for predicting the interruption rate.

Dimension Number	14 18 4 8 27 24 28 7 30 16
Sorted voting score (Smaller is better)	20 23 24 27 27 32 32 34 37 38
Dimension Number	6 17 10 26 12 13 1 3 11 2
Sorted voting score (Smaller is better)	39 40 42 43 46 46 47 47 52 53
Dimension Number	21 19 20 23 5 15 22 25 9 29
Sorted voting score (Smaller is better)	53 55 55 56 62 63 69 69 77 87

(c) The application characteristics that each dimension of the feature vector represents. Dimensions larger than 9 have the information of instruction execution order using the N-gram technique.

Dimension#	1, 11, 21	2, 12, 22	3, 13, 23	4, 14, 24	5, 15, 25
Meaning of dimension#	CFI	FPI	II	MI	Condition
Dimension#	6, 16, 26	7, 17, 27	8, 18, 28	9, 19, 29	10, 20, 30
Meaning of dimension#	Shift	Truncation	DO	DLR	RA

that in most of the benchmarks we study. Predicting the interruption rate accurately depends on accurately counting load instructions because loading data from an incorrect address often cause segmentation faults (or interruptions). However, we do not accurately count load instructions during feature construction, because load and other memory-related instructions are counted together as an instruction group (see Table 5.1). Thus using a group (as opposed to a single instruction class) for counting causes low prediction accuracy in this case.

In summary, while the method of using instruction groups as features may cause high prediction error, we use groups to limit the number of features to reduce training time and the necessity of using many training samples. Hence, there is a tradeoff between training efficiency and prediction accuracy.

Discussion. We achieve a high prediction accuracy for predicting success rate in contrast to the prediction on SDC and interruption rates. Our quick (See Section 5.5.4

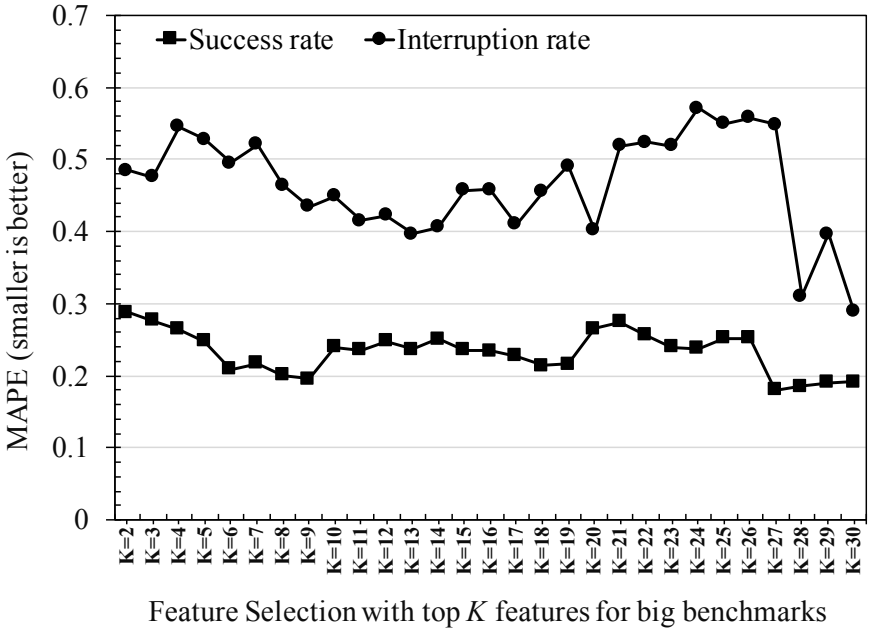


Figure 5.6: The ablation study result: the average prediction error for predicting the rates of success and interruption when the best k features are selected (k ranges from 2 to 30).

for the efficiency study) and accurate prediction on success rate is valuable in practical. For example, when deciding the application-level fault tolerance mechanism for a code, the resilience (or success rate) of the code in the presence of errors is the key concern [33]. When the success rate is high (close to 1), which means the code has a high resilience to errors. In this case, one would use cheap fault tolerance mechanisms rather than expensive ones. Therefore, having an efficient and accurate way to estimate resilience (or success rate) of the code is beneficial for directing fault tolerance mechanisms.

5.5.2 Feature Selection and Analysis

Recalling that we use a voting strategy for feature selection. With the voting strategy, we have a global index for each feature. The global index aggregates voting results of the three feature selection methods (p-value, mutual information, and variance). Table 5.3 shows the global indexes for all 30 dimensions of the feature vector. The application characteristics that each dimension of the feature vector represents is summarized in Table 5.3.c. Table 5.3.a reveals that the 4th dimension (the memory-related instructions), 24th dimension (the memory-related instructions in bigram), and 8th dimension (the pattern of overwriting) in $\mathcal{F}_{30}^{ave}$ rank the highest; Table 5.3.b reveals that the 14th dimension (the memory-related instructions in bigram), 18th dimension (the pattern of overwriting in bigram), and 4th dimension (the memory-related instructions) in $\mathcal{F}_{30}^{ave}$ rank the highest. Those dimensions are memory-related instructions, which seem to matter most to the application resilience.

In addition, both tables reveal that the 9th dimension (i.e., the pattern of dead loca-

tion), 19th dimension (i.e., the pattern of dead location in bigram), and 29th dimension (i.e., the pattern of dead location in bigram) rank relatively low. This result indicates that dead location seems to have less impact to application resilience than the other features.

Ablation study. In this study, we show the effect of using the best k features to make a prediction ($k = 2, 3, ..., 30$) to prediction accuracy. This study can also help us understand the contributions of each feature to prediction accuracy. Figure 5.6 shows the result of the ablation study. The figure shows the prediction error for the rates of success and interruption.

In Figure 5.6, the prediction error decreases by 17% (from 0.3 to 0.25) for predicting the success rate when adding MI-related features (4th and 24th dimensions in $\mathcal{F}_{30}^{ave}$) and data overwriting related features (8th and 28th dimensions in $\mathcal{F}_{30}^{ave}$). Moreover, the prediction error decreases another 24% (from 0.25 to 0.19) after adding truncation in bigram (17th dimension) into features. We then conclude that MI-related instructions, data overwriting-related instructions and truncation have a significant impact on application resilience in terms of the success rate. This finding is consistent with our findings for feature voting scores for predicting the success rate in Table 5.3.a.

When predicting the interruption rate, the prediction error decreases by 20% (from 0.5 to 0.4), when adding the 2nd dimension in $\mathcal{F}_{30}^{ave}$ to features. The 2nd dimension is the floating point instructions. When k is 28, the prediction error decreases 45% (from 0.55 to 0.3) when adding the 25th dimension in $\mathcal{F}_{30}^{ave}$ to features. The 25th dimension is the conditional statement in bigram. This suggests that floating point instructions and conditional statement significantly affect application resilience in terms of interruption.

On the other hand, we see an increase of MAPE after adding a new feature to the feature vector. For example, after adding the 23rd dimension in $\mathcal{F}_{30}^{ave}$ to features when k is 24 for predicting the interruption rate, the MAPE of interruption rate goes up to 0.57 from 0.51. However, this does not necessarily mean that this feature plays a less important role to predict application error resilience. This feature together with the successive features can make a significant contribution to application resilience with respect to interruption. For example, we can see a significant decrease in MAPE when k is 28 for predicting the interruption rate (the MAPE decreases to 0.31 from 0.55). Lacking this feature, we may not achieve such a big decrease in MAPE when k is 28.

In Figure 5.6, we notice that the MAPE value is the lowest for both success rate and interruption rate when k is 30. At this point, MAPE for predicting the success and interruption rates are 0.19 and 0.28, respectively. In consequence, we choose k equal to 30 for both the success and interruption rates. We also notice that the MAPE values when k is 30 in Figure 5.6 are different from those in Table 5.2. The reason is as follows. The MAPE when k is 30 in Figure 5.6 is the result of feature selection before applying the two model tuning techniques: hyperparameter tuning and bagging. However, the MAPE in Table 5.2 is the final result after applying all model tuning techniques and feature construction optimizations. Therefore, the MAPE values in Table 5.2 is smaller than those when k is 30 in Figure 5.6. Also note that the two results in Table 5.2 (0.08 for success rate and 0.22 for interruption rate) are consistent with the results in Figure 5.7.

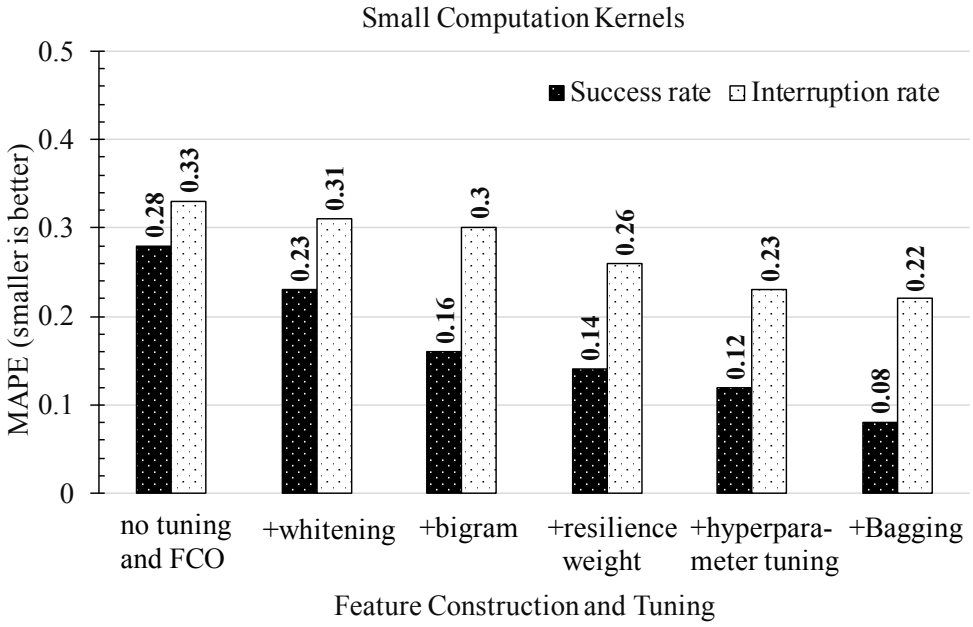


Figure 5.7: Evaluating the impact of model tuning and feature construction optimization on the prediction error for the two fault manifestation rates. FCO = "feature construction optimization". In terms of MAPE, Lower is better.

5.5.3 Evaluation of Model Tuning and Feature Construction Optimization

We study the impact of our model tuning (whitening, bagging and tuning hyperparameters) and feature construction techniques (bigram and resilience weight) to prediction accuracy. We use 100 small computation kernels (for training) for our study. We start with the model without using any of the five techniques, and then apply them one by one in each step.

Figure 5.7 shows the results. We can see that the prediction error continues decreasing after we apply all these techniques. Overall, the MAPE of predicting success rate decreases by 71%; the MAPE of predicting interruption rate decreases by 33%. This demonstrates the effectiveness of all the five techniques in predicting application resilience. Among the five techniques, the most effective ones are bigram and bagging for predicting success rate, and resilience weight for predicting interruption rate.

We notice that after introducing bigram, the MAPE decreases by 30% when predicting success rate. Despite the MAPE reduces slightly when predicting interruption rate after introducing bigram, we find that 58% of kernels have lower prediction error, with up to 20% decrease in MAPE. After introducing resilience weight, the MAPE decreases by 12% when predicting success rate and by 13% when predicting interruption rate. We also observe that the MAPE decreases by 33% when predicting success rate after introducing bagging. After considering resilience weight, the MAPE reduces 12.5% when predicting success rate and 13.3% when predicting interruption rate. The above results demonstrate the effectiveness of bigram, resilience weight, and bagging in predicting application resilience.

5.5.4 Efficiency Study–Comparing PARIS to Random Fault Injection and Trident

We compare the execution time of using FI, using Trident, and using PARIS to predict the rate of manifestations on the 16 big benchmarks. The number of FIs is determined by using a statistical approach [95] with the confidence level of 99% and the margin of error 1%. The number of FIs is about 3000. We measure the execution time of 3000 random FIs as the execution time of FI for each benchmark. When measuring the execution time of using PARIS, we measure the execution time spent on the whole workflow of predicting application resilience for a new, unseen application, including dynamic instruction trace generation, feature extraction, and making prediction with the trained model.

It is important to note that the model training time is not counted into the execution time of the whole workload of predicting application resilience, because once the model is trained, it can be reused repeatedly for an unlimited number of applications, which amortizes the cost of training.

Table 5.4 shows the results. In general, the speedup of using PARIS over using FI is up to 450x (see LULESH) and 49x on average. PARIS is faster than FI for all 16 benchmarks. Furthermore, PARIS is faster than Trident for 12 out of the 16 benchmarks with 15x speedup on average. For the four benchmarks (Sad, Bfs-parboil, PuReMD, and Bfs-rodinia), PARIS is slower, due to the time-consuming trace generation.

We further break down the execution time for the workflow of PARIS and compute the speedup of using PARIS over FI in Table 5.4. The execution time of FI is in the second column. The execution time of FI can be affected by instruction profiling and the complexity of the FI tool. Furthermore, the time can be significantly affected if the program hangs after FI. The time breakdown of PARIS is shown in the third, fourth, and fifth columns. The time spent on making the prediction is constant, which is always around 0.3 seconds. The time spent on dynamic instruction trace generation changes significantly across benchmarks, which is correlated to input problem size and computation complexity of the benchmark. The time spent on feature extraction varies significantly for different benchmarks, which is affected by instruction trace size and complexity of computations in the application. We plan to improve performance of trace generation by using trace compression for better performance in our future work.

Table 5.4: The efficiency comparison between FI, Trident, and PARIS. The table includes breakdown of execution time for the PARIS workflow and speedup (using FI as the baseline).

Benchmarks	FI (s)	Trident (s)	PARIS (s)	Trace generation (s)	Feature construction(s)	Prediction (s)	Speedup over FI
IS	15740	5158	4765	712.3	4052.5	0.3	3x
Nn	8860	4820	395	16.5	378.5	0.3	20x
Myocyte	16380	1215	582	87.2	494.8	0.3	28x
MG	9270	10980	4915	1359.3	3555.9	0.3	2x
Kmeans	4680	1083	234	51.8	182.2	0.3	20x
Libquantum	4714.3	1179	558	0.4	557.6	0.3	8x
Blackscholes	4793	918	23.3	1.1	21.9	0.3	205x
Sad	58890.8	9723	13408	4187.6	9220.4	0.3	4x
Bfs-parboil	11340.4	2835	10450	553.2	9896.8	0.3	1x
Hercules	4703.2	1170	194	7.6	186.4	0.3	24x
PuReMD	1099350	4410	360947	48640.3	312307.2	0.3	3x
Lulesh	9089.3	1896	20.3	1.8	18.2	0.3	450x
Hotspot	43650	15740	10480	3749.7	6730.3	0.3	4x
Bfs-rodinia	36630	10913	15952	6051.7	9900.3	0.3	2x
Nw	16470	4618	4232	859	3373	0.3	4x
Pathfinder	102960	16509	8240	2507	5733	0.3	13x

5.6 Discussions

Use of PARIS. To use PARIS, the user only needs to train the prediction model once, and then the trained model can be repeatedly used for predicting error resilience of any application. Predicting application resilience is useful for improving application resilience [34, 65] and optimizing fault tolerance mechanisms [158, 80, 98, 44]. To train the prediction model, the user must follow the training workflow in Figure 5.1. Given a new application, the user needs to generate a dynamic instruction trace and feed it to PARIS, and PARIS will output three numerical values: the predicted success, SDC, and interruption rates.

Furthermore, PARIS can work on different hardware architectures and for parallel applications with different input problems. We discuss these scenarios as follows.

Support for Different Hardware Architectures. To use PARIS on a new architecture, the user needs to generate new LLVM IR traces. Since the LLVM IR instructions are (micro)architecture-independent, any other workflow in PARIS remains the same. Furthermore, since PARIS users training data sets to train the prediction model and collecting training data sets requires FI, the user is required to perform FI on the new architecture to create training data sets. However, once the prediction model is created and trained, FI will not be required any more. In conclusion, PARIS has no problem to work on a different hardware architecture.

Support for Different Input Problems. PARIS can work on applications with different input problems. Given an input problem to the application, the user is required to run the application to generate the dynamic instruction trace and then build the feature

vector to feed into the prediction model in PARIS. With the traditional FI, the user has to perform an FI campaign, which is usually slower than PARIS.

Support for Parallel Code. PARIS can work for MPI programs. This is supported by our extension to LLVM-tracer that enables LLVM-tracer to generate a trace for each MPI process. Also, the prediction model in PARIS has to be trained using parallel programs, in order to capture the effects of error propagation across MPI processes. If the user cannot train the prediction model using parallel programs, the user can still use the prediction model to make the prediction for serial programs, and then make the prediction for parallel programs based on recent work [154, 86].

5.7 Conclusions

Understanding application resilience to errors becomes increasingly important to ensure result correctness for HPC applications. The traditional method (FI) to understand application resilience is too expensive. Analytical models are faster but they are not as accurate as FI. This chapter introduces PARIS, a new solution based on ML to solve the above problems. We discuss feature constructions, extraction and selection, which are the keys to enable high-performance ML for predicting application resilience. Using a broad spectrum of benchmarks for evaluation, we show that PARIS is much faster than FI, and provides better accuracy (at least 63% better) than the state-of-the-art analytical model. PARIS provides comparable execution time (on average) than the analytical model, but is faster for 12 out of the 16 evaluated benchmarks.

Chapter 6

Evaluating the Performance of Global-Restart Recovery Methods For MPI Fault Tolerance

6.1 Introduction

In this chapter, we present an extensive evaluation using three HPC proxy applications to contrast the two leading global-restart recovery approaches–ULFM and Reinit. Specifically, our contributions are three folds: (1) A new design and implementation of the Reinit approach, named Reinit^{++}, using the latest Open MPI runtime. Our design and implementation supports recovery from either process or node failures, is high performance, and deploys easily by extending the Open MPI library. Notably, we present a precise definition of the failures it handles and the scope of this design and implementation. (2) An extensive evaluation of the performance of the possible recovery approaches (CR, Reinit^{++}, ULFM) using three HPC proxy applications (CoMD, LULESH, HPCCG), and including file and in-memory checkpointing schemes. (3) New insight from the results of our evaluation which show that recovery under Reinit^{++} is up to 6$\times$ faster than CR and up to 3$\times$ faster than ULFM. Compared to CR, Reinit^{++} avoids the re-deployment overhead, while compared to UFLM, Reinit^{++} avoids interference during fault-free application execution and has less recovery overhead.

6.2 Overview

This section presents an overview of the state-of-the-art approaches for MPI fault tolerance. Specifically, it provides an overview of the MPI recovery models.

6.2.1 Existing Approaches for MPI Recovery

ULFM

One of the state-of-the-art approaches for fault tolerance in MPI is User-level Fault Mitigation (ULFM) [19]. ULFM extends MPI to enable failure detection at the application level and provide a set of primitives for handling recovery. Specifically, ULFM taps to

the existing error handling interface of MPI to implement user-level fault notification. Regarding its extensions to the MPI interface, we elaborate on communicators since their extensions are a superset of other communication objects (windows, I/O). Following, ULFM extends MPI with a *revoke* operation (`MPI_Comm_revoke(comm)`) to invalidate a communicator such that any subsequent operation on it raises an error. Also, it defines a *shrink* operation (`MPI_Comm_shrink(comm, newcomm)`) that creates a new communicator from an existing one after excluding any failed processes. Additionally, ULFM defines a collective *agreement* operation (`MPI_Comm_agree(comm,flag)`) which achieves consensus on the group of failed processes in a communicator and on the value of the integer variable `flag`.

Based on those extensions, MPI programmers are expected to implement their own recovery strategy tailored to their applications. ULFM operations are general enough to implement any type of recovery discussed earlier. However, this generality comes at the cost of complexity. Programmers need to understand the intricate semantics of those operations to correctly and efficiently implement recovery and restructure, possibly significantly, the application for explicitly handling failures. Although ULFM provides examples that prescribe the implementation of global-restart, the programmer must embed this in the code and refactor the application to function with the expectation that communicators may change during execution due to shrinking and merging, which is not ideal.

Reinit

Reinit [92, 35] has been proposed as an alternative approach for implementing global-restart recovery, through a simpler interface compared to ULFM. The most recent implementation [35] of Reinit is limited in several aspects: (1) it requires modifying the job scheduler (SLURM), besides the MPI runtime, thus it is impractical to deploy and skews performance measurements due to crossing the interface between the job scheduler and the MPI runtime; (2) its implementation is not publicly available; (3) it bases on the MVAPICH2 MPI runtime, which makes comparisons with ULFM hard, since ULFM is implemented on the Open MPI runtime. Thus, we opt for a new design and implementation[1], named Reinit^{++}, which we present in detail in the next section.

6.3 Reinit^{++}

This section describes the programming interface of Reinit^{++}, the assumptions for application deployment, process and node failure detection, and the recovery algorithm for global-restart. We also define the semantics of MPI recovery for the implementation of Reinit^{++} as well as discuss its specifics.

6.3.1 Design

Programming Interface of Reinit^{++}

Figure 6.1 presents the programming interface of Reinit^{++} in the C language, while figure 6.2 shows sample usage of it. There is a single function call, `MPI_Reinit`,

[1] Available open-source at `https://github.com/ggeorgakoudis/ompi/tree/reinit`

```
1   typedef enum {
2     MPI_REINIT_NEW, MPI_REINIT_REINITED, MPI_REINIT_RESTARTED
3   } MPI_Reinit_state_t
4
5   typedef int
6   (*MPI_Restart_point)
7     (int argc, char **argv, MPI_Reinit_state_t state);
8
9   int MPI_Reinit
10    (int argc, char **argv, const MPI_Restart_point point);
```

Figure 6.1: The programming interface of Reinit^{++}

```
1  int foo(int argc, char **argv, MPI_Reinit_state_t state)
2  {
3    /* Load checkpoint if it exists */
4    while(!done) {
5      /* Do computation */
6      /* Store checkpoint */
7    }
8  }
9
10 int main(int argc, char **argv)
11 {
12   MPI_Init(&argc, &argv);
13   /* Application specific initialization */
14   // Entry point of the resilient function
15   MPI_Reinit(&argc, &argv, foo);
16   MPI_Finalize();
17 }
```

Figure 6.2: Sample usage of the interface of Reinit^{++}

for the programmer to call to define the point in code to rollback and resume execution after a failure. This function must be called after `MPI_Init` so ensure the MPI runtime has been initialized. Its arguments imitate the parameters of `MPI_Init`, adding a parameter for a pointer to a user-defined function. Reinit^{++} expects the programmer to encapsulate in this function the main computational loop of the application, which is restartable through checkpointing. Internally, `MPI_Reinit` passes the parameters `argc` and `argv` to this user-defined function, plus the parameter `state`, which indicates the MPI state of the process as values from the enumeration type `MPI_Reinit_state_t`. Specifically, the value `MPI_REINIT_NEW` designates a new process executing for the first time, the value `MPI_REINIT_REINITED` designates a survivor process that has entered the user-defined function after rolling back due to a failure, and the value `MPI_REINIT_RESTARTED` designates that the process has failed and has been re-spawned to resume execution. Note that this state variable describes only the MPI state of Reinit^{++}, thus has no semantics on the application state, such as whether to load a checkpoint or not.

Application Deployment Model

Reinit^{++} assumes a logical, hierarchical topology of application deployment. Figure 6.3 shows a graphical representation of this deployment model. At the top level, there

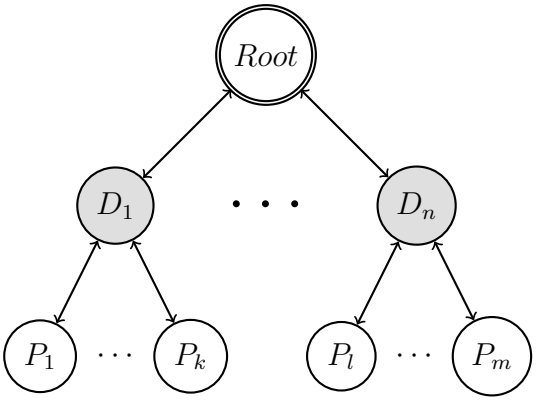


Figure 6.3: Application deployment model

is a single *root* process that spawns and monitors *daemon* processes, one on each of the computing nodes reserved for the application. Daemons spawn and monitor *MPI processes* local to their nodes. The root communicates with daemons and keeps track of their liveness, while daemons track the liveness of their children MPI processes. Based on this execution and deployment model, Reinit^{++} performs fault detection, which we discuss next.

Fault Detection

Reinit^{++} targets *fail-stop* failures of either MPI processes or daemons. A daemon failure is deemed equivalent to a node failure. The causes for those failures may be transient faults or hard faults of hardware components.

In the design of Reinit^{++}, the root manages the execution of the whole applications, so any recovery decisions are taken by it, hence it is the focal point for fault detection. Specifically, if an MPI process fails, its managing daemon is notified of the failure and forwards this notification to the root, without taking an action itself. If a daemon process fails, which means either the node failed or the daemon process itself, the root directly detects the failure and also assumes that the children MPI processes of that daemon are lost too. After detecting a fault the root process proceeds with recovery, which we introduce in the following section.

MPI Recovery

Reinit^{++} recovery for both MPI process and daemon failures is similar, except that on a daemon failure the root chooses a new host node to re-instate failed MPI processes, since a daemon failure proxies a node failure. For recovery, the root process broadcasts a *reinit* message to all daemons. Daemons receiving that message roll back survivor processes and re-spawn failed ones. After rolling back survivor MPI processes and spawning new ones, the semantics of MPI recovery are that only the world communicator is valid and any previous MPI state (other communicators, windows, etc.) has been discarded. This is similar to the MPI state available immediately after an application calls `MPI_Init`. Next, the application restores its state, discussed in the following section.

Application Recovery

Reinit^{++} assumes that applications are responsible for saving and restoring their state to resume execution. Hence, both survivor and re-spawned MPI processes should load a valid checkpoint after MPI recovery to restore application state and resume computation.

6.3.2 Implementation

We implement Reinit^{++} in the latest Open MPI runtime, version 4.0.0. The implementation supports recovery from both process and daemon (node) failures. This implementation does not presuppose any particular job scheduler, so it is compatible with any job scheduler the Open MPI runtime works with. Introducing briefly the Open MPI software architecture, it comprises of three frameworks of distinct functionality: (i) the OpenMPI MPI layer (OMPI), which implements the interface of the MPI specification used by the application developers; (ii) the OpenMPI Runtime Environment (ORTE), which implements runtime functions for application deployment, execution monitoring, and fault detection, and (iii) the Open Portability Access Layers (OPAL), which implements abstractions of OS interfaces, such as signal handling, process creation, etc.

Reinit^{++} extends OMPI to provide the function `MPI_Reinit`. It extends ORTE to propagate fault notifications from daemons to the root and to implement the mechanism of MPI recovery on detecting a fault. Also, Reinit^{++} extends OPAL to implement low-level process signaling for notifying survivor process to roll back. The following sections provide more details.

Application Deployment

Reinit^{++} requires the application to deploy using the default launcher of Open MPI, `mpirun`. Note that using the launcher `mpirun` is compatible with any job scheduler and even uses optimized deployment interfaces, if the scheduler provides any. Physical application deployment in Open MPI closely follows the logical model of the design of Reinit^{++}. Specifically, Open MPI sets the root of the deployment at the process launching the `mpirun`, typically on a login node of HPC installations, which is deemed as the Head Node Process (HNP) in Open MPI terminology. Following, the root launches an ORTE daemon on each node allocated for the application. Daemons spawn the set of MPI processes in each node and monitor their execution. The root process communicates with each daemon over a channel of a reliable network transport and monitors the liveness of daemons through the existence of this channel.

Launching an application, the user specifies the number of MPI processes and optionally the number of nodes (or number of processes per node). To withstand process failures, this specification of deployment is sufficient, since Reinit^{++} re-spawns failed processes on their original node of deployment. However, for node failures, the user must *over-provision* the allocated process slots for re-spawning the set of MPI processes lost due to a failed node. To do so, the most straightforward way is to allocate more nodes than required for fault-free operation, up to the maximum number of node failures to withstand.

Algorithm 1 Root: HandleFailure

Data: $\mathcal{D}$: the set of daemons,
$Children(x)$: returns the set of children MPI processes of daemon x,
$Parent(x)$: returns the parent daemon of MPI process x
Input: The failed process f (MPI process or daemon)
`// failed process is a daemon`
if $f \in \mathcal{D}$ **then**
  $\mathcal{D} \leftarrow \mathcal{D} \setminus \{f\}$
  $d' \leftarrow d \mid \underset{d \in D}{arg\,min}\ Children(d)$
  `// broadcast REINIT to all daemons`
  Broadcast $\mathcal{D}$ message $\langle$`REINIT`, $\{\ \langle d', c\rangle \mid \forall c \in Children(f)\ \}\ \rangle$
`// failed process is an MPI process`
else
  Broadcast $\mathcal{D}$ message $\langle$`REINIT`, $\{\ \langle Parent(f), f\rangle\ \}\ \rangle$
end

Fault Detection

In Open MPI, a daemon is the parent of the MPI processes on its node. If an MPI process crashes, its parent daemon is notified, by trapping the signal `SIGCHLD`, in POSIX semantics. Implementing the fault detection requirements of Reinit^{++}, a daemon relays the fault notification to the root process for taking action. Regarding node failures, the root directly detects them proxied through daemon failures. Specifically, the root has an open communication channel with each daemon over some reliable transport, e.g., TCP. If the connection over that communication channel breaks, the root process is notified of the failure and regards the daemon at fault, thus assuming all its children MPI process lost and its host node is unavailable. For both types of failures (process and node), the root process initiates MPI recovery.

MPI Recovery

Algorithm 1 shows in pseudocode the operation of the root process when handling a failure. On detecting a failure, the root process distinguishes whether it is a faulty daemon or MPI process. For a node failure, the root selects the *least loaded node* in the resource allocation, that is the node with the fewest occupied process slots, and sets this node's daemon as the parent daemon for failed processes. For a process failure, the root selects the original parent daemon of the failed process to re-spawn that process. Next, the root process initiates recovery by broadcasting to all daemons a message with the `REINIT` command and the list of processes to spawn, along with their selected parent daemons. Following, when a daemon receives that message it signals its survivor, children MPI processes to roll back, and re-spawns any processes in the list that have this daemon as their parent. Algorithm 2 presents this procedure in pseudocode.

Regarding the asynchronous, signaling interface of Reinit^{++}, Algorithm 3 illustrates the internals of the Reinit^{++} in pseudocode. When an MPI process executes `MPI_Reinit`, it installs a *signal handler* for the signal `SIGREINIT`, which aliases `SIGUSR1` in our implementation. Also, `MPI_Reinit` sets a non-local goto point using the POSIX

Algorithm 2 Daemon $\hat{d}$: HandleReinit

Data: $Children(x)$: returns the set of children MPI processes of daemon x,
$Parent(x)$: returns the parent daemon of MPI process x
Input: List $\{\langle d_i, c_i\rangle, \cdots\}$
`// Signal survivor MPI processes`
for $c \in Children(\hat{d})$ **do**
  $c.state \leftarrow$ `MPI_REINIT_REINITED`
  Signal `SIGREINIT` to c
end
`// Spawn new process if` $\hat{d}$ `is parent`
foreach $\{\langle d_i, c_i\rangle, \cdots\}$ **do**
  if $\hat{d} == d_i$ **then**
    $Children(\hat{d}) \leftarrow Children(\hat{d}) \cup c_i$
    $c_i.state \leftarrow$ `MPI_REINIT_RESTARTED`
    Spawn c_i
  end
end

Algorithm 3 Reinit^{++} internals

Function `OnSignalReinit()`:
  goto Rollback
end
Function `MPI_Reinit`$(argc, argv,$ `foo`$)$:
  Install signal handler `OnSignalReinit` on `SIGREINIT`
Rollback:
  if $this.state ==$ *`MPI_REINIT_REINITED`* **then**
    Discard MPI state
    Wait on barrier
    Re-initialize world communicator
  end
  return `foo` $(argc, argv, this.state)$
end

function `setjmp()`. The signal handler of `SIGREINIT` simply calls `longjmp()` to return execution of survivor processes to this goto point. Rolled back survivor processes discard any previous MPI state and block on a ORTE-level barrier. This barrier replicates the implicit barrier present in `MPI_Init` to synchronize with re-spawned processes joining the computation. After the barrier, survivor processes re-initialize the world communicator and call the function `foo` to resume computation. Re-spawned processes initialize the world communicator as part of the MPI initialization procedure of `MPI_Init` and go through `MPI_Reinit` to install the signal handler, set the goto point, and lastly call the user-defined function to resume computation.

Application Recovery

Application recovery includes the actions needed at the application-level to resume computation. Any additional MPI state besides the repaired world communicator, such as

sub-communicators, must be re-created by the application's MPI processes. Also, it is expected that each process loads the latest consistent checkpoint to continue computing. Checkpointing lays within the responsibility of the application developer. In the next section, we discuss the scope and implications of our implementation.

Discussion

In this implementation, the scope of fault tolerance is to support recovery from failures *happening after* `MPI_Reinit` has been called by all MPI processes. This is because `MPI_Reinit` must install signal handlers and set the roll-back point on all MPI processes. This is sufficient for a large coverage of failures since execution time is dominated by the main computational loop. In the case a failure happens before the call to `MPI_Reinit`, the application falls back to the default action of aborting execution. Nevertheless, the design of Reinit^{++} is not limited by this implementation choice. A possible approach instead of aborting, which we leave as future work, is to treat any MPI processes that have not called `MPI_Reinit` as if failed and re-execute them.

Furthermore, signaling `SIGREINIT` for rolling back survivor MPI processes asynchronously interrupts execution. In our implementation, we render the MPI runtime library *signal and roll-back safe* by using masking to defer signal handling until a safe point, i.e., avoid interruption when locks are held or data structures are updating. Since application code is out of our control, Reinit^{++} requires the application developer to program the application as signal and roll-back safe. A possible enhancement is to provide an interface for installing cleanup handlers, proposed in earlier designs of Reinit [89], so that application and library developers can install routines to reset application-level state on recovery. Another approach is to make recovery synchronous, by extending the Reinit^{++} interface to include a function that tests whether a fault has been detected and trigger roll back. The developer may call this function at safe points during execution for recovery. We leave both those enhancements as future work, noting that the existing interface is sufficient for performing our evaluation.

6.4 Experimentation Setup

This section provides detailed information on the experimentation setup, the recovery approaches used for comparisons, the proxy applications and their configurations, and the measurement methodology.

Recovery approaches

Experimentation includes the following recovery approaches:

- *CR*, which implements the typical approach of immediately restarting an application after execution aborts due to a failure.
- *ULFM*, by using its latest revision based on the Open MPI runtime v4.0.1 (4.0.1ulfm2.1rc1).
- *Reinit*$^{++}$, which is our own implementation of Reinit, based on OpenMPI runtime v4.0.0.

Table 6.1: Proxy applications and their configuration

Application	Input	No. ranks
CoMD	-i4 -j2 -k2 -x 80 -y 40 -z 40 -N 20	16, 32, 64, 128, 256, 512, 1024
HPCCG	64 64 64	16, 32, 64, 128, 256, 512, 1024
LULESH	-i 20 -s 48	8, 64, 512

Table 6.2: Checkpointing per recovery and failure

Failure	Recovery		
	CR	ULFM	Reinit
process	file	memory	memory
node	file	file	file

Emulating failures

Failures are emulated through fault injection. We opt for random fault injection to emulate the occurrence of random faults, e.g., soft errors or failures of hardware components, that lead to a crash failure. Specifically, for process failures, we instrument applications so that at a random iteration of the main computational loop, a random MPI process suicides by raising the signal `SIGKILL`. The random selection of iteration and MPI process is the same for every recovery approach. For node failures, the method is similar, but instead of itself, the MPI process sends the signal `SIGKILL` to its parent daemon, thus kills the daemon and by extension all its children processes. In experimentation, we inject a *single* MPI process failure or a *single* node failure.

Applications

We experiment with three benchmark applications that represent different HPC domains: *CoMD* for molecular dynamics, *HPCCG* for iterative solvers, and *LULESH* for multi-physics computation. The motivation is to investigate global-restart recovery on a wide range of applications and evaluate any performance differences. Table 6.1 shows information on the proxy applications and scaling of their deployed number of ranks. Note *LULESH* requires a cube number of ranks, thus the trimmed down experimentation space. The deployment configuration has 16 ranks per node, so the smallest deployment comprises of one node while the largest one spans 64 nodes (1024 ranks). Application execute in *weak scaling* mode – for *CoMD* we show its input only 16 ranks and change it accordingly. We extend applications to implement global-restart with Reinit^{++} or ULFM, to store a checkpoint after every iteration of their main computational loop and load the latest checkpoint upon recovery.

Checkpointing

For evaluation purposes, we implement our own, simple checkpointing library that supports saving and loading application data using in-memory and file checkpoints. Table 6.2 summarizes checkpointing per recovery approach and failure type. In detail, we implement two types of checkpointing: *file* and *memory*. For file checkpointing, each MPI process stores a checkpoint to globally accessible permanent storage, which is the networked, parallel filesystem Lustre available in our cluster. For memory checkpointing, an MPI process stores a checkpoint both locally in its own memory and remotely to the memory of a *buddy* [163, 162] MPI process, which in our implementation is the (cyclically) next MPI process by rank. This memory checkpointing implementation is applicable only to single process failures since multiple process failures or a node failure can wipe out both local and buddy checkpoints for the failed MPI processes. CR necessarily uses file checkpointing since re-deploying the application requires permanent storage to retrieve checkpoints.

Statistical evaluation

For each proxy application and configuration we perform 10 independent measurements. Each measurement counts the total execution time of the application breaking it down to time needed for writing checkpoints, time spent during MPI recovery, time reading a checkpoint after a failure, and the pure application time executing the computation. Any confidence intervals shown correspond to a 95% confidence level and are calculated based on the t-distribution to avoid assumptions on the sampled population's distribution.

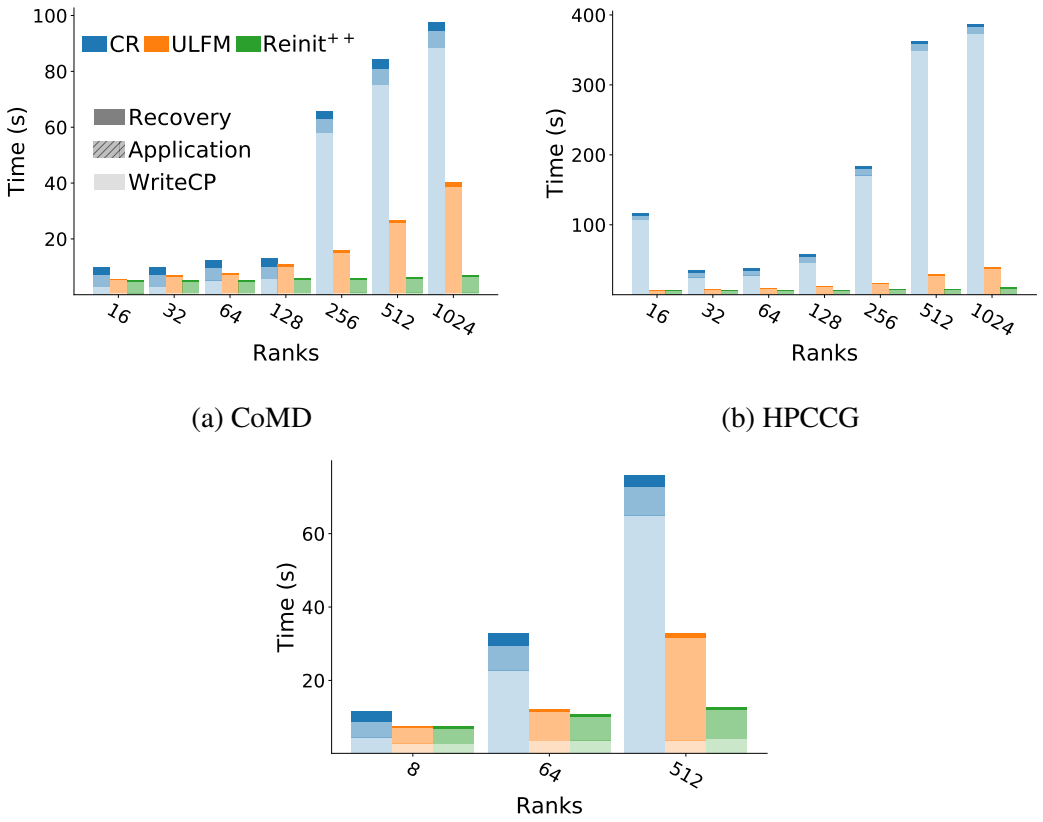
6.5 Evaluation

For the evaluation we compare CR, Reinit^{++} and ULFM for both process and node failures. Results provide insight on the performance of each of those recovery approaches implementing global-restart and reveal the reasons for their performance differences.

6.5.1 Comparing total execution time on a process failure

Figure 6.4 shows average total execution time for process failures using file checkpointing for CR and memory checkpointing for Reinit^{++} and ULFM. The plot breaks down time to components of writing checkpoints, MPI recovery, and pure application time. Reading checkpoints occurs one-off after a failure and has negligible impact, in the order of tens of milliseconds, thus it is omitted.

The first observation is that Reinit^{++} scales excellently compared to both CR and ULFM, across all programs. CR has the worse performance, increasingly so with more ranks. The reason is the limited scaling of writing checkpoints to the networked filesystem. By contrast, ULFM and Reinit^{++} use memory checkpointing, spending minimal time writing checkpoints. Interestingly, ULFM scales worse than Reinit^{++}; we believe that the reason is that it inflates pure application execution time, which we illustrate in the next section. Further, in the following sections, we remove checkpointing overhead



(a) CoMD

(b) HPCCG

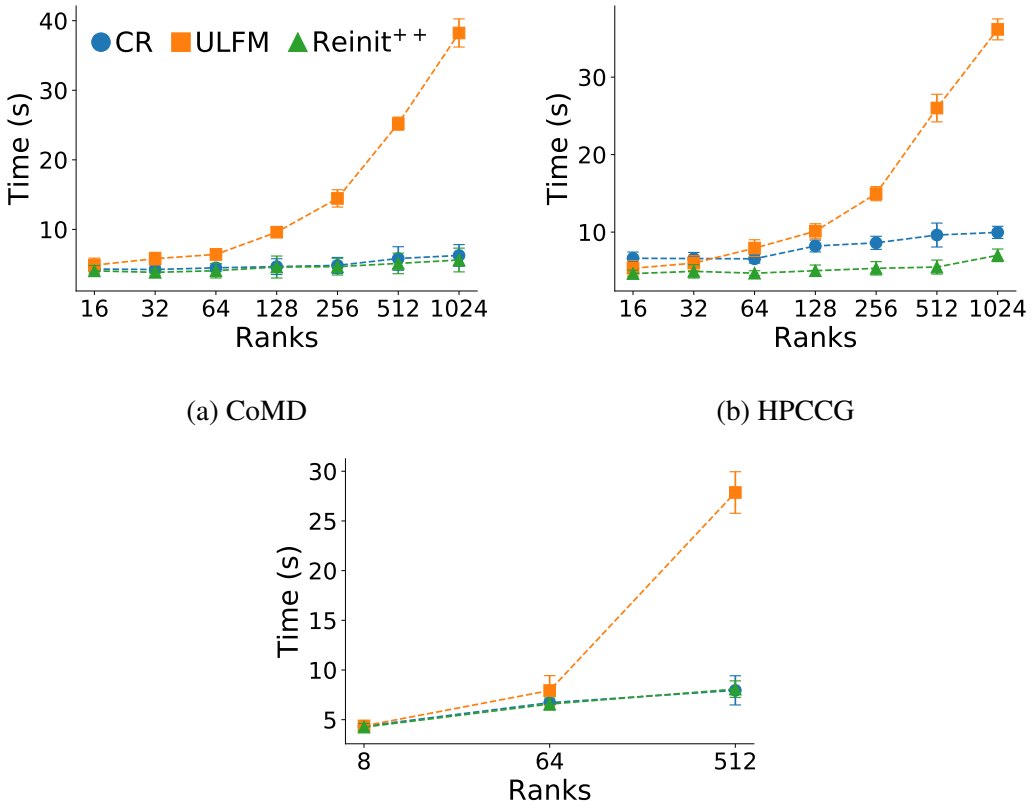
(c) LULESH

Figure 6.4: Total execution time breakdown recovering from a process failure

from the analysis to highlight the performance differences of the different recovering approaches.

6.5.2 Comparing pure application time under different recovery approaches

Figure 6.5 shows the pure application time, without including reading/writing checkpoints or MPI recovery. We observe that application time is on par for CR and Reinit^{++}, and that all applications scale weakly well on up to 1024 ranks. CR and Reinit^{++} do not interfere with execution, thus they have no impact on application time, which is on par to the fault-free execution time of the proxy applications. However, in ULFM, application time grows significantly as the number of ranks increases. ULFM extends MPI with an always-on, periodic heartbeat mechanism [22] to detect failures and also modifies communication primitives for fault tolerant operation. Following from our measurements, those extensions noticeably increase the original application execution time. However, it is inconclusive whether this is a result of the tested prototype implementation or a systemic trade-off. Next, we compare the MPI recovery times among all the approaches.



(a) CoMD

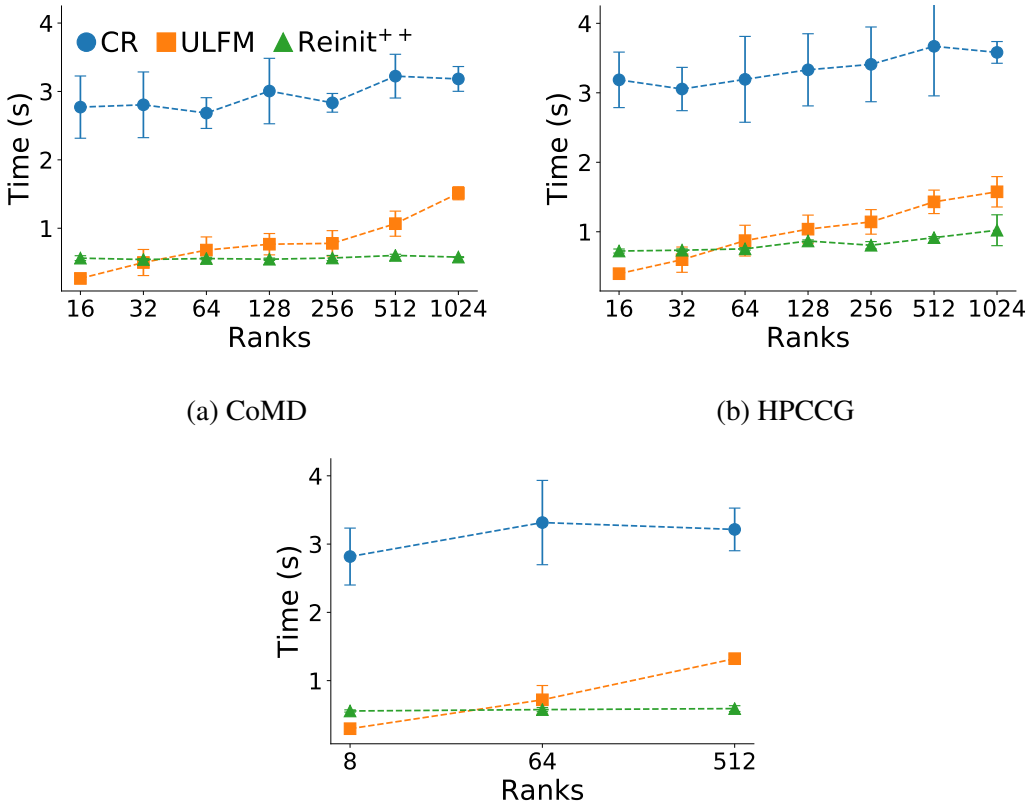
(b) HPCCG

(c) LULESH

Figure 6.5: Scaling of pure application time

6.5.3 Comparing MPI recovery time recovering from a process failure

Though checkpointing saves applications computation time, reducing MPI recovery time saves overhead from restarting. This overhead is increasingly important the larger the deployment and the higher the fault rate. In particular, figure 6.6 shows the scaling of time required for MPI recovery across all programs and recovery approaches, again removing any overhead for checkpointing to focus on the MPI recovery time. As expected, MPI recovery time depends only on the number of ranks, thus times are similar among different programs for the same recovery approach. Commenting on scaling, CR and Reinit^{++} scale excellently, requiring almost constant time for MPI recovery regardless the number of ranks. However, CR is about 6$\times$ slower, requiring around 3 seconds to tear down execution and re-deploy the application, whereas Reinit^{++} requires about 0.5 second to propagate the fault, re-initialize survivor processes and re-spawn the failed process. ULFM has on par recovery time with Reinit^{++} up to 64 ranks, but then its time increases being up to 3$\times$ slower than Reinit^{++} for 1024 ranks. ULFM requires multiple collective operations among all MPI processes to implement global-restart (shrink the faulty communicator, spawn a new process, merge it to a new communicator). By contrast, Reinit^{++} implements recovery at the MPI runtime layer requiring fewer operations and confining collective communication only between root and daemon processes.



(a) CoMD

(b) HPCCG

(c) LULESH

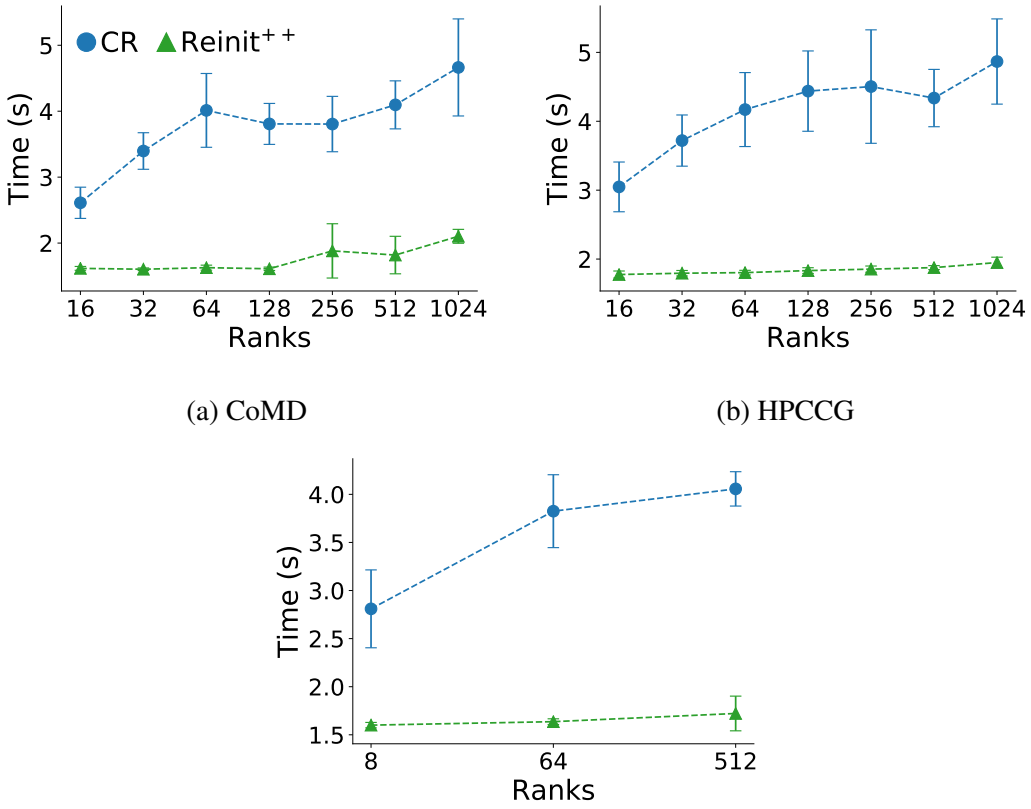
Figure 6.6: Scaling of MPI recovery time recovering from a process failure

6.5.4 Comparing MPI recovery time recovering from a node failure

This comparison for a node failure includes only CR and Reinit^{++}, since the prototype implementation of ULFM faced robustness issues (hanging or crashing) and did not produce measurements. Also, since both CR and Reinit^{++} use file checkpointing and do not interfere with pure application time, we present only results for MPI recovery times, shown in figure 6.7. Both CR and Reinit^{++} scale very well with almost constant times, as they do for a process failure. However, in absolute values, Reinit^{++} has a higher recovery time of about 1.5 seconds for a node failure compared to 0.5 seconds for a process failure. This is because recovering from a node failure requires extra work to select the least loaded node and spawn all the MPI processes of the failed node. Nevertheless, recovery with Reinit^{++} is still about 2$\times$ faster than with CR.

6.6 Conclusion

We have presented Reinit^{++}, a new design and implementation of the global-restart approach of Reinit. Reinit^{++} recovers from both process and node crash failures, by spawning new processes and mending the world communicator, requiring from the programmer only to provide a rollback point in execution and have checkpointing in place.



(a) CoMD

(b) HPCCG

(c) LULESH

Figure 6.7: Scaling of MPI recovery time recovering from a node failure

Our extensive evaluation comparing with the state-of-the-art approaches Checkpoint-Restart (CR) and ULFM shows that Reinit^{++} scales excellently as the number of ranks grows, achieving almost constant recovery time, being up to 6× faster than CR and up to 3× faster than ULFM. For future work, we plan to expand Reinit for supporting more recovery strategies besides global-restart, including shrinking recovery and forward recovery strategies, to maintain its implementation, and expand the experimentation with more applications and larger deployments.

6.7 Acknowledgment

This work is performed when I was doing an internship at Lawrence Livermore National Laboratory. I am one of the primary contributor to this work. I helped the design and evaluation of Reinit^{++} and the development of the fault tolerance frameworks into the three applications and collection of evaluation results.

Chapter 7

A Benchmark Suite to Characterize and Model MPI Application Resilience

7.1 Introduction

In this chapter, we design and develop a benchmark suite MATCH aiming to study the performance efficiency of a variety of MPI fault tolerance configurations. MATCH contains six proxy applications from the Exascale Computing Project (ECP) Proxy Apps Suite and LLNL Advanced Simulation and Computing (ASC) proxy application suite; MATCH uses Fault Tolerance Interface (FTI) for the data recovery interface and uses ULFM and Reinit for the MPI recovery interface. We pick a representative set of HPC applications but our methodology is extensible to more HPC applications. In evaluation, we break down the execution time and compare the performance overhead, when using FTI with Restart, when using FTI with ULFM, and when using FTI with Reinit, respectively. All the above experiments are running in four different scaling sizes (64 processes, 128 processes, 256 processes, and 512 processes on 32 nodes), in three different input sizes (small, median, and large), and when with or without injecting process failures.

In particular, our contributions are three-fold: (1) we present MATCH, an MPI fault tolerance benchmark suite. This is the first benchmark suite designed to evaluate multiple fault tolerance techniques for MPI. We illustrate the process and manifest the details of implementing a range of different fault tolerance designs to HPC proxy applications; (2) we develop a data dependency analysis tool for identifying the data objects for checkpointing, which are the only data objects necessary to guarantee the restoring of application state for the application execution correctness for the first time; (3) we comparatively and extensively investigate the performance efficiency of different configurations and different combinations of fault tolerance designs. Our evaluation reveals that, for MPI global-restart recovery, using FTI with Reinit is the most efficient design within the three evaluated fault tolerance designs, and Reinit recovery is 4 times faster than ULFM recovery on average, and 16 times faster than restarting on average.

7.2 Overview

7.2.1 MATCH

There is not an existing benchmark suite aiming at benchmarking of MPI fault tolerance. We design, implement, and test a benchmark suite MATCH to understand, study, and comparatively evaluate the performance efficiency of different MPI fault tolerance designs and configurations. MATCH is composed of HPC proxy applications coming from representative HPC benchmark suites. MATCH contains six representative HPC applications. Our fault tolerance design has two interfaces: the checkpointing interface to preserve and protect the data, and the failure recovery interface to protect and repair the MPI communicator. We use the Fault Tolerance Interface (FTI) for checkpointing and ULFM and Reinit for MPI process recovery in this work.

7.2.2 Workloads

Our workloads are proxy applications getting from well-known benchmark suites: ECP proxy applications suite [125] and LLNL ASC proxy applications suite [112]. Proxy applications are small and simplified applications that allow HPC practitioners, operators, and domain scientists to explore and test key features of real applications with a quick turnaround. Our workloads represent the most important HPC application domains in scientific computing, such as iterative solvers, multi-grid, molecular dynamics, etc. We describe the six proxy applications used in MATCH below.

AMG: An algebraic multi-grid solver dealing with linear systems in unstructured grids problems. AMG is built on top of the BoomerAMG solver of the Hypre library which is a large-scale linear solver library developed at LLNL. AMG provides a number of tests for a variety of problems. The default one is an anisotropy problem in the Laplace domain.

CoMD: A proxy application in Molecular Dynamics (MD) commonly used as a research platform for particle motion simulation. Different than previous MD proxy applications such as MiniMD, the design of CoMD is significantly modularized which allows performing analyses individual modules.

LULESH: A proxy application that solves the hydrodynamics equation in a Sedov blast problem. LULESH solves the hydrodynamics equation separately by using a mesh to simulate the Sedov blast problem which is divided into a composition of volumetric elements. This mesh is an unstructured hex mesh, where nodes are points connected by mesh lines.

miniFE: A proxy application that solves unstructured implicit finite element problem. miniFE aims at the approximation of an unstructured implicit finite element.

miniVite: A proxy application that solves the graph community detection problem using the distributed Louvain method. The Louvain method is a greedy algorithm for the community detection problem.

HPCCG: A preconditioned conjugate gradient solver that solves the linear system of partial differential equations in a 3D chimney domain. HPCCG approximates practical physical applications that simulate unstructured grid problems.

7.2.3 Checkpointing Interface - FTI

Fault Tolerance Interface (FTI) [15] is a multi-level checkpointing interface for efficient multilevel checkpointing in large-scale high-performance computing systems. FTI provides programmers a number of APIs which are easy to use, and allows programmers to choose checkpointing strategy that fits the application. FTI enables multiple levels of reliability with different performance efficiency by utilizing local storage, data replication, and erasure codes. FTI is an application-level checkpointing. It requests users to decide which data objects to be checkpointed. Furthermore, FTI hides data processing details from users. Users only tell FTI the memory address and data size of the date object to be protected to enable checkpointing of the data object. Because failures can corrupt single or multiple nodes during the execution of an application, FTI provides multiple levels of resiliency to recover from failures of different severities. Namely the levels are the following:

- L1: This level stores checkpoints locally to each compute node. In case of a node failure, the application states cannot successfully restore.
- L2: This level is built on top of L1 checkpointing. In this level each application stores their checkpoint locally as well as to a neighboring node.
- L3: In this level, the checkpoints are encoded by the Reed-Solomon (RS) erasure code. This implementation can survive the breakdown of half of the nodes. The lost data can be restored from the RS-encoded files.
- L4: This level flushes checkpoints to parallel file system. This level enables differential checkpointing.

FTI have proposed a multi-level checkpointing model, and have conducted an extensive study of correctness and reliability of the proposed checkpointing model. In our work, we use FTI in the context of MPI recovery which is for the first time.

7.3 Design

We present the design details in this section. In particular, we describe the algorithm that we use to find data objects for checkpointing through data dependency analysis.

7.3.1 Find Data Objects for Checkpointing

Different than many fault tolerance frameworks that request programmers to decide data objects for checkpointing, we develop a practical analytic tool to guide programmers to identify data objects to be checkpointed, in order to recover the application execution to the same state as before the failure. We identify data objects for checkpointing through data dependency analysis across iterations following three **principles**.

- The data objects for checkpointing across iterations must be defined before the iterative computation. Data objects defined locally within the main computation loop must be excluded for checkpointing.

Algorithm 4 Find Data Objects for Checkpointing

Input: $Locs\_in\_loop$: the set of locations used in the main computation loop; $Locs\_before\_loop$: the set of locations defined or allocated before the main computation loop

Output: CPK_Locs: the set of locations for checkpointing

```
// Check values of locations in Locs_in_loop
for l ∈ Locs_in_loop do
    if The invocation values of l are not the same then
        Keep l in Locs_in_loop
    else
        Remove l from Locs_in_loop
    end
end
// Remove repetition in Locs_in_loop and Locs_before_loop
for l ∈ Locs_in_loop do
    Remove repetition
end
for l ∈ Locs_before_loop do
    Remove repetition
end
// Check if locations in Locs_in_loop can find a match in
   Locs_before_loop
for l_i ∈ Locs_in_loop do
    for l_j ∈ Locs_before_loop do
        if l_i matches l_j then
            CPK_Locs ← l_i
        end
    end
end
```

- The data objects for checkpointing must be used (read or written) across iterations of the main computation loop.
- The value of data objects for checkpointing must vary across iterations of the main computation loop.

Following the three principles, we design and develop the data dependency analysis tool. The **input** to the tool is a dynamic execution instruction trace generated using LLVM-Tracer [136]. The trace contains detailed information of dynamic operations, such as the register name and memory address, the operator, and the line number in the source code where the operation performs. We describe the algorithm of the data dependency analysis tool in Algorithm 4. The input to the algorithm is the set of locations used within the main computation loop, and the set of locations allocated before the main computation loop. Here locations are registers and memory locations. We create the two sets of locations by traversing the instruction trace once. After that, we first check values of locations, and make sure the invocation values of the same location within the main computation loop are different. We then remove repetitions from both

```
int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);

    // Initialize FTI
    FTI_Init(argv[1], MPI_COMM_WORLD);

    // Right before the main computation loop
    // Add FTI protection to data objects
    FTI_Protect();

    // the main computation loop
    while (...) {
        // At the beginning of the loop
        // If the execution is a restart
        if ( FTI_Status() != 0){
            FTI_Recover();
        }

        // do FTI checkpointing
        if (Iter_Num % cp_stride == 0) {
            FTI_Checkpoint();
        }
    }

    FTI_Finalize();
    MPI_Finalize();
}
```

Figure 7.1: A sample implementation of FTI.

sets of locations. Lastly, for each location in the set of the main computation loop we search for a match in the location set before the main computation loop. If a match is found, the matched location is used to localize data objects for checkpointing. The **output** of the tool is a set of locations for checkpointing. Note that the tool only outputs the locations for checkpointing, runs separately, and has not supported automatic generation of checkpointing code at this stage. We leave it for future work.

7.4 Implementation

7.4.1 FTI Implementation

The Fault Tolerance Interface (FTI) is a checkpointing library widely used by HPC developers for checkpointing. We illustrate a sample usage of FTI in Figure 7.1. We find a challenge while implementing FTI to `MATCH` workloads.

The challenge is the programming complexity of enabling FTI checkpointing to data objects, when the number of data objects for checkpointing is large. FTI requests users to manually add FTI checkpointing to every data object. This significantly increases the programming effort when the number of data objects for checkpointing is large and when the data object is a complicated data structure. This is a common issue in application level checkpoint libraries such as FTI, VeloC, and SCR. These libraries cannot automatically enable checkpointing to target data objects.

```
1  int main(int argc, char *argv[])
2  {
3      MPI_Init(&argc, &argv);
4      OMPI_Reinit(argc, argv, resilient_main);
5      MPI_Finalize();
6      return 0;
7  }
8  // Move the original main() into resilient_main()
9  int resilient_main(int argc, char** argv, OMPI_reinit_state_t state) {
10     FTI_Init(argv[1], MPI_COMM_WORLD);
11     ...
12     // the main computation loop
13     ...
14     FTI_Finalize();
15     return 0;
16 }
```

Figure 7.2: A sample implementation of Reinit.

7.4.2 FTI with Reinit Implementation

Reinit is the state-of-the-art MPI global non-shrinking recovery framework. Reinit hides all recovery implementations to the MPI runtime, which makes it ease-to-use. We provide a sample implementation of Reinit with FTI checkpointing in Figure 7.2. We can see that Reinit recovery only adds less than five lines of code. Line 4 and 5 are for Reinit recovery, while Line 14 is used for other functionalities. FTI is completely independent of Reinit. To implement FTI with Reinit, the only thing to notice is to move the FTI_Init() and FTI_Finalize() functions into the resilient_main() function as well.

7.4.3 FTI with ULFM Implementation

ULFM is a pioneer MPI recovery framework. ULFM provides five new MPI interfaces to support MPI fault tolerance. ULFM gives flexibility to programmers to use the provided interfaces to implement the MPI recovery functionality. Also, ULFM allows programmers to use both shrinking and non-shrinking recovery. However, it takes a significant learning and programming effort before a programmer can successfully implement ULFM process recovery. As most HPC applications follow the Bulk Synchronous Parallel (BSP) paradigm, we focus on ULFM global non-shrinking recovery. In order to implement ULFM non-shrinking recovery, we add more than 200 lines of code for each benchmark, which is less efficient comparing to the implementing effort (less than five lines of code) for Reinit recovery. We provide a sample implementation of ULFM global non-shrinking recovery with FTI in Figure 7.3.

When combining ULFM global non-shrinking recovery with FTI, it is important to notice that the MPI_COMM_WORLD at Line 4 in Figure 7.1 must be implemented as a global variable with external declaration. Such that, the world communicator is immediately updated after repaired by ULFM recovery, and FTI is able to use the repaired world communicator for MPI communication without incurring communication faults.

```c
/* world will swap between worldc[0] and worldc[1] after each respawn */
MPI_Comm worldc[2] = { MPI_COMM_NULL, MPI_COMM_NULL };
int worldi = 0;

//the MPI communicator must be implemented as a global variable to enable
     immediately update after ULFM recovery for FTI to use
#define world (worldc[worldi])

int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    // set long jump
    int do_recover = _setjmp(stack_jmp_buf);
    int survivor = IsSurvivor();
    /* set an errhandler on world, so that a failure is not fatal anymore
       */
    MPI_Comm_set_errhandler(world);
    FTI_Init(argv[1], world);
    ...
    // the main computation loop
    ...
    FTI_Finalize();
    MPI_Finalize();
}

/* error handler: repair comm world */
static void errhandler(MPI_Comm* pcomm, int* errcode, ...)
{
    int eclass;
    MPI_Error_class(*errcode, &eclass);

    if( MPIX_ERR_PROC_FAILED != eclass &&
        MPIX_ERR_REVOKED != eclass ) {
        MPI_Abort(MPI_COMM_WORLD, *errcode);
    }

    /* swap the worlds */
    worldi = (worldi+1)%2;

    MPIX_Comm_revoke(world);
    MPIX_Comm_shrink();
    MPI_Comm_spawn();
    MPI_Intercomm_merge();
    MPIX_Comm_agree();

    _longjmp( stack_jmp_buf, 1 );
}
```

Figure 7.3: A sample implementation of ULFM non-shrinking recovery.

7.4.4 Fault Injection

We emulate MPI process failures through fault injection. In particular, we raise a SIGTERM signal at the selected MPI process in the selected iteration of the main computation loop. We illustrate the fault injection code in Figure 7.4. Note that we choose to evaluate different fault tolerance techniques by triggering a process failure, which does not mean that the MPI recovery frameworks do not support recovery in a node failure. Reported in a recent study [60], Reinit can recover in a node failure, while ULFM

```
// simulation of proc failures
if (procfi == 1 && numIters==Selected_Iter){
    if (myrank == Selected_Rank){
        printf("KILL rank %d\n", myrank);
        kill(getpid(), SIGTERM);
    }
}
```

Figure 7.4: A sample implementation of fault injection.

cannot. In our case, it is sufficient to evaluate on MPI process failures to compare the performance difference when using FTI checkpointing in ULFM and Reinit.

7.5 Evaluation

We seek for answers for a few questions in the analyses and discussion of the evaluation results with respect to fault tolerance efficiency.

- Can fault tolerance interfaces (such as ULFM) delay the application execution or not?
- Can the checkpointing interface and the MPI recovery interface interfere with each other?
- Can ULFM perform better or Reinit perform better in different scaling sizes and different input problem sizes?

Table 7.1: Experimentation configuration for proxy applications (**default scaling size: 64 processes; default input problem: small**)

Application	Small Input	Medium Input	Large Input	No. of processes
AMG	-problem 2 -n 20 20 20	-problem 2 -n 40 40 40	-problem 2 -n 60 60 60	64, 128, 256, 512
CoMD	-nx 128 -ny 128 -nz 128	-nx 256 -ny 256 -nz 256	-nx 512 -ny 512 -nz 512	64, 128, 256, 512
HPCCG	64 64 64	128 128 128	192 192 192	64, 128, 256, 512
LULESH	-s 30 -p	-s 40 -p	-s 50 -p	64, 512
miniFE	-nx 20 -ny 20 -nz 20	-nx 40 -ny 40 -nz 40	-nx 60 -ny 60 -nz 60	64, 128, 256, 512
miniVite	-p 3 -l -n 128000	-p 3 -l -n 256000	-p 3 -l -n 512000	64, 128, 256, 512

7.5.1 Artifact Description

We run experiments on a large-scale HPC cluster having 752 nodes. Each node is equipped of two Intel Haswell CPUs, 28 CPU cores, 128 GB shared memory, and 8 TB local storage.

7.5.2 Experimentation Setup

This section provides the configuration details of the experimentation setup. We aim to test, evaluate, and compare the performance efficiency of different combinations and configurations of fault tolerance designs. In our experiments, we evaluate three fault tolerance designs. They are FTI checkpointing only, FTI checkpointing with ULFM recovery, and FTI checkpointing with Reinit recovery. “FTI checkpointing only" means that we restart the execution in a process failure for MPI recovery.

For FTI checkpointing, we use the L1 checkpointing mode. FTI L1 checkpointing allows users to store checkpoints to the local SSD or to do in-memory checkpointing. In our evaluation, we use the faster way that saves checkpoints to the local memory associated with the nodes in use using RAMFS through “/dev/shm". Although there are L1, L2, L3, and L4 modes for checkpointing, we do not evaluate all of them. The efficiency comparison between the four FTI checkpointing modes has been fully investigated in the FTI paper [15]. We save checkpoints every **ten** iterations. For ULFM, we use the latest version “ULFM v4.0.1ulfm2.1rc1" based on OpenMPI 4.0.1. For Reinit, we use its latest version based on OpenMPI 4.0.0.

We implement all the three fault tolerance designs to the `MATCH` benchmarks. Each evaluation is run on *three input problem sizes with the default scaling size (64 processes)* with and without fault injection. Also, each evaluation is run on *four scaling sizes (64 processes on 32 nodes, 128 processes on 32 nodes, 256 processes on 32 nodes, and 512 processes on 32 nodes) with the default input problem size (small)* with and without fault injection. We show the experimentation configuration in Table 7.1. Note that LULESH needs to run on a cube number of processes. We can only run LULESH on 64 and 512 processes.

For fault injection, we choose a certain iteration and a certain process to inject a fault. This enables us to fairly compare the efficiency of different fault tolerance configurations.

Notably, we run experiment of each configuration for five times, and calculate the average execution time to avoid any system noise. We use ‘-O3’ for mpicc or mpicxx compilation.

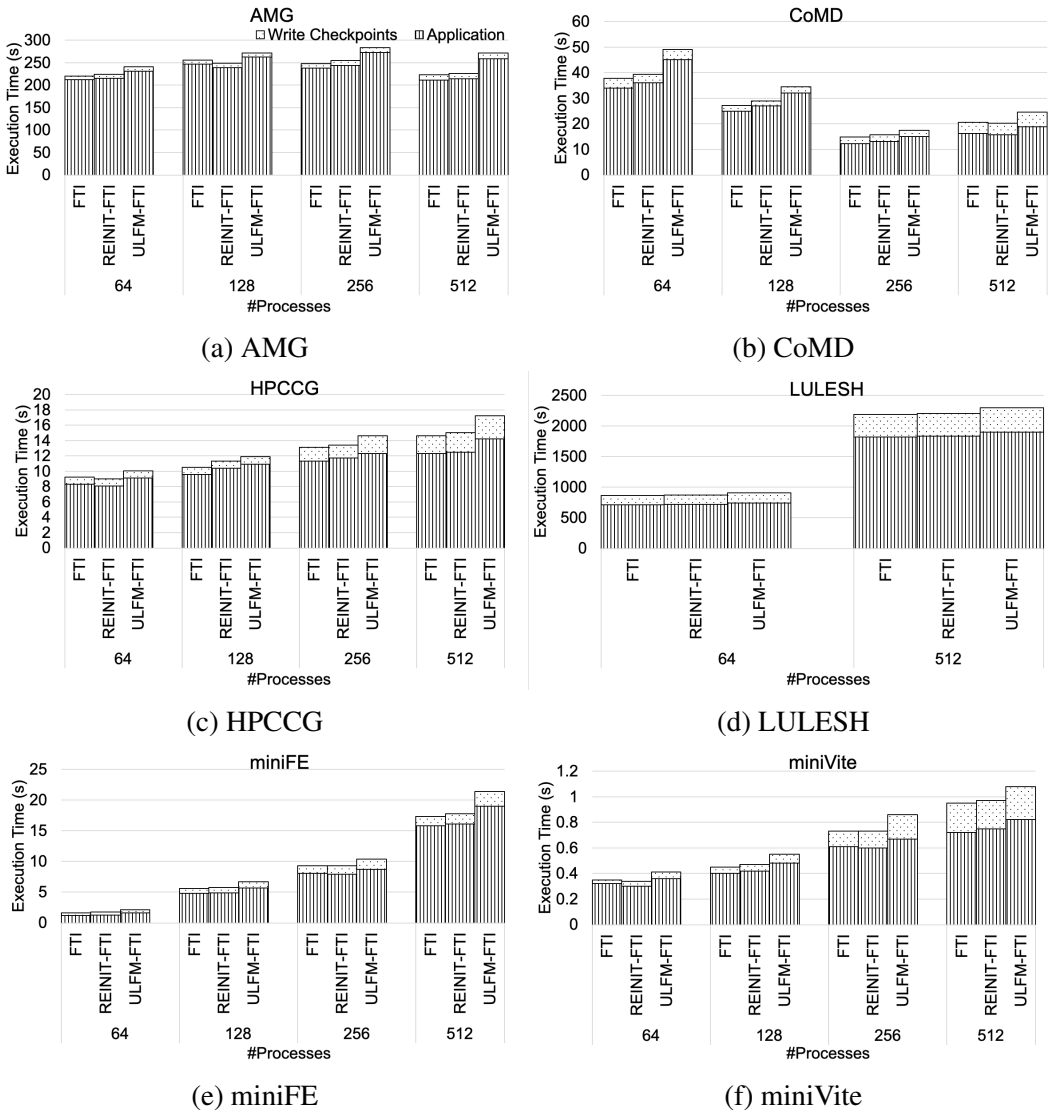
7.5.3 Performance Comparison on Different Scaling Sizes

In this experiment, we run each evaluation on four scaling sizes with the default input problem size (small). We seek to compare the scaling efficiency of the three fault tolerance designs with and without process failures.

Without A Failure: Figure 7.5 shows the average execution time when no failure occurs. We break down the execution time to *the pure application execution time* and *the time for writing checkpoints*.

Overall, we can see that among the three fault tolerance designs, the FTI checkpointing with ULFM recovery case performs worst. The FTI checkpointing only and the FTI checkpointing with Reinit recovery perform similar and better than “ULFM-FTI".

We first observe that FTI L1 checkpointing scales well. The time spent on writing checkpoints gently increases with more processes. This verifies that there are a number of collective operations implemented in FTI L1 checkpointing. The average time for writing checkpoints is accounted for 13% of the total execution time.



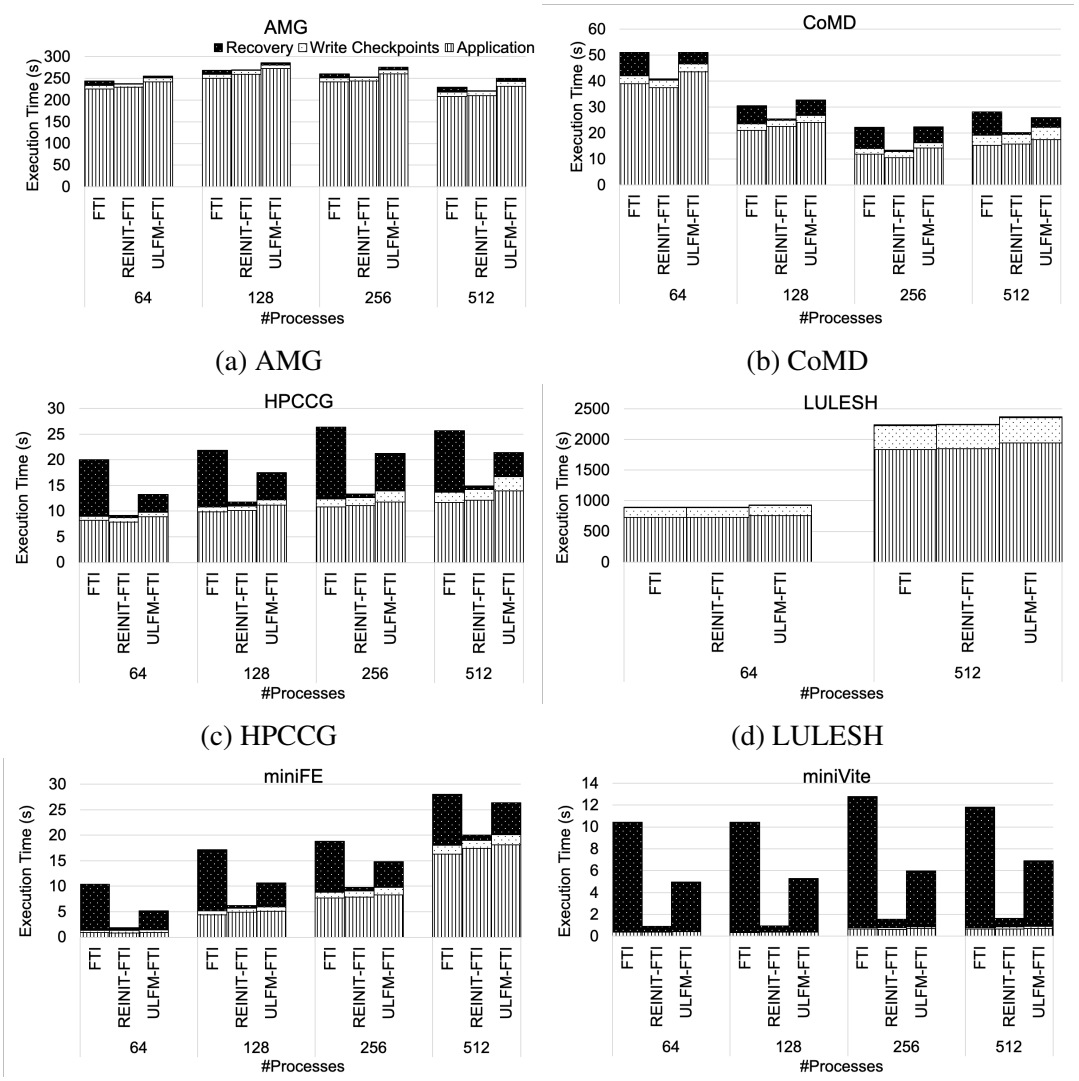
(a) AMG (b) CoMD

(c) HPCCG (d) LULESH

(e) miniFE (f) miniVite

Figure 7.5: Execution time breakdown recovering in different scaling sizes with no process failures

Second, we observe that Reinit has no impact to application execution when there is no failure. We use the FTI application execution time as the baseline for comparison because FTI is an application-level checkpointing library, whereas ULFM and Reinit modify the MPI runtime. We can see that the *application execution time* of "REINIT-FTI" is very close to the *application execution time* of cases using FTI checkpointing only. However, the "ULFM-FTI" cases using ULFM recovery introduce some overhead to the application execution time. This overhead increases as the number of processes goes up. This is understandable. ULFM is known as a framework implemented across MPI runtime and application levels. It can introduce memory and communication latency to the application execution and further affect the application execution efficiency. As reported in a ULFM paper [22], ULFM implements a constantly heartbeat mechanism for failures detection, and also amends MPI communication interfaces for failure recovery operations. These changes must have impact on the application execution. Dif-



(a) AMG (b) CoMD

(c) HPCCG (d) LULESH

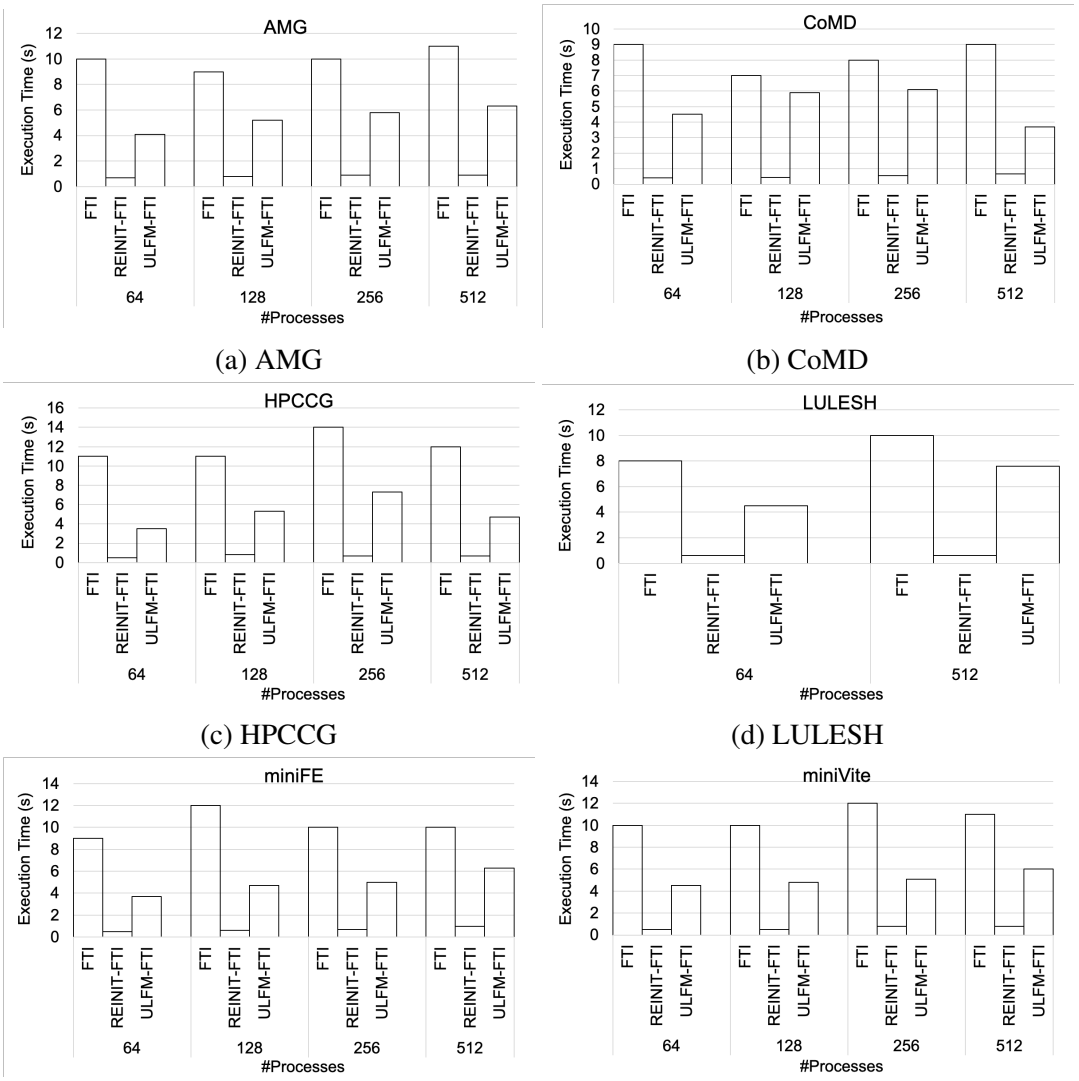
(e) miniFE (f) miniVite

Figure 7.6: Execution time breakdown recovering from a process failure in different scaling sizes

ferent than ULFM, Reinit incurs overhead only when a failure happens because it does not perform any other background operation in the MPI runtime during execution.

Furthermore, we observe that the times for writing checkpoints in FTI checkpointing only and “REINIT-FTI" cases are close. This indicates that Reinit has no interference on FTI checkpointing, yet ULFM has a small impact on FTI checkpointing in some cases such as HPCCG and miniVite. This is reasonable. Reinit implements the process recovery at the MPI runtime level, which has minimal impact on application-level operations, where the FTI operations run. Whereas ULFM does a significant amount of collective operations for periodic heartbeat in the MPI runtime, which leads to background overhead.

Conclusion 1. *“REINIT-FTI" cases achieve similar performance to “FTI checkpointing only" cases. This suggests using “REINIT-FTI" and “FTI checkpointing only" when there is no MPI process failure.*



(a) AMG (b) CoMD

(c) HPCCG (d) LULESH

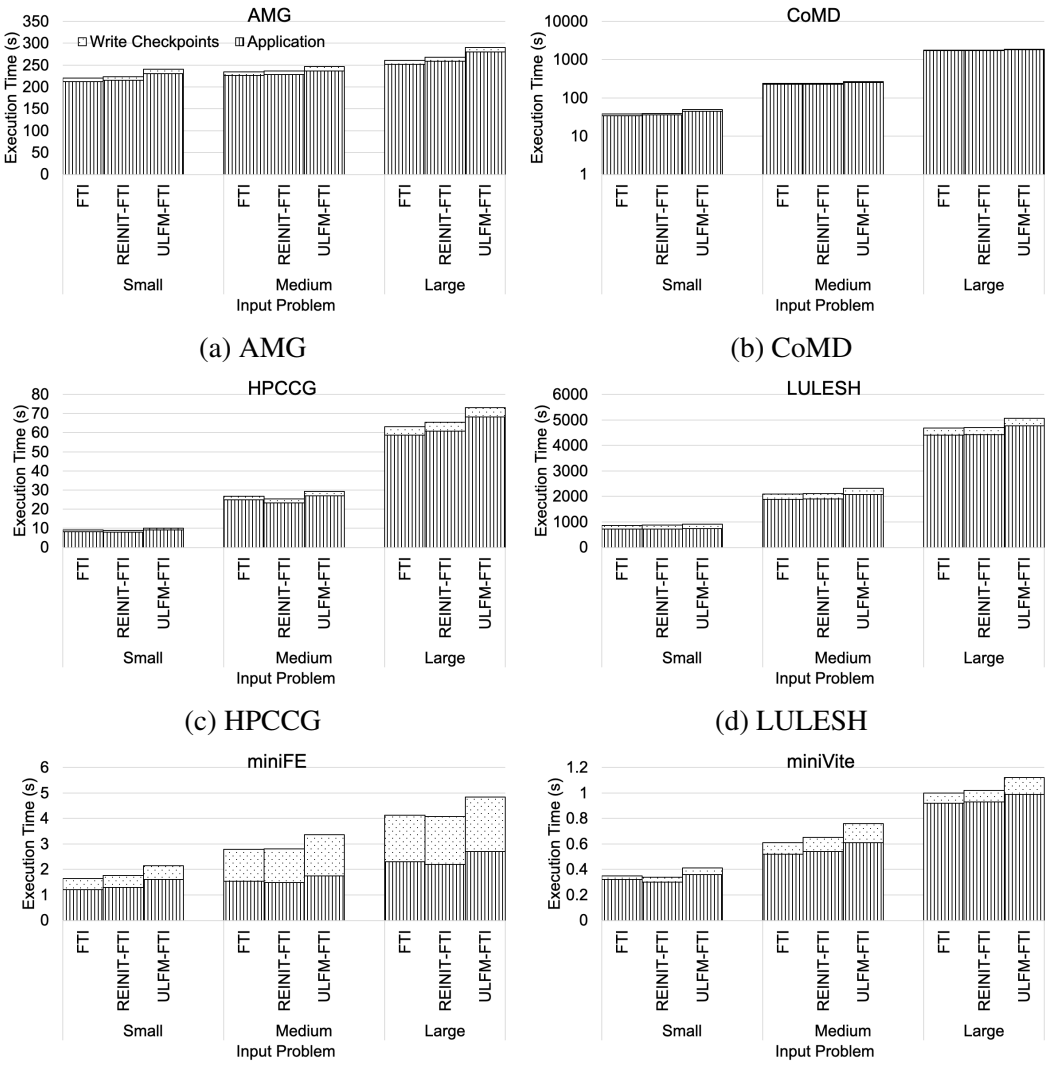
(e) miniFE (f) miniVite

Figure 7.7: Recovery time for different scaling sizes

With A Failure: Figure 7.6 shows the breakdown of execution time recovering from a process failure on different scaling sizes. Note that reading checkpoints only happens once in the execution, and has values in the order of milliseconds, which is difficult to observe, and we exclude it from the figure. Figure 7.7 shows the MPI recovery time for different scaling sizes.

Overall, we observe that “REINIT-FTI" achieves the best performance compared to the other two cases “FTI checkpointing only" and “ULFM-FTI". There are two essential reasons. First, “REINIT-FTI" does not affect the performance of writing checkpoints. Second, Reinit recovery achieves the best performance for MPI recovery than restarting and ULFM recovery. We can make the similar observations we made from Figure 7.5. Furthermore, we can make new observations. First, we can compare the time of MPI recovery for cases using restarting, Reinit, and ULFM. Also, we can find that restarting and ULFM recovery are significantly slower than Reinit recovery in many cases.

ULFM recovery vs. Reinit Recovery. By observation, we find that the ULFM



(a) AMG (b) CoMD

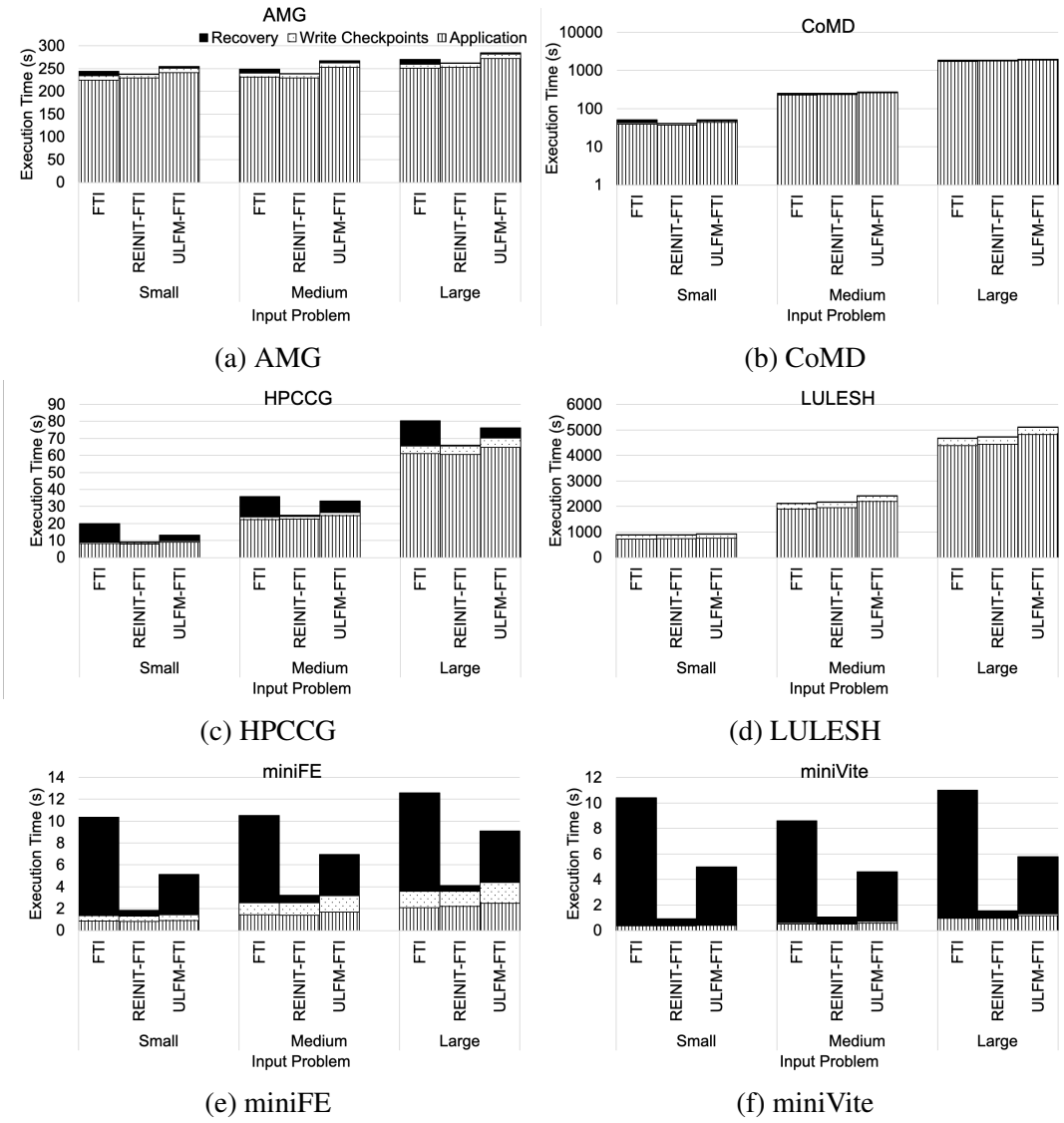
(c) HPCCG (d) LULESH

(e) miniFE (f) miniVite

Figure 7.8: Execution time breakdown in different input problem sizes with no process failures

recovery time can be up to 13 times larger than Reinit recovery time, and 4 times larger on average. We can also see a trend that the ULFM recovery time increases as the number of processes increases, not scaling well. *Different than ULFM, after counting numbers, we find that Reinit recovery time looks constant in many cases and is independent of the number of processes.* This makes sense. ULFM enforces a variety of fault tolerance collective operations on all MPI processes to enable the MPI global non-shrinking recovery. Even worse, ULFM implements these fault tolerance operations at the application level, which needs to synchronize with other fault tolerance operations implemented at the MPI runtime. On the contrast, Reinit is implemented at the MPI runtime level, which requests much fewer collective operations.

Restarting vs. Reinit recovery. By calculation, we find that the restarting recovery can be up to 22 times slower than Reinit recovery, and 16 times slower on average. This is acceptable. Redeployment of the MPI setup and allocation of resources for restarting



(a) AMG (b) CoMD

(c) HPCCG (d) LULESH

(e) miniFE (f) miniVite

Figure 7.9: Execution time breakdown recovering from a process failure in different input problem sizes

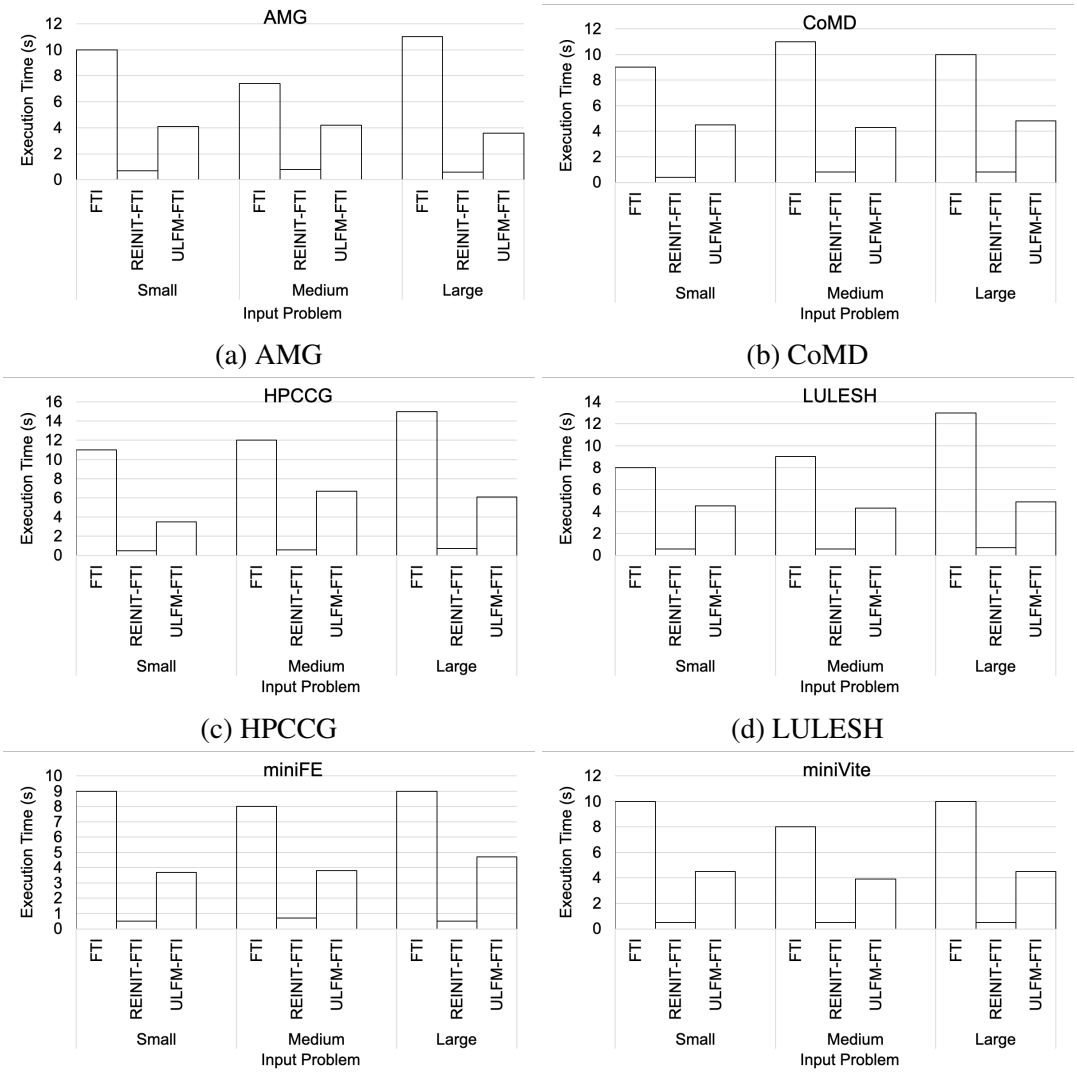
the execution is very expensive. Whereas Reinit recovery repairs the MPI state online.

Restarting vs. ULFM recovery. Restarting recovery is 2 to 3 times slower than ULFM recovery. Similarly, ULFM recovery is online recovery, which is much more efficient than redeployment.

Conclusion 2. “REINIT-FTI" outperforms “FTI checkpointing only" and “ULFM-FTI" in case of a failure. This suggests using “REINIT-FTI" for MPI fault tolerance.

7.5.4 Performance Comparison on Different Input Sizes

In this experiment, we perform the performance comparison of three fault tolerance designs on three input problem sizes with the default scaling size (64 processes), with and without fault injection. Each configuration runs for five times, and we count the average of the five runs to avoid any system noise.



(a) AMG (b) CoMD

(c) HPCCG (d) LULESH

(e) miniFE (f) miniVite

Figure 7.10: Recovery time for different input problem sizes

Without A Failure: Figure 7.8 presents the results of application execution in different input problem sizes with no process failures. The execution time is divided into the time of writing checkpointing and pure application execution time. We make several observations. Again, we use the pure application execution time of FTI as the baseline for comparison.

First, we can see an increment on the pure application execution time and FTI checkpointing time when running on larger input problem sizes because the amount of data to process increases.

We can also observe the performance latency in application execution time in “ULFM-FTI" cases using ULFM recovery. This latency increases as the input problem size grows. This indicates that ULFM is intensively involved into the application execution, where ULFM fault tolerance operations run a large number of collective MPI operations. These inefficient operations significantly affect the application execution, causing a huge communication latency, especially when there is a large amount of data

to process and communicate. Different than ULFM, Reinit does not delay the application execution. We can observe that the application execution time of “REINIT-FTI" cases is very close to the execution time of the “FTI checkpointing only" cases. This is expected as Reinit is implemented in the MPI runtime. Also, Reinit uses much fewer collective operations than ULFM used.

With A Failure: Figure 7.9 shows the results of execution time breakdown when recovering from a process failure in different input problem sizes. Note that we omit the time of reading checkpoints because it is in the order of milliseconds. Also, Figure 7.10 shows the recovery time for different input problem sizes.

From the results, we can make the same observation we make through Figure 7.8 and results of the scaling experimentation. However, the new observation is that, after counting numbers, we find that either the recovery times of ULFM or Reinit only has a negligible change when the input problem sizes increase. This is an interesting finding, but makes sense. When a failure occurs, ULFM starts collecting messages among daemons and processes, which cannot be affected by application execution because the application stops computing and communicating data. Reinit is fully implemented in the MPI runtime, which is even more difficult to be affect. We find that ULFM and Reinit process recovery are independent of input problem size.

Conclusion 3. *Through the performance comparison results on different input sizes, we again find that “REINIT-FTI" is the most efficient design within the three fault tolerance designs.*

7.6 Conclusions

MPI fault tolerance is becoming an increasingly critical problem as supercomputers continue to grow in size and add new components. We have designed and implemented a benchmark suite `MATCH` with an emphasis on MPI fault tolerance. Our benchmark suite has six representative HPC proxy applications selected from flagship benchmark suites. We comprehensively evaluate and compare the performance efficiency of the three fault tolerance designs we implement into the six workloads. The evaluation results reveal that FTI checkpointing with Reinit recovery is the most efficient fault tolerance design within the three designs. Our analytics and insights will inspire future MPI fault tolerance designs.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

HPC systems are widely used in industrial, economical, and scientific applications, and many of these applications are safety- and time-critical. We must ensure that the application execution is reliable and the scientific simulation outcome is highly trustful. My Ph.D. research has been focusing on characterizing, modeling, developing, and advancing fault tolerance strategies and tools in HPC systems to allow scientific applications to better survive system failures.

In this dissertation, we have systematically characterized and modeled application-level error resilience in HPC at multiple granularities, from the data object, to the code region, and to the entire application. The characterization and modeling have covered a diversity of application types including serial programs, GPU applications, and MPI applications. We have investigated a collection of fault tolerance techniques that aims at two types of system faults–transient faults and process/node failures. With lessons and insights we have learned through characterization and modeling of HPC fault tolerance, we summarize the following takeaway messages which we believe would be helpful for future HPC researchers.

- HPC systems are suffering from an increasing number of system errors as the complexity and heterogeneity of next-generation HPC systems grows. Future HPC systems are expected to see system errors in diverse and unseen forms. This creates new and critical challenges for future fault tolerance design.
- Natural error resilience comes for free that we can leverage to improve the efficiency of fault tolerance designs. We have found and studied six resilience computation patterns from serial HPC code, which we believe are generic and can be applied to parallel HPC code as well. Natural error resilience can not only be found in serial HPC applications, we believe we can find more and special natural error resilience patterns in parallel programming models too. In addition, we have seen natural error resilience examples in architecture and microarchitecture levels. We should discover and investigate these resilience patterns, and use them in our fault tolerance designs.
- In Chapter 4, we have proposed a methodology to definitely measure application by quantifying error masking events. Following this methodology, we develop a

metric to measure application error resilience on data objects. More importantly, we believe this methodology is generic and can be used to measure application resilience at a larger granularity as well.

- Machine Learning-based prediction of application resilience shows its value, which can help filter program properties that matter most to application resilience. However, we cannot rely on the prediction results to decide the fault tolerance mechanism until we achieved a persistently high prediction accuracy in practice.
- Reinit^{++} is so far the more efficient fault tolerance framework for global-restart non-shrinking MPI recovery. Although many HPC applications follow the Bulk Synchronous Parallel paradigm, it is helpful to support local-restart shrinking MPI recovery in Reinit^{++} as well.

8.2 Future Work

HPC systems tend to be heterogeneous in the era of big data to have the capability to process a variety of data resources. Emerging HPC systems are becoming domain-specific systems with a diversity of software and architecture components adding to the system. Examples of proceeding software include machine learning toolkits such as Google TensorFlow [2] and LLNL LBANN [149], and big data frameworks such as Spark [159]. Examples of novel hardware include GPU, FPGA, Google TPU, and emerging IoT devices. HPC practitioners are developing new platforms such as RAJA [74] to advance the portability, flexibility, and scalability of the emerging software to enable them to run on large-scale parallel systems and also run on the emerging hardware. All these new software and hardware components have significantly increased the complexity of the system. More importantly, these changes bring new challenges and opportunities to system designs with respect to fault tolerance. The out-of-date fault tolerance designs are unlikely to fit into the emerging HPC systems. To solve the problem, we must propose new fault tolerance techniques and designs to be able to fit into these changes.

8.2.1 Next-Generation Fault Tolerance Mechanisms for Big Data Frameworks

Big Data (BD) is significantly changing our life and work and is becoming the key driver for scientific research. For example, the DOE Inertial Confinement Fusion (ICF) simulation is using 3.8 billion images for machine learning training, which is more than 200 times larger than ImageNet (the largest commercial dataset for visual recognition), to advance the simulation accuracy to ensure that the current and future nuclear stockpile is safe and reliable. Given the fact that BD has been intensively deployed and processed in security-critical scientific applications, the primary concern is to ensure that the BD processing outcome is highly dependable in the presence of system errors.

Existing fault tolerance designs for BD are coarse-grained and application semantics-agnostic. For example, Resilient Distributed Datasets (RDDs) [159], which enable distributed and parallelized checkpoint/restart in Spark—a production BD framework. Later, Flint [137] proposes the optimized RDDs, which develop policies and mechanisms for selectively checkpointing. Even though the new RDDs design saves up to

90% checkpointing overhead, the new design does not understand fault tolerance at a fine granularity, such as how errors are propagated from the initial corrupted location to other locations and to the application outcome, and how and where errors are tolerated during error propagation. A potential research opportunity could be to research the fault tolerance design that takes high-level application resilience into account to enable fault tolerance at a fine granularity to enable efficient and effective data processing.

The major research challenges include how to find effective and efficient BD domain-specific characteristics and how to efficiently apply these characteristics to fault tolerance designs. The goal is to characterize representative BD frameworks to research and identify fundamentally new ways to design and build effective and efficient fault tolerance mechanisms for BD frameworks by leveraging BD domain-specific characteristics at both the system- and application-level. The research aims to develop theorem and practical tools for highly dependable and substantial BD systems that can provide rigorous and meaningful guarantees.

8.2.2 Application-Aware AVF Analysis

Architectural Vulnerability Factor (AVF) [111] measures the probability that a fault occurring in a hardware architecture leads to a visible error in the program output. AVF is typically used to guide the architecture-level fault tolerance design. However, AVF does not understand high-level application resilience and therefore cannot guide the future fault tolerance design for domain-specific architectures.

A collection of domain-specific hardware architectures (such as FPGA and Cloud TPU) are developed to further drive the execution and advance the execution performance of domain- specific applications. These architectures are merely protected by architecture protection mechanisms such as parity and ECC code. The design (or decision) of architecture protection mechanisms is directed by AVF. AVF takes microarchitecture and architecture level fault tolerance into consideration. However, AVF does not consider higher-level application resilience information (e.g., resilience computation patterns), which is particularly important for directing fault tolerance designs for domain-specific architectures.

The other research possibility is to introduce higher-level application resilience information into AVF to complement the capability of AVF to support the design of domain-specific architectures. The research challenges include how to characterize application resilience for domain-specific applications in different domains at system- and application-level, how to identify unique resilience patterns in domain-specific applications, and how to apply these domain-specific application characteristics and resilience patterns into AVF calculation and improve fault tolerance designs. The goal is to propose a systematic methodology for developing more efficient and effective architecture- and application-level fault tolerance mechanisms for domain-specific hardware architectures.

Bibliography

[1] Coral Benchmark Codes. `https://asc.llnl.gov/CORAL-benchmarks/`.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[3] Julien Adam, Jean-Baptiste Besnard, Allen D Malony, Sameer Shende, Marc Pérache, Patrick Carribault, and Julien Jaeger. Transparent high-speed network checkpoint/restart in mpi. In *Proceedings of the 25th European MPI Users' Group Meeting*, page 12, 2018.

[4] Julien Adam, Maxime Kermarquer, Jean-Baptiste Besnard, Leonardo Bautista-Gomez, Marc Pérache, Patrick Carribault, Julien Jaeger, Allen D Malony, and Sameer Shende. Checkpoint/restart approaches for a thread-based mpi runtime. *Parallel Computing*, 85:204–219, 2019.

[5] Tejaswi Agarwal and Michela Becchi. Design of a hybrid mpi-cuda benchmark suite for cpu-gpu clusters. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE, 2014.

[6] Hasan Metin Aktulga, Joseph C Fogarty, Sagar A Pandit, and Ananth Y Grama. Parallel reactive molecular dynamics: Numerical methods and algorithmic techniques. *Parallel Computing*, 2012.

[7] Farhana Aleen, Monirul Sharif, and Santosh Pande. Input-driven dynamic execution prediction of streaming applications. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2010.

[8] Md Mohsin Ali, Peter E Strazdins, Brendan Harding, and Markus Hegland. Complex scientific applications made fault-tolerant with the sparse grid combination technique. *The International Journal of High Performance Computing Applications*, 30(3):335–359, 2016.

[9] Cesare Alippi, Vincenzo Piuri, and Mariagiovanna Sami. Sensitivity to errors in artificial neural networks: A behavioral approach. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 42(6):358–361, 1995.

[10] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: A language and compiler for algorithmic choice. In *PLDI*, 2009.

[11] Rizwan Ashraf, Roberto Gioiosa, Gokcen Kestor, Ronald F. DeMara, Chen-Yong Cher, and Pradip Bose. Understanding the propagation of transient errors in HPC applications. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[12] D. H. Bailey, L. Dagum, E. Barszcz, and H. D. Simon. NAS Parallel Benchmark Results. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 1992.

[13] Roberto Battiti. Using mutual information for selecting features in supervised neural net learning. *IEEE Transactions on neural networks*, 5(4):537–550, 1994.

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

[15] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. Fti: high performance fault tolerance interface for hybrid systems. In *International conference for high performance computing, networking, storage and analysis (SC)*, 2011.

[16] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 2012.

[17] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008.

[18] Arijit Biswas, Paul Racunas, Razvan Cheveresan, Joel Emer, Shubhendu S. Mukherjee, and Ram Rangan. Computing Arch. Vulnerability Factors for Address-Based Structures. In *International Symposium of Computer Architecture (ISCA)*, 2005.

[19] Wesley Bland, Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack Dongarra. Post-failure recovery of mpi communication capability: Design and rationale. *The International Journal of High Performance Computing Applications*, 27(3):244–254, 2013.

[20] Wesley Bland, Huiwei Lu, Sangmin Seo, and Pavan Balaji. Lessons learned implementing user-level failure mitigation in mpich. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2015.

[21] George Bosilca, Aurelien Bouteiller, Amina Guermouche, Thomas Herault, Yves Robert, Pierre Sens, and Jack Dongarra. Failure detection and propagation in hpc systems. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 312–322, 2016.

[22] George Bosilca, Aurelien Bouteiller, Amina Guermouche, Thomas Herault, Yves Robert, Pierre Sens, and Jack Dongarra. A failure detector for hpc platforms. *The International Journal of High Performance Computing Applications*, 32(1):139–158, 2018.

[23] Aurelien Bouteiller, George Bosilca, and Jack J Dongarra. Plan b: Interruption of ongoing mpi operations to support failure recovery. In *Proceedings of the 22nd European MPI Users' Group Meeting*, page 11, 2015.

[24] Paul S Bradley and Olvi L Mangasarian. Feature selection via concave minimization and support vector machines. In *ICML*, volume 98, 1998.

[25] Johan Bring. How to standardize regression coefficients. *The American Statistician*, 48(3):209–213, 1994.

[26] J Mark Bull, James P Enright, and Nadia Ameer. A microbenchmark suite for mixed-mode openmp/mpi. In *International Workshop on OpenMP*. Springer, 2009.

[27] Devendar Bureddy, Hao Wang, Akshay Venkatesh, Sreeram Potluri, and Dhabaleswar K Panda. OMB-GPU: a micro-benchmark suite for evaluating MPI libraries on GPU clusters. In *European MPI Users' Group Meeting*. Springer, 2012.

[28] Jon Calhoun, Luke Olson, and Marc Snir. Flipit: An LLVM based fault injector for HPC. In *Euro-Par 2014 International Workshops*, 2014.

[29] Jon Calhoun, Marc Snir, Luke N. Olson, and William D. Gropp. Towards a more complete understanding of sdc propagation. In *International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2017.

[30] Jiajun Cao, Kapil Arya, Rohan Garg, Shawn Matott, Dhabaleswar K Panda, Hari Subramoni, Jérôme Vienne, and Gene Cooperman. System-level scalable checkpoint-restart for petascale computing. In *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*, 2016.

[31] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward exascale resilience. *International Journal of High Performance Computing Applications*, 23(4):374–388, 2009.

[32] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward Exascale Resilience: 2014 Update. *International Journal of Supercomputing Frontiers and Innovations*, 1(1), 2014.

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

[34] Marc Casas, Bronis R. de Supinski, Greg Bronevetsky, and Martin Schulz. Fault Resilience of the Multi-grid Solver. In *ICS*, 2012.

[35] Sourav Chakraborty, Ignacio Laguna, Murali Emani, Kathryn Mohror, Dhabaleswar K. Panda, Martin Schulz, and Hari Subramoni. Ereinit: Scalable and efficient fault-tolerance for bulk-synchronous mpi applications. *Concurrency and Computation: Practice and Experience*, 0(0):e4863. e4863 cpe.4863.

[36] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[37] Xinchi Chen, Xipeng Qiu, Chenxi Zhu, and Xuanjing Huang. Gated recursive neural network for chinese word segmentation. In *ACL*, 2015.

[38] Zizhong Chen. Algorithm-based Recovery for Iterative Methods without Checkpointing. In *HPDC*, 2011.

[39] Zizhong Chen. Online-ABFT: An Online ABFT Scheme for Soft Error Detection in Iterative Methods. *PPoPP*, 2013.

[40] C.-Y. Cher, M. S. Gupta, P. Bose, and K. P. Muller. Understanding Soft Error Resiliency of BlueGene/Q Compute Chip Through Hardware Proton Irradiation and Software Fault Injection. In *SC*, 2014.

[41] I-Hsin Chung, Robert E Walkup, Hui-Fang Wen, and Hao Yu. Mpi performance analysis tools on Blue Gene/L. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2006.

[42] Adam Coates, Andrew Ng, and Honglak Lee. An analysis of single-layer networks in unsupervised feature learning. In *AISTATS*, 2011.

[43] J. J. Cook and C. Zilles. A Characterization of Instruction-Level Error Derating and its Implications for Error Detection. In *International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, 2008.

[44] Anwesha Das, Frank Mueller, Charles Siegel, and Abhinav Vishnu. Desh: deep learning for system health prediction of lead times to failure in hpc. In *HPDC*, 2018.

[45] Teresa Davies and Zizhong Chen. Correcting Soft Errors Online in LU Factorization. In *International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2013.

[46] Arnaud De Myttenaere, Boris Golden, Bénédicte Le Grand, and Fabrice Rossi. Mean absolute percentage error for regression models. *Neurocomputing*, 192:38–48, 2016.

[47] Daniel Alfonso Goncalves De Oliveira, Laercio Lima Pilla, Mauricio Hanzich, Vinicius Fratin, Fernando Fernandes, Caio Lunardi, José María Cela, Philippe Olivier Alexandre Navaux, Luigi Carro, and Paolo Rech. Radiation-induced error criticality in modern hpc parallel accelerators. In *HPCA*, 2017.

[48] Debra Werner. HPE Supercomputer in Orbit is Ready for Researchers. https://spacenews.com/hpe-supercomputer.

[49] 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 *IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014.

[50] Pedro Domingos. Bayesian averaging of classifiers and the overfitting problem. In *ICML*, 2000.

[51] J Dongarra. Emerging heterogeneous technologies for high performance computing. In *International Heterogeneity in Computing Workshop*, 2013.

[52] Matthew GF Dosanjh, Taylor Groves, Ryan E Grant, Ron Brightwell, and Patrick G Bridges. RMA-MT: a benchmark suite for assessing MPI multithreaded RMA performance. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 2016.

[53] Peng Du, Aurelien Bouteiller, George Bosilca, Thomas Herault, and Jack Dongarra. Algorithm-based Fault Tolerance for Dense Matrix Factorizations. In *PPoPP*, 2012.

[54] L. Duan, B. Li, and L. Peng. Versatile Prediction and Fast Estimation of Architectural Vulnerability Factor from Processor Performance Metrics. In *HPCA*, 2009.

[55] J. Elliott, M. Hoemmen, and F. Mueller. Evaluating the Impact of SDC on the GMRES Iterative Solver. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1193–1202, 2014.

[56] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[57] Marc Gamell, Daniel S. Katz, Hemanth Kolla, Jacqueline Chen, Scott Klasky, and Manish Parashar. Exploring automatic, online failure recovery for scientific applications at extreme scales. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 895–906, Piscataway, NJ, USA, 2014. IEEE Press.

[58] Marc Gamell, Keita Teranishi, Michael A Heroux, Jackson Mayo, Hemanth Kolla, Jacqueline Chen, and Manish Parashar. Local recovery and failure masking for stencil-based applications at extreme scales. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[59] Emden R. Gansner and Stephen C. North. Graphviz: an open graph visualization system and its applications to software engineering. *Software - Practice and Experience*, 30(11):1203–1233, 2000.

[60] Giorgis Georgakoudis, Luanzheng Guo, and Ignacio Laguna. Reinit++: Evaluating the performance of global-restart recovery methods for mpi fault tolerance. In *ISC*, 2020.

[61] Giorgis Georgakoudis, Ignacio Laguna, Dimitrios S. Nikolopoulos, and Martin Schulz. REFINE : Realistic Fault Injection via Compiler-based Instrumentation for Accuracy , Portability and Speed. In *SC*, 2017.

[62] Siavash Ghiasvand, Florina M Ciorba, Ronny Tschüter, and Wolfgang E Nagel. Lessons learned from spatial and temporal correlation of node failures in high performance computers. In *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, 2016.

[63] Luanzheng Guo and Dong Li. MOARD: Modeling Application Resilience to Transient Faults on Data Objects. In *International Parallel and Distributed Processing Symposium*, 2019.

[64] Luanzheng Guo, Dong Li, and Ignacio Laguna. PARIS: Predicting application resilience using machine learning. *arXiv preprint arXiv:1811.10379*, 2018.

[65] Luanzheng Guo, Dong Li, Ignacio Laguna, and Martin Schulz. Fliptracker: Understanding natural error resilience in hpc applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2018.

[66] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *JMLR*, 3(Mar):1157–1182, 2003.

[67] HackRank. HackRank Home Page. *https://www.hackerrank.com/* (Since 2009).

[68] Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *Journal of Physics: Conference Series*, volume 46, page 494, 2006.

[69] Siva Kumar Sastry Hari, Sarita V. Adve, Helia Naeimi, and Pradeep Ramachandran. Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[70] V. E. Henson and U. M. Yang. BoomerAMG: A Parallel Algebraic Multigrid Solver and Preconditioner. *Appl. Num. Math*, 41, 2002.

[71] Thomas Herault, Aurelien Bouteiller, George Bosilca, Marc Gamell, Keita Teranishi, Manish Parashar, and Jack Dongarra. Practical scalable consensus for pseudo-synchronous distributed systems. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[72] Justin Holewinski, Ragavendar Ramamurthi, Mahesh Ravishankar, Naznin Fauzia, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. Dynamic Trace-based Analysis of Vectorization Potential of Applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[73] Atsushi Hori, Kazumi Yoshinaga, Thomas Herault, Aurélien Bouteiller, George Bosilca, and Yutaka Ishikawa. Sliding substitution of failed nodes. In *Proceedings of the 22nd European MPI Users' Group Meeting*, page 14. ACM, 2015.

[74] Richard D Hornung and Jeffrey A Keasler. The raja portability layer: overview and status. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2014.

[75] Kenneth Hoste and Lieven Eeckhout. Comparing benchmarks using key microarchitecture-independent characteristics. In *2006 IEEE International Symposium on Workload Characterization*, 2006.

[76] Kuang-Hua Huang and J. A. Abraham. Algorithm-Based Fault Tolerance for Matrix Operations. *IEEE Transactions on Computers*, C-33(6):518–528, 1984.

[77] A. A. Hwang, I. A. Stefanovici, and B. Schroeder. Cosmic Rays don't Strike Twice: Understanding the Nature of DRAM Errors and the Implication for System Design. In *Proceedings of the International Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, 2012.

[78] Andhi Janapsatya, Aleksandar Ignjatovic, Sri Parameswaran, and Joerg Henkel. Instruction trace compression for rapid instruction cache simulation. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2007.

[79] Manolis Kaliorakis, Dimitris Gizopoulos, Ramon Canal, and Antonio Gonzalez. MeRLiN: Exploiting Dynamic Instruction Behavior for Fast and Accurate Microarchitecture Level Reliability Assessment. In *ISCA*, 2017.

[80] Charu Kalra, Fritz Previlon, Xiangyu Li, Norman Rubin, and David Kaeli. Prism: predicting resilience of gpu applications using statistical methods. In *PRISM: Predicting Resilience of GPU Applications Using Statistical Methods*, 2018.

[81] Ian Karlin, Abhinav Bhatele, and etc. Exploring Traditional and Emerging Parallel Programming Models using a Proxy Application. In *IEEE International Parallel and Distributed Processing Symposium*, 2013.

[82] Ian Karlin, Jeff Keasler, and Rob Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, August 2013.

[83] Johan Karlsson, Peter Liden, Peter Dahlgren, Rolf Johansson, and Ulf Gunneflo. Using heavy-ion radiation to validate fault-handling mechanisms. *IEEE micro*, 1994.

[84] Amogh Katti, Giuseppe Di Fatta, Thomas Naughton, and Christian Engelmann. Scalable and fault tolerant failure detection and consensus. In *Proceedings of the 22nd European MPI Users' Group Meeting*, page 13, 2015.

[85] Amogh Katti, Giuseppe Di Fatta, Thomas Naughton, and Christian Engelmann. Epidemic failure detection and consensus for extreme parallelism. *The International Journal of High Performance Computing Applications*, 32(5):729–743, 2018.

[86] Gokcen Kestor, Ivy Bo Peng, Roberto Gioiosa, and Sriram Krishnamoorthy. Understanding Scale-Dependent soft-Error Behavior of Scientific Applications. In *International Symposium on Cluster, Cloud and Grid Computing*, 2018.

[87] Nils Kohl, Johannes Hötzer, Florian Schornbaum, Martin Bauer, Christian Godenschwager, Harald Köstler, Britta Nestler, and Ulrich Rüde. A scalable and extensible checkpointing scheme for massively parallel simulations. *The International Journal of High Performance Computing Applications*, 33(4):571–589, 2019.

[88] Siva Kumar Sastry Hari, Timothy Tsai, Mark Stephenson, Stephen W. Keckler, and Joel Emer. Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation. In *ISPASS*, 2017.

[89] Ignacio Laguna, David F. Richards, Todd Gamblin, Martin Schulz, and Bronis R. de Supinski. Evaluating user-level fault tolerance for mpi applications. In *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, pages 57:57–57:62, New York, NY, USA, 2014. ACM.

[90] Ignacio Laguna, David F. Richards, Todd Gamblin, Martin Schulz, and Bronis R. de Supinski. Evaluating user-level fault tolerance for mpi applications. In *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, pages 57:57–57:62, New York, NY, USA, 2014. ACM.

[91] Ignacio Laguna, David F Richards, Todd Gamblin, Martin Schulz, Bronis R de Supinski, Kathryn Mohror, and Howard Pritchard. Evaluating and extending user-level fault tolerance in mpi applications. *The International Journal of High Performance Computing Applications*, 30(3):305–319, 2016.

[92] Ignacio Laguna, David F Richards, Todd Gamblin, Martin Schulz, Bronis R de Supinski, Kathryn Mohror, and Howard Pritchard. Evaluating and extending user-level fault tolerance in mpi applications. *The International Journal of High Performance Computing Applications*, 30(3):305–319, 2016.

[93] Ignacio Laguna, Martin Schulz, David F Richards, Jon Calhoun, and Luke Olson. IPAS: Intelligent protection against silent output corruption in scientific applications. In *CGO*, 2016.

[94] Larkhoon Leem, Hyungmin Cho, Jason Bau, Quinn A Jacobson, and Subhasish Mitra. ERSA: Error Resilient System Architecture for Probabilistic Applications. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2010.

[95] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. Statistical Fault Injection: Quantified Error and Confidence. In *Conference on Design, Automation and Test in Europe (DATE)*, 2009.

[96] Dong Li, Jeffrey S. Vetter, and Weikuan Yu. Classifying Soft Error Vulnerabilities in Extreme-Scale Scientific Applications Using a Binary Instrumentation Tool. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[97] Guanpeng Li, Karthik Pattabiraman, Chen-Yong Cher, and Pradip Bose. Understanding Error Propagation in GPGPU. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2016.

[98] Guanpeng Li, Karthik Pattabiraman, Siva Kumar Sastry Hari, Michael Sullivan, and Timothy Tsai. Modeling soft-error propagation in programs. In *DSN*, 2018.

[99] X. Li and D. Yeung. Application-level Correctness and Its Impact on Fault Tolerance. In *International Symposium on Computer Arch.*, 2007.

[100] Xiaodong Li, Sarita V. Adve, Pradip Bose, and Jude Rivers. Online Estimation of Arch Vulnerability Factor for Soft Errors. In *ISCA*, 2008.

[101] LLVM. LLVM Language Reference Manual. `http://llvm.org`.

[102] Nuria Losada, Iván Cores, María J Martín, and Patricia González. Resilient mpi applications using an application-level checkpointing framework and ulfm. *The Journal of Supercomputing*, 73(1), 2017.

[103] Yixin Luo, Sriram Govindan, Bikash Sharma, Mark Santaniello, Justin Meza, Aman Kansal, Jie Liu, Badriddine Khessib, Kushagra Vaid, and Onur Mutlu. Characterizing application memory error vulnerability to optimize datacenter cost via heterogeneous-reliability memory. In *DSN*, 2014.

[104] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The hpc challenge (hpcc) benchmark suite. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006.

[105] Kiwan Maeng, Alexei Colin, and Brandon Lucia. Alpaca: Intermittent execution without checkpoints. *Proceedings of ACM Programming Language*, 2017.

[106] Harshitha Menon and Kathryn Mohror. Discvar: discovering critical variables using algorithmic differentiation for transient faults. In *PPOPP*, 2018.

[107] Thomas P. Minka. Bayesian linear regression. Technical report, 2010.

[108] Subrata Mitra, Greg Bronevetsky, Suhas Javagal, and Saurabh Bagchi. Dealing with the unknown: Resilience to prediction errors. In *PACT*, 2015.

[109] K. Mohror, A. Moody, G. Bronevetsky, and B. R. de Supinski. Detailed modeling and evaluation of a scalable multilevel checkpointing system. *IEEE Transactions on Parallel and Distributed Systems*, 25(9):2255–2263, Sep. 2014.

[110] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R. de Supinski. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *Conference on High Performance Computing Networking, Storage and Analysis (SC)*, 2010.

[111] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In *International Symposium on Microarchitecture*, 2003.

[112] J Robert Neely and Bronis R de Supinski. Application modernization at llnl and the sierra center of excellence. *Computing in Science & Engineering*, 2017.

[113] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Network and Distributed Systems Security Symposium (NDSS)*, 2005.

[114] Bin Nie, Ji Xue, Saurabh Gupta, Tirthak Patel, Christian Engelmann, Evgenia Smirni, and Devesh Tiwari. Machine learning models for gpu error prediction in a large scale hpc system. In *DSN*, 2018.

[115] Bin Nie, Lishan Yang, Adwait Jog, and Evgenia Smirni. Fault site pruning for practical reliability analysis of gpgpu applications. In *Proceedings of the International Symposium on Microarchitecture MICRO*, 2018.

[116] Michael Noeth, Prasun Ratn, Frank Mueller, Martin Schulz, and Bronis R. de Supinski. Scalatrace: Scalable compression and replay of communication traces for high-performance computing. *Journal of Parallel Distributed Computing*, 69(8):696–710, 2009.

[117] Konstantinos Parasyris, Georgios Tziantzoulis, Christos D Antonopoulos, and Nikolaos Bellas. Gemfi: A fault injection tool for studying the behavior of applications on unreliable substrates. In *DSN*, 2014.

[118] Karthik Pattabiraman, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Application-Based Metrics for Strategic Placement of Detectors. In *Pacific Rim International Symposium on Dependable Computing*, 2005.

[119] Karthik Pattabiraman, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Automated derivation of application-aware error detectors using static analysis: The trusted illiac approach. *IEEE Transactions on Dependable and Secure Computing*, 8(1):44–57, 2011.

[120] Stefan Pauli, Manuel Kohler, and Peter Arbenz. A fault tolerant implementation of multi-level monte carlo methods. *Parallel computing: Accelerating computational science and engineering (CSE)*, 25:471–480, 2014.

[121] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[122] Wenzhe Pei, Tao Ge, and Baobao Chang. Max-margin tensor neural network for chinese word segmentation. In *ACL*, 2014.

[123] Vincenzo Piuri. Analysis of fault tolerance in artificial neural networks. *Journal of Parallel and Distributed Computing*, 61(1):18–48, 2001.

[124] Ralf Reussner, Peter Sanders, Lutz Prechelt, and Matthias Müller. SKaMPI: A detailed, accurate MPI benchmark. In *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*. Springer, 1998.

[125] DF Richards, O Aaziz, J Cook, S Moore, D Pruitt, and C Vaughan. Quantitative performance assessment of proxy apps and parentsreport for ecp proxy app project milestone adcd-504-9. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2020.

[126] Martin Rinard, Henry Hoffmann, Sasa Misailovic, and Stelios Sidiroglou. Patterns and Statistical Analysis for Understanding Reduced Resource Computing. In *OOPSLA*, 2010.

[127] Behrooz Sangchoolie, Karthik Pattabiraman, and Johan Karlsson. One Bit is (Not) Enough: An Empirical Study of the Impact of Single and Multiple Bit-Flip Errors. In *International Conference on Dependable Systems and Networks*, 2017.

[128] Sriram Sankaran, Jeffrey M Squyres, Brian Barrett, Vishal Sahay, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The lam/mpi checkpoint/restart framework: System-initiated checkpointing. *JHPCA*, 19(4):479–493, 2005.

[129] Piyush Sao and Richard Vuduc. Self-stabilizing iterative solvers. In *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA, 2013.

[130] Siva Kumar Sastry Hari, Radha Venkatagiri, Sarita V. Adve, and Helia Naeimi. GangES: Gang Error Simulation for Hardware Resiliency Evaluation. In *International Symposium on Computer Arch.*, 2014.

[131] Kento Sato, Dong H Ahn, Ignacio Laguna, Gregory L Lee, and Martin Schulz. Clock delta compression for scalable order-replay of non-deterministic parallel applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[132] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: a large-scale field study. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems, SIGMETRICS/Performance (SIGMETRICS)*, 2009.

[133] Sangmin Seo, Gangwon Jo, and Jaejin Lee. Performance characterization of the NAS parallel benchmarks in opencl. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization (IISWC)*, 2011.

[134] Faisal Shahzad, Jonas Thies, Moritz Kreutzer, Thomas Zeiser, Georg Hager, and Gerhard Wellein. Craft: A library for easier application-level checkpoint/restart and automatic fault tolerance. *IEEE Transactions on Parallel and Distributed Systems*, 30(3):501–514, 2018.

[135] M. Shantharam, S. Srinivasmurthy, and P. Raghavan. Characterizing the Impact of Soft Errors on Iterative Methods in Scientific Computing. In *International Conference on Supercomputing (ICS)*, 2011.

[136] Yakun Sophia Shao and David Brooks. ISA-Independent Workload Characterization and its Implications for Specialized Architectures. 2013.

[137] Prateek Sharma, Tian Guo, Xin He, David Irwin, and Prashant Shenoy. Flint: Batch-interactive data-intensive processing on transient servers. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–15, 2016.

[138] V. C. Sharma, A. Haran, Z. Rakamaric, and G. Gopalakrishnan. Towards Formal Approaches to System Resilience. In *Pacific Rim International Symp. on Dependable Computing*, 2013.

[139] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, et al. Addressing failures in exascale computing. *The International Journal of High Performance Computing Applications*, 28(2):129–173, 2014.

[140] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, and Sudhanva Gurumurthi. Mem Errors in Modern Systems: The Good, The Bad, and The Ugly. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[141] Vilas Sridharan and David R. Kaeli. Eliminating Microarchitectural Dependency from Architectural Vulnerability. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2009.

[142] Vilas Sridharan and Dean Liberty. A study of dram failures in the field. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[143] Vilas Sridharan, Jon Stearley, Nathan DeBardeleben, Sean Blanchard, and Sudhanva Gurumurthi. Feng Shui of Supercomputer Memory: Positional Effects in DRAM and SRAM Faults. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

[144] Omer Subasi, Tatiana Martsinkevich, Ferad Zyulkyarov, Osman Unsal, Jesus Labarta, and Franck Cappello. Unified fault-tolerance framework for hybrid task-parallel message-passing applications. *The International Journal of High Performance Computing Applications*, 32(5):641–657, 2018.

[145] Nawrin Sultana, Martin Rüfenacht, Anthony Skjellum, Ignacio Laguna, and Kathryn Mohror. Failure recovery for bulk synchronous applications with mpi stages. *Parallel Computing*, 84:1 – 14, 2019.

[146] Ricardo Taborda and Jacobo Bielak. Large-scale earthquake simulation: computational seismology and complex engineering systems. *Computing in Science & Engineering*, 2011.

[147] Keita Teranishi and Michael A Heroux. Toward local failure local recovery resilience model using mpi-ulfm. In *Proceedings of the 21st european mpi users' group meeting*, page 51, 2014.

[148] Anna Thomas and Karthik Pattabiraman. Llfi: An intermediate code level fault injector for soft computing applications. In *SELSE*, 2013.

[149] Brian Van Essen, Hyojin Kim, Roger Pearce, Kofi Boakye, and Barry Chen. Lbann: Livermore big artificial neural network hpc toolkit. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*, 2015.

[150] R. Venkatagiri, A. Mahmoud, S. K. S. Hari, and S. V. Adve. Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency. In *MICRO*, 2016.

[151] A. Vishnu, H. v. Dam, N. R. Tallent, D. J. Kerbyson, and A. Hoisie. Fault Modeling of Extreme Scale Applications Using Machine Learning. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016.

[152] Zhigang Wang, Lixin Gao, Yu Gu, Yubin Bao, and Ge Yu. A fault-tolerant framework for asynchronous iterative computations in cloud environments. *IEEE Transactions on Parallel and Distributed Systems*, 29(8):1678–1692, 2018.

[153] Jiesheng Wei, Anna Thomas, Guanpeng Li, and Karthik Pattabiraman. Quantifying the Accuracy of High-Level Fault Injection Techniques for Hardware Faults. In *DSN*, 2014.

[154] Kai Wu, Wenqian Dong, Qiang Guan, Nathan DeBardeleben, and Dong Li. Modeling application resilience in large scale parallel execution. In *International Conference on Parallel Processing (ICPP)*, 2018.

[155] Panruo Wu, Chong Ding, and etc. On-line Soft Error Correction in Matrix Multiplication. *J. of Computational Sci.*, 4(6), 2013.

[156] Xin Xu and Man-Lap Li. Understanding soft error propagation using efficient vulnerability-driven fault injection. In *International Conference on Dependable Systems and Networks (DSN)*, 2012.

[157] Ruini Xue, Xuezheng Liu, Ming Wu, Zhenyu Guo, Wenguang Chen, Weimin Zheng, Zheng Zhang, and Geoffrey Voelker. MPIWiz: Subgroup reproducible replay of MPI applications. *ACM Sigplan Notices*, 44(4):251–260, 2009.

[158] Li Yu, Dong Li, Sparsh Mittal, and Jeffrey S. Vetter. Quantitatively Modeling App. Resiliency with Data Vulnerability Factor. In *SC*, 2014.

[159] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 2016.

[160] Junyuan Zeng, Yangchun Fu, Kenneth A. Miller, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Obfuscation resilient binary code reuse through trace-oriented programming. In *Proceedings of the ACM SIGSAC Conference on Computer & Communications Security (CCS)*, 2013.

[161] Xuegong Zhang, Xin Lu, Qian Shi, Xiu-qin Xu, E Leung Hon-chiu, Lyndsay N Harris, James D Iglehart, Alexander Miron, Jun S Liu, and Wing H Wong. Recursive svm feature selection and sample classification for mass-spectrometry and microarray data. *BMC bioinformatics*, 7(1):197, 2006.

[162] G. Zheng, Xiang Ni, and L. V. Kalé. A scalable double in-memory checkpoint and restart scheme towards exascale. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)*, pages 1–6, June 2012.

[163] Gengbin Zheng, Chao Huang, and Laxmikant V. Kalé. Performance evaluation of automatic checkpoint-based fault tolerance for ampi and charm++. *SIGOPS Oper. Syst. Rev.*, 40(2):90–99, April 2006.