Approaches for Building Error Resilient Applications

by

Bo Fang

B.Eng. , Wuhan University, 2006

M.A.Sc., University of British Columbia, 2014

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Electrical and Computer Engineering)

The University of British Columbia

(Vancouver)

March 2020

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

Approaches for Building Error Resilient Applications

submitted by **Bo Fang** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy** in **Electrical and Computer Engineering**.

Examining Committee:

Karthik Pattabiraman, Electrical and Computer Engineering
Supervisor

Matei Ripeanu, Electrical and Computer Engineering
Co-supervisor

Alexandra Fedorova, Electrical and Computer Engineering
University Examiner

Norman Hutchinson, Computer Science
University Examiner

Bianca Schroeder, University of Toronto
External Examiner

Additional Supervisory Committee Members:

Sathish Gopalakrishnan, Electrical and Computer Engineering
Supervisory Committee Member

Abstract

Transient hardware faults have become one of the major concerns affecting the reliability of modern high-performance computing (HPC) systems. They can cause failure outcomes for applications, such as crashes and silent data corruptions (SDCs) (i.e. the application produces an incorrect output). To mitigate the impact of these failures, HPC applications need to adopt fault tolerance techniques.

The most common practices of fault tolerance techniques include: *(i)* characterization techniques, such as fault injection and architectural vulnerability factor (AVF)/program vulnerability factor (PVF) analysis; *(ii)* run-time error detection techniques; and *(iii)* error recovery techniques. However, these approaches have the following shortcomings: *(i)* fault injections are generally time-consuming and lack predictive power, while the AVF/PVF analysis offers low accuracy; *(ii)* prior techniques often do not fully exploit the program's error resilience characteristics; and *(iii)* the application constantly pays a performance/storage overhead.

This dissertation proposes comprehensive approaches to improve the above techniques in terms of effectiveness and efficiency. In particular, this dissertation makes the following contributions:

First, it proposes ePVF, a methodology that distinguishes crash-causing bits from the architecturally correct execution (ACE) bits and obtains a closer estimate of the SDC rate than PVF analysis (by 45% to 67%). To reduce the overall analysis time, it samples representative patterns from ACE bits and obtains a good approximation (less than 1% error) for the overall prediction. This dissertation applies the ePVF methodology to error detection, which leads to a 30% lower SDC rate than well-accepted hot-path instruction duplication.

Second, this dissertation combines the roll-forward recovery and the roll-back

recovery schemes and demonstrates the improvement in the overall efficiency of the C/R with two systems: LetGo (for faults affecting computational components) and BonVoision (for faults affecting DRAM memory). Overall, LetGo is able to elide 62% of the crashes caused by computational faults and convert them to continued execution (out of these 80% result in correct output while a majority of the rest fall back on the traditional roll-back recovery technique). BonVoision is able to continue to completion 30% of the DRAM memory detectable but uncorrectable errors (DUEs).

Lay Summary

Reliability, availability and serviceability (RAS) are important goals for designing high-performance computing (HPC) platforms. Hardware faults, especially when they are transient and lead to independent one-time errors, are considered one of the major sources affecting the RAS of the HPC systems. Applications running on such systems need to take comprehensive actions to deal with the transient hardware faults, including the following: first, the error resilience characterization reveals how serious the impact would be when encountering the faults; then, the error detection techniques allow the applications to inform when the faults cause an erroneous program state. Last, the error recovery techniques rescue the applications from failure. In my thesis, I identify various drawbacks existing in the state-of-the-art solutions and propose advanced approaches that improve the overall effectiveness and efficiency of the current solutions.

Preface

This thesis is the result of work carried out by me, in collaboration with my advisors (Dr. Karthik Pattabiraman and Dr. Matei Ripeanu), my colleagues (Qining Lu, Hassan Halawa), and other research scientists from the Los Alamos National Laboratory, Pacific Northwest National Laboratory and IBM. Chapters 3, 4 and 5 are based on research work published in the conferences of DSN, HPDC and ICS, respectively. For each project, I was responsible for shaping the key ideas, leading the project, designing and implementing the systems, conducting experiments to evaluate the approaches, analyzing the data, and writing the paper.

The following is a list of the publications related to each chapter:

- **Chapter 3** I was the main contributor for the research presented in this chapter. Qining Lu offered feedback on the methodology design and other collaborators were responsible for proofreading the manuscript, providing guidance and feedback.

  Bo Fang, Qining Lu, Karthik Pattabiraman, Matei Ripeanu and Sudhanva Gurumurthi, "ePVF: An Enhanced Program Vulnerability Factor Methodology for Cross-Layer Resilience Analysis",Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2016 (acceptance rate 21%).

- **Chapter 4** I was the main contributor for the research presented in this chapter. Other collaborators provided feedback and made edits in the manuscript.

  Bo Fang, Qiang Guan, Nathan Debardeleben, Karthik Pattabiraman and Matei Ripeanu, "LetGo: A Lightweight Continuous Framework for HPC Applications upon Failures", The ACM International Symposium on High-performance Par-

allel and Distributed Computing (HPDC) June 2017 (acceptance rate 18%).

- **Chapter 5** I was the main contributor for the research presented in this chapter. Hassan Halawa offered feedback in the machine learning aspect. Other collaborators participated in project discussions and helped edit the manuscript.
  Bo Fang, Hassan Halawa, Karthik Pattabiraman, Matei Ripeanu and Sriram Krishnamoorthy, "BonVoision: Leveraging Spatial Data Smoothness for Recovery from Memory Soft Errors", ACM International Conference on Supercomputing (ICS), 2019 (acceptance rate 23%).

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would first to thank my advisors, Dr. Karthik Pattabiraman and Dr. Matei Ripeanu. I would not have reached this point if it weren't for you. I have received from your valuable suggestions, feedback, and guidance along my PhD journey, both academic and for life in general. You inspire me in the pursuit of an academic career, and I hope to become a researcher and the advisor of your level of excellence.

Special thanks goes to Dr. Sathish Gopalakrishnan for his support and mentorship during the CSRG seminar and my days at UBC. I also want to thank all my mentors, collaborators and lab mates. You have helped me in many ways.

Sincere thanks goes to Dr. Bianca Schroeder for providing valuable feedback and suggestions on the dissertation.

I am grateful to the mentors I have had during my internships: Dr. Qiang Guan from Kent State University, Dr. Nathan DeBardeleben from the Los Alamos National Laboratory and Dr. Sriram Krishnamoorthy from the Pacific Northwest National Laboratory.

Finally, I want to thank my parents and the whole family for their unconditional support and love throughout my life. My wife, Yuan Qin; my daughters Emma and Anne; and my son, Edwin—you make my dream come true.

Chapter 1

Introduction

Hardware faults, caused by particle strikes, aging, temperature, manufacturing variations, etc. are one of the major sources of failures that compromise the reliability of the computer systems. As feature sizes shrink, the smaller amount of charge per hardware component makes hardware more vulnerable to the particle strikes, while the reduced cross-section decreases the likelihood of experiencing such strikes. Although these two effects may cancel each other out, more condensed chips, especially those used extensively in high performance computing (HPC) systems, are more prone to experience hardware faults [21, 25, 37, 132]. Handling hardware faults effectively and efficiently is essential to designing computer systems with high levels of **r**eliability, **a**vailability and **s**erviceability (RAS).

1.1 Transient hardware faults

Hardware faults can be classified as permanent, intermittent and transient by their duration; a transient fault exists for a short period of time and is nonrecurring [113], which is different from an intermittent fault (i.e. periodically recurring) and a permanent fault (i.e. stable and continuously occurring). Unlike permanent and intermittent faults, which are likely to have visible symptoms, transient hardware faults occur in a non-deterministic way, and thus it is challenging to predict when and where a transient hardware fault will occur. Due to these properties of transient hardware faults, their handling demands more sophisticated fault tolerance

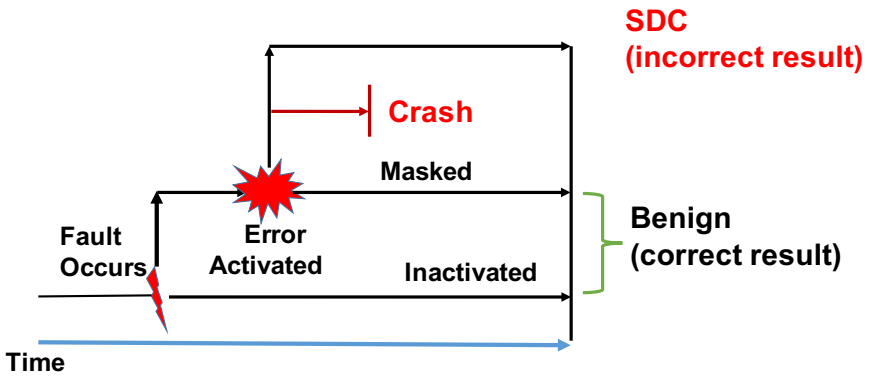


Figure 1.1: Types of failure outcomes for an application.

solutions than simply replacing the failing hardware components.

1.2 Tolerating transient hardware faults with software-based solutions

When a transient hardware **fault** affects an application, it causes an **error(s)** (i.e. deviation) in the program state, and may eventually lead to a **failure** in the application. Two of the most critical failure outcome types are crashes—an application terminates before it finishes, and silent data corruptions (SDCs)—an application produces incorrect output. Other possible outcomes are that the application hangs (which is rare in practice [34, 56, 69]), or the application finishes without any error (i.e. benign). The consequences of crashes and SDCs are serious: crashes result in a waste of computational resources, and SDCs cause the applications to generate wrong results without any warning. Therefore, mitigating the impact of the crashes and SDCs on the applications is the main focus of fault tolerance techniques. Figure 1.1 [70] illustrates the fault-error-failure chain and the failure outcome types for an application.

Hardware-based solutions have been used to counter transient hardware faults. In mission-critical systems such as satellites, spacecraft, aircraft, etc., radiation-hardened components are widely used to tolerate radiation effects. However, in more general-purpose environments, the radiation-hardened devices are not popular due to high costs, manufacturing productivity constraints and degraded perfor-

mance, and hardware-based redundancy techniques are commonly applied. Hardware-based redundancy techniques can leverage either the replication of hardware components (e.g. **t**riple **m**odular **r**edundancy) or redundant information (e.g. code word) to form a collection of methods for error masking, error detection and error correction/recovery tasks (detailed descriptions of these techniques are presented in Section 2.1).

While some hardware-based techniques (e.g. error-correcting code for DRAM) are widely adopted, in general, they face the following problems:

(i) They constantly incur extra energy costs. Hardware-based redundancy techniques consist of either additional hardware resources for redundant hardware components or additional hardware operations such as determining the validity of the code word. Running redundant hardware components or operations requires continuously additional energy consumption, therefore it is not desirable for modern computing systems, which need to carefully take the power constraints into account.

(ii) They may result in inefficient error detection and correction practice. The hardware-based solution normally makes no assumption on the outcome of the application when it is affected by an error, hence the behavior of the solution depends on whether the hardware-implemented error detection criteria have been violated (e.g. the checksum is incorrect) or how the recovery mechanisms can be applied (e.g. applying the checksum to correct an error). However, this solution is rather conservative. As shown by Cho et al. [34], 89.1% of hardware faults originating from the flip-flop level do not reach the software level and thus have no impact on the correctness of the application's execution. That said, the hardware-based solution may waste cycles for transient hardware faults that stay off the application layer.

Compared to hardware-implemented techniques, software-implemented hardware fault tolerance techniques [116] offer a more desirable solution for modern computing systems for the following reasons:

(i) Software-implemented techniques are flexible and effective under various circumstances/configurations;

(ii) They do not require hardware modifications, hence they fit well with HPC systems that are built with general-purpose hardware components;

(iii) Many applications mask an erroneous program state during execution due to their inherent error-masking abilities [56, 58, 75]. Since the software-implemented techniques share the same level of abstraction as the application does, they can efficiently take the advantage of the error-masking abilities of an application to achieve the selectivity in error protection (i.e. only protect a fraction of program states that do not mask errors) and thus reduce the overall costs of fault tolerance techniques.

1.3 The challenges in software-implemented fault tolerance solutions and opportunities for advancements

To design the software-based solutions for transient hardware faults for applications, the following two main aspects need to be taken into consideration: *(i)* what are the outcomes of an application when affected by transient hardware faults?; *(ii)* is it possible to efficiently and effectively mitigate the impact of an error on the application?

These two aspects steer the research of fault tolerance techniques in the following directions: error resilience characterization, error detection and error recovery. In the following section, we describe the main task conducted in each direction and explain why these tasks are essential to the success of the mitigation of the failures caused by transient hardware faults. We then focus on the challenges and the space one can explore in each research direction.

1.3.1 Error resilience characterization

Error resilience is defined as the probability that the application will not a failure outcome (i.e., crash, hang or SDC) after a hardware fault occurs. Error resilience characterization provides a fundamental understanding of how an application reacts to hardware faults. Two mainstream approaches for error resilience characterization are as follows:

Fault injection: This is a procedure to introduce faults in a systematic, controlled manner and study the corresponding system behavior. Fault injection techniques can be generally categorized as hardware-based and software-based [79]. Software-based fault injection techniques typically emulate the effects of hard-

ware faults on the software by perturbing the values of selected data/instructions in the program. Compared to hardware-based techniques such as accelerated testing using neutron beams [61], software-based techniques do not require additional devices and are adjustable to different fault models. Fault injection is a mature and well-understood technique, and there are many tools to help automate the process [5, 27, 56, 134, 144].

AVF/PVF analysis: Mukherjee et al. [109] introduced the architectural vulnerability factor (AVF) analysis, which quantifies the vulnerability of micro-architectural components to errors caused by transient hardware faults. Sridharan et al. [131] study the vulnerability of software independent of hardware by introducing program vulnerability factor (PVF) analysis. Although AVF and PVF measure properties that are different from error resilience, such as the probability that a fault in the particular structure or in a program state will result in an error, they can be used for a first-order estimate of the error resilience of an application.

Challenges: There are challenges associated with both fault injection and AVF/PVF analysis. Fault injection techniques generally face two key challenges. First, fault injection has limited predictive power in terms of determining the impact of code transformations on vulnerability—thus it is challenging to use it for guiding code optimization. Second, fault injection campaigns are typically resource intensive, as thousands of faults need to be injected in complete executions of the program to get statistically significant results, and a standard practice is to inject only one fault per fault injection run for controllability. For HPC applications, this usually entails many hours to finish a single fault injection run, and the cost to get such statistically significant results is unacceptable.

On the other side, the key limitation in AVF/PVF is that AVF and PVF treat all failure outcomes equally; that is, they do not differentiate between crashes, SDCs and the benign outcomes from an application's perspective. This causes those techniques to have significant inaccuracies in the estimate of the error resilience characteristics of an application.

Research opportunity: Error resilience characterization techniques need to consider the trade-off space between the effort required to characterize, and the accuracy of the estimate obtained. The two approaches discussed in this section represent the trade-offs of the two choices—fault injection is accurate but costly,

while the AVF/PVF analysis is fast but much less accurate. My research aims to design an approach that offers low-cost and accurate estimates of the error resilience characteristics for applications.

1.3.2 Error detection

The main goal of error detection through software-implemented techniques is to determine if a transient hardware fault that manifests as an error in the software layer subsequently causes a deviation in the program's behavior before the program finishes.

A commonly-used approach to catch errors caused by transient hardware faults is through duplication [60, 75, 123]. The process is as follows: The detection technique duplicates a portion of the execution of a program and compares the results between the original and the duplicated programs; if the results differ, this means that an error has affected one of the executions. This approach works under two assumptions: i) the piece of the program that has been duplicated is idempotent, meaning that running this piece of the program twice would generate the same results/leads to the same changes to the program state or ii) the probability that the errors affect both programs and produce the same outcomes is low (i.e. the errors caused by transient hardware faults), hence it is unlikely for the error to bias the error detection decision.

Challenges: There are two fundamental challenges associated with duplication-based detection. First, it is computationally costly. The basic strategy potentially provides 100% error detection coverage but costs about 100% performance overhead. This is especially impractical for HPC applications, which usually run for hours or days. Second, duplication-based error detection may cause fault positives. This is because the detection mechanism makes the decision of the existence of an error based on the intermediate results of the original and duplicated execution, while errors may get masked later in the execution and lead to a benign outcome. Therefore, duplication-based detection techniques need to balance between the cost of the duplication and the effectiveness in catching errors. Since different program states affected by errors may have different impacts on the outcome of the application, increasing costs (i.e. duplicating more program states) does not necessarily

indicate a linear increase in error coverage.

Research opportunity: Most state-of-the-art error detection mechanisms do not focus on establishing an indicative connection between the detection strategies and the error resilience characteristics of an application (see detailed discussion in Chapter 2.2.2). Therefore, they cannot be applied extensively in general scenarios. My research takes a different approach; it leverages the error resilience characteristics of an application to guide the duplication-based error detection approach in an application-neutral manner. In particular, with the error resilience characterization technique proposed in Chapter 3, it is possible to obtain a per-program state—a quantitative estimate of how likely an error affecting that state would cause a certain type of failure (e.g. an SDC). When one wants to mitigate the impact of that type of failure for the application, these estimates can indicate the priority of the program states to be selected by the duplication technique: the portion of the program states that have a high correlation with the target failure type would be the candidates to be chosen for duplication.

1.3.3 Error recovery

If the error detection techniques detect and localize the error, they normally signal the run-time and cause a noticeable symptom, such as an OS exception for the application to handle. Immediate termination is a widely acceptable option by the application; however, for many scenarios where the application needs to be robust and provide high availability, terminating the application is not a viable solution. The main task of the error recovery technique is to help the application overcome such symptoms, potentially fix the corrupted program states and, if possible, resume the application's execution.

In general, the error recovery techniques can be classified as "roll-back" and "roll-forward", where the main difference is in their actions upon the symptom: roll-back recovery strategies use the prior information to replace the corrupted program states (i.e. rolling back the application to a previous state(s)), while roll-forward recovery strategies attempt to continue the execution of the application from the failure point. Today, the most commonly used error recovery system is ***checkpoint/restart (C/R)***, which follows the roll-back paradigm—it stores the

essential program states on non-volatile storage periodically (i.e. on every checkpoint interval) as checkpoints and rewinds the application to a “safe" state by loading checkpoints when the application crashes [54, 138]. In the context of HPC systems, C/R is considered as the best practice for recovering applications from fail-stop failures [23, 51].

Challenges: The standard C/R system can incur potentially significant overhead in terms of performance, energy and storage originating from the following two sources: *(i)* the periodically taken checkpoints, and *(ii)* the recovery operations when the failure occurs. In *(i)*, the overhead is determined by the frequency of taking the checkpoint and how fast the necessary program states can be saved to the non-volatile storage per checkpoint; in *(ii)*, the overhead comes from the extra I/O operations to load the checkpoint back to memory and the re-computation of the work from the restored state up to the failure point. There have been numerous techniques for reducing the overall overhead of C/R (discussed in Chapter 2.2.3) via mitigating the costs caused by each of the above aspects.

Research opportunity: In this thesis, we investigate the possibility of introducing roll-forward recovery into the standard C/R systems. The roll-forward recovery strategy implicitly trades off faster time to solution and energy consumption for higher risk of having SDCs in the results compared to the roll-back recovery. My research aims to first characterize such risk, and provide comprehensive solutions for the HPC applications to lower the risk when roll-forwarding is deployed upon failures and hence to potentially reduce the overhead caused by C/R. The roll-forward recovery scheme offers a novel view that fundamentally improves the C/R efficiency, and can potentially work on top of any optimization techniques for roll-back C/R.

1.4 Research questions

As discussed in previous sections, software-based fault tolerance techniques face fundamental challenges. For error resilience characterization, the challenge is to find a reasonable trade-off between the effort required to characterize and the accuracy, such that *(i)* the characterization process is released from the heavy cost caused by fault injection, and *(ii)* the error resilience characteristics of an appli-

cation are estimated more accurately than via AVF/PVF analysis; for error detection, it is not clear how one can correlate the error resilience characteristics of an application with a selective error detection scheme; for error recovery, the existing “roll-back" recovery strategy may cause unnecessary performance overheads, which need to be mitigated. The overall goal of this thesis is to develop comprehensive techniques to address the above issues. To achieve this goal, we are interested in answering the following research questions:

- **RQ1: Can one find a different trade-off point that requires a low effort to characterize and offers acceptable accuracy to a specific end-goal such as SDC protection for an application? Can the obtained information be used to guide the error detection approach, in particular, the selective duplication, to efficiently and effectively detect SDCs?** To avoid the use of costly fault injections, we adapt the PVF methodology to achieve a faster estimate of the resilience characteristics. More precisely, we developed a crash model to predict which faults in a program cause a crash and a propagation model to reason about propagating ranges of crash-causing bits in the program's dependence graph. Together, these two models separate the crash-causing state from the vulnerable program state inferred by the PVF methodology and obtain an accurate estimate on the probability of crash outcomes. By removing the crash-causing state, the rest of the program state represents a more accurate estimate on the SDC-causing state than the PVF methodology. Next, informed by the results of the two models, we implement a selective duplication approach that protects only the program state that is unlikely to cause crashes.

- **RQ2: On a standard checkpoint/restart enabled system, how much performance overhead reduction for an application is obtained by the roll-forward recovery when a crash happens? Does the roll-forward recovery scheme work for all categories of applications?** Depending on the root causes of crashes, we further diversify the research efforts to target different types of fault modes: transient hardware faults manifest in the computational components of CPUs and in memory systems such as DRAM. Therefore, we explore two research directions to achieve a comprehensive answer to the

RQ2:

(i) Recover applications from crashes caused by errors affecting the computational components. As a fundamental modification on the standard “roll-back" strategy, we propose a “roll-forward" C/R system, LetGo, which a) monitors the application at run-time and when a crash-causing error occurs, pauses the program to avoid termination; and b) uses heuristics to attempt to repair the program state and enable continued program execution. To show how much improvement the “roll-forward" scheme brings to the overall efficiency of the C/R system, we model an HPC system in the context of a C/R scheme using a state machine and evaluate the end-to-end performance impact of the “roll-forward" recovery.

(ii) Recover applications from memory errors that are detectable but uncorrectable. We leverage spatial data smoothness preserved in many scientific applications' data space to repair the corrupted memory values that caused memory DUEs. To this end, we employ the assumption that the data that live close to each other in physical space (modeled by a HPC application) are likely to be mapped to the nearby memory locations and compute the repair values for the memory errors based on the neighboring data elements of that corrupted memory location. To alleviate the concern of increasing the SDC proneness of the application due to the estimated repair values, we use spatial data smoothness as the feature and train an online classifier to predict the outcome of a repair for the application before applying it.

1.5 Summary of contributions

This section summarizes the contributions of the dissertation at a high-level. Chapter 3,4 and 5 elaborate on these points and present the individual contributions by answering each RQ in detail.

- *Enabling different failure modes in the PVF analysis.* The PVF analysis treats all obtained bits as vulnerable bits, where it does not consider the actual outcomes caused by errors occurring on these bits. Our approach, ePVF, proposes the crash model and propagation model that identify the

crash-causing bits within the scope of all vulnerable bits. With the separation of the crash-causing bits from the vulnerable bits, ePVF analysis comes to a tighter conservative estimate of the SDC-causing bits.

- *Closing the gap between the analytical model (i.e. PVF) and experimental assessment (i.e. fault injection) on the estimate of error resilience characteristics.* The crash and propagation models together offer the estimate of crash-causing bits with 89% recall and 92% precision, when evaluated over a set of 10 benchmarks. Our analysis narrows the estimated vulnerable bits for the goal of analyzing a program's resilience to SDCs: the number of vulnerable bits estimated by the ePVF analysis is lower than that estimated by the standard PVF analysis by 61% on average. Compared to fault injection results, ePVF offers accurate crash rate estimates over eight benchmarks and a much closer estimate on SDC rate than PVF analysis.

- *Offering the quantitative measure on the SDC-proneness of the program state to improve the selective duplication technique.* We present an ePVF-informed protection heuristic for the selective duplication technique. The heuristic computes the ePVF value as a proxy of the SDC-proneness per instruction and ranks the instructions based on their ePVF values. With duplication on the high-rank instructions for a given performance overhead cap, ePVF does 30% better SDC-coverage than a hot-path-based duplication strategy.

- *A roll-forward recovery implementation that supports HPC applications.* The prototype framework, LetGo, is implemented with the production-level tools that are widely adopted and available in HPC systems. LetGo works directly with the application's executable, hence no modification is needed on the application's compilation procedure. We also extend the LetGo implementation to support MPI applications. We expect that our design choices made for LetGo would accelerate the adoption of LetGo in the HPC community.

- *Demonstrating the successful use of the proposed roll-forward recovery schemes on a diverse set of applications.* Our evaluation shows that LetGo is able to

continue the application's execution in about 62% of the cases and produce 100% correct results when it encounters a crash-causing error (for the remaining 38%, it gives up and the application can be restarted from a checkpoint as before). The increase in the SDC rate (undetected incorrect output) is as low as less than 1% on average. The overall performance overhead incurred by LetGo is trivial.

- *Demonstrating that the spatial data smoothness is preserved in scientific applications.* We hypothesize that for scientific applications, spatial data smoothness in physical space, if it exists, would be observed across the logical data space of the application. We profile the application and find that given a random point in the memory location, the standard deviation calculated across the neighbouring elements of that point is small for all benchmark applications.

- *Proposing a line of approaches that achieve an accurate online prediction over the memory soft errors, with negligible overhead incurred.* Our evaluation shows that the proposed approach, BonVoision, constructs the repairs that result in 0.8-2.5x more benign outcomes, and only marginal increases in SDC outcomes, compared to using 0 and random values for the repairs; and on average BonVoision incurs 6 milliseconds across all benchmarks. Moreover, with the machine-learning guided optimization, BonVoision is able to conduct online predictions on the outcome of each repair, which eliminates most SDCs and other types of failure outcomes.

- *Showing a significant efficiency gain for the application that employs a roll-forward recovery scheme in the standard C/R system for both computational and memory soft errors.* We use the state machine to model the HPC system that uses the standard C/R scheme, and when LetGo and BonVoision are in place, we alter the states according to the semantics of LetGo and BonVoision separately. We simulate the system running for over 10 years and find that overall the standard C/R system with either LetGo or BonVoision obtains a significantly higher estimate on the efficiency than the standard C/R system alone, across a wide range of configurations and applications.

1.6 Scope of the contributions

This section highlights the key assumptions, and the properties of the target applications explored in my thesis to answer each of the following research questions:

- **RQ1**—*Low-cost, high-accuracy error resilience characterization and error detection techniques.* The main assumption made in this part of my research is as follows: During the execution of an application, different program states, if affected by the errors, are responsible for different failure outcomes. Therefore, it is possible to improve the AVF/PVF analysis in terms of achieving significantly more accurate estimates while remaining low-cost by bounding the subsets of program states that lead to different types of failure outcomes. Further, if the target applications exhibit *repetitive computation patterns* that can be leveraged by estimates to prioritize the choices of program state protection for error detection, the program states that have high crash-proneness are not likely to lead to other types of failures such as SDCs.

- **RQ2**—*(i) Recover applications from crashes caused by errors affecting the computational components.* The class of applications that are likely to significantly benefit from the proposed roll-forward recovery technique—LetGo has the two following properties: (a) The applications conduct iterative methods to require the computation to *converge*, thus errors can be masked inherently during the computation. (b) The applications feature application-specific *acceptance checks* to verify the correctness of the computation, which also helps filter erroneous output caused by LetGo, if any.

- **RQ2** *(ii) Recover applications from memory errors that are detectable but uncorrectable.* BonVoision targets a similar class of applications as LetGo does, while exhibiting the following two differences in terms of the expected application's properties: (a) *spatial data smoothness*: the physical phenomena often modeled by the target applications exhibit inherent continuities in their modeled physical space, and further, the data structures used by these applications tend to keep physically close data points close together in memory as well to exploit locality; and (b) the applications using direct methods

Table 1.1: The scope of target applications

Proposed techniques	Target application domain	Main sources	Expected property
ePVF	general scientific applications	Rodinia	repetitive computation † pattern
LetGo	scientific modelling and simulation	DOE mini-app	iterative, convergence, acceptance check
BonVoision	scientific modelling and simulation, linear algebra	DOE mini-app	convergence (relaxed), acceptance check, spatial data smoothness

† A repetitive computation pattern allows the ePVF methodology to use a partial data dependence graph to estimate the error resilience for the entire application state and hence significantly accelerate the modeling.

for computation may also benefit from the recovery schemes proposed by BonVoision.

Table 1.1 summarizes the scope of the target applications and the corresponding properties that are in favor of each of my proposed techniques.

1.7 Dissertation organization

The rest of this dissertation is organized as follows:

- Chapter 2 presents the related work associated with each direction of my research: error resilience characterization, error detection and error recovery.
- Chapter 3 presents the ePVF methodology, an advancement over the state-of-the-art solution, PVF analysis, for a more accurate estimate of an application's error resilience characteristics, and selective duplication informed by the analysis.
- Chapter 4 motivates the demand of the roll-forward recovery C/R scheme, proposes the LetGo framework as a demonstration of such schemes and evaluates its overall effectiveness and efficiency.

- Chapter 5 describes how the spatial data smoothness can be employed to implement the roll-forward recovery C/R systems for HPC applications to mitigate the impact of memory DUEs.
- Chapter 6 concludes the thesis and presents the impact of the dissertation and potential future work.

Chapter 2

Background and Related Work

This chapter presents the background information on hardware fault characterization and tolerance systems and highlights the novelty of this thesis. The chapter is organized as follows: First, we present the prior studies on hardware-based fault characterization and tolerance techniques and explain the benefit of using software-implemented techniques; then we discuss the efforts made in the software-based techniques as follows: *(i)* various techniques proposed by the community aiming to characterize the error resilience of applications; *(ii)* different error detection techniques and the opportunity we seek to connect the error resilience characterization to the error detection techniques and *(iii)* the general failure recovery strategies, with the particular interests in the existing lightweight recovery techniques, the recovery techniques for memory DUEs and, finally, the studies towards more efficient C/R schemes.

2.1 Hardware-based fault injection and tolerance techniques

Hardware-based solutions have been proposed for various purposes in the reliability community over decades. In this section, we present the advantages and disadvantages of hardware-based solutions and motivate the need for software-implemented techniques.

2.1.1 Fault characterization with hardware

Hardware-implemented fault injection relies on extra hardware to introduce faults into the system. As defined in [79], there are two categories of injections—with and without contact of the system:

- With contact: The fault injection tool has direct contact with the target system so that the tool has the control over the design of the injection procedure, such as the frequency of the occurrence of the fault, the location of the fault or what type of fault is modeled. A general way to implement the tools in this category is through the circuit pins [8, 85, 122], while others [33] choose to directly flip the content in a flip-flop.

- Without contact: In contrast, techniques such as accelerated testing with neutron beams [61] issue neutron strikes on the system to mimic the natural physical phenomenon and study how the system reacts.

Hardware-implemented fault injection is commonly used for tasks such as mimicking stuck-at or permanent faults. The software-implemented techniques are not suitable for such tasks because implementing these types of faults in software requires significant efforts and is often impossible. However, if the goal is to understand how a fault propagates during a program's execution, or to evaluate the impact of the fault on the program's behavior, software-implemented techniques are much more desirable. In addition, since the hardware-implemented approaches involve extra hardware components, they are not as flexible as software-implemented techniques.

2.1.2 Hardware-based redundancy

Hardware-based redundancy offers reliable computation or storage at various levels of abstraction. Below, we describe some effective solutions at each level:

- *System level*: The typical approach for system-level redundancy is to employ the extra processor(s) (i.e. the watchdog processor [104]) to validate the results of the main processor. For example, Benso et al. [16] propose a watchdog to check if there is an error during the data exchange between the

main processor and the memory, and to check the control-flow integrity of the execution of a program. Austin [12] proposes a dynamic implementation verification architecture (DIVA) that extends the speculation mechanism of a modern microprocessor to detect errors in the computation of the processor core. On the multi-threading/multi-core architectures, redundant threads/cores are employed to detect transient hardware faults by comparing the results of two threads/cores that contain the same input [111].

- *Circuit level*: Circuit level techniques normally use code words to detect errors, where the complexity of the code word ranges from simple parity bits, error detection and correction codes, residue codes for ALUs [7], etc. As for today's system, the most commonly used technique is ECC. ECC has been an industry standard technology that considerably improves the reliability of the DRAM or SRAM on the system. The examples of ECC schemes include SEC-DED ECCs, chipkill-detect ECCs and chipkill-correct ECCs. SEC-DED ECCs support the correction of single-bit errors and the detection of double-bit errors. As an advanced version, chipkill-detect and chipkill-correct ECCs employ RAID (Redundant Arrays of Inexpensive Disks)-like DRAM architectures, and such a scheme is used to calculate the ECC checksum for the contents of the entire set of chips for each memory access. The chipkill ECC then stores the result in extra memory space on the protected DIMM, and when a memory chip on the DIMM fails, the RAID result is used to reload the lost data.

The hardware-based redundancy approaches require additional hardware components and the support of extra hardware logic for fault detection and recovery. Compared to software-implemented approaches, they incur a fixed cost in area and complexity regardless of whether the application uses them.

2.2 Software-implemented characterization and fault tolerance techniques

In this section, we investigate the existing software-implemented characterization and fault tolerance techniques and explain why the approaches proposed in this

thesis have the potential for improvement.

2.2.1 Error resilience characterization via vulnerability analysis

There has been a considerable amount of work on estimating the error resilience of a program either through fault injection or through vulnerability analysis techniques. The main advantage of fault-injection is that it is simple and allows distinguishing between different failure outcomes, yet it has limited predictive power and is slow. Alternatively, the main advantage of vulnerability analysis is that it has predictive power and is faster, but it does not distinguish between different kinds of failures.

Biswas et al. [20] separate the overall AVF of processor structures into SDC AVF and Detected Unrecoverable Errors (DUE) AVF by considering whether bit-level error protection mechanisms such as ECC or parity are enabled in those structures. While DUE is similar to the notion of the crash and causes a fail-stop symptom, it is defined at the hardware level only and does not consider software-level mechanisms. Furthermore, like AVF, DUE-AVF is highly hardware dependent.

Wang et al. [142] compare the ACE analysis estimation of the AVF on a processor component with the result of a fault injection analysis and find that ACE analysis is significantly conservative. However, they do not attempt to improve the ACE analysis methodology.

Bronovetsky et al. [24] use standard machine learning algorithms to predict the vulnerability profiles of different routines under soft errors to understand the vulnerability of the full applications. However, their technique is confined to linear algebra applications. Lu et al. [102] and Laguna et al. [92] identify SDC-causing code regions through a combination of static analysis and machine learning. However, their technique does not provide a sound understanding of why some faults cause SDCs and others do not. A common issue with machine learning techniques is that they require extensive training with representative data, which analytic techniques do not.

Finally, Yu et al. [152] introduce a novel resilience metric called the *data vulnerability factor* (DVF) to quantify the vulnerability of individual data structures. By combining the DVF of different data structures, the vulnerability of an applica-

tion can be evaluated. While useful, this technique requires the program to be written in a domain-specific language, which is restricted in terms of its expressiveness. Further, DVF does not distinguish between crash-causing errors and other errors.

The main question we ask in this thesis is whether it is possible to combine the advantages of the two approaches by building an architecture-neutral vulnerability analysis technique to distinguish different failure outcomes, and especially SDCs. Therefore, we use fault injection to gather the ground truth of the error resilience characteristics of an application and compare it with the result of the ePVF methodology.

2.2.2 Error detection techniques

There has been a significant amount of prior work in the design of efficient error detection mechanisms and the main direction towards this goal is through selective duplication. For example, SWIFT [123] removes the need for replicating the memory subsystem when detecting errors and hence improves the overall performance of error detection. Hari et al. [75] present a low-cost, program-level fault detection mechanism for reducing SDCs in applications. They use their own prior work, Relyzer [76], to profile applications and select a small portion of the program fault site to identify static instructions that are responsible for SDCs. By placing program-level error detectors on those SDC-causing sections, they can achieve high SDC coverage at a low cost. However, application-specific behaviours are major contributors of SDCs for half of their benchmarks, which makes it difficult to extend their technique to other applications.

Shoestring [60] defines “high value" instructions (instructions which, when affected by errors, are likely to cause SDCs), and selectively duplicates those instructions. In this work, any instructions that can potentially impact global memory are considered high-value, and any instructions that can produce arguments passed to function calls are also high-value. The problem with these heuristics is that they are coarse in terms of classifying the instructions with their SDC-proneness. Lu et al. [102] propose an empirical model to predict the SDC-proneness of a program's data. The proposed framework, SDCTune, is based on static and dynamic features of the program and achieves high accuracy in predicting the relative SDC prone-

ness of applications, which limits its usefulness in identifying the most SDC-prone instructions for selective duplication. Recently, Li et al. [96] proposed TRIDENT, which is a compiler-based approach that models the error propagation and predicts both the overall SDC probability of a given program and the SDC-proneness of individual instructions of programs, without fault injections. The resultant estimate of the SDC probability, however, is flickering around the estimate obtained from the fault injection across various applications, which limits its usefulness to guide the design of the error resilience techniques.

There are other detection approaches taking advantage of the parallelism of an application. For example, Saxena et al. [128] were the first to use redundant threads to detect and tolerate errors in a multi-threading environment, which relies on full duplication (of a thread). Wei [143] take advantage of the similarities between parallel processes of a program to detect errors in the program's control data. Selective duplication techniques can be integrated with these approaches to further improve the coverage of errors.

In this thesis, we focus on improving the selective duplication techniques for the following aspects: *(i)* the generality of the approach, hence the techniques do not need knowledge of the application-specific behavior, and *(ii)* the explainability of the approach, for building an explicit and quantitative correlation between the chosen instructions for duplication and the SDC-proneness of the program.

2.2.3 Failure recovery strategies

Overview

There have been many efforts to provide comprehensive solutions for HPC systems to recover from failures. Recovery strategies can be grouped along the following two dimensions: first, on whether they are application-aware or application-agnostic, and second, on whether they roll-back to a previously correct state or use heuristics to attempt to repair the state and roll forward. An example of the application-agnostic approach is the global view resilience (GVR) [52] framework, which allows applications to store and compute an approximation from the current and versioned application data for forward recovery. Aupy et al. [11] dis-

cuss how to combine the silent error detection and checkpointing and schedule the checkpointing and verification in an optimal-balanced way with a rollback recovery. Other approaches rely on a detailed understanding of the application. Gamel et al. [62] design and implement a local recovery scheme specialized for stencil-based applications for a fast roll-back at the minimum scale, while algorithm-based fault tolerance (ABFT) techniques, such as [43, 148], compute checksums for the intermediate states of the LU factorization problems and enable forward recovery. This thesis aims to provide an application-agnostic and forward recovery solution in the same vein as GVR [52]. However, it is more general and efficient as there is no need to store previous program states and no need for any data structure or interface support.

Lightweight recovery techniques

The work proposed in this thesis, LetGo, is inspired by the following two key areas: *failure-oblivious computing*, which focuses on recovering from failures caused by software bugs [125, 126], and *approximate computing*, which makes the assumption that neglecting some errors during application execution will still lead to producing acceptable results. We discuss below how LetGo relates to these areas and highlight the differences.

Failure-oblivious computing: Rinard et al. [125, 126] propose failure oblivious computing, an approach that continues application execution when memory-related errors occur during execution. They introduce an abstraction called boundless memory block: when there is an out-of-bound write, the written value is stored in a hash table indexed by its memory location, then for out-of-bound reads it retrieves the value from the hash table if the same memory address is used (or uses a default value if the hash table has not been initialized for that value). This is enabled by compile-time instrumentation and checks for all memory accesses at runtime. There are two major problems in their approach, as follows: *(i)* the technique above focuses only on out-of-bound memory accesses, a subset of sources of crashes in HPC systems, and *(ii)* their approach does not focus on HPC applications, where we believe such techniques are likely to have a high impact.

Recently, Long et al. [101] proposed a run-time repair and containment in the

same style as the original failure-oblivious computing work. They expand the solution used to drop assumptions on application structure and impose no instrumentation on a program during execution. This technique works for errors including divide-by-zero and null pointer de-referencing. Both Rinard et al. [126] and Long et al. [101] find that one of the main reasons to the success to their techniques is the common computational pattern that occurs in all of their benchmark applications—that the input of the applications can be divided into units, and there is no interaction between computations on different input units. This is, however, not a typical computation model for HPC applications.

Failure escaping: Carbin et al. [26] designed Jolt, a system that detects when an application has entered an infinite loop and allows the program to escape from the loop and continue. Jolt has a similar philosophy as our proposed approach (i.e. LetGo): adjusting the program state when a program appears to be trapped in a failing state. However, Jolt is designed only to help programs escape from an infinite loop (a hang), a relatively infrequent failure scenario, as our fault-injection experiments indicate.

Approximate computing [72, 108]: This starts with the observation that, in many scenarios, an approximate result is sufficient for the purpose of the task, hence energy and performance gains may be achieved when relaxing the precision constraints. The philosophy of approximate computing is applied to different system levels, including circuit design [71, 149], architectural design [66, 120] and application characterization and kernel design [32]. This concept, along with the related body of research such as probabilistic computing [30, 117], offers potential platforms for applications that can tolerate imprecision in the final answer. Miguel et al. [107] propose the Bunker Cache, a design that maps similar data to the same cache storage location for better cache efficiency while degrading the quality of the computation. However, approximate computing, or probabilistic computing, aggressively relaxes the accuracy of the computation, which is a different philosophy from that of our approach.

Recovery techniques for memory DUEs

Levy et al. [94] propose a methodology that leverages the data similarity between memory pages to perform compression on memory and use the compressed memory pages to repair the ones that are impacted by DUEs. At a high level, their goal is similar to our goal, while two main limitations exist: *(i)*, as stated in their paper, the memory compression may not always be a viable operation, as they can only protect pages that are read-only at the beginning of each protection interval; *(ii)* when a DUE occurs, and the corrupted page is eligible to recover, they essentially roll that corrupted page back to its previous state, which can cause a cascading recovery.

Gottscho et al. [64, 65] propose a software-defined ECC technique (SDECC) that uses side information to estimate the original message by filtering and ranking possible candidate code words. The SDECC technique is shown to be able to successfully recover most DUEs for integer applications. Along this direction, Schoeny et al. [129] focus on the coding side of SDECC and a priori designate specific messages to protect the system from errors with stronger confidence. There are two problems involved, as follows: first, both papers assume that the ECC syndrome is available for investigation, while the availability of the syndrome bits is limited in reality; second, both papers need hardware-level support and modifications, which are not possible for most HPC systems. Poulos et al. [53] leverage both the candidate code-words and the application-specific contextual information to attempt to repair DUEs without hardware modification. However, they require application-specific information to guide the selection of the code word, which makes the approach less general.

Improving C/R efficiency

The C/R systems can incur high overheads for HPC applications. Mitigating these overheads depends on several factors, including the algorithm that determines the frequency, the content/size of the program state that needs to be stored and the actual implementation across the whole spectrum of the system. Below, we place emphasis on the research directions towards the improvement of the C/R system and explain the potential opportunities for better efficiency of the C/R system.

Optimize the checkpoint interval: Young et al. [151] and Daly et al. [42] came up with analytic models that aim to find the optimal theoretical checkpoint interval. EI-Sayed et al. [50] show that checkpointing under Young's formula achieves a performance almost identical to that of more sophisticated schemes, based on exhaustive observations on the production systems. Wang et al. [139] model the coordinated checkpointing for large scale supercomputers and find that there is an optimum number of processors for which total useful work is maximized.

Shorten the checkpoint and restart time: Unlike the traditional checkpoint/restart systems that store the checkpoints on disks, several virtual machine checkpointing techniques, including those of Remus [40] and Wang et al. [140], save checkpoints in the memory, which achieves an orders of magnitude reduction in the checkpointing and restart time for VMs.

Shrink the size of the checkpoint: TICK (Transparent Incremental Checkpointer at Kernel Level) [63] uses kernel threads to implement the system-level incremental checkpoint for serial applications, which significantly shrinks the size of the checkpoint. Moreover, Wang [137] proposes a C/R system that transparently supports both full and incremental checkpoints for a parallel application (i.e. MPI-based).

The above discussed techniques, however, are designed for C/R schemes that exercise the roll-back recovery strategies. Our optimization, which allows the application to roll-forward its execution upon failures, would yield potential benefits on the overall efficiency of the C/R system, under the following hypotheses *(i)* as the application crashes less often, the mean time between failures (i.e. MTBF) of an application becomes longer. According to the analytical models [42, 151], the longer MTBF would lead to an increased checkpoint interval, thus the application can take fewer checkpoints during its execution; *(ii)* since the application directly continues under the roll-forward scheme, it does not need to pay the overhead of rolling back to a previous checkpoint or restarting upon a failure.

Today's large HPC systems have an MTBF of tens of hours [78], even with hardware- and software-based protection techniques employed together. The failure rate is expected to further increase, which mandates that a larger portion of computation cycles are used for fault tolerance [45].

Chapter 3

ePVF: An Enhanced Program Vulnerability Factor Methodology for Cross-layer Resilience Analysis

This chapter answers the RQ1 and presents my work on first predicting the error resilience characteristics of an application, and use the prediction results to guide the design of selective duplication systems. We propose ePVF, an enhancement of the original PVF methodology, which filters out the crash-causing bits from the ACE bits identified by the traditional PVF analysis. The ePVF methodology consists of an error propagation model that reasons about how the error propagates in the program, and a crash model that encapsulates the platform-specific characteristics for handling hardware exceptions. ePVF reduces the vulnerable bits estimated by the original PVF analysis by between 45% and 67% depending on the benchmark, and has high accuracy (89% recall, 92% precision) in identifying the crash-causing bits. We demonstrate the utility of ePVF by using it to inform selective protection of the most SDC-prone instructions in a program.

3.1 Introduction

As discussed in Chapter 1, a transient hardware fault can affect an application and cause an SDC or a crash failure. For certain domains like physics simulation, applications validate the computational result to filter out obvious deviations, while such validation does not offer the guarantee of no data corruption. Moreover, there is no generic method to detect SDCs without re-executing the entire application and checking for a mismatch, or without a significant amount of hardware redundancy, both of which are expensive. Users will typically trust the application's output in the absence of an error indication, which makes SDCs one of the major concerns.

One of my research goals is to develop a systematic method to inform the design of software protection techniques (e.g., code transformations) to make applications resilient to SDCs. A first and essential step towards this goal is estimating the SDC rates of programs. As SDCs are caused by a combination of application-specific and system-specific factors, in this chapter, we focus on the system-specific factors that lead to SDCs. The main insight underlying this part of the thesis is that a fault that leads to a crash cannot (by definition) lead to an SDC. Because crashes are caused by a combination of the hardware and Operating System (OS) features, they can be systematically reasoned about in an application-independent manner. By removing the crash-causing faults from the set of all faults, one can obtain a tighter estimate of the SDC rate. This is as important as crashes are often the dominant failure outcome, and hence significantly outnumber both SDCs and hangs [1, 10, 67, 150].

We propose a new method, ePVF (enhanced PVF), that builds on the original Program Vulnerability Factor (PVF) analysis methodology proposed by Sridharan et al. [131] to remove crash-causing faults from the set of all faults. PVF is a systematic method to efficiently evaluate the error resilience of software under hardware faults. PVF can also be used for predictive and comparative analysis studies to understand the effect of different protection techniques or code transformations on the error resilience. However, PVF does not distinguish between fault outcomes and, essentially, treats crashes, SDCs and hangs as equally severe. Therefore, using PVF to estimate application error resilience and inform the protection mechanisms often leads to overprotecting applications, thereby resulting in unnec-

essary performance and energy overheads. By distinguishing between crashes and other failures, ePVF allows protection techniques to better focus on the program's bits that if corrupted, can potentially cause SDCs.

There are two challenges in identifying crash-causing bits. Firstly, crashes are caused by OS and architecture-specific factors, which we need to understand and model. Secondly, the crash-related OS state varies during program execution (e.g., segment boundaries for segmentation faults). Hence we need a dynamic model for predicting whether a particular fault will cause a crash. We find that the majority of crashes are caused by illegal memory addressing, and that by capturing and reasoning about the state of a program's memory segments in a platform-specific manner, we can accurately find almost all crash-causing bits. We therefore extend the original PVF estimation to estimate the ranges of values that may generate crashes, propagate them on the backward slices of the loads and stores, and efficiently compute the set of bits that can result in program crashes.

Contributions. To the best of our knowledge, our work is the first to consider the effects of different failure modes through a PVF-like analysis for the goal of analyzing a program's resilience to SDCs. Our work is also the first to close the gap between analytical models such as PVF, and experimental assessment techniques such as fault injections. This chapter:

1. Develops a *crash model* (§3.3.4) to predict which faults in a program cause a crash, a *propagation model* (§3.3.3) to reason about propagating ranges of crash-causing bits in the program's dependence graph, and integrates them with the PVF methodology;

2. Implements the method in the LLVM compiler [93] and its intermediate representation (IR) which offers the ability to support multiple platforms and architectures;

3. Evaluates the accuracy of the proposed ePVF method vis-a-vis fault injection (§3.4) at the same abstraction level using LLFI, an open-source fault injector [144]. It finds that ePVF estimates crash-causing bits with 89% recall and 92% precision, when evaluated over a set of ten benchmarks. More importantly, the number of vulnerable bits estimated by the ePVF analysis is

lower than that estimated by the standard PVF analysis by 61% on average. Thus ePVF leads to a tighter estimate of the SDC rate, and a close estimate of the crash rate compared to fault injection.

4. Demonstrates the utility of the ePVF analysis through a case study involving selective instruction-level protection for SDC mitigation (§3.5). We find that the SDC rate reduction achieved using ePVF is, on average, 30% better than that achieved by hot-path duplication (i.e., duplicating the most frequently executed program paths), for the same performance overhead.

3.2 Background

This section offers background information on error resilience, the dependability metric we estimate (§3.2.1), past work on estimating it through fault-injection (§3.2.2) and vulnerability analysis techniques (§3.2.3), the abstraction level that our technique works at (§3.2.4) and our fault model (§3.2.5).

3.2.1 Dependability metric: error resilience

Not all faults in a program result in failures due to masking at different layers of the system stack. As we focus on software resilience techniques, we do not consider hardware masking [105], but only take into account faults passing the hardware and seen by the software. This is in line with other work in this area [44, 60, 86, 123].

In the context of this work, *we define error resilience as the probability that the application does not have an SDC after a transient hardware fault occurs and impacts the application state.* Note that error resilience does not take into account the probability of a fault occurring and affecting the software (which depends on the base fault rate in the hardware and the application execution time). In §3.5, we estimate the impact of protection techniques by taking into account their effect on application performance in addition to the resilience.

3.2.2 Effort in accelerating fault injection

Hari et al. [73, 76] have proposed approaches to reduce the cost of fault injection campaigns by pruning the fault injection space. However, fault injection campaigns

are still costly and cannot be used in situations where predictive power is needed to choose between the multiple options available for code optimization. Instead, we need an automated characterization of error resilience that does not use fault injections.

3.2.3 Program vulnerability factor (PVF)

The AVF of a hardware component is the probability that a fault occurring in the component leads to a visible error in the final output of a set of executed instructions. Mukherjee et al. [110] introduced the Architecturally Correct Execution (ACE) analysis for estimating the AVF of processor structures (e.g., Reorder buffer) based on a dynamic execution trace on a specific microarchitecture. By combining ACE analysis with the raw error rate of a processor structure and its AVF, microprocessor designers can estimate the FIT (Failures In Time) of each processor structure and take appropriate action in the design stage. However, AVF is intricately tied to the microarchitectural design of a processor, and cannot be used to reason about software resilience in isolation.

Sridharan et al. [131] separate the hardware-specific component of AVF from the software-specific component: the Program Vulnerability Factor (PVF). They show that the PVF can be used to explain the error resilience behaviour of a program independent of the processor. Moreover, they show that by using techniques that are used in computing the PVF [131], programmers are able to pinpoint the vulnerability of different segments of the program, and gain insights for designing application-specific fault tolerance mechanisms. However, the key drawback of PVF is that it does not distinguish between different kinds of failures, i.e. crashes and SDCs.

PVF abstracts out timing information and includes only the relative instruction order in the instruction flow. This makes the process of computing the PVF largely microarchitecture neutral, thus making it a function of the program and the architecture alone (when executed with a specific input). Sridharan et al. [131] estimate the PVF of an architectural resource 'R' as the ratio between the Architecturally Correct Execution (ACE) bits in the resource when executing the set of instructions I in a trace and the number of total bits involved in R (i.e., B_R) (Equation 3.1). The

ACE bits are the bits in which a fault would potentially affect the correctness of the execution of the instructions in I.

$$PVF_R = \frac{\sum_{i=0}^{I} (ACE\ bits\ in\ R\ at\ instruction\ i)}{B_R \times |I|} \tag{3.1}$$

3.2.4 LLVM IR

LLVM is a compiler infrastructure that provides support for different hardware platforms [93]. The key component of LLVM is its intermediate representation (IR), an assembly-like language that abstracts out the hardware and ISA-specific information. Our methodology is built on LLVM IR level, and we choose to work at this abstraction level for the following reasons:

i) LLVM abstraction level (i.e., LLVM IR) offers an uniform representation of a program; hence the ePVF methodology is architecture neutral and easy to port across different architectures/ISAs, thereby eliminating the influence of architecture or ISA-specific factors.

ii) LLVM IR maps closely to constructs of a program and preserves source-level program properties, which makes it easy to help understand the inherent fault masking.

iii) LLVM infrastructure provides extensive support for program analysis and instrumentation. Prior work focusing on selective duplication techniques [60, 123] uses the LLVM compiler for both static and dynamic analysis, and program instrumentation.

To validate our methodology, we use fault injection experiments at the same abstraction level (LLVM IR) as our ePVF implementation using LLFI [144]. Cho et al. [33] have found that high-level fault injections can directly model a subset of system-level behaviors caused by transient faults. However, our focus (and the focus of the original PVF paper) is on the subset of faults that do manifest in architecturally visible state (e.g., registers) - these faults can be modeled by high-level fault injections.

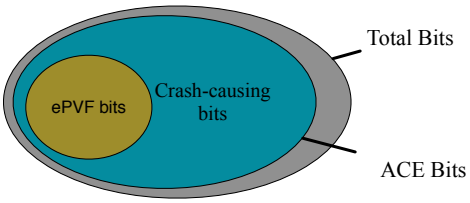


Figure 3.1: Venn diagram that highlights the crash-causing and the ePVF bits that the ePVF methodology identifies as a subset of ACE bits. SDC-causing bits will be a subset of the ePVF bits.

3.2.5 The fault model

For this part of the dissertation, we consider transient faults that occur within the processor (i.e., register file, ALUs). We use the single-bit-flip model to represent transient faults as in other related work [60, 67, 123, 143, 150]. Our technique can be easily extended to multiple-bit flips. In recent work, Cho et al. [33] find that low-level hardware faults manifest as both single- and multiple-bit flips at the application level. However, recent work [13, 103] has shown that the difference between single- and multiple-bit flips occurring in program states is marginal in terms of their impact on SDCs. Therefore for this study, we stick to single bit flips, as SDCs are our main concern.

3.3 ePVF methodology

Our goal is to obtain a comprehensive estimate of the program resilience that does not entail the full-blown costs of fault injection. We aim to adapt the PVF methodology (see §3.2) for this purpose. As pointed out earlier, PVF does not distinguish between crashes and SDCs, and hence is overly conservative, as SDCs are the main concern in practice.

Definition: We define ePVF by analogy to PVF as the ratio of *non-crashing* ACE bits over the total bits involved. Figure 3.1 shows the ePVF bits: they are a subset of all ACE-bits, and a superset of the SDC-causing bits. It is a superset because not all non-crashing bits cause SDCs.

$$ePVF_R = \frac{\sum_{i=0}^{I} (ACEBits \text{ - } CrashBits \text{ in } R \text{ at instruction } i)}{B_R \times |I|} \tag{3.2}$$

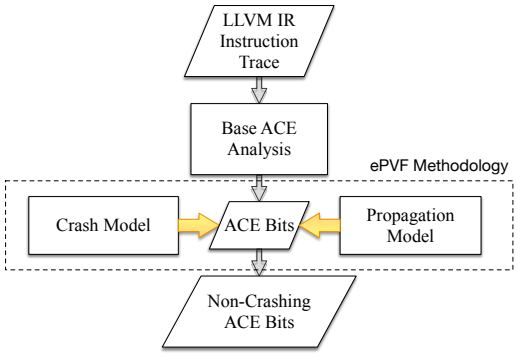


Figure 3.2: The overall workflow of the ePVF methodology to compute the non-crashing ACE (ePVF) bits.

Methodology overview: At a high level, the ePVF methodology consists of three major components (Figure 3.2): (i) Base ACE analysis to estimate all the vulnerable bits of the program (§3.3.1); (ii) a *crash model* to identify the ranges of bit-level faults that cause crashes (§3.3.4); and (iii) a *propagation model* that propagates these ranges along the backward slices of each operand (§3.3.3). This methodology is supported by our identification of incorrect memory addressing as the most common cause for crashes (§3.3.2).

3.3.1 Base ACE analysis

ACE analysis is used to determine the set of all bits in an architectural resource (e.g., a register file) that are not masked and can affect application's final state. The basic idea is to first identify the instructions that are responsible for the output of the program (called output instructions), and then find all the instructions in their backward slice. We use the program's dynamic dependency graph (DDG) [4] to keep track of the data dependencies among the program's instructions. The DDG is a representation of data flow in the program, and is constructed based on the program's dynamic instruction trace [4]. In the DDG, a vertex can be a register, a memory address or even a constant value. An edge records the instruction (i.e., an operation) and links source operand(s) to destination operand(s).

We implement the DDG analysis at the LLVM compiler's intermediate representation (IR) level. Note that since the LLVM IR abstracts out the hardware/ISA-

specific information, it contains an infinite number of virtual registers [1]. As a result, at this level there is no notion of a register file. We model the architectural resource as the set of virtual registers used in the IR of a program. This definition also matches our fault injection experiments as only activated faults are considered.

One further issue is deciding how to express register-based memory addressing instructions. This needs special care as it is common to have the same register to store many different memory addresses, or different registers store to the same address. Since we create new DDG nodes for each newly written memory address, it is also common for a register to store multiple uses of the same memory address. To handle these cases, we create an edge in the DDG to link the memory address used and the register. This edge is *virtual* to differentiate this case from direct data data dependencies.

Running example: Figure 3.3 shows a small portion of the DDG constructed from a dynamic IR instruction trace of the *pathfinder* benchmark [29]. We rename the IR registers for readability. Figure 3.3a presents the corresponding static instruction in the LLVM IR of the program[2]. Figure 3.3b illustrates the DDG obtained after executing the static instructions in Figure 3.3a. Nodes representing memory are labelled with the address values recorded during the run-time. Memory addresses that correspond to the output are highlighted.

From each memory location that is part of the output, in this case *0x15FB174* (highlighted in Figure 3.3b), we run a reverse breadth-first search on the DDG that contains all the dependent vertices of *0x15FB174*. This step will exclude the node *r8* as it does not contribute to the output. We call the resulting graph the ACE graph (Figure 3.3c). Then the total ACE bits are calculated as (the size of each operand is defined in IR):

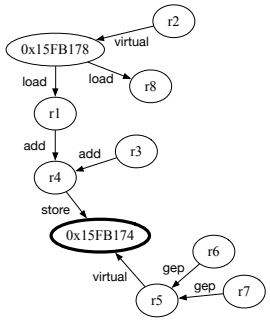
$$ACE\ Bits_{used\ registers} = \sum_{i=1}^{7} Bits\ in\ R_i$$
$$= 32 + 64 + 32 + 32 + 64 + 64 + 64 = 352$$

[1]The register allocator will take the physical register file size into account when mapping the virtual registers to physical ones in the compiler backend.

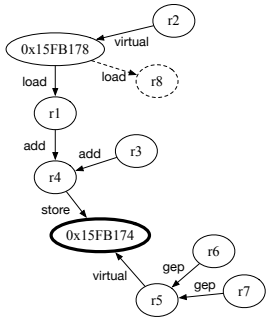
[2]LLVM has a special IR instruction named *getelementptr* (gep) to abstract memory address computations, which corresponds to a combination of several instructions in a assembly language such as MOV and ADD instructions.

```
r1 = load i32* r2, align 4
r4 = add nsw i32 r1, r3
r5 = getelementptr inbounds i32* r6, i64 r7
store i32 r4, i32* r5, align 4
r8= load i32* r2, align 4
```

a A small portion of static IR-level representation of the pathfinder benchmark



b The DDG constructed based on the dynamic trace resulting from executing the code presented in (a).



c The ACE graph used to capture the ACE bits. Note that dynamically dead code is eliminated.

Figure 3.3: An example of computing PVF for the register file

To compute the PVF we also need to compute the total bits for used registers, summing the total bits used in operations within this sequence of instructions.

$$Total\ Bits_{used\ registers} = \sum_{i=1}^{8} Bits\ in\ R_i$$
$$= 32 + 64 + 32 + 32 + 64 + 64 + 64 + 64 = 416$$

Then, the PVF of used registers for this example is:

$$PVF_{used\ registers} = \frac{ACEBits_{used\ registers}}{TotalBits_{used\ registers}} = 0.846$$

3.3.2 Finding the crash-causing bits

We aim to identify the bits that cause the program to crash (i.e., lead to hardware exceptions), and subtract these bits from the overall ACE bits. To this end, it is important to determine the types of crashes we observe in practice and their relative frequencies. We perform a fault injection experiment by injecting faults into ten benchmark applications (§3.4 describes our fault injection methodology and benchmarks).

We observe four types of exceptions resulting in crashes (Table 3.1). Table 3.2 shows their relative frequencies. Our results show that *segmentation faults are the predominant source of crashes with a 99% average frequency and a 96% minimum over all benchmarks.* This observation suggests that, for the class of workloads with similar properties as these benchmarks, we only need to model the mechanisms that generate segmentation faults to identify almost all crash-causing bits. We note that other workloads, architectures, or operating systems may change these precise findings, but a similar methodology can be followed.

Segmentation faults result from memory access violations. Although different operating systems may implement violation detection mechanisms in different ways, segmentation faults are determined based on checking memory accesses against segment boundaries.

There are two main challenges in determining which bit flips would lead to a segmentation fault: first, we need to find the ranges of the bits that, if flipped, would

Table 3.1: Types of exceptions resulting in crashes

Type	Description
Segmentation fault (SF)	Memory access that exceeds the legal boundary of a memory segment
Abort (A)	Programs aborted by themselves or OS
Misaligned memory access (MMA)	Memory accesses are not aligned at four bytes
Arithmetic errors (AE)	Divided by 0, Overflow, Floating point error, etc

Table 3.2: Relative crash frequency analysis for each benchmarks

Benchmark	Types of crashes (%)			
	SF	A	MMA	AE
hotspot	97.6%	0.0%	2.3%	0.1%
bfs	98.8%	0.0%	0.7%	0.5%
lavaMD	99.5%	0.0%	0.0%	0.5%
nw	99.6%	0.0%	0.4%	0.0%
pathfinder	99.9%	0.1%	0.0%	0.0%
lud	100.0%	0.0%	0.0%	0.0%
srad	96.0%	0.0%	4.0%	0.0%
mm	99.8%	0.1%	0.1%	0.0%
particlefilter	100.0%	0.0%	0.0%	0.0%
lulesh	99.0%	1.0%	0.0%	0.0%

result in an out-of-bounds memory access. This includes both faults in the memory instructions themselves (i.e., load and store), and faults in their backward slices used for memory addressing. Second, we need to predict if an incorrect memory access will generate an access violation. To this end, all segment boundaries at the time of the memory access need to be known.

To overcome the first challenge, we implement an algorithm that propagates the ranges of crash-causing bits along the backward slice of the memory access operation (§3.3.3). To overcome the second challenge, we instrument the program to embed a probe for each memory access and capture all the dynamic segment boundaries (§3.3.4).

3.3.3 The propagation model

We model fault propagation for crash-causing faults starting from a memory addressing operation and going backwards in the DDG. This analysis is triggered each time a load/store instruction is encountered during the iteration over the ACE graph (i.e., the subgraph that contains all ACE nodes in the DDG) to compute ACE bits. The aim is to find all bits that can generate an out-of-bound address on the backward slice of the memory address calculation. The model assumes that only one fault happens during the course of a program. (§3.2.5).

The propagation model consists of two algorithms. Algorithm 1 describes when and how the propagation model is triggered. It consists of two procedures: ITERATE_OVER_ACE_GRAPH and CRASH_CALC. The ITERATE_OVER_ACE_GRAPH procedure takes the ACE graph as input and iterates over the vertices in the ACE graph. When it reaches a load/store instruction (line 3), the backward slice for the address used in the load/store instruction is calculated (line 5) and passed to the procedure CRASH_CALC (line 6). Inside CRASH_CALC, all the instructions along the backward slice are visited and, for each instruction, the ranges for crash-causing bits in operands are computed by invoking GET_RANGE_FOR_CRASH_BITS.

Algorithm 1 Iterates over the ACE graph and invokes CRASH_CALC whenever a load or store instruction is encountered

```
1: procedure ITERATE_OVER_ACE_GRAPH(ACEGraph)
2:     for all inst in ACE Graph do
3:         if inst.opcode == load/store then
4:             backwardslice =
5:             CALCULATE_BACKWARD_SLICE(inst)
6:             CRASH_CALC(backwardslice)
7: procedure CRASH_CALC(backwardslice)
8:     for all inst in backwardslice do
9:         GET_RANGE_FOR_CRASH_BITS(inst)
```

The second algorithm (Algorithm 2) consists of the procedure GET_RANGE_FOR_CRASH_BITS that models the execution of each instruction along the backward slice to calculate the range for crash-causing bits. Specifically, for a load/store instruction, the crash model is invoked (CHECK_BOUNDARY) to determine the range of bits that generate an out-of-bound memory access (at line 6) to obtain a range of crash-causing bits for the destination register (line 9). Then, for each source operand, the procedure calculates the range for the crash-causing bits by taking into account the range of

Algorithm 2 Calculates the range of the crash-causing bits for memory access instructions based on the backward slice of the address used

```
1: procedure GET_RANGE_FOR_CRASH_BITS(inst)
2:     crashing_bits ← 0
3:     global crash_bits_list
4:     oplist ← inst.source_operands
5:     if inst == load/store then
6:         (max,min) = CHECK_BOUNDARY(inst.addres)
7:         crash_bits_list[inst.address] = (max,min)
8:     else
9:         (max,min) = crash_bits_list[inst.dest_opernd]
10:    for all op in oplist do
11:        (new_max,new_min) = lookup_table(op,inst)
12:        crash_bits_list[op] = (new_max,new_min)
13:        crashing_bits += bits that make the value of op outside (new_max,new_min)
       return crashing_bits
```

the corresponding destination operand and the semantics of that instruction (line 10 to line 13). The semantics of the instruction are determined by the *lookup_table* function in line 12.

Table 3.3 shows the common instruction types encountered on the backward slice of a memory address calculation and how the *lookup_table* is used to compute the range for each operand. We assume that all values of operands are positive integers. The algorithm propagates these ranges along the backward slice by storing them in the **CRASHING_BIT_LIST** for further reference by the next instructions, as shown in line 7.

Table 3.3: Range calculation operations that commonly occur on the backward slice of memory addresses

No.	Opcode	Operand	Semantic	Range Calculation for operands
1	add	dest, op1, op2	dest = op1 + op2	Max(op1) = Max(dest) - op2 ; Min(op1) = Min(dest) - op2
				Max(op2) = Max(dest) - op1; Min(op2) = Min(dest) - op1
2	sub	dest, op1, op2	dest = op1 - op2	Max(op1) = Max(dest) + op2 ; Min(op1) = Min(dest) + op2
				Max(op2) = op1 - Min(dest) ; Max(op2) = op1 - Max(dest)
3	mul	dest, op1, op2	dest = op1 * op2	Max(op1) = Max(dest)/op2 ; Min(op1) = Min(dest)/op2 (if op2 != 0)
				Max(op2) = Max(dest)/op1; Min(op2) = Min(dest)/op1
4	div	dest, op1, op2	dest = op1 / op2	Max(op1) = Max(dest)*op2 ; Min(op1) =Min(dest)*op2
				Max(op2) =Max(op1)/dest ; Min(op2) = Min(op1)/dest
5	getelementptr	dest, op1, op2	dest = op1 + sizeof(op1.type)*op2	Max(op1) = Max(dest) - sizeof(op1.type)*op2 ; Min(op1) = Min(dest) - sizeof(op1.type)*op2
				Max(op2) = (Max(dest) - op1)/sizeof(op1.type) ; Min(op2) = (Min(dest) - op1)/sizeof(op1.type)
6	srem	dest, op1, op2	dest = op1 % op2	Max(op1) = Max(for all bits in op1: bitflip(op1)%op2 >Max(dest) and bitflip(op1)%op2 <Min(dest))
				Min(op1) = Min(for all bits in op1: bitflip(op1)%op2 >Max(dest) and bitflip(op1)%op2 <Min(dest))
				Max(op2) = Max(for all bits in op2: op1%bitflip(op2) >Max(dest) and op1%bitflip(op2) <Min(dest))
				Min(op1) = Min(for all bits in op2: op1%bitflip(op2) >Max(dest) and op1%bitflip(op2) <Min(dest))
7	bitcast	dest, op1	dest = op1	Max(op1) = Max(dest); Min(op1) = Min(dest)

We explain the details of these algorithms using our running example. In Figure 3.3b, *r5* stores the address *0x15FB174* for the instruction *store i32 r4, i32**

r5, align 4. Suppose our bound-determination technique (described in the next subsection) returns the bound (0x15FB800, 0x15FA000), meaning that addressing outside this bound will generate a segmentation fault. The ACE graph indicates that *r5*, *r6* and *r7* are used in addressing (or computing the address). Together, these three registers belong to the instruction *getelementptr* in LLVM. The instruction semantics are based on row 6 of the Table 3.3 - ranges are obtained by applying the corresponding equations.

$$r5_{.value} = r6_{.value} + sizeof(r6)_{.type}) \times (r7)$$
$$max_{r6} = max_{r5} - sizeof((r6)_{.type}) \times (r7)$$
$$min_{r6} = min_{r5} - sizeof((r6)_{.type}) \times (r7)$$

The range of *r6* can be computed as follows: min: *0x15FB800* - 4*1 = 0x15FB7FC; max: *0x15FA000* - 4*1 = 0x15F9FFC. The resulting range (0x15FB7FC, 0x15F9FFC) of r6 is stored into **CRASHING_BIT_LIST** as the reference for operands on the backward slice of *r6*, if any. Similarly, we can compute the range for register *r7* and for other registers in the backward slice.

Algorithm 3 Obtains the boundary of the segment

```
1:  procedure CHECK_BOUNDARY(inst.address)
2:      global crash_bits_list
3:      max ← 0
4:      min ← 0
5:      vma_start = locate_segment_start(inst.address)
6:      if inst.address ⊂ stack&&vma_start < ESP − 65536 − 128 then
7:          min ← ESP − 65536 − 128
8:      else
9:          min ← vma_start
10:     vma_end = locate_segment_end(inst.address)
11:     max ← vma_end
12:     crash_bits_list[inst.address] = (max, min)
        return (max, min)
```

3.3.4 Crash model

The goal of the crash model is to determine the ranges of the addresses for which a memory access will trigger a segmentation fault. While this is platform-specific i.e., specific to the hardware and operating system (OS) on which the program is running, the technique described here can be adapted to any architecture that uses memory segmentation. This includes most modern architectures such as x86 and

ARM.

Algorithm 3 describes how to compute the range of allowable addresses for a memory segment. It undertakes two main tasks: (i) obtains the boundary of the memory segment through the underlying OS interface (see line 5 and line 10) and, (ii) determines the run-time range of valid memory addresses for each load/store instruction (from line 6 to line 9). We explain the steps below.

Obtaining the segment boundaries. Modern operating systems organize process memory over multiple segments (e.g., text, data, heap, stack). To identify the boundary of each segment we instrument the program to embed a run-time probe that probes the "/proc" system of Linux to record the segment boundaries at each load and store instruction.

Determining allowed ranges. Once we determine segment boundaries at the time of each load or store, we need to determine which accesses would result in segmentation faults. We initially hypothesized that all accesses outside segment boundaries will trigger a segmentation fault. Unfortunately, this is not the case, as we found through a fault injection experiment: a segmentation fault occurred only for about 85% of the out-of-segment accesses we generated. The remaining 15% of accesses did not result in a segmentation fault even though they were outside the segment boundaries, suggesting that our hypothesis was incorrect.

To better understand this behaviour, we studied the source code of the Linux kernel in our platform, an x86-based machine (Figure 3.4)[3]. The "vma_start" and "vma_end" indicate the start and end addresses of Linux **v**irtual **m**emory **a**rea (vma) and the "addr" indicates the memory address used. In the code, the label "common case" shows the kernel code for when *addr* is within the valid bound. Note that if *addr* is within the last page of the stack, Linux will add one page below the current last page until the 8 megabyte limit is reached. The label "case I" is when *addr* is smaller than the "vma_start" and is still bigger than the $(ESP - 64KB - 128B)$. Linux treats such an address as valid and will expand the stack for it. However, if *addr* is smaller than $(ESP - 64KB - 128B)$, a segmentation fault occurs. The label "case II" occurs when *addr* is greater than the"vma_end", and will result in a segmentation fault.

[3]Similar code can be found for both x86 and PowerPC kernel versions.

```
/* "vma" means virtual memory area it is an abstraction
    used in Linux  for memory segments. */
/*  addr represents the accessed memory address */
/*  regs->sp stores the current stack pointer */
Common case:
    if   vma_start <= addr <= vma_end:
        //everything is fine
        addr &= page_mask;
        // for stack:
        if addr == the last page of the vma:
            expand_stack(vma, addr)
Case I:
    // only for stack:
    if vma_start > addr:
        if addr + 65536 + 32 * sizeof(unsigned long)
           < regs->sp:
            // SEGFAULT
        else:
            expand_stack(vma, addr)
Case II:
    if vma_end < addr
        // SEGFAULT
```

Figure 3.4: Linux kernel implementation for determining which memory accesses result in segmentation faults. Linux kernel version: 3.15. File locations: mm/ and arch/x86/mm.

Thus, for a non-stack segment, Linux determines the boundaries using its "vma_start" and "vma_end", while for stack segments, it compares the "vma_start" with the current stack pointer plus an offset to determine the lower bound of the stack. If *addr* is inside an invalid memory region, a segmentation fault occurs.

We implement our crash model to mirror the handling of these different cases. *Upon re-evaluating the accuracy of the model through the same fault injection experiment, we find that we can now accurately predict crashes for over* 99.5% *of the accesses, pointing to the correctness of the crash model.* We use this crash model in our experiments.

3.4 Evaluation

Our evaluation is guided by the following four questions:

Q1 How accurate is the ePVF methodology when predicting the bits in which faults lead to program crashes?

Q2 How close are estimated crash rates to the actual crash rates obtained through fault injection?

Q3 Can the methodology be used to obtain a significantly tighter estimate for the SDC rate than the conventional PVF methodology?

Q4 How fast and scalable is the ePVF analysis?

3.4.1 Experimental setup

Benchmarks. We evaluate the ePVF methodology on ten benchmarks (Table 3.4): these include eight OpenMP-based scientific applications picked from the Rodinia benchmark suite [29], our basic implementation of the matrix multiplication kernel, and Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (i.e. lulesh) [83, 84], a DOE proxy application. The applications range from 100 lines of code (*mm*) to 3000 lines of code (*lulesh*). Note that we target scientific applications and hence we do not consider SPEC programs.

Platform. All of our experiments are conducted on a machine with a x86 CPU running at 2.67GHz and Linux v3.15.

Table 3.4: Benchmarks used and their complexity (lines of C code).

Benchmark	Description	Domain	LOC
lulesh	Unstructured Lagrangian Explicit Shock Hydrodynamics	Physics Modelling	3,000
particlefilter	Statistical Estimator of the state of a particle	Image Analysis	602
srad	Speckle Reducing Anisotropic Diffusion	Image Processing	388
nw	Needleman-Wunsch Method	Bio Informatics	285
hotspot	Processor Temperature Estimation	Physics Simulation	272
lavaMD	MDLAVA Molecular Dynamics	Molecular Dynamics	218
bfs	Breadth-First Search	Graph Algorithm	203
lud	LU Decomposition	Linear Algebra	174
pathfinder	Dynamic Programming	Grid Traversal	135
mm	Matrix Multiplication	Linear Algebra	100

Fault injection. To build a ground truth, we use the publicly available, open-source LLFI fault injector [144] to inject faults at the LLVM Intermediate Code (IR) level. We inject faults into the source registers for the executed instructions to emulate faults in the used registers of the instructions, and hence all faults are activated as they are used in the instruction. Only one fault is injected in each run. We perform over 3,000 fault injection runs for each benchmark. The 95% confidence levels are reported as error bars for statistical significance.

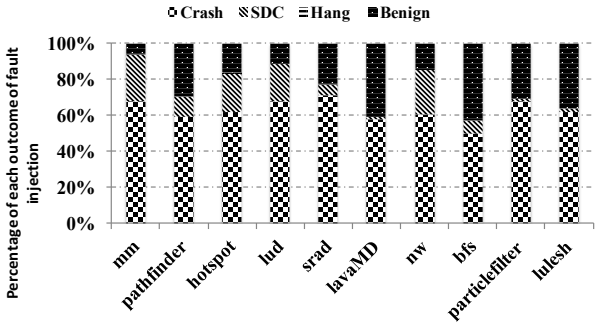


Figure 3.5: Fault injection results for each benchmark.

3.4.2 Q1: What is the accuracy of ePVF methodology?

To answer this question, we evaluate ePVF *recall* and *precision*. We use fault injection experiments to obtain the ground-truth, and compare the outcome of each fault injection experiment with the outcome predicted by the ePVF methodology. Figure 3.5 shows the outcome (i.e., SDC, crash, hang and benign fault) frequency for each benchmark: crashes are the dominant outcome, on average, 63% of injections result in crashes, while 12% result in SDCs, and less than 1% in hangs. The dominance of crashes highlights the importance of separating the crash-causing bits from the other failure bits.

Recall. We define *recall* as the ratio of crash runs that our model predicts correctly to be crashes, to all fault injection runs that lead to crashes in reality. To estimate recall, for each fault injection run that leads to a crash, we record the instruction counter and the register that the fault is injected into, as well as the bit that was flipped. We then run the crash and propagation models for the entire program and check whether the location appears in the final *crash_bits_list* (described in Algorithm 2) that stores the bits that lead to a crash if the bit is corrupted.

Figure 3.6 presents the recall for each benchmark. Overall, our methodology achieves an average of 89% recall across the ten benchmarks (ranging from 85% to 92%). We manually analyzed the crash-causing bits that were not identified as crashes by ePVF methodology. The main reason is that our validation technique introduces approximations due to non-determinism in execution environment: the segment boundaries may be slightly shifted. As a result, it cannot be guaranteed

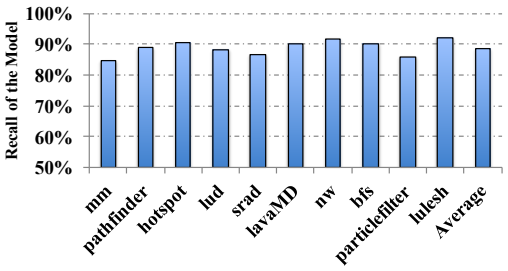


Figure 3.6: ıethodology.

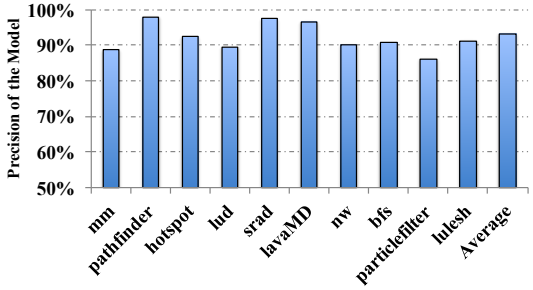


Figure 3.7: Precision for the crash bits predicted using the ePVF methodology.

to execute fault injection runs with exactly the same environment, particularly the same memory allocation and the profiling. Through manual verification we found that, depending on benchmark, this factor accounts for 92% to 99% of incorrect predictions.

Precision. We define *precision* as the ratio of the number of *correctly* predicted crash-causing bits to the total number of predicted crash-causing bits. To estimate precision, we randomly choose over 1,200 different bits from those identified by the model as crash-causing (i.e., appear in the **CRASHING_BIT_LIST**), and perform a targeted fault injection experiment. Similar to the recall study, this time for each bit, we specify the dynamic instruction and the register to inject the fault into, as well as the bit that should be flipped. Precision is calculated as the number of observed crashes over the total number of fault injections performed.

Figure 3.7 shows the results of the evaluation. The average precision across all benchmarks is 92% (ranges from 86% to 98%). As in the case of recall, after manual inspection we have confirmed that the main reason for not hitting 100% precision is the difference between the run-time and modeled environments, i.e., non-deterministic memory allocation.

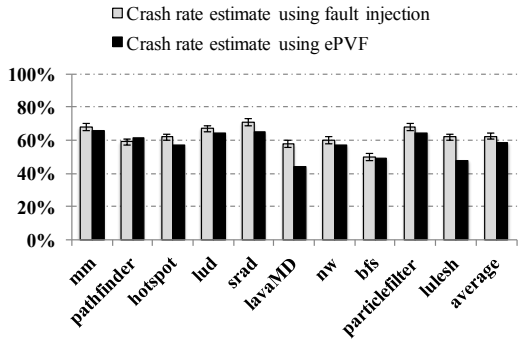


Figure 3.8: The crash rates estimates using ePVF (right bars) and using fault injection experiments (left bars) are close. For fault injection experiments, the error bars indicate the 95% confidence intervals.

3.4.3 Q2: How close are the crash rates estimated using ePVF and fault injection?

The ePVF methodology is able to identify crash-causing bits with high accuracy. This can be used to estimate the crash rate of a program as the fraction of crash-causing bits over the total number of bits in an application. Such an estimate can be important for techniques that use crash rates to determine the level of protection to be provided, e.g., choosing the checkpoint interval.

Figure 3.8 shows that estimating crash rates this way is a good approximation for crash rates obtained through fault injection experiments. The differences are within or close to the 95% confidence interval bounds, except for *lavaMD* and *lulesh*. The reason the crash rate predictions are off for these two applications is that ePVF calculates the crash bits only based on the ACE graph, which contains 70% and 80% of the whole DDG for *lavaMD* and *lulesh* respectively. On the other hand, the fault injection uses the full program execution corresponding to the whole DDG.

3.4.4 Q3: Does ePVF lead to a tighter estimate of SDC rate than the original PVF?

We have shown that the ePVF methodology can accurately estimate the crash bits of an application. We now ask whether it can lead to better SDC rate estimates. As explained earlier, ePVF provides an upper bound (i.e., overestimate) for the SDC

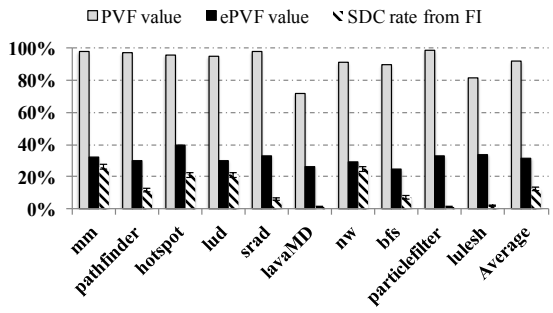


Figure 3.9: ePVF (center bars) offers a much better upper bound estimate for the SDC rate (right bars) than the original PVF methodology (left bars). For SDC rates, error bars represent 95% confidence intervals.

rate like PVF does. We compare the tightness of these two upper bounds.

Figure 3.9 shows the original PVF and the ePVF values for the ten benchmarks. The original PVF ranges from 71% to 98%, with an average of 92%. In contrast, the ePVF estimate ranges from 25% to 40%, with an average value of 31%. The average difference between PVF and ePVF is 61%, ranging from 45% to 67% depending on the benchmarks.

Figure 3.9 also shows, for each benchmark, the SDC rate obtained through the fault injection experiments described earlier. The SDC rate ranges from 1 to 25% depending on the benchmark, with an average value of about 12% across benchmarks. *ePVF significantly lowers the upper bound of estimated SDC vulnerability of a program.* The above evaluation suggests that our technique has higher predictive power than the original PVF analysis to understand the SDC behaviour of a program (we demonstrate that this can be used in practice in §3.5). That said, there is still room for a tighter bound as we will discuss in §3.6.

3.4.5 Q4: How fast is the ePVF analysis?

Table 3.5 shows, for each benchmark, the number of dynamic LLVM IR instructions, the number of nodes in the ACE graph, and the total time to compute ePVF. The running time ranges from less than a minute (*lavaMD*) to 5 hours (*pathfinder*). As expected, the time taken correlates with the ACE graph size. We also measured the time spent by various parts of the ePVF analysis: most time is spent in the crash

Table 3.5: Number of nodes in the ACE graph and time taken by the ePVF analysis for each benchmark

Benchmarks	# of Dynamic IR instructions	ACE nodes	Modelling time (s)
hotspot	954,920	1,102,265	14,400
pathfinder	839,163	967,836	18,000
mm	464,438	597,604	3,987
particlefilter	352,866	479,994	3,956
nw	376,022	453,998	3,800
lulesh	322,738	319,253	953
bfs	274,170	269,019	900
lud	75,543	93,089	205
srad	72,041	91,385	172
lavaMD	17,814	16,779	30

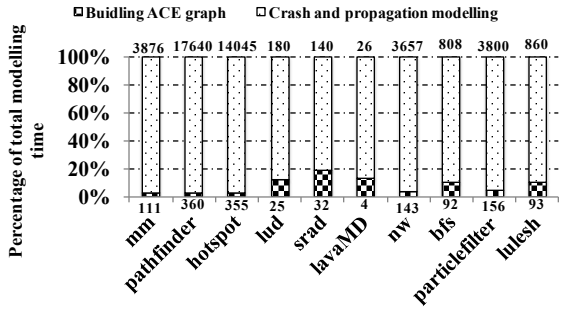


Figure 3.10: Breakdown of execution time between graph construction (bottom bar) and running the crash and propagation models (top bar). Labels on bars present absolute time in seconds.

and propagation models.

We discuss scalability in detail in Section §3.6. Here we propose an optimization to reduce the time to compute ePVF, based on sampling the ACE graph. This approach is based on the intuition that many HPC applications consist of repetitive program states and patterns, and hence a small sample of the ACE graph will be representative of the overall application behaviour. Since a dynamic instruction trace preserves the temporal ordering of the instructions executed by the program, the output nodes in the ACE graph can be ordered based on their presence in the

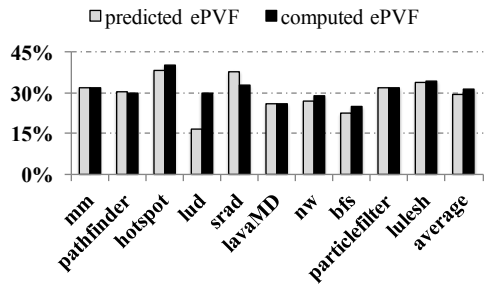


Figure 3.11: The predicted ePVF value based on sampling only 10% of the ACE graph and ePVF computed based on the entire graph are close.

trace. To validate if the sampling works, we pick the first $p\%$ of the output nodes, and based on the resulting partial ACE graph we estimate ePVF. For regular applications, we find that there is a strong linear relationship and we can linearly extrapolate the partial ePVF to the entire application and thus estimate the overall ePVF accurately.

Figure 3.11 shows the extrapolated ePVF values based on analyzing only 10% of the ACE graph. As can be observed, for most benchmarks, the extrapolated ePVF values are a good approximation for the overall ePVF: on average the error is less than 1%, suggesting that these programs exhibit repetitive behaviors as we expected.

Importantly, we can also estimate whether an application displays repetitive behaviours and thus whether the ACE-graph sampling be useful, without completing the full ACE analysis. To demonstrate this, we randomly select multiple small sub-sample of the ACE graph nodes (each 1%) and compute for each of them the ePVF estimates. The normalized variance is relatively low for benchmarks with repetitive behaviours (e.g., 0.6 for *lavaMD* and 0.04 for *particlefilter*), but high for applications where the ACE-graph sampling technique does not offer high accuracy (e.g., 1.9 for *lud*).

3.5 Case study: selective duplication

To demonstrate the practical usability of the ePVF methodology to improve application resilience, we use ePVF to guide a selective instruction duplication tech-

nique to protect against SDCs. The intuition is that a technique that prioritizes protecting instructions with high ePVF values will offer good SDC protection as the faults occurring in crashing bits are unlikely to lead to SDCs. To establish a baseline, we compare the SDC rate of a program protected by duplicating the high ePVF instructions, with that protected by duplicating the hot paths of the program. Prior studies [74, 103] have shown that protecting hot paths is an effective technique (i.e., instructions on the top 20% of most executed paths are responsible for most of the SDCs - these constitute the hot paths).

We also attempted to use PVF to drive the choice of instructions to duplicate. However, we found that the PVF values of most instructions are clustered around 1, which means that PVF has little discriminative power to inform the choice of which instructions to protect. As an example, we plot the CDF (Cumulative Distribution Function) of the PVF and ePVF values of every instruction for two benchmark programs, namely *nw* and *lud* in Figure 3.12. As can be seen in the figure, the CDF for PVF has a sharp spike near 1, while the ePVF values are distributed more evenly throughout the range. Therefore, we did not consider PVF-informed duplication as a comparison point in this study.

To make the comparison fair, we control the performance overhead incurred by both techniques we compare (by controlling the number of instructions we protect and measuring execution time). *Our hypothesis is that for a given performance overhead bound, the ePVF based duplication scheme can offer higher SDC coverage than hot-path duplication*. A full-duplication technique (i.e., duplication of every instruction) would offer 100% detection coverage, but incur significant performance overheads [60, 103]. Hence we do not consider full-duplication technique in this work.

An ePVF-informed protection heuristic We first compute the ePVF value of each dynamic instruction using the equation 3.3. Then, we compute the ePVF value of all static instructions in the program by averaging the ePVF values of all their dynamic instances, and rank the static instructions in descending order of their ePVF values. We then select the static instruction at the top of the list, and extract its backward slice. Finally, we selectively duplicate the instructions in the slice, and insert a comparison of the duplicated value with the original value following the chosen instruction. Because we need to limit protection within the overhead

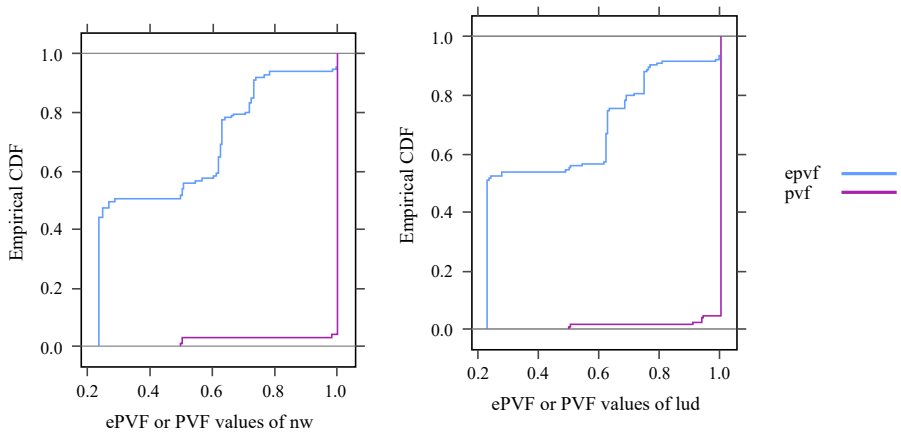


Figure 3.12: The figure presents the CDF for the ePVF and PVF values of registers used in each instruction of *nw* (left) and *lud* (right) benchmarks. PVF values for most instructions are clustered around 1 and thus can not inform protection mechanisms based on instruction level protection.

budget, we measure the performance overhead incurred by duplication. If the performance overhead bound is not exceeded, we choose the next instruction on the list and repeat the procedure. Thus this is a greedy algorithm for choosing instructions to duplicate. We use the same process for hot-path instructions, with the difference that the instructions are ranked in a decreasing order of their execution frequencies instead of their ePVF values.

$$ePVF_{inst} = \frac{\sum_{register\ in\ inst} (ACEBits - CrashBits)}{Total\ bits\ in\ inst} \tag{3.3}$$

Evaluation Methodology. We evaluate the coverage of the above schemes through fault injection experiments. We only consider the five benchmarks whose SDC rates were higher than 10% in Figure 3.9 (i.e. *mm, pathfinder, hotspot, lud and nw*) so as to discriminate the effects of the two schemes better. Further, we run the fault injection campaigns with different inputs than the ones we used to get the ePVF values (these are much larger in size) to get stable performance numbers.

Evaluation Results. Figure 3.13 shows the SDC rate of the original application (no protection), the SDC rate when using hot-path protection, and the SDC rate

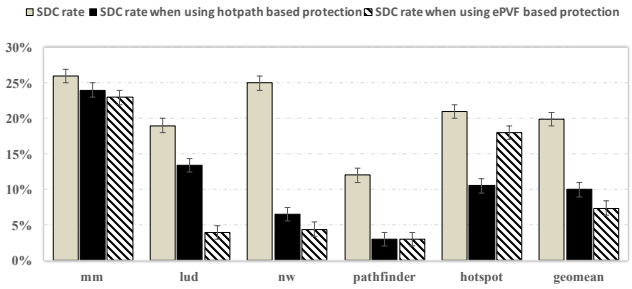


Figure 3.13: SDC rates for the original application (left bars) and when using hotpath (center) and ePVF-based protection (right) for an overhead bound of 24%. Error bars present 95% confidence intervals.

when using ePVF-informed protection, given a performance overhead bound of 24%[4]. Overall, we find that ePVF based protection does better than hot-path based protection, and reduces the SDC rate from 20% to 7% (geometric mean), while hot-path based protection reduces it to about 10%. Thus, ePVF based protection does 30% better than hot-path based protection, on average across benchmarks, showing that it has better discriminative power than execution frequencies for protection. Further, we find that ePVF-based protection outperforms hot-path based protection for all benchmarks except *hotspot*. This is due to the presence of many control-flow structures in *hotspot* all of which are marked as sensitive by ePVF though they do not cause SDCs.

3.6 Discussion

A. Scalability is an important issue as most applications will likely generate ACE graphs with billions of vertices. The ACE-graph sampling technique we describe in §3.4.5 offers a significant speedup for applications that contain repetitive patterns. We believe that scaling to handle larger applications is a matter of good engineering and not a fundamental barrier for the following reasons. First, the current ePVF infrastructure (including building/processing the DDG) is implemented in Python. A tuned C/C++ implementation will likely be orders of magnitude faster and consume less memory. Second, the most time-consuming portion of the ePVF

[4]We considered performance overheads of 8%, 16% and 24% as well. We report here only the results using 24% overhead due to space constraints - the qualitative results were similar all cases.

analysis is running the crash and propagation models. These start from each load/-store individually, and search along their backward slices. This process is trivially parallelizable (threads can be assigned to one backward slice each with minimum coordination required). Additionally the work allocated for each thread (i.e., the size of the backward slice) scales sub-linearly with the size of the graph. Third, if the DDG does not fit in memory, it can be partitioned to support the parallel backward slice exploration suggested above.

B. Sources of Inaccuracy. §3.4.3 shows that ePVF is a much closer upper bound than PVF for the SDC rate of an application. However, ePVF still overestimates the SDC rate, in some cases by a significant amount. This overestimate is generated mainly by the following three factors:

1. Lucky loads: ePVF assumes that any fault that causes a load to deviate from its intended source address (but still stays within the bounds of the program's allocated memory) will lead to an SDC. However, as prior work has found [38], this is not always true. For example, the value loaded from the incorrect address may still be correct, and hence have no effect on the program. The likelihood of this case increases if the value loaded is 0, as memory typically has large areas initialized to zeroes [38].

2. Y-branches: Y-branches are branches that do not affect the outcome of the application even when the program executes the wrong part of a branch due to a fault [142]. The ePVF analysis assumes that all branches lead to SDCs if flipped. However, only about 20% of branch flips lead to SDCs in practice, as prior work has found [142].

3. Application-specific correctness checks: Similar to PVF, the ePVF model, considers as ACE bits all bits that lead to visible changes to the application output. Some of these faults, however, may be characterized as benign by application-specific correctness checks (e.g., based on precision thresholds for floating-point computations).

C. Conservativeness: While ePVF may overestimate the SDC rate, it will never underestimate it (barring the case below). This is because ePVF conservatively labels every non-crash causing operation as potentially leading to an SDC. Being conservative is important as it can drive decisions about how much state to protect in the worst-case for the application. However, in Section 3.4.2, we found

that our implementation of the ePVF methodology may yield false positives i.e., it may identify a failure as a crash when in fact it is an SDC. This occurs because of differences between the program's memory structures in the golden run (on which the ePVF analysis is based) and the fault injected runs. However, the differences are very small in practice (at most 8% in our experiments).

3.7 Summary

This chapter presents ePVF, a methodology that extends the PVF analysis to distinguish crash-causing bits from the ACE bits for obtaining a tighter bound on SDC rate. With ePVF, we demonstrate that one can predict the error resilience characteristics of an application with "good enough" accuracy at a low cost and use such information to guide the selective duplication to detect SDCs, which answers the RQ1. Our methodology consists of two models: (1) a propagation model to predict the dependent bits of memory address calculations based on a range propagation analysis; and (2) a crash model to predict the platform-specific behaviour of program crashes. We implement the ePVF methodology in the LLVM compiler. The results show that ePVF can predict crashes with high confidence (89% recall and 92% precision on average). Further, ePVF significantly lowers the upper bound of the estimate of the SDC rate of a program (61% on average) compared to the original PVF. Finally, we present a use-case study with this methodology: an ePVF-informed selective duplication technique, which leads to 30% lower SDCs than hot-path instruction duplication.

Chapter 4

LetGo: A Lightweight Continuous Framework for HPC Applications Under Failures

Requirements for reliability, low power consumption, and performance place complex and conflicting demands on the design of high performance computing (HPC) systems. Checkpoint/restart (C/R) schemes are the most commonly-used fault-tolerance techniques to protect HPC applications against hardware faults. Such fault tolerance techniques, however, have non-negligible overheads particularly when the fault rate exposed by the hardware is high: it is estimated that in today's HPC systems, 20% or more of the computational cycles have been used for providing fault tolerance [18].

To mitigate the overall overhead of C/R systems, we propose the roll-forward C/R system to work for comprehensive types of errors. In particular, in this chapter, we present *LetGo*, an approach that attempts to continue the execution of a HPC application when crashes would otherwise occur due to the errors affecting the computational components, which answers the first part of the RQ2. Our hypothesis is that many classes of HPC applications have good enough intrinsic fault tolerance so that it is possible to re-purpose the default mechanism that terminates an application once a crash-causing error is signaled. Instead, LetGo repairs the corrupted application state and continues the application execution. This chapter

explores this hypothesis, and quantifies the impact of using this observation in the context of checkpoint/restart (C/R) mechanisms.

4.1 Introduction

Applications crash on transient hardware faults (i.e., bit flips) due to the runtime system detecting the error and terminating the application, thereby losing the application's work. Checkpoint/restart (C/R) is one of the most popular methods to recover from such faults [54, 138, 139] by loading a previously saved intermediate state of the application (i.e., a checkpoint), and restarting the execution. While useful, checkpoint/restart techniques incur high overheads in terms of performance, energy and memory, which will be exacerbated as the failure rate increases [42, 151].

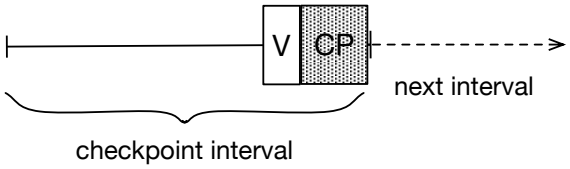
This chapter proposes LetGo, which upon detecting an impending crash, attempts to repair the application state to enable it to continue its execution (instead of recovering from a checkpoint). LetGo is based on three observations. First, a large class of HPC applications are, intrinsically, resilient to localized numerical perturbations as they require computation results to converge over time. As a result, they are able to mask some data corruptions. For example, Casas et al. [28] show that the algebraic multi-grid (AMG) solver, which is based on iterative methods, has high intrinsic resiliency. Second, a certain number of classes of HPC applications have application-specific acceptance checks (e.g., based on energy conservation laws). These checks can be used to filter out obvious deviations in the application's output, and reduce the probability of producing incorrect results. For example, High Performance Linpack (HPL) solves a linear system using LU decomposition [119] and tests the correctness of the result by checking the residual of the linear system as a norm-wise backward error [77, 106]. Third, most crash-causing errors lead to program crashes within a small number of dynamic instructions, and are hence unlikely to propagate to a large part of the application state [95, 121]. Therefore, the impact of crash-causing faults is likely to be confined to a small portion of the application's state, thus allowing recovery.

Taken together, these observations offer an optimistic hypothesis that it may be possible to re-purpose the default mechanism that terminates an application once

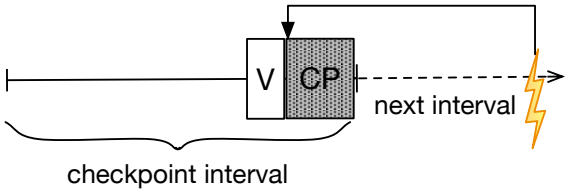
a crash-causing error is signalled, and attempt to repair the corrupted application state and continue the application execution. This paper explores this hypothesis, proposes heuristics to repair the application state, and quantifies the impact of using this observation in the context of C/R mechanisms.

To enable this exploration we design and implement *LetGo*. LetGo works by monitoring the application at runtime; when a crash-causing error occurs, LetGo intercepts the hardware exception (e.g., segmentation fault), and does not pass the exception on to the application. Instead, it advances the program counter of the application to the next instruction, bypassing the crash-causing instruction. Further, LetGo employs various heuristics to adjust the state of the application's register file to hide the effects of the ignored instruction and ensure, to the extent possible, that the application state is not corrupted.

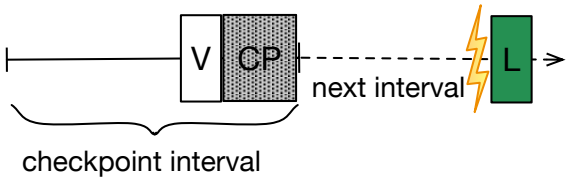
Figure 4.1 illustrates how LetGo can be used in the context of a checkpoint/restart (C/R) scheme. As shown in Figure 4.1a and 4.1b, the default action of a C/R scheme on fail-stop failures is to rewind to the last checkpoint. LetGo allows the HPC run-time to continue the execution of an application once a crash-causing error occurs (Figure 4.1c) and later use application-level correctness test to detect possible state corruption. If the application passes these checks, LetGo assumes that intermediate/final states of an application are correct, and hence no recovery is needed. This reduces checkpoint overheads in two ways: first, LetGo avoids the overhead of restarting from a previous checkpoint upon the occurrence of a crash-causing error; second, since crashes are less frequent, checkpoints can be taken less frequently as well (or not at all if the developer is prepared to accept the risk of unrecoverable failures). The potential cost of LetGo is an increased rate of Silent Data Corruption (SDC) leading to incorrect results. We argue that this may be acceptable for two reasons: first, our experiments indicate that this increase is low (the resulting SDC rate is in the same range as the SDC rate of the original application), and, second, since the possibility of undetected incorrect results exists even with the original application (i.e., without using LetGo), application users independently need to develop efficient techniques to increase confidence in the application results. By leveraging these application checks, LetGo reduces the chances of an error causing a SDC. *To the best of our knowledge, LetGo is the first system that applies the idea of tolerating errors by repairing application state in*



a A standard checkpoint interval without LetGo.



b If a crash occurs, the HPC run-time loads the last checkpoint. In existing solutions a crash occurs each time a crash-causing error is signalled.



c LetGo continues the execution of the application when a crash-causing error occurs, and the HPC run-time does not load the checkpoint.

Figure 4.1: Illustration of how LetGo changes the behavior of an HPC application that uses checkpointing by continuing the execution when a crash-causing error occurs. Axes indicate time. The labels used for time intervals: CP - checkpoint; V - application acceptance check/verification; L - LetGo framework, lightning bolt: crash-causing error

the context of C/R in HPC applications.

This chapter makes the following contributions:

- We propose a methodology to reduce the overhead of C/R techniques for HPC applications by resuming the execution of an application upon the occurrence of a crash-causing error without going back to the last checkpoint.
- We design LetGo, a light-weight run-time system that consists of two main components: a *monitor* that intercepts and handles operating system signals generated when a crash-causing error occurs, and a *modifier* that employs

heuristics to adjust program state to increase the probability of successful application continuation (Section 4.3.1). Importantly, LetGo requires neither modifications to the application, nor the availability of the application's source code for analysis (Section 4.3.3 and Section 4.3.2). therefore, it is practical to deploy in today's HPC context.

- We evaluate LetGo through fault-injection experiments using five DoE mini-applications. We find that LetGo is able to continue the application's execution in about 62% of the cases when it encounters a crash-causing error (for the remaining 38%, it gives up and the application can be restarted from a checkpoint as before). The increase in the SDC rate (undetected incorrect output) is low: 0.913% arithmetic mean. (Section 4.5.1).

- Finally, we evaluate the end-to-end impact of LetGo in the context of a C/R scheme and its sensitivity to a wide range of parameters. We find that LetGo offers significant efficiency gains (1.01x to 1.20x) in the ratio between the time spent for useful work and the total time cost, compared to the standard C/R scheme, across a wide range of parameters.

Our evaluation shows that, on average, LetGo is able to continue to completion 62% of the crashes while increasing the overall application SDC rates from 0.75% to 1.6%. This highlights a key contribution of LetGo: it creates the opportunity to trade off confidence in results for efficiency (time to solution). Certainly, for some applications - or for some operational situations - confidence in results is the user's primary concern, and LetGo will not be used. We believe, however, that there are many situations that make LetGo attractive: Firstly, since Silent Data Corruptions (SCD) can occur anyways (due to bit-flips even when LetGo is not used), users of HPC applications are already taking the risk of getting incorrect results, and have developed techniques to validate their results. Application-specific checks to diminish this risk are an active area of research [57, 60, 76, 102] and LetGo will benefit from all these efforts. Secondly, for some applications LetGo performs extremely well (e.g., for CLAMR and SANP all faults that would lead to crashes can be elided by LetGo, without resulting in any additional SDCs). In these cases, LetGo certainly represents an appealing solution. Finally, note that it is trivial to

collect information on whether a run has benefited from LetGo repair heuristics and thus offers users additional information base on which to reason about confidence.

4.2 Background

Context: The effectiveness of LetGo is influenced by two factors: (1) the application-level acceptance checks that detect whether the application state is corrupted before delivering results to users; (2) the resilience characteristics of the HPC application making it able to withstand minor numerical perturbations. This section argues that a large class of HPC applications present these characteristics, and offers an example that illustrate how these two factors affect application fault-tolerance.

Factor 1: Application acceptance checks. Since the rate of hardware faults is expected to increase and applications become increasingly complex (and, as a result, the design and implementation process is error-prone), there is an increased awareness for the need of result acceptance tests, to boost the confidence in the results offered by HPC applications. Result acceptance checks are usually written by application developers to ensure that computation results do not violate application-specific properties, such as energy conservation or numeric tolerance for result approximation. These acceptance checks are typically placed at the end of the computation (i.e. the residual check performed in HPL application [119]), but they can be also placed during application execution to detect earlier possible state corruption such as [114].

Factor 2: Fault masking in HPC applications. A large class of HPC applications are based on iterative processes (For example, stencil computations iteratively compute physical properties at time T+1 based on values at time T; iterative solvers work by improving the accuracy of the solution at each step. For an iterative method that is convergent, numerical errors introduced by a hardware fault can be eliminated during this convergence process (although it may take longer). Prior studies such as [28] show that the algebraic multi-grid solver always masks errors if it is not terminated by a crash.

4.3 System design

Our goal is to demonstrate the feasibility and evaluate the potential impact of a run-time framework that allows the program to avoid termination and correct its state after a crash-causing error occurs. The four main requirements of LetGo are:

a) *Transparency:* LetGo should be able to transparently track the system behavior, monitor for crash-causing errors, and modify the application state to enable application continuation once a crash-causing error occurs, all without modifying the application's code (R1).

b) *Convenience:* As HPC applications tend to be conservative and sensitive to the computation environment, LetGo should not make any assumption about the application's compilation level or require changes the application's compilation process (R2).

c) *Low overhead:* To be attractive for deployment in production systems, LetGo should incur minimum overheads in terms of performance, energy and memory footprint (R3).

d) *A low rate of newly introduced failures:* LetGo inherently trades the ability to continue application execution for the risk of introducing new failures. For LetGo to be practical, the increase in the rate of undetected incorrect results should be low (R4).

The rest of this section describes LetGo design in detail and, shows how LetGo satisfies the above requirements.

4.3.1 Overall design

LetGo is activated when a crash-causing error occurs. LetGo detects the exceptions raised by the OS, intercepts the OS signals, and modifies the default behavior of the application for these signals. Then it diagnoses which states of the program have been corrupted, and modifies the application state to ensure, to the extent possible, application continuation. Figures 4.2 and 4.3 show this process.

LetGo contains two components: the *monitor* and the *modifier*.

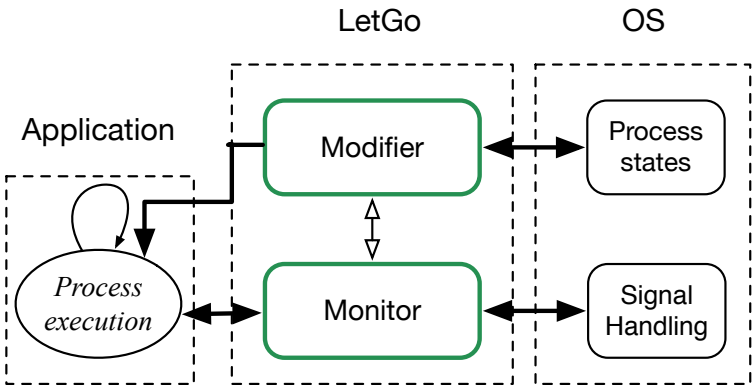


Figure 4.2: LetGo architecture overview.

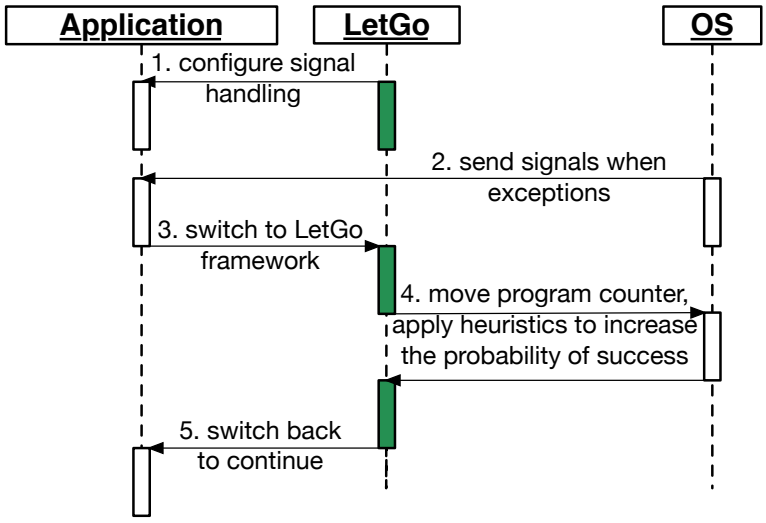


Figure 4.3: A sequence diagram highlighting LetGo use: the step 1-5 describe the interactions between LetGo, the application, and the operating system: LetGo starts by installing the monitor - i.e., configuring the the signal handling, and launches the application in a debugger (step 1). If the application encounters a signal, LetGo detects it (step 2) and takes the control of the application (step 3). To avoid the failure, LetGo increments the program counter (i.e., instruction pointer) of the application and adjusts the necessary program states (step 4). After the modification, LetGo lets the application continue without any further interference (step 5).

The monitor is attached to the application at startup. It changes the default behavior of the application from termination to pausing when operating system signals such as SIGSEGV and SIGBUS are received. The monitor intercepts these

signals and hands the control over to the modifier.

The modifier kicks in when executing the current application instruction would lead to an application termination (crash). The modifier attempts to avoid the crash: it advances the program counter (i.e., instruction pointer) to the next instruction, and inspects and modifies application state (e.g., the stack pointer) to increase the probability of a successful continued execution. The details of the modifier are discussed in Section 4.3.2. Note that the application might still not be able to finish successfully and it may crash again - if so, LetGo does not intervene and allows the application to terminate.

4.3.2 Heuristics

We describe the modifications that the LetGo modifier makes to the application state (in Step 4 of Figure 4.3). These modifications have two goals: first, increase the likelihood that, once the application continues execution, it does not crash again; and, second, reduce the chance that data corruption propagates further.

There are two issues to deal with: first, advancing the program counter may bypass memory loads or stores, and hence the destination register that is supposed to hold the value from the memory load (or the memory location used for store) may contain an incorrect value , which may cause subsequent errors in case this register is later used; and, second, if the fault has corrupted the stack pointer register *sp* (i.e., *rsp* in X86-64) or the base pointer register *bp* (i.e., *rbp* in X86-64), and the application continues due to LetGo, the likelihood of receiving another system exception due to a memory-related violation is high because *sp* and *bp* are repeatedly used. To mitigate these challenges, LetGo employs two heuristics (to satisfy R4).

Heuristic I - This heuristic deals with memory load/store instructions. If the program crashes due to the error in a memory-load instruction, LetGo feeds the to-be-written register(s) (which holds the data loaded from the memory) with a “fake" value(s). In practice, 0 is chosen as the value to feed to the register. We choose 0 by default because the memory often contains a lot of 0s as initialization data [39]. For the case where the program stops at a memory-store instruction,

the value in that memory location remains the same because the memory-store operation is not successful. In this case, we do nothing - our empirical experience suggests that this is a more practical decision than assigning a random value. In the future, this heuristic can be combined with run-time analysis for more realistic and application-dependent behaviour.

Heuristic II - As discussed above, if a fault affects the values in the stack pointer register or the base pointer register, the corrupted registers may cause consecutive memory access violations. Since LetGo avoids performing run-time tracking, determining the correctness of the values in *sp* and *bp* statically becomes challenging. To overcome this challenge, LetGo implements the following heuristic that include a detection and a correction phase:

1. *Detection*: for each function, the difference between the values in *sp* and *bp* can be approximately bound in a range via static analysis, hence this range can be calculated with minimum effort and can be used to indicate the corruption in *sp* or *bp* at run-time.

2. *Correction:*: since *sp* and *bp* usually hold the same value at the beginning of each function, one can be used to correct the error in the other one if necessary.

We explain the intuition behind this heuristic with two observations based on the code in Listing 4.1. First, *bp* is normally pointed to the top of the stack (line 2), hence *sp* and *bp* usually carry the same value at the beginning of every function call. Second, based on the size of the memory allocated on the stack (line 3), the range of *bp* can be inferred as $sp < bp < sp + 0x290$ (*bp* is always greater than *sp* because the stack grows downwards. Therefore, when the program receives an exception and stops at an instruction that involves stack operation, LetGo runs the following steps: First, it gets the size of the allocated memory by searching for the beginning of the function that the instruction belongs to and then locating the instruction that shows how much memory the function needs on the stack (by analyzing the assembly code). Second, it calculates the valid range based on the size and checks if the *bp* is in it, and, finally, if the range constraint is invalid, LetGo copies the value of the *sp* to the *bp* (or vice versa depending on which one is used in the instruction causing the crash).

Listing 4.1: Example of a common sequence of X86 instructions at the beginning of a function

```
push \%rbp
mov  \%rsp,\%rbp
sub  \$0x290, \%rsp
```

For the rest of this paper, we refer to the version of LetGo that applies these heuristics as **LetGo-E**(nhanced) and the version without heuristics as **LetGo-B**(asic). We evaluate the effectiveness of LetGo-B and LetGo-E in Section 4.5.

4.3.3 Implementation

We implement the LetGo prototype with three production-level tools that are widely adopted and readily available on HPC systems: gdb, PIN [118] and pexpect [130].

gdb: LetGo relies on *gdb* to control the application's execution. *gdb* provides the interfaces to handle operating system signals and to change the values in the program registers. We describe these two aspects in turn. LetGo uses *gdb* to redefine the behaviour of an application against OS signals as described in Table 4.1. Since most of application crashes are due to memory-related errors such as segmentation faults or bus errors [57, 68], LetGo currently supports three signals related to memory errors: SIGSEGV, SIGBUS and SIGABRT, and can be easily extended for more signals if needed. (Satisfying R1).

Note that the LetGo use of *gdb* does not require any source-code level analysis (or changes to the application). Applications therefore do not need to run in the debug mode, which inhibits code optimization and often results in significant performance degradation (satisfying R2 and R3). Applications can run with LetGo for any optimization/compilation requirement levels they need. We evaluate the generated overhead in Section 4.4.

b) PIN: It is a tool that supports dynamic instrumentation of programs. *PIN* can insert arbitrary code at arbitrary locations of an executable during its execution. LetGo uses *PIN* to conduct instruction-level analysis, such as obtaining the next PC, parsing an instruction and finding the size of allocated memory on the stack. Since LetGo only needs the static information of a program, there is no need for LetGo to keep track of dynamic program states and only dissembler inside PIN is needed. Therefore, LetGo incurs minimum performance overhead (Satisfying R3).

Table 4.1: *gdb* signal handling information redefined by LetGo. 'Stop' means the program will stop upon a signal, and 'Pass to program' means this signal will not be passed to the program

Signal	Stop	Pass to program	Description
SIGSEGV	Yes	No	Segfault
SIGBUS	Yes	No	Bus error
SIGABRT	Yes	No	Aborted

It is possible to use other lightweight tools for parsing instructions instead of PIN.

c) pexpect: expect [99] is a tool that automates interactive applications (e.g. telnet, ftp, etc.) and it is widely for testing. LetGo uses *pexpect*, the Python extension of *expect* to automate all interactions between LetGo and the application: e.g., configuring signal handlers and updating register values. Since these are relatively rarely executed operations, the overall performance impact is small.

All the interactions between a *gdb* process and the target application are automated via *pexpect*, and confined to a limited number of *gdb* commands such as "print" or "set". When heuristics need to be applied, LetGo relies on *PIN* to analyze the program and feed the result to *gdb*. As a prototype, the current implementation of LetGo is used to support the experimentation, to demonstrate the ability of automation, and to investigate the overheads incurred - for a production version, one can directly and efficiently implement the functionality offered by each of these tools, so the overhead estimates we offer are conservative.

4.4 Evaluation methodology

This section focuses on evaluating the ability of LetGo to transform crashes into successful application runs To this end, this section first describes the fault model and the fault injection methodology we use, then explains how the various failure outcome categories are impacted by LetGo, and proposes metrics to quantitatively evaluate LetGo effectiveness. Using this information, the next section evaluates LetGo impact on reducing C/R overheads.

4.4.1 Fault model

As introduced earlier, in this work, we consider faults occurring in the computational units of processors, such as the ALUs, pipeline latches and register files. Our methodology is agnostic to whether a fault arises in the register file or is propagated to the registers from elsewhere. We use the single-bit-flip model as it is the most common transient fault model in today's systems [133]. We also assume that at most one fault occurs in an application run leading to a crash-causing error, as soft errors are relatively rare compared to typical application execution times.

4.4.2 Categories of fault outcomes

The traditional outcomes of a fault affecting an application can be categorized as crashes, detected by the application acceptance check, hangs, SDCs, and benign outcomes. When applying LetGo, (some of the) crash outcomes are transferred to other categories, thus, to evaluate LetGo we further categorize the outcomes that correspond to a crash in a non-LetGo context in multiple new classes as presented in Figure 4.4.

At the top level of our taxonomy (Figure 4.4), a fault either causes a program to crash, or not. In Figure 4.4 we label these two classes - *Finished* and *Crash*.

1. A *finished* run can result further in two outcomes: the program contains errors in the output that are detected by the application's acceptance checks (labeled as *Detected*), or the output of the program passes those checks (labeled as *Pass check*). If the output passes the check, it may differ from the golden run, in which case we consider it an *SDC* [1]; or the output matches the golden run, labeled as *Benign*.

2. A *Crash* run is where LetGo has impact. When LetGo is deployed, it may fail to continue the application and lead to a second crash (labeled as *DoubleCrash*) or make the application to finish successfully labeled *C-Finished*. In this case, the program may have similar outcomes as in the *Finished* case (when no crash occurs) - we label these as *C-Benign* (no observable outcome

[1] This is a conservative assumption as we do not know how the results of the application are used. The application output also includes the application data that is compared between such data from the golden run, as defined in Table 4.2

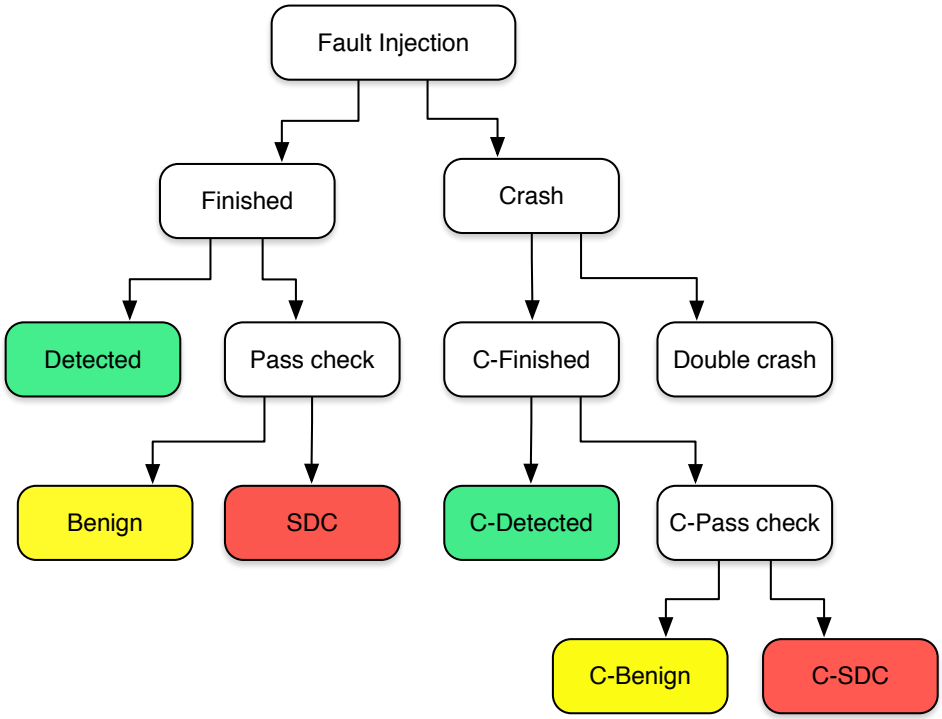


Figure 4.4: Classification of the fault injection outcomes. LetGo has impact only on the right side of the tree above as it attempts to avoid a crash outcome.

of the fault), *C-Detected* (incorrect output detected by application acceptance checks), and *C-SDC* (incorrect output not detected by those checks, but different from the golden run). Compared to a situation when LetGo is not used, LetGo is able to transfer some of the crash outcomes to C-Benign, C-SDC, or C-Detected outcome.

4.4.3 Metrics of effectiveness

The effectiveness of LetGo can be estimated by answering questions like “How many crashes can be converted to continued execution?", “what is the likelihood of producing correct results after continuation?", “How often the application check can catch the errors after continuation?" and “How many incorrect results will be produced after continuation?". To this end, we define four metrics to quantitatively evaluate LetGo effectiveness:

Table 4.2: Benchmark description. The last two columns present which data is used for bit-wise comparison to determine SDCs (undetected incorrect results), and, respectively describe the result acceptance check used by each application. All benchmarks are compiled with g++ 4.93 using O3 except for SNAP, which is a FORTRAN program.

Application	**Domian**	**# dynamic instructions (billions)**	**Application data used to check for SDCs**	**Criteria used in application acceptance check**
LULESH	Hydrodynamics	1.0	Mesh	Number of iterations: exactly the same Final origin energy: correct to at least 6 digits Measures of symmetry: smaller than 10^{-8}
CLAMR	Adaptive mesh refinement	2.8	Mesh	Threshold for the mass change per iteration
HPL	Dense linear solver	1.2	Solution vector	Residual check on the solution vector
COMD	Classical molecular dynamics	5.1	Each atom's property	Energy conservation
SNAP	Discrete ordinates transport	1.6	Flux solution	The flux solution output should be symmetric
PENNANT	Unstructured mesh physics	1.7	Mesh	Energy conservation

Continuability is the likelihood that LetGo is able to convert a crashing program into the program that would finish (regardless of the correctness of the output).

$$Continuability = \frac{C\text{-}Pass\ check + C\text{-}Detected}{Crash} \tag{4.1}$$

Continued_detected is the likelihood that the application acceptance check catches errors in the application (if any) after continuation.

$$Continued\_detected = \frac{C\text{-}Detected}{Crash} \tag{4.2}$$

Continued_correct is the likelihood that the programs result in the correct output after continuation.

$$Continued\_correct = \frac{C\text{-}Benign}{Crash} \tag{4.3}$$

Continued_SDC indicates the likelihood that application finished but results in SDCs - undetected incorrect results.

$$Continued\_SDC = \frac{C\text{-}SDC}{Crash} \tag{4.4}$$

Note that the *Continuability* is the sum of *Continued_detected*, *Continued_correct* and *Continued_SDC* metrics. All four values range from 0 to 1.

For an application to benefit from LetGo, it needs to satisfy the following properties: first, there is a low probability that the key program states are not affected by the failure (indicated by high Continuability), and second, there is a high probability that the program states adjusted by LetGo converge to the original path (indicated by the high Continued_correct and low Continued_SDC).

4.4.4 Fault injection methodology

To evaluate LetGo, we implement a software-based fault injection tool based on *gdb* and *PIN* [2]. Our injector does not require the application's source code, nor does it need the application to be compiled using special compilers or compilation flags.

The fault injection experiments for an application consists of two phases: we first perform *a one-time profiling phase* to count the number of dynamic instructions using the PIN tool. As we assume all dynamic instructions have equal likelihood of being affected by a fault, we use the number of total instructions to randomly choose an instruction to inject a fault for each fault injection run. During this phase, we also profile the number of times each static instruction in the program is executed during the profiling phase so at to be able to inject a fault at the appropriate dynamic instruction. For example, if we choose to inject into the 5th dynamic instance of an instruction, we need to skip the first 4 instances when the breakpoint is reached (using the *continue* command of *gdb*).

During the *fault injection phase*, we then use *gdb* to set a breakpoint at the randomly-chosen dynamic instruction and inject a fault at the instruction by flipping a single bit in its destination register (after the instruction completes). This emulates the effect of a fault in the computational units involved in the instruction.

The profiling phase is run once per application and is relatively slow. The injection phase, on the other hand, is executed tens of thousands of times, and is much faster as it does not involve running the application inside a virtual machine as PIN does. *We perform a total of* $20,000$ *fault injections per application, one per run, to obtain tight error bounds of 0.1% to 0.2% at the 95% confidence interval.*

[2] The tool is publicly available at https://github.com/flyree/pb_interceptor

Table 4.3: Fault injection results for five iterative benchmarks when using LetGo-E. The value for each outcome category is normalized using the total number of fault injection runs for the application. Error bars range from 0.1% to 0.2% at the 95% confidence level.

Benchmark	Finished			Crash			
	Detected	Pass check		Double crash	C-Detected	C-Pass check	
		Benign	SDC			C-Benign	C-SDC
LULESH	0.90%	22.00%	0.13%	25.00%	2.30%	49.50%	0.17%
CLAMR	0.50%	33.30%	0.50%	25.00%	1.10%	39.60%	0.00%
SNAP	0.02%	43.94%	0.01%	20.77%	0.06%	35.20%	0.00%
COMD	1.00%	55.00%	1.10%	18.32%	0.85%	22.13%	1.60%
PENNANT	1.00%	50.00%	2.00%	19.00%	2.50%	22.70%	2.80%
AVERAGE	**0.68%**	**40.85%**	**0.75%**	**21.62%**	**1.36%**	**34.02%**	**0.91%**

4.4.5 Benchmarks

We use six HPC mini-applications namely LULESH [83], CLAMR [114], HPL [119], SNAP [90], PENNANT [89] and COMD [35] (details in Table 4.2). Note that these benchmarks meet the assumption that application-level acceptance checks are well defined/implemented. All benchmarks exhibit convergence-based iterative computation patterns except for HPL, which is implemented with a direct method[3] [47]. Therefore, we separate the results of HPL from others and discuss them in Section 4.7.

Table 4.2 briefly describes the acceptance checks for each benchmark. For *CLAMR, HPL, PENNANT*, we use the built-in acceptance checks (written by the developers), while for *LULESH, COMD and SNAP*, we wrote the checks ourselves based on their verification specifications: Section 6.2 in [82] for *LULESH*, "Verification correctness" section in [36] for *COMD* and "Verification of Results" section in [91] for *SNAP*.

4.5 Experimental results

This section presents experiments that aim to understand whether LetGo is indeed able to continue application execution when a crash-causing error occurs with minimal impact on application correctness and efficiency. The next section evaluates the impact of LetGo in the context of an C/R mechanism.

[3] A direct method computes the exact answers after a finite number of steps (in the absence of roundoff)

4.5.1 Effectiveness of LetGo

We run the fault injection experiments for both LetGo-B (the basic version that uses minimal repair heuristics) and LetGo-E (the version that uses the advanced heuristics described in Section 4.3). Table 4.3 shows the fault injection result for the five benchmarks that use iterative, convergence-based solutions when using LetGo-E. We discuss HPL, a direct method, separately in the discussion section.

We note the following: First, the the average crash rate over all applications is 56%, showing that more than half of the time when a fault occurs the application will crash (i.e., in the table this shows as the sum of values in the four columns under the “Crash" category). Second, with LetGo-E, on average, 62% of these crashes can be transformed to continue running the application to termination (only 38% are double crashes). We first discuss the results for LetGo-E, and then compare these results with those for LetGo-B to understand the effectiveness of the heuristics introduced by LetGo-E. We observe the following:

1. **The ability of LetGo-E to enable continued execution when facing a crash-causing error**: The mean *continuability* for the benchmark set is 62%, which indicates that 62% of the time when the benchmark program receives a crashing signal, LetGo-E resumes the execution and the application completes successfully without crashing again.

2. **LetGo-E is able to convert more than half of the crashes to produce correct results** (and thus possibly offer a solution to lower checkpoint overheads for a long-running applications).

3. **Low rate of undetected incorrect results.** The rate of *Continued_SDC* cases for all benchmarks is on average in the same range as the SDC rate of the unmodified application. For *CLAMR* and *SNAP*, we do not observe new SDCs after applying LetGo-E. Overall, LetGo-E maintains the low SDC rate of the original application (yet it doubles it only 1.6% of the cases did the program produce incorrect results after continuing it with LetGo-E, compared with 0.75% when not using LetGo). We further discuss the impact of the increased SDC rates, and techniques to mitigate it, in Section 4.7.

4. **Continued_detected of the application-level acceptance checks**. The Continued_detected of LetGo-E across the five benchmarks is 2.4%: for our benchmarks, after LetGo-E continues the execution, the application acceptance checks would detect the errors 2.4% of the time - this is slightly higher than the case without LetGo-E.

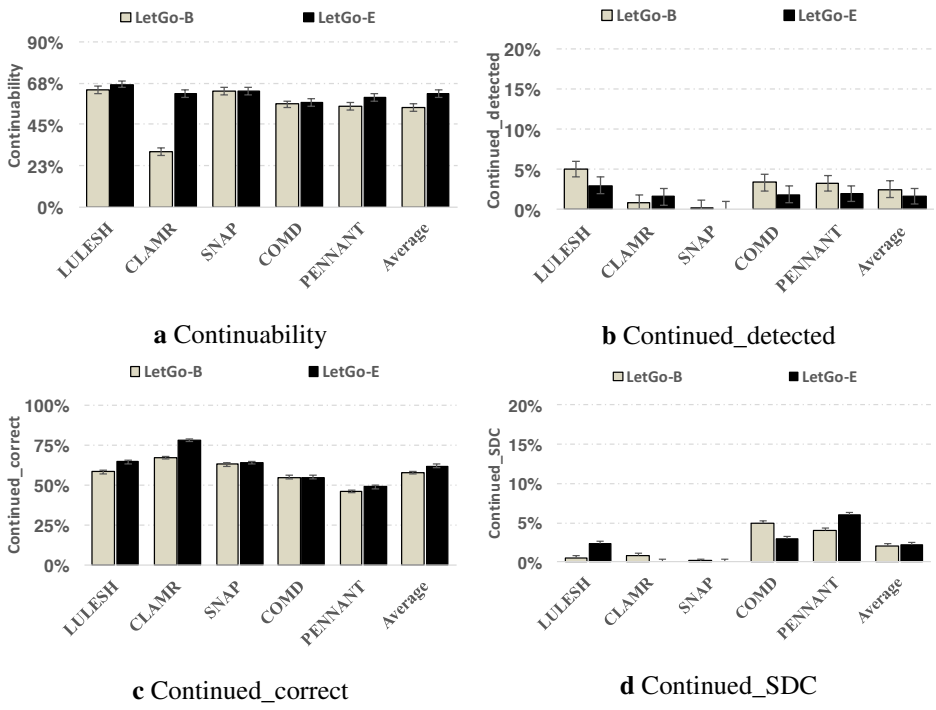
Thus, we find that LetGo-E has a high likelihood to convert crashes into either benign or detected states, while only marginally increasing the SDCs produced.

Figure 4.5 compares LetGo-B and LetGo-E over the four metrics. Figure 4.5a shows that LetGo-E achieves an improvement in Continuability for CLAMR by 32% and for PENNANT by 5% over LetGo-B , but not much for the other benchmarks (considering the error bars). Overall, LetGo-E achieves 14% on average higher Continuability than LetGo-B. Figure 4.5b shows that the Continued_detected declines by 1% from LetGo-B to LetGo-E on average and with only 0.8% increase in *CLAMR*. Therefore, the efficacy of the acceptance checks is not much affected by the heuristics employed by LetGo-E. Figure 4.5c shows that LetGo-E has higher Continued_correct over LetGo-B by 4% on average across all benchmarks. This shows that it allows more crashes to be converted into correct results than LetGo-B. In Figure 4.5d, we find that Continued_SDC ratio for LetGo-E remains the same as that of LetGo-B on average. In Figure 4.5d, we can observe that LetGo-E totally eliminate the SDCs for *CLAMR* and *SNAP*, and has almost the marginally different values of the Continued_SDC metric for all benchmarks - the worst case is 2% higher Continued_SDC faults for PENNANT. Thus, the heuristics used by LetGo-E does not add much to the incorrect executions.

Overall, the heuristics introduced by LetGo-E lead to better continuablility (by about 14%) over LetGo-B for continuing the programs, and producing 5% more correct results than LetGo-B.

4.5.2 Performance overhead

To estimate performance overhead, we experimentally measure the performance LetGo for a single application run outside the context of a C/R scheme. In Section 4.6, we will evaluate the end-to-end impact of LetGo when used in the context of an C/R scheme, that is in the presence of failures and considering C/R overheads.



a Continuability

b Continued_detected

c Continued_correct

d Continued_SDC

Figure 4.5: Comparison of Continuability, Continued_detected, Continued_correct and Continued_SDC between the LetGo-B and LetGo-E. LetGo-E has a higher likelihood of converting crashes into correctly executions for our benchmarks than LetGo-B but no increase in Continued_SDC cases.

There are two source of overhead in LetGo: a). Running the program with *gdb*. b). Adjusting program states after a crash happens. We report the time overhead for each part below. Since LetGo-E is a superset of the operations performed by LetGo-B, we report only the LetGo-E overheads to get the worst-case time overhead.

We first measure the execution time of LULESH with LetGo, under three input sizes. We find that for the three different input sizes, the number of dynamic instructions range from 1 billion to 180 billion, and LULESH with *gdb* exhibits consistently low overhead (i.e., less than 1% compared to running it without *gdb* for each case). We have observed a similar trend for the rest of the benchmarks as well. As explained in Section 4, this is because LetGo neither changes the applications' compilation levels nor does it set breakpoints on the application.

We also measured the time overhead of adjusting program states after a crash by measuring how much time is spent in LetGo-E for each benchmark (i.e., the time spent in step 4 of the Figure 4.3). We find that across all of our benchmarks, the wall-clock time spent in LetGo is roughly around 2-5 seconds, and, as expected, it stays constant when we increase the input size. This time is trivial compared to the overall execution time of most HPC programs. Recall that LetGo takes two actions to adjust the program states: 1). finding the next PC, 2) applying the two heuristics if necessary. As explained in Section 4.3.2 and Section 4.3.3, both actions only need a disassembler to acquire the static instruction-level information of a program - we use PIN. With a more efficient disassembler, the time overhead can be even further reduced. *Thus, LetGo incurs insignificant performance overheads in most cases, and this overhead does not increase with increase in applications' input sizes.*

4.6 LetGo in a C/R context

The previous section demonstrated that LetGo is indeed able to often continue application execution with minimal impact on application correctness and efficiency. This section aims to evaluate the end-to-end impact of LetGo in the context of a long-running parallel application using a C/R mechanism. The main challenge in this evaluation is that there are multiple configuration scenarios that need to be considered, and hence direct measurement is prohibitively expensive. To address this issue, we model a typical HPC system using C/R as a state machine and have built a continuous-time event simulation of the system. This simulation framework enables us to compare resource usage efficiency with and without LetGo. We predict the overall performance gains using LetGo, based on the effectiveness of LetGo estimated with fault injections on an application-specific basis in the previous section. We focus on LetGo-E as we found that it achieves higher Continuability and Continued_correct compared to LetGo-B. In the rest of this section, when we say LetGo, we mean LetGo-E.

Model assumptions. We make a number of assumptions that are standard for most of the checkpointing literature [23, 42, 49, 151]. Our models assume that all crashes are due to transient hardware faults, and hence restarting the application

Table 4.4: The description of parameters of the models

Parameter	Description	Value
T	Checkpoint interval (useful work)	$\sqrt{2 * T\_chk * MTBF}$†
T_r	Time spent for recovery from a previous checkpoint	T_chk
T_chk	Time spent writing a checkpoint	System-dependent
T_sync	Time spent in synchronization across nodes	50%*T_chk and 10%*T_chk
T_v	Time spent in application acceptance check	1%*T_chk
T_letgo	Time spent in LetGo	5s
P_crash	The probability that a application crashes when a fault occurs	Application-dependent
P_v	The probability that an application passes the verification check	Application-dependent
P_v'	The probability that an application passes the verification check with LetGo	Application-dependent
P_letgo	The Continuability of LetGo	Application-dependent
MTBF	Mean time between failure	System-dependent
MTBF_letgo	Mean time between failure with LetGo	MTBF/(1-62%)
MTBFaults	Mean time between faults	System-dependent

† EI-Sayed et al. [49] show that checkpointing under Young's formula achieves almost identical performance as more sophisticated schemes, based on exhaustive observations on the production systems.

from a checkpoint will be sufficient to recover from the crash. In a similar vein, we assume that the checkpointing process itself is not corrupted by a fault. We further assume that the application does not have any other fault-tolerance mechanism than C/R (and LetGo), and that it does not modify its behaviour based on the faults encountered. Finally, when modelling a multi-node platform, we assume the HPC system uses synchronous coordinated checkpointing, which implies that checkpoints are taken at the same time across different nodes via synchronization; and that, when one node crashes, all nodes in the system have to fall back to the last checkpoint and re-execute together.

Parameter description We categorize the model parameters into three classes, summarized in Table 4.4:

1. Configured: The time to write a checkpoint (*T_chk*), and the mean time between faults (*MTBFaults*) are configured by the model users based on the characteristics of the platform and the application;

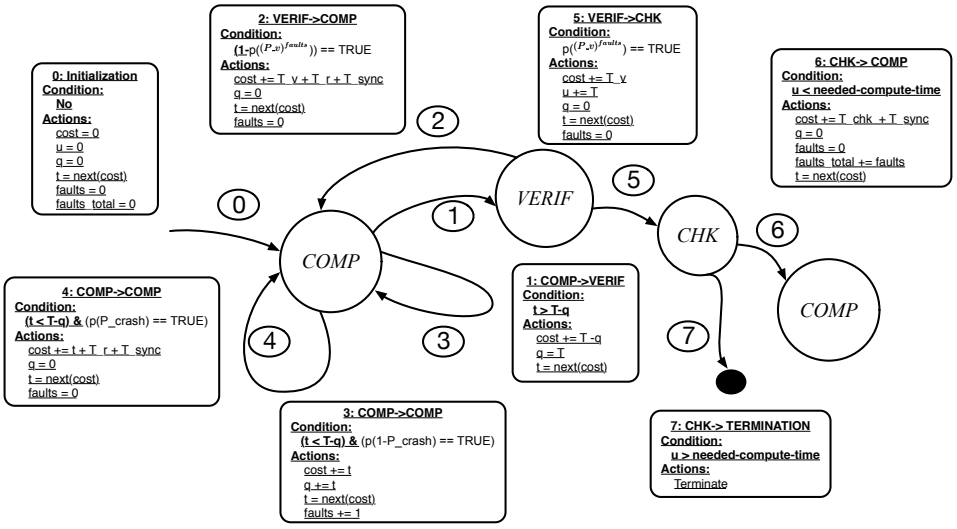
2. Estimated: The probability of a crash after a fault occurs (*P_crash*), the probability that an application passes an application-defined acceptance check (*P_v*, the probability that an application passes an an application-defined acceptance check after LetGo has been used to repair state (*P_v'*), and LetGo Continuability (*P_letgo*) are obtained from the fault injection experiments

on a per application basis.

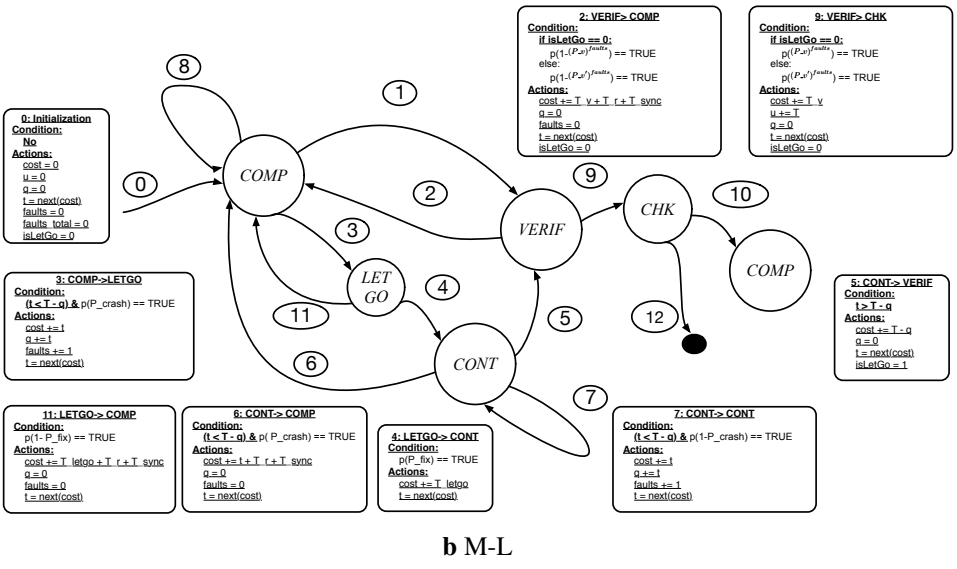
3. Derived: The checkpoint interval T is determined to be a value that maximizes efficiency for the current configuration based on Young's formula [151] (when not explicitly mentioned otherwise in the experiment description). The recovery time (T_r) is, conservatively, chosen to be equal to the checkpoint overhead T_c as we assume the equal write and read speed access to the stable storage (also used in prior work [23]), and neglect the additional coordination overhead. We assume that the time spent for an acceptance check (T_v) is proportional to the checkpoint overhead because the size of the data to check is the same. We use: $T\_v = 0.01 * T\_c$. The overhead for synchronizing multiple nodes (T_sync) to take a coordinated checkpoint is (optimistically, as we do not consider system scaling effects) a constant fraction of the per/node checkpointing time (T_chk). We use two values for the synchronization overhead as 10% and 50% of the checkpointing overhead.

   Finally we can derive mean time between failures (*MTBF*) based on the experiments in the previous section. As we observe from the fault injection experiments in the previous section, on average 56% of faults lead to crashes. Thus for simplicity, we use *MTBFaults* = 2**MTBF*. The C/R scheme with LetGo helps the application avoid crashes, which results in a longer MTBF. We refer to this new MTBF of the system after LetGo is applied as MTBF_letgo, and since the Continuability of LetGo is about 62% on average, we set *MTBF_letgo* equal to *MTBF*/(1-62%).

Model description. The state machine modelling a system that does not use LetGo is depicted in Figure 4.6a and has three states: **COMP**utation, **Ch**ec**k**point, and **VERIF**-ication. In the beginning, the application enters the COMP state for normal computation. A transition is made from the state COMP to VERIF if no crash happens (①), and the acceptance check is applied on the application data/output. If this check passes, a transition is made from the state VERIF to CHK (⑤) and a checkpoint is taken immediately. If the application does not pass the check, it transits from the state VERIF to COMP (②). A transition from the state COMP back to itself occurs when a failure is detected (④), or faults occur when the application is in the COMP state but none of them cause crashes, so that the application



a M-S



b M-L

Figure 4.6: The state machines for the standard C/R scheme (a) and the C/R scheme with LetGo (b). The black circle represents the termination state of the model. We use u/cost to represent the efficiency of the model. *t: time interval till the next fault; cost: accumulated runtime; u: accumulated useful work; q: accumulated useful work within the current checkpoint interval; faults: number of faults that did not lead to crashes since the last checkpoint; faults_total: total number of faults that did not lead to crashes; isLetGo: a flag that indicates that if the P_v' is chosen or not*

stays in the COMP state and the number of faults will be increased (③). When faults are accumulated in the system, the probability that the application passes the verification check is modeled as $(P\_v)^{faults}$, given the assumption that hardware transient faults occur as independent events.

Figure 4.6b illustrates the model for the C/R scheme when using LetGo. The state machine contains two more states: "LETGO" and "CONT"inue. Due to space limitations, we emphasize here only the transitions related to the new states. When there is a failure (i.e., crash) occurring during the computation (i.e., the application stays in the COMP state), a transition is made to the LETGO state (③). The application moves from the LetGo state to CONT if LetGo continues the execution of the application (④), otherwise, the application transits back to the COMP state (①). While the application stays in the CONT state, the occurring fault can either cause another crash and make the application transit to the COMP state (⑥), or not cause a crash and make the application proceed to the state VERIF. The "isLetGo" flag is set for choosing the different base probabilities (P_v or P_v') that the application passes the verification check (⑤). The base probability is used in the conditions of the state transitions ② and ⑨. The actual probability is then calculated using $(P\_v)^{faults}$ or $(P\_v')^{faults}$ in ⑧ and ⑦.

Evaluation metric. The goal of the simulation is to understand the impact of LetGo on resource usage efficiency in the context of a long running application using a C/R scheme in the presence of faults. We define resource usage efficiency as the ratio between the accumulated useful time and the total time spent (i.e., u/cost). To evaluate the efficiency of both setups (with and without LetGo), we perform simulations for configuration parameters corresponding to different benchmarks and different platforms.

Choice of parameters. We justify the choices for the checkpointing overhead, and the MTBF. We first discuss the checkpointing overhead: the time spent to write a checkpoint to the persistent storage depends on the characteristics of the hardware. For more advanced hardware, the checkpointing overhead becomes less significant. However, on one side, advanced hardware support such as burst buffers represent additional costs. To the degree these are added to reduce checkpointing overheads, a checkpointing scheme with lower overhead would enable provision-

ing systems for lower overall cost. On the other side, even in the presence of burst-buffers, checkpointing is still a major bottleneck on deployed systems as our simulations show. We use two criteria for choosing the checkpoint overheads. Here are two data points that justify our choice of parameters to seed our simulations:

- *Back-of-the-envelope calculation*: For each checkpointing overhead value we pick for our simulations we assume that the system-level checkpointing writes some portion of the main memory to the persistent storage. A modern HPC node normally features 32 to 128GB memory. For a burst buffer implemented with SSD, the average I/O bandwidth for write is around 1GiB/s, and the peak value is 6GiB/s [17]. For spinning disks, the I/O bandwidth is usually around 50MiB/s to 500MiB/s. As a result, our choices for the checkpoint overhead of (12s, 120s or 1200s) respectively represent, (i) a well provisioned system using burst buffers, (ii) and averagely provisioned system (e.g., using burst buffers, or compression and spinning disks); and (iii) a naive, under-provisoned system. We note that a similar set of values is also used in prior work [23, 49].

- *Future system requirements*: The Alliance for application Performance at EXtreme scale (APEX) 2020 document [88] requires that the systems delivered in 2020 have a single job mean time to interrupt of more than 24 hours, and for a delta-checkpoint scheme (i.e., the time to checkpoint 80% of aggregate memory of the system to persistent storage), the time for writing the checkpoint to be less than 7.2 minutes (432s). This suggests that our parameters are in the same ballpark as those of the current and future systems.

Along similar lines, we derive *MTBFaults* for existing systems from previously reported studies: we start with the system presented by [23] as a baseline. This system contains about $10,000$ nodes, and usually experiences around 2 failures per day [21] (*MTBF* of 12 hours). Based on this data, we scale *MTBF* for larger systems, and also consider systems with lower node-level reliability.

Running the Simulation. We assume that the hardware transient faults hitting the system are governed by a Poisson process and we generate a random time sequence for the occurrences of the hardware faults. Then, we seed the models

with various sets of parameters and run the simulation over the generated sequence for a long simulation time (10 years), to determine the asymptotic efficiency value for each benchmark.

Experimental Results. We first show the efficiency for the C/R system with and without LetGo under different checkpoint overheads. We take the system that has a MTBFaults of 21600 seconds (i.e., $MTBF = 12hours$) and a synchronization overhead of 10% of the checkpoint interval as an example.

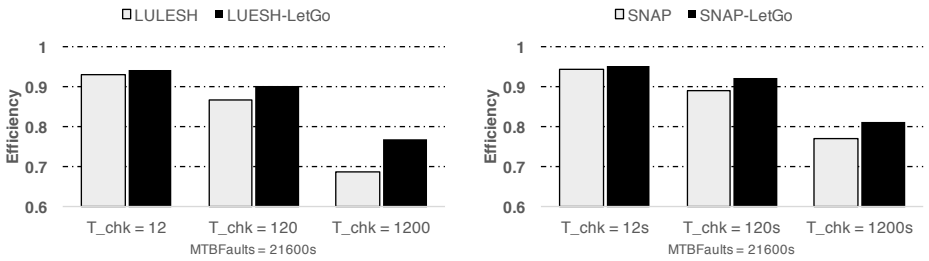
The efficiency improvement enabled by LetGo is between 1% to 11% across our benchmarks (absolute values, the relative values are much higher (nearly 20%)). As the checkpoint overhead scales up (from 12s to 1200s), the efficiency gain increases for all applications, while, at the same time, the absolute efficiency per application decreases. Figure 4.7 shows the applications that have the highest and lowest efficiency gains respectively, *LULESH* and *SNAP*, when the checkpoint overhead is 1200 seconds. This trend is consistent across all applications, and across different synchronization overheads. Our results thus show that LetGo offers significant efficiency gains.

We then scale the system from $100,000$ nodes to $200,000$ and $400,000$ nodes. Scaling results in lower MTBF for the whole system: 6 and 3 hours. Again, we use two benchmarks namely *CLAMR* and *PENNANT* as examples, shown in Figure 4.8. As the scale of the system increases, the efficiency of the system both with and without LetGo decreases, as expected. Importantly, the rate of decrease of efficiency is lower for the system with LetGo than without. This trend is consistent with all our benchmarks, suggesting that LetGo offers better efficiency as the system scale increases.

4.7 Discussion

We address a number of interrelated issues.

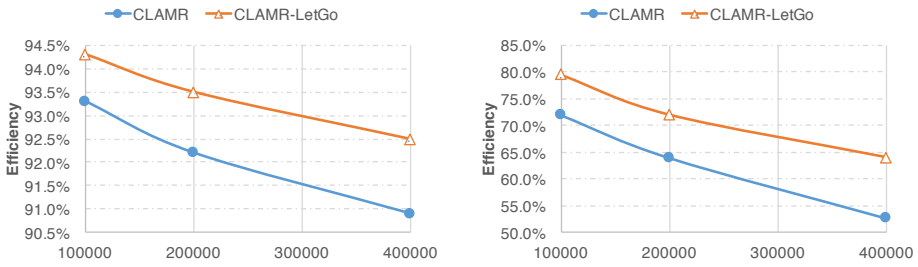
How effective is LetGo when used for applications that do not use convergent methods? We have evaluated LetGo on five benchmarks that use convergent methods. We have also evaluated LetGo on *HPL*, which is a linear algebra solver that uses a direct method. Our fault injection experiments show that, without the presence of LetGo, 34% of faults lead to crashes, 38% lead to incorrect output de-



a Efficiency of LULESH with and without LetGo

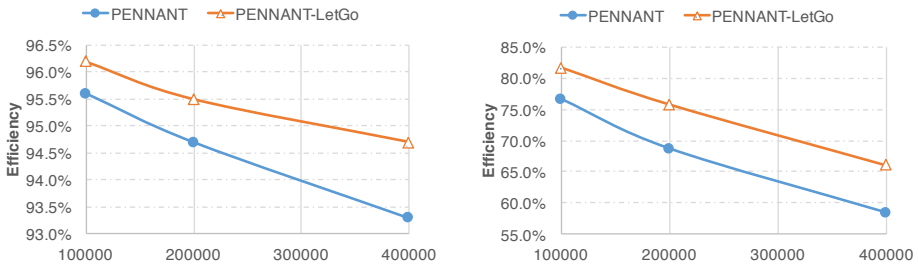
b Efficiency of SNAP with and without LetGo

Figure 4.7: Efficiency with and without LetGo under different checkpoint overheads for LULESH and SNAP.



a The efficiency trend of CLAMR when T_chk = 12s

b The efficiency trend of CLAMR when T_chk = 1200s



c The efficiency trend of PENNANT, T_chk = 12s

d The efficiency trend of PENNANT, T_chk = 1200s

Figure 4.8: The trend of the efficiency for the C/R scheme with and without LetGo when the system scales from 100,000 nodes to 200,000 nodes and 400,000 nodes

tected by the HPL residual check, about 1% lead to SDCs, and 27% lead to correct output. We find that the application-level acceptance checks are much more selective than for the applications in our initial set, and this would make HPL a good candidate for use with LetGo. However, while the crash rate is still high (34%), it is lower than that of the other applications, where it was around 60% - this would potentially reduce the impact of LetGo. When using LetGo with HPL, we obtain around 70% Continuability and 2x increases in SDC rate (from 1% to 3%). To understand the overall performance, we run the simulation of a C/R scheme with the potential results from applying LetGo-E to HPL. Our simulation results show that the efficiency of the standard C/R scheme applied to *HPL* is around 40%, and LetGo-E only marginally improves efficiency. Thus, LetGo by itself is not a good fit for applications like HPL - other error correcting mechanisms (e.g. ABFT) may be needed for such programs.

Determining when/how to use LetGo An operator will decide whether to use LetGo depending on a mix of factors: i) how frequently the system experience hardware faults that crashes the application, ii) what is the likelihood of the application to experience additional SDCs given the use of LetGo, iii) what is the checkpoint overhead for a specific C/R scheme for that application and deployment, iv) what is the acceptable increase in the SDC rate. It is reasonable to assume that the operator has (an approximation for) some of the information above as she needs to configure the checkpoint interval when LetGo is not used. Additionally, the operator needs information to estimate the increase in SDC rate due to LetGo. A large characterization study with applications from multiple categories that extends the preliminary data provided in this paper is necessary to provide these estimates.

Moreover, since LetGo allows applications to continue with errors, it may be possible to use it for application-specific (re)configuration of the hardware fault tolerance mechanisms to enable energy savings.

4.8 Summary

This chapter answers the first part of RQ2. We demonstrate that it is feasible to continue an HPC application's execution rather than to terminate it on crashes due to transient hardware faults affecting computational components. We have imple-

mented the LetGo prototype, which monitors the execution of an application and modifies its default behavior when it receives a termination-causing OS signal. When used in the context of a C/R scheme, LetGo enables sizable resource usage efficiency gains. More specifically, for a set of HPC benchmarks, LetGo offers over 50% chance that the application can continue and produce correct results without performing a roll-back/recovery. We evaluate the impact of LetGo for long-running applications that use C/R and find that LetGo enables sizeable efficiency gains. The efficiency gains increase with both the system scale and checkpointing overheads, and thus suggesting that LetGo will likely be even more important for the future large-scale HPC applications.

Chapter 5

BonVoision: Leveraging Spatial Data Smoothness for Recovery from Memory Soft Errors

In Chapter 4, we presented *LetGo*, which optimizes the standard C/R system to support the roll-forward scheme and demonstrate the feasibility and performance of LetGo over failures due to errors affecting computational components. This chapter focuses on the second part of the RQ2 (Chapter 1.4) and investigates if the roll-forward recovery scheme can be applied to the C/R system when the application encounters memory detectable but uncorrectable errors (DUEs).

In this chapter, we start from two high-level observations made on certain classes of HPC applications (e.g., stencil computations on regular grids or irregular meshes): First, these applications display a property referred as *spatial data smoothness*: i.e., data items that are nearby in the application's logical space are relatively similar; Second, since these data items are generally used together, programmers go to great lengths to place them in nearby memory locations to improve the application's performance by improving access locality. Based on these observations we explore the feasibility of a roll-forward recovery scheme that leverages spatial data smoothness to repair the memory location corrupted by a DUE and continues the application execution. We present BonVoision, which intercepts DUE events and analyzes the application binary at runtime. BonVoision identi-

fies the data elements in the neighborhood of the memory location corrupted by a memory DUE and uses those data to repair the memory corruption. Our evaluation demonstrates that BonVoision is: *(i) efficient*—it incurs negligible overhead; *(ii) effective*—it is frequently successful in continuing the application with benign outcomes; and *(iii) user friendly*—it does not require programmer input to expose the data layout, nor does it need access to source code. We show that using BonVoision can lead to significant savings in the context of a checkpointing/restart scheme by enabling longer checkpoint intervals.

5.1 Introduction

Current memory devices (both DRAM and SRAM) are subject to increasing error rates [2, 3], due to the shrinking feature sizes, reduced supply voltages, lower refresh rates, and the overarching need to reduce the energy footprint. This scenario is exacerbated in a HPC context, where a large application memory footprint and system scale make it more likely that the system experiences memory errors. For example, Martino et al. [21] report that on the Blue Waters supercomputer, on average 160 memory errors per hour are observed across half of the total Blue Waters nodes. A recent study [132] predicts that the uncorrected error rate for a future exascale system could be 3.6–69.9$\times$ the uncorrected error rate currently observed in existing supercomputers.

The most common kind of error correcting codes (ECC) used to protect against these errors are single-error correction double-error detection (SECDED): they correct single-bit flips, but can only detect (and not correct) double-bit flips. More advanced ECC schemes such as chipkill can extend the capacity of the correctable bits in memory [46, 98], but they generally incur high cost in terms of energy and performance. Therefore, a portion of the memory errors remain detected but not corrected. These Detectable but Uncorrectable Errors are known as DUEs. While DUEs occur less frequently than correctable errors (e.g., Martino et al. [21] observe that 29.8% of memory errors are DUEs without chipkill, while with chipkill this percentage drops to less than 1%), they can lead to critical consequences for long-running applications. when a DUE occurs, the CPU raises a machine check exception (MCE) to the Operating System (OS), which typically crashes the appli-

cation. Higher crash likelihood directly impacts time-to-solution and the cost of mitigating strategies (e.g., checkpoint frequency).

Prior studies [9, 106, 141] have observed that many applications inherently mask some errors, and still produce acceptable results. This observation motivated us to first investigate whether the fail-stop model overprotects the application in the context of HPC applications. We therefore asked whether *is it feasible to simply ignore DUEs* and continue operating. Unfortunately, our fault injection experiments (Section 5.3.1) revealed that while not all DUEs lead to failures, the rate of subsequent failures is too high for this technique to be practical. Therefore, we develop heuristics to repair the state affected by the DUE and enable forward application progress.

The key insight that guides the design of our repair mechanism is that some classes of HPC applications (e.g., stencil applications) are likely to have spatial data smoothness [15]. This is because the physical phenomena often modeled by HPC applications exhibit inherent continuities in their modeled physical space e.g., temperature, pressure, or velocity for a climate modelling application. Further, the data structures used by these applications tend to keep physically close data points close together in memory as well to exploit locality. We leverage these two observations to propose a repair scheme for DUEs in HPC applications that is: *(i) effective* - it is frequently successful in continuing the application with benign outcomes, *(ii) efficient* - it incurs negligible performance overhead, *(iii) automated* - it does not require additional programmer involvement to expose the data layout, and *(iv) practical* as it neither requires access to the application's source code, nor runtime instrumentation.

While conceptually elegant, there are two challenges in designing a repair technique based on the above intuition and offering the above characteristics. The first challenge is to identify the application-level data *element* corresponding to an observed DUE (recall that the ECC memory would only identify the faulty memory *word* in case of a DUE), as HPC applications are deployed as binaries on the target system, and the source code is not available at runtime when faults occur. Even if the source code was available, it is not straightforward to infer the data layout and the nearby data elements in the application's physical space without relying on either programmer annotations (which would imply a significant burden), or on

runtime instrumentation (which would slow down the execution). After identifying the application-level data element corresponding to the fault, the second challenge is to decide on the value to use for repair by finding uncorrupted elements that are spatially close to the location of the fault.

We present BonVoision, a lightweight and automated approach for DUE repair. BonVoision requires neither programmer annotations nor instrumentation of the application code. To address the first challenge, we propose a set of heuristics to infer the neighboring application-level data elements of a faulty word, based on runtime analysis of the application's binary. To address the second challenge, we propose a number of repair heuristics, and experimentally evaluate them.

BonVoision can be used in conjunction with classic checkpoint-restart (C/R) schemes typically deployed on HPC systems. On a DUE, a C/R system rolls-back the application and recovers it from a previously collected checkpoint. With BonVoision, the system will first attempt to reconstruct the application state and roll-forward (i.e., continue execution). Only in case of an another failure (either a crash or incorrect state detected by application-level checks), will it attempt to recover from the checkpoint. From the perspective of a C/R scheme, the impact of using BonVoision is similar to increasing the mean time between failures (MTBFs), leading to lower checkpointing overheads and faster execution times.

This chapter makes the following contributions:

- Introduces BonVoision, a software methodology that leverages data spatial similarity to repair memory DUEs. (Sections 5.3 and 5.4)
- Evaluates and compares the end-to-end application outcomes for three repair schemes: *(i)* using BonVoision's stride-based values, *(ii)* 0s, and *(iii)* random values. Results show that BonVoision repairs lead to 0.8 - 2.5x more benign outcomes, and only marginal increases in Silent Data Corruption (SDC) outcomes compared to using 0 values and random values for the repairs. (Section 5.6.1)
- Proposes an enhancement to BonVoision to use an online classifier, to predict the outcome of a individual repair. We demonstrate that the optimized BonVoision (called BonVoision-E) leads to a significant improvement in repair effectiveness (on average 23% higher rate of benign-only outcomes). (Section 5.6.2)

- Evaluates the overall BonVoision efficiency with different program input sizes and number of MPI processes. We find that BonVoision incurs a negligible overhead (about 6 milliseconds on average per corrected DUE). (Section 5.6.3)
- Demonstrates the performance gains enabled by BonVoision-E when used in the context of a C/R scheme. (Section 5.6.4)

5.2 Background

ECC Memory. Dynamic random-access memory (DRAM) may incur single- or multi-bit flips due to, for example, alpha particle strikes, background radiation, or neutrons from cosmic rays [115]. ECC uses extra memory bits to detect and correct bit-flip errors. SECDED, the most widely-used ECC scheme, uses 8 redundant bits for every 64 bits. More powerful schemes such as chipkill ECCs can correct more errors at a higher cost. While generally applicable, we demonstrate BonVoision in the context of the SECDED scheme. A memory location is checked for an error when an application reads it or during memory scrubbing, which periodically performs memory reads and writes. The memory controller computes the checksum for the data on every read, and compares it with the stored ECC bits. For SECDED, when more than one bit is flipped, the controller logs the event as a DUE and passes it to the processor.

MCE for memory DUEs. On x86 processors, the most common mechanism to deal with a memory DUE is to generate a machine check exception (MCE). In the case of DUEs, MCE encapsulates the DUE-specific hardware information (e.g., socket, channel, DIMM, etc). In some scenarios, the computed ECC syndrome and the value of the corrupted data are available, but we do not assume that this information is available. MCEs are generally directly handled by the OS. For example, Linux provides an "MCE tolerance level" option for users to determine how the OS should react to MCEs. By default the kernel would either “panic” or deliver a SIGBUS exception to the applications on uncorrected errors, and log the corrected errors. We assume that only the application experiencing the DUE receives the SIGBUS and terminates (this covers most cases). In user space, mcelog [87] is used for Linux to decode and consolidate MCE messages.

Checkpoint/Restart. C/R schemes [54, 137] are frequently used to recover

from crash (i.e., fail-stop) failures. A C/R scheme consists of two main operations: *(i)* it periodically saves program state to non-volatile storage (i.e., it 'checkpoints' the data necessary for the application to recover); *(ii)* if a crash failure occurs, it retrieves one of the checkpoints (often the latest one) to re-create the current state, and continues the application. Many studies have focused on determining the "optimal" check-pointing interval, which optimizes application's expected runtime. Young et al. [151] and Daly et al. [42] present analytic models to find the optimal checkpoint interval as a function of *(i)* the time to generate the checkpoint (i.e., the checkpoint overhead), and *(ii)* the mean time between failure (MTBF). EI-Sayed et al. [51] empirically show that checkpointing guided by Young's formula achieves almost identical performance as more sophisticated schemes, based on exhaustive observations on production systems. BonVoision essentially attempts to convert the fail-stop DUE failures into continued and successful executions. Since the application fails less often, its MTBF is increased, reducing checkpointing overheads when BonVoision is used.

5.3 Motivating studies

Our approach to repair memory soft errors that result in DUEs is based on two observations: *(i)* simply ignoring the DUEs does not work for HPC applications, and *(ii)* some classes of HPC applications exhibit spatial data smoothness that can be leveraged for efficient DUE repairs. We describe the experiments based on which we make the two observations. The benchmarks used and the other experimental details are described in Section 5.5.

5.3.1 Does ignoring DUEs work?

A typical application's virtual memory space is organized as segments such as stack, heap, etc. Typically, the heap segments consumes the largest size of the virtual memory space, and hence if a DUE occurs in an application's memory, it most likely occurs in the heap segment. To verify this, we have profiled the use of stack and heap segments for the HPC applications in our benchmark and found that the heap segments indeed dominate the memory usage.

We inject DUE errors into the heap segment, and observe their impact on appli-

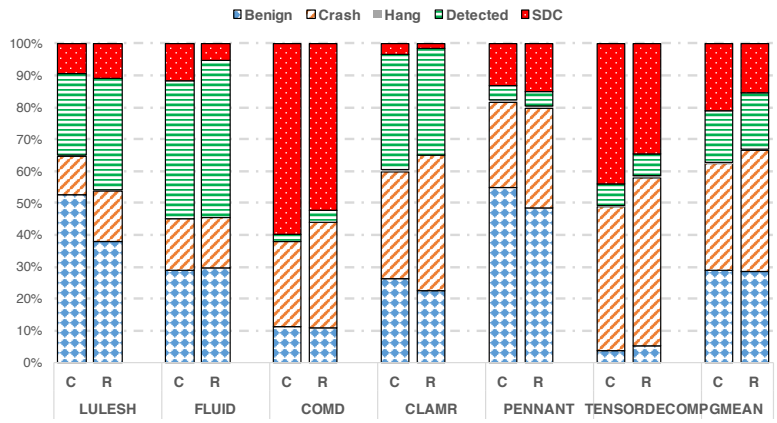


Figure 5.1: Error injection results for six HPC applications. C means consecutive bits, while R means random bits.

cation outcomes. We perform the following steps in the error injection campaign: *(i)* we profile each application to find out how many dynamic memory read instructions access the heap segment – these are our *fault injection sites*. We pick memory read instructions only as they are the sole sources that trigger DUEs considered in this paper [1]; *(ii)* we randomly (uniform probability distribution) select an instruction from the set of all error injection sites, and flip[2] two bits in the memory data accessed by the selected instruction before the instruction executes. We consider two error modes: the flipped bits are consecutive or at random; and *(iii)* we allow the program to complete execution and study its outcome.

The outcomes are classified into five categories: (1) benign: correct output, identical to that of a 'golden run' - a run with no fault injected, (2) detected: the application's own correctness checks detects some violation of the properties in the results (see section Section 5.5 for details on correctness checks), (3) Silent Data Corruption (SDC): the output data is different from that of the 'golden run', (4) crash: exceptions, and (5) hang: the program does not stop for a long period of time. We perform 5,000 double-bit fault injection runs for each application to obtain a statistically significant estimate of outcome probabilities: the 95% confi-

[1]Memory scrubbing is likely to occur during idle period to prevent decreasing performance, so we do not consider it as the source of the DUEs in this study

[2]We modify PINFI [145], a PIN-based instruction-level fault injection tool, to inject 2-bit-flip errors into memory on the memory instructions that read from the heap.

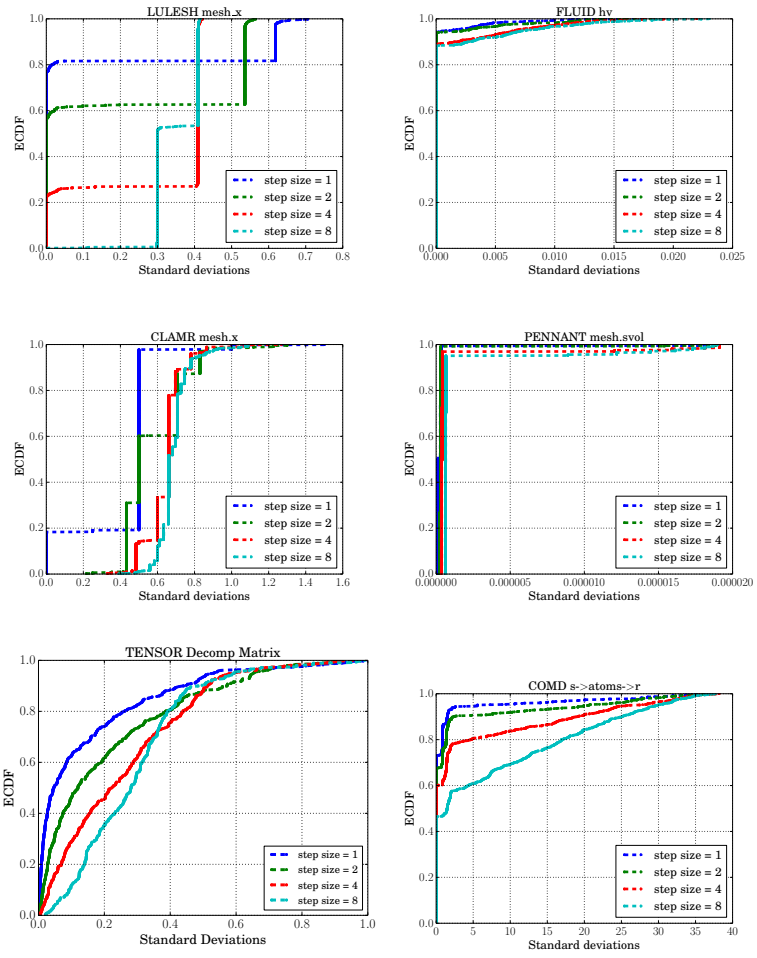


Figure 5.2: Examples of the ECDF of the standard deviations for six benchmarks (LULESH, FLUID, CLAMR (top-row), PENNANT, TENSORDECOMP, and COMD (bottom row). The corresponding data structures are indicated on the top of the figures.

dence interval yields error bars of around 1%.

Figure 5.1 shows the results of the error injection experiments. On average, when injecting into consecutive bits (left bars), 30.5% of the injected errors lead to benigns, 26.4% to crashes, 19.3% to detected cases, 0.3% to hangs and 23.5% to SDCs. The errors injected into random bits (right bars) cause slightly fewer benigns (25.4%) and SDCs (20%) but higher crashes (32.2%) and detected cases (22%). Overall, the results are independent of how we choose the 2 bits. We also observe that not a single benchmark exhibits a high benign rate combined with a low SDC rate. Thus it is not practical for these applications to just ignore memory DUEs.

5.3.2 Exploring spatial data smoothness

This paper makes two assumptions about smoothness : *(i)* that it exists in the application's logical space; and *(ii)* that data structures preserve it when mapped to actual memory. To verify these assumptions together, we quantitatively estimate the spatial smoothness of an application's core data structures (i.e., the data structures that model the physical space the application works over) once they are mapped to memory. If spatial smoothness exists, then the standard deviation of the differences between consecutive elements in the memory layout of the data structure is small. Therefore, given a data structure and a sets of elements S, we form a new set $diff(S) = \{ d_i | d_i = s_{i+1} - s_i, \forall s_i \in S \mid \}$ by first computing the difference between successive elements, and then computing the standard deviation over $diff(S)$ - the smaller the standard deviation, the smoother is the space. To study how spatially far from an anchor element smoothness extends, we vary the size of the set S. More concretely, we pick in S the data elements at distance (in stride) $\pm 2^i$ for $i = 1..K$ around an anchor element.

To estimate smoothness for our benchmark applications, we build a custom profiler that pauses the application execution at multiple random time points and, at each pause, randomly selects a number of anchor elements from the core data structures of the application. It then constructs the set S and applies the metric described above. Note that since the profiler works directly at the data structures, it makes no mistake in finding the neighbouring elements of that data structure. In

our experimental settings, the profiler pauses each application 10 times and picks 1,000 random anchor elements during each pause.

Figure 5.2 shows the empirical cumulative distribution function (ECDF) of standard deviations (SD) under different step sizes (values for K above). We present the ECDFs sampled from one of the core data structures of each application as an example. We make two main observations: *(i) generally the values for SDs are low, indicating smoothness.* Around 80% to 100% SDs are close to 0 in LULESH, FLUID and PENNANT; 80% of SDs are less than 0.3 in TENSORDECOMP, and nearly 100% of SDs are less than 0.5 in CLAMR; for COMD, around 75% of SDs are close to 0, but the remaining 25% vary in a large range. *(ii) comparing the SDs across different step sizes, step size 1 always offers the smallest SDs for all applications*. This indicates that BonVoision should use the closest neighbours for repair. In summary, we observe that significant spatial data smoothness exists across the elements of the core data structures.

5.4 Design and implementation of BonVoision

Overview. Section 5.3.2 shows that application-level data structures exhibit spatial data smoothness. Armed with this information, we explore DUE repair strategies that harness smoothness and use the data values from nearby locations to propose values used for repair. BonVoision embodies the results of this exploration.

BonVoision consists of three main components (Figure 5.3): *(i)* information parsing, *(ii)* stride speculation, and *(iii)* repair write-back. BonVoision is invoked upon a DUE signal. The information parsing component parses the information from the DUE signal handler and feeds it to the stride speculation component. This component runs a set of heuristics and outputs the possible stride. Finally, the repair write-back component uses the stride to access the neighboring elements, computes their average value, and writes that value back to the corrupted memory location. The rest of this section focuses mainly on the stride speculation component, as it is the most challenging part. We first explain the challenges of stride speculation, followed by a synthetic study we designed to devise heuristics. Finally, we present the heuristics for stride speculation.

Stride speculation challenges. We explain the challenge for stride speculation

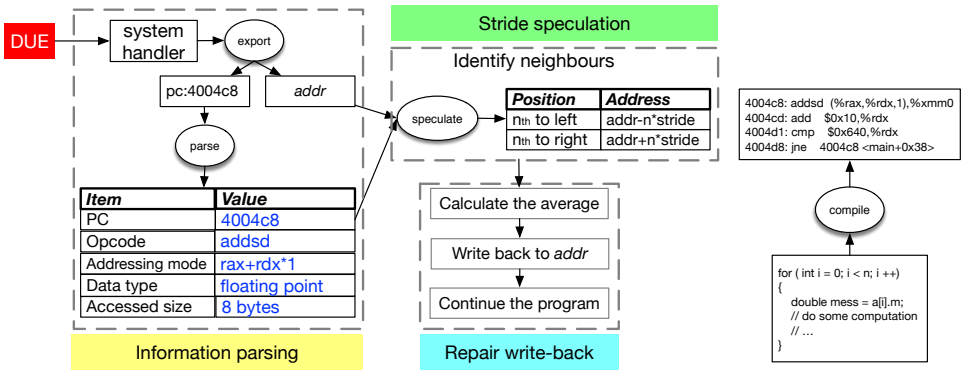


Figure 5.3: Illustration of BonVoision components, the information available upon a DUE and why this information is insufficient to determine the a data structure's stride.

Table 5.1: The patterns and stride observed for accessing to the elements of the different data structures. Reading elements from Foo3 and Foo5 results in the same addressing mode as Foo2.

Data structure	Compilation option	Accessed elements	Addressing mode	Register for address shifting	Instruction(s) modify the register	Stride
Foo1	O3	bar2	(base)	rdx	add $(0x8), %rdx	8
Foo2	O3	bar3	displacement(base)	rdx	add $(0x10), %rdx	16
Foo4	O3	bar1.bar2	displacement(base,index,scale)	rdx*4 (scale)	add $0x1, %rdx	4
Foo4	O0	bar1.bar2	displacement(base)	rax, rdx	shl $0x4, %rdx add $rdx, %rax	16

with an example: the program iterates over the array and reads its elements for the later computation. Figure 5.3 shows the C code and corresponding x86 instructions (compiled with gcc and -O3). Suppose a memory DUE occurs when the memory read instruction (*4004c8*) is executed, and a recovery handler is in place. For the recovery strategy to exploit data smoothness and repair the corrupted location using information from nearby elements, it needs to identify the neighbours of the data residing in *addr*. At the source code level, this task is trivial: knowing the address of an element &a[k].m = *addr*, its neighbours should be addressed as &a[k-1].m = *addr* - sizeof(a[k]) and &a[k+1].m = *addr* + sizeof(a[k]), where in our example the sizeof(a[k]) = 2*sizeof(double). However, it is challenging to infer this at the assembly code level, with no support from the programmer or other program instrumentation. This is because the only information available to the recovery handler is the assembly instruction itself (referred to as *pc* in Figure 5.3) and the

corrupted memory location *addr*. We need to speculate the stride based on this information.

5.4.1 Synthetic study

We conduct a synthetic study that explores several typical scenarios to find common patterns for inferring the stride.

Addressing modes. The addressing modes in x86 assembly consist of several segments including base, index, displacement and scale, where the base and index are x86 registers, and displacement and scale are immediate numbers (scale can be 1,2,4 or 8). Depending on the compilation options, program semantics and other factors, the compiler needs to choose a particular addressing mode for accessing the elements within a data structure. For example, the instruction *4004c8* in Figure 5.3 contains no displacement segment, while other memory read instructions may contain no index or scale segments, no base segments, etc.

Observations. We made a preliminary observation on the assembly code in Figure 5.3: the register *rdx* is the index register of instruction *4004c8*, and gets added by 0x10 (which is the size of the data structure *Atom*) every time the instruction *4004cd* gets executed. This observation leads to two insights: 1) there is an observable update on the instruction's register to locate different elements in the data structure - we call this register the **stride register**; 2) the value of the stride can be inferred based on the modification to the stride register. Therefore, it is possible to infer the stride from the corresponding address calculation operations on the memory instruction.

For a more comprehensive understanding of how a program calculates the addresses for different elements in a data structure, we study several typical data structures in C/C++: vector, array, array of structures, structure of arrays, namely foo: vector<double>; foo1: {int bar1; double bar2}; foo2:{float bar1; float bar2;}; foo3:{foo2 *bar1; int bar2;}; foo4:{foo2 bar1[n]; int bar2[n];}; [3]. We implement a simple program that consecutively accesses elements of each data structure, and examine the program's assembly code for the instructions associated with the address calculations. Table 5.1 summaries our findings. There are two main insights:

[3]The allocation, initialization, access, and free operations, implemented in the driver code, are not shown here due to space limits.

(i) both the base and the index registers can participate in the element's address calculations, *(ii)* while the compilation options (i.e., O0 and O3) generate distinctive addressing modes, the stride information can still be inferred from the corresponding address calculation instructions.

5.4.2 Heuristics

Based on the insights from the above study, we design 2 heuristics to speculate the stride information from the program's assembly code.

Basic heuristic. We observed that many core data structures are comprised of basic-type elements such as double, float, integer etc. For those data structures, the stride information can be directly inferred by the accessed size from the memory read instruction. We call this heuristic the size-based approach.

Advanced heuristic. For more complicated cases, we leverage the insight from the above study, that the stride information may be found by tracing the modification to the stride register, i.e., either the base or the index register used for accessing different elements in a data structure. Thus, we propose the following heuristic to look for the instruction/instructions that write to the stride register, and extract the value in the operands as the indicator of how consecutive elements are accessed. The heuristic contains two main processes:

Backward and forward slicing. The goal of this process is to traverse all the instructions that use the stride register as the destination register at the assembly level, and find which one(s) contain the information of accessing to the neighbouring elements. As observed in the the synthetic study, these instructions can be either before or after the memory instruction, so both directions need to be searched. In practice, when a instruction modifies the register with an intermediate number, or a *lea* (load effective address) instruction modifies the register, the stride is inferred from the instruction(s), and the process terminates. The process also terminates when all the static instructions in the same function are iterated, or there is a memory read instruction that overwrites the register with the memory data. In these cases, BonVoision shifts to the basic heuristic and uses the accessed size as the stride.

Handling transitive relations. There are cases where the data dependency

Table 5.2: Benchmark description, including SDC determination approach and application correctness check criteria

Application	Domain	Main data struct	LOC	# dynamic mem read inst (10^9)	Application data used for acceptance check	Criteria for acceptance check
LULESH	Hydrodynamics	Struct mesh	2.9k	1.0	Mesh points	Same # of iterations, Energy conservation, Measures of symmetry: smaller than 10^{-8}
CLAMR	Hydrodynamics	Adaptive mesh	60.1k	6.4	Mesh points	Threshold for the mass change per iteration
PENNANT	Hydrodynamics	Unstruct. mesh	5.0k	9.3	Mesh points	Energy conservation
COMD	Molec. dynamics	Grid	5.6k	8.6	Atom properties	Energy conservation
FLUID	Fluid dynamics	Newtonian particle grid	5.7k	1.8	Particle properties	Particles' positions, velocities, bounding boxes with absolute tolerance
TENSORDECOMP	Sparse tensor decomposition	Multi-D Array	10.2k	0.2	Output matrices	Decomposition fit > 0.65 and delta < 10^{-3}

propagates to another register: when the backward or the forward slicing process identify it as (i.e. the transitive register) being used to update the root register of the current slices, the modification on the new transitive register needs to be considered. When such relation is captured, the current slice needs to be discarded, and new backward and forward slicing processes based on the transitive register are launched. It might be possible for BonVoision to encounter one or more levels of transitive relations. Therefore, BonVoision sets the limit to the depth of the transitivity, based on the observations on various applications' assembly codes.

Algorithm 5 shows how the two heuristics are implemented in BonVoision. It

Algorithm 4 Backward instruction slicing

```
1:  procedure BACKWARD_SLICING(reg, inst)
2:      inst_b ← inst
3:      while inst_b ≠ first inst in the function do
4:          ▷ the function contains the inst
5:          inst_b ← prev(inst_b)
6:          if reg == dest_reg(inst_b) then
7:              ▷ dest_reg outputs the destination register of a inst
8:              if has_immediate(inst_b)oris_lea(inst_b) then
9:                  if is_lea(inst_b) then
10:                     scale ← parse_lea(inst_b)
11:                     if scale == 0||scale == 1 then
12:                         index_lea = parse(inst_b)
13:                         ▷ goes one more level of transitivity
14:                         backward_slicing(index_lea)
15:                         forward_slicing(index_lea)
16:                     else
17:                         stride ← scale
18:                 else
19:                     stride = handle_immediate(inst_b)
20:             else
21:                 ▷ Transitive relation begins
22:                 backward_slicing(source_reg(inst_b))
23:                 forward_slicing(source_reg(inst_b))
24:     return stride
```

takes the instruction that triggers the DUE as the input, determines its addressing

mode, and calls the backward and forward slicing procedures respectively. Algorithm 4 describes how the backward slice is computed. The forward slicing process is identical to the backward slicing process except that the $prev(inst\_b)$ needs to be replaced with $next(inst\_b)$.

Algorithm 5 The main procedure for stride speculation

```
1:  procedure SPECULATE(inst)                          ▷ the inst that triggers the DUE
2:      stride ← 0
3:      (base, index, size) ← parse(inst)
4:      ▷ size means the accessed size
5:      if index ≠ "" then                             ▷ the addressing mode includes index
6:          stride ← backward_slicing(base, inst)
7:          if stride == 0 then
8:              stride ← forward_slicing(base, inst)
9:      else
10:         stride ← backward_slicing(index, inst)
11:         if stride == 0 then
12:             stride ← forward_slicing(index, inst)
13:     if stride == 0 then
14:         stride ← size                              ▷ takes the basic heuristic
15:     return stride
```

Impact on the run-time overhead. Prior studies [41, 57] show that conducting backward/forward slicing processes requires building dynamic data dependence graphs, which could be extremely time-consuming for large-scale applications. This problem, however, has been largely mitigated in BonVoision's stride speculation heuristic, due to the following reasons: *(i)*. the heuristic runs over static instructions, so it is unlikely to cause state explosion; *(ii)*. it looks for the dependence chains of the index (and/or transitive index) register(s), *(iii)*. it is limited to the scope of the function the DUE-causing instruction belongs to (i.e., it is intra-procedural).

5.4.3 Computing and writing-back the repair value

Based on the results of the motivating study presented in Section 5.3.2, the closest elements offer the smallest standard deviations across all applications. BonVoision operates as follows: once the stride is determined, BonVoision uses the stride to chooses the closest neighbours located at (*addr*-1*stride, *addr*+1*stride) and obtains the data (l,r) from those elements correspondingly (assume the DUE occurs at *addr*). It then computes the arithmetic mean of l and r, and writes the mean back to the address *addr* as the repaired value. After this, BonVoision lets the program

continue the execution. Because a typical word is 8 bytes, if the inferred stride is smaller than 8 bytes (e.g. 4 bytes) , BonVoision needs to repair the entire memory word. For each unit (e.g. 4 bytes) in the word that needs to be repaired, BonVoision attempts to locate the closest elements beyond the scope of the same memory word, and use the average of those elements to write back to the unit.

Handling vector instructions Modern CPUs provide direct support for vector operations where a single instruction is applied to multiple data (i.e. SIMD). For example, MOVAPD in SSE2 instruction sets loads 16 bytes of data from the memory to a 128-bit register like XMM. If an ECC exception is triggered when executing such instructions, BonVoision assumes that the corrupted bits are in the first 8 bytes of the memory and repairs them correspondingly. Repairs that cover neighbouring memory word will be explored in the future work.

Implementation-level mechanisms. For the system to adopt BonVoision, the following mechanisms should be in place. Firstly, the applications should not react (i.e., terminate) to the SIGBUS signal. This can be implemented by overwriting the signal handler from the application, or by reconfiguring the application's signal-handling actions [59]. Secondly, when performing the repair, the application should bypass the cache and directly overwrite the corrupted data in the memory [4]. This can be implemented with a class of intrinsics (i.e. _mm_stream_si32, _mm_stream_si128, etc) supported by compilers such as *gcc* or the Intel compiler, or the binary can be instrumented directly with the corresponding instructions such as MOVNTI from Intel SSE2. Thirdly, the OS kernel needs to perform a translation from the physical memory address to the virtual memory address for the DUE and pass the virtual memory address to the run-time. We assume that the DUE is triggered by an application read and not by scrubbing, since memory scrubbing is likely to occur during the idle periods.

Limitations. There are a few cases that adversely affect BonVoision's performance. First, if a program implements loop unrolling on some loops, then the elements are accessed in a user-defined fashion which makes the stride speculation much more difficult from the corresponding assembly code. Second, the advanced heuristics assume that frequently accessed elements on the heap are likely to be

[4]Otherwise there could be recurrent ECC errors due to the mismatched ECC. A memory write can trigger the computation of the new ECC for the memory data

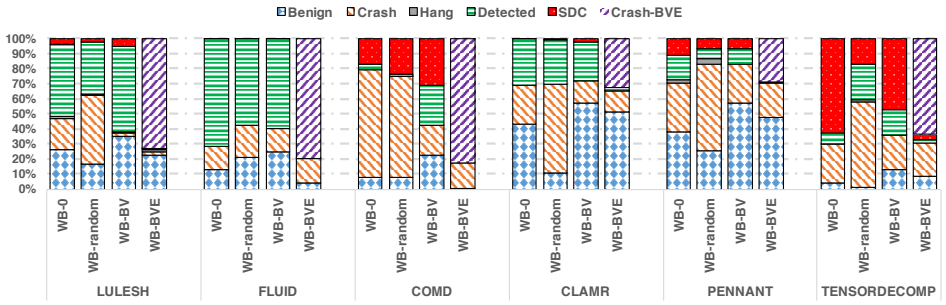


Figure 5.4: Application outcome ratios when DUEs are repaired by WB-0 (left), WB-random (center-left), BonVoision (center-right), and BonVoision-E (right). Crash-BVE class denotes that BonVoision-E predicts the repair will lead to detected or SDC outcome, a situation similar to a crash in the context of a C/R, as applications would continue from a checkpoint. For BonVoision-E, the crash and SDC rates are low or zero for most benchmarks, as a result they are not visible in some plots.

iterated over in a loop so that the stride can be speculated from the operations on certain registers during the address calculation. Our assumptions are based on two common practices in scientific applications: *(i)* many simulation problems tend to access nearby data iteratively, *(ii)* applications tend to exploit locality of reference.

5.5 Experimental methodology

We conduct large-scale *fault injection* campaign to evaluate BonVoision's effectiveness. We follow the methodology described in Section 5.3.1, with the slight difference that in each experimental trial, we do not explicitly "inject" a two-bit error into the memory as the corrupted data in the location will be completely overwritten by the repair anyways.

Benchmarks. We use (1) four representative DOE mini-applications, namely, LULESH [83], CLAMR [114], PENNANT [89] and COMD [35], (2) two scientific applications: FLUID [112] from the PARSEC benchmark suite [19] and, (3) a state-of-the-art CANDECOMP/PARAFAC decomposition implementation (denoted as TENSORDECOMP) [97]. The details of these applications are presented in Table 5.2. Note that our benchmarks use representative internal data structures (structured grids / meshes, adaptive meshes, etc.)

Application-specific correctness checks. For many scientific applications, developers write acceptance checks that increase confidence that the result was not impacted by a numerical error, an SDC, or simply a software bug. These acceptance checks are often based on energy conservation or numeric tolerance for result approximation. In practice, the check is typically placed at the end of the execution of the application [83], or if the application uses a C/R scheme, at the end of each checkpoint interval [114]. For *CLAMR*, *FLUID*, and *PENNANT*, we use the built-in acceptance checks (written by the developers). For *LULESH*, *COMD*, we wrote the checks ourselves, based on application verification specifications: Section 6.2 in [82] for *LULESH*, "Verification correctness" section in [36] for *COMD*. For *TENSORDECOMP*, we consulted with the benchmark's developer and wrote the check that examines the decomposition fit rate at the end of each iteration for convergence. Table 5.2 describes the the criteria used in the acceptance checks for each benchmark.

Outcome classification. A first decision is made based on whether the application appears to complete its run successfully after a repair. If not, the application either receives an OS signal and terminates before it finishes (a *crash*), or the application does not finish in an expected time (a *hang*) and is terminated. If yes, then the output of the final application state is checked. If the application's results do not pass the acceptance check, the outcome is labelled as *detected*, while if the application passes the check, the core program output is compared bit-wise with the output from a fault-free run: if they differ then the outcome is *SDC*, otherwise the outcome is *benign*. Note that this is conservative estimate as we do not consider application-specific semantics in interpreting the output.

Implementation and experimental setup. BonVoision is implemented using Intel Pin tool [118] 3.5. All experiments are run on a server with 40 Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz processors running Linux 4.13.0. *We perform* $5,000$ *injections per application to obtain tight error bounds of 0.2% - 1.3% at the 95% confidence interval.* All benchmarks are compiled with gcc 8.2 using O3, which auto-generates X86-64 SSE2 instructions.

5.6 Evaluation results

We first evaluate the effectiveness of BonVoision in repairing memory errors that result in DUEs (Section 5.6.1 and Section 5.6.2). We then measure BonVoision's performance overheads while scaling up the application (Section 5.6.3). Finally, we demonstrate the use of BonVoision in the context of a typical C/R scheme and evaluate its impact on reducing C/R overheads (Section 5.6.4).

5.6.1 Q1: How effective is BonVoision?

Comparison baselines. We introduce two repair strategies other than BonVoision, namely write-back with 0 (labeled WB-0 in plots) and write-back with random (WB-random), both attempting to repair the corrupted memory data and continue the execution of an application. Based on past related work [59, 101], both strategies are viable, and are hence reasonable as baselines.

Overall effectiveness. To measure BonVoision's effectiveness we focus on the benign and SDC outcomes of a repair: the more benigns and the fewer SDCs BonVoision converts DUEs into, the more effective are its repairs. Figure 5.4 (left three bars for each benchmark) shows the outcome of our error injection campaign after repairing DUEs with BonVoision (indicated as WB-BV), WB-0 and WB-random. For each benchmark, we present the percentage of each type of outcome under different repair techniques. *Overall, applications repaired by BonVoision lead to highest benign rates among three repair strategies across all benchmarks.*

In particular, the improvements of benign rates range from 5% to 20% compared to WB-0, and from 5% to 47% comparing WB-random (all expressed as absolute value). On average, BonVoision achieves on average 12% and 21% improvements over WB-0 and WB-random, respectively. We observe, on average, 0.8× and 2.5× improvement in benign rates when comparing WB-BV with WB-0 and WB-random, respectively (expressed as relative values).

For LULESH, FLUID, CLAMR and PENNANT, the SDC rates are less than 7% across all repair strategies; although BonVoision leads to higher SDC rates, the difference is rather marginal, ranging from 0.1% to 2% (absolute). For COMD and TENSORDECOMP, we observe relatively high SDC rates from three strate-

gies: 23%(WB-0), 27%(WB-random) and 30%(WB-BV) for COMD, 63%(WB-0),16%(WB-random) and 45%(WB-BV) for TENSORDECOMP.

WB-BV outperforms the strategy that ignores DUEs: Comparing the results of BonVoision with the results of ignoring DUEs in Figure 5.1, BonVoision is able to reduce the SDC rates from 22% to 14%, and increase the benign rates from 26% to 36% on average.

WB-BV offers more effective repairs than WB-random Although WB-random offers the lowest SDC rates, it performs poorly for converting DUEs to benign outcomes. Specifically, the repairs lead to the highest crash and detected rates among the three strategies on average, significantly degrading the benefit of the repairs in the context of the C/R: the application either crashes or violates the application's checks (if any), and rolls back to the checkpoint.

To summarize, BonVoision outperforms both WB-0 and WB-random in offering the most effective repairs: BonVoision results in the highest benign rates for all benchmarks, and only incurs marginal increases in SDC rates in four benchmarks. The results were similar when we tested with different data inputs for LULESH, CLAMR, and PENNANT. For COMD and TENSORDECOMP, none of the three techniques: WB-BV, WB-0, and WB-random, offers effective repairs in terms of minimizing the SDC rates.

5.6.2 Q2: Can machine learning help improve BonVoision?

One of the main uses for BonVoision is in the context of a C/R scheme, thus, the following discussion will be guided by this context. When using BonVoision, upon a memory DUE, the application can roll forward instead of rolling backward since the corrupted data likely gets repaired by BonVoision. However, as shown in Figure 5.4, the chance that the repair is followed by a crash, detected, or SDC is not negligible, which raises concerns[5] when using BonVoision with a C/R system. We elaborate these concerns and explain why they need to be addressed for BonVoision to run with C/R: *(i)* most importantly SDCs are the worst case as the application produces incorrect results without notifying the users; *(ii)* the detected errors result in the waste of resources: as the correctness checks are typically placed at the

[5]The similar concerns are also discussed in the recent study [55]

end of the checkpoint interval, a BonVoision repair that triggers an application correctness check error at the end of the interval is costlier than a crash triggered by the DUE; and finally *(iii)* crashes are similar to detected though they incur lower cost, as a crash would generally re-occur before the end of the checkpoint interval [6]. Therefore, we consider the crashes to be not harmful in this study and focus on the other outcomes.

One way to alleviate these concerns is to predict the outcome of a BonVoision repair, and just recover from a checkpoint if the predicted outcome is SDC, detected or crash. To this end, we explore the effectiveness of an online classifier to predict the outcome of BonVoision repairs. The classifier uses as features our estimators of space smoothness around the repair site and is trained based on ground truth obtained during the error injection campaigns.

The classifier is built using the standard supervised machine learning process: we collect the standard deviation calculated from each trial of the fault injection experiments for each benchmark, and assign the label to each standard deviation based on the outcome class of the trial. We follow standard practice and split the dataset 80/20 as the training and testing set, train the classifier on the training set with a couple of classification algorithms (e.g., linear-regression and decision-tree), and test the models on the testing set to determine the best-performing model.

Model tuning. We prioritize avoiding primarily SDCs, and, secondly detected outcomes. Table 5.3 shows the corresponding confusion matrix. We note that, depending on the operational context, the classifier can be tuned to prioritize other success metrics.

Table 5.3: The confusion matrix for the classifier, associated with mis-classification costs. Low cost for misclassifying a repair that leads to a benign outcome (in these cases the C/R scheme will restart from a checkpoint similarly to when BonVoision is not used), high cost if SDCs or detected are predicted as benigns.

Predicted \ Actual	Benign	Detected or SDC (not benign)
Benign	True Positive Cost: low	False Positive Cost: high
Detected or SDC (not benign)	False Negative Cost: not high	True Negative Cost: low

[6] We measure the interval between the time the application gets continued after the repair and the time the crash occurs, and observe that all the crashes occur nearly right after the repair.

Classifier performance. Table 5.4 shows the classifier performance on each benchmark. Overall, the decision-tree classifier provides the best results: depending on application, the accuracy of the classifiers are from 70% to 92.4%, offering good accuracy to separating benign and not benign outcomes (SDC and detected). In particular, as the classifier is optimized for a conservative prediction on benigns when the actual class of the repairs are also benigns, the recalls (i.e., the true positive rates) for the benchmarks can be relatively low, while the selectivity (i.e., the true negative rate) soars from 88% to 100%, meaning that when the classifier predicts that a repair would not lead to a benign outcome, it is likely that the prediction leads to detected or SDC. Thus, a C/R scheme is confident to crash the application for those cases. It is observed that, for TENSORDECOMP, the mis-classification rate of detected cases can be a little higher than 10%, and mis-classification rate of SDCs is 8%. In fact, for this case, the absolute number of SDCs and detected cases is relatively small [7].

Comparing with vanilla BonVoision. Figure 5.4 highlights the impact of using the online classifier: the outcomes of using the online predictor are labelled BonVoision-E(nhanced) - right bars in the plots. The ratio of predicted SDC or crash outcomes is identified separately. We assume that the crash rate stays constant between BonVoision and BonVoision-E, so all the types of outcomes are adjusted proportionally, which leads to a conservative estimation The key takeaway is that BonVoision-E is capable of eliminating most SDC and detected cases for all the benchmarks and for some benchmarks it completely eliminates SDCs.

Table 5.4: Classifier quality. The mis-classification rate of SDC shows the fraction of predicted benigns but the true class are SDCs divided by the total number of SDCs in the testing set. Similarly for the misclassification rate of detected.

Benchmark	Accuracy	Recall	Precision	Selectivity	False Positive Rate	Misclassification Rate of SDC	Misclassification Rate of Detected
LULESH	88.5%	55.4%	99.0%	99.7%	0.2%	3.0%	0%
FLUID	85.7%	22.0%	84.2%	98.8%	1.1%	0	1.0%
COMD	70.0%	5.8%	100.0%	100.0%	0	0	0
CLAMR	91.2%	89.2%	96.6%	94.8%	5.0%	0	6.0%
PENNANT	92.4%	90.0%	99.2%	98.3%	1.6%	5.0%	0.0%
TENSOR-D.	85.3%	48.5%	54.0%	88.4%	2.5%	8.0%	13.0%

[7] We applied this process on LULESH and PENNANT for additional inputs (scale and values), and we observed qualitatively similar results.

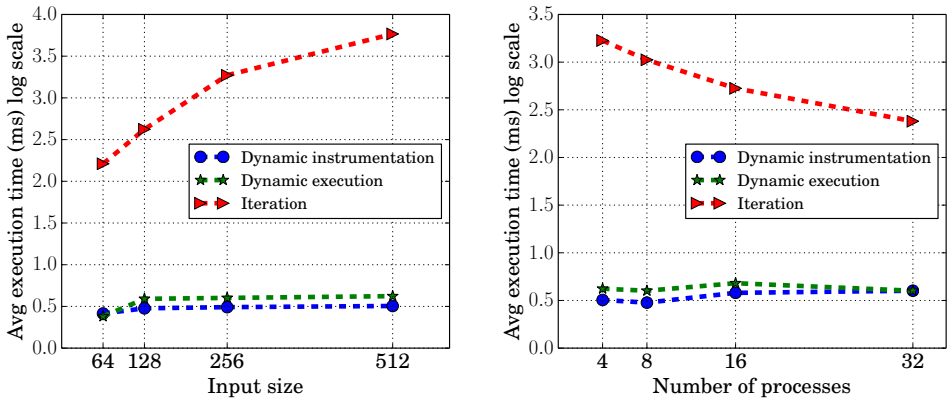
5.6.3 Q3: What is BonVoision's efficiency?

BonVoision runs in two phases: the dynamic instrumentation phase for stride speculation on the program's assembly code, and the dynamic execution phase that performs the actual repairs. We aim to estimate the overhead of each of these phases, and how overheads scale when the application scales up in either input size or the number of processes. We present the results for only a single application CLAMR, though similar results were observed for all benchmarks. We measure the execution time of each phase under different input sizes and numbers of MPI processes with CLAMR, and compare those times with the time spent for each iteration by CLAMR. For each configuration, we run the application over 100 times, and measure the average of the execution time and the iteration time. We use CLAMR in this experiment because it has a basic C/R feature enabled, and the checkpoint is taken at the end of every certain number of iterations (configurable) as used in real world.

Figure 5.5a shows that the execution times of both phases remain approximately stable across different input sizes, while the iteration time increases much faster. Figure 5.5b shows a similar trend in the executions times of BonVoision while the number of MPI processes increases. The dynamic instrumentation time and the dynamic execution time adds up to a total execution time around 6 milliseconds overall for BonVoision. The overheads for the other benchmarks are in the same order of magnitude as well (0.5 to 2 ms). This level of overhead is negligible for most HPC applications.

5.6.4 Q4: What is the impact in the context of C/R?

For long-running scientific applications, the ability to tolerate memory DUEs is essential to make progress to completion - thus they usually employ C/R schemes. To evaluate the impact of BonVoision-E in this context we have developed we use a finite state machine to model the applications that run with a typical C/R system with and without BonVoision. We conduct experiments using an in-house continuous-time event simulator [59] on the applications in both scenarios to estimate resource usage efficiency (i.e., the ratio between the time the application does useful work and the total application runtime), a metric widely used for C/R's



a Execution time for each BonVoision phase and the iteration time (including checkpointing) while scaling input size

b Execution time for each BonVoision phase and the iteration time while scaling number of processes

Figure 5.5: The performance overhead incurred by BonVoision scales well and is overall negligible while the application scales up.

efficiency [22, 42, 151]).

We make the following assumptions to simplify the model. First, all crashes are due to memory DUEs, other memory soft errors are corrected by the ECC/chipkill, and no DUEs occur during the checkpointing process. Second, no other fault tolerance mechanisms than C/R are used. Third, we assume the applications take synchronous coordinated checkpoints. Thus, when a crash occurs on one node, all the nodes used by the application have to roll back to the last checkpoint and re-execute from that point. These are standard assumptions [42, 59, 151].

Parameter description. *Checkpoint interval.* The checkpoint interval defines the frequency to take checkpoints. Young's formula [151] indicates that the optimal checkpoint interval is determined by the MTBF and the checkpointing overhead.

MTBF. We extrapolate the MTBF based on the error rates occuring on the Blue Waters supercomputer. Martino et al. [21] report that during the measurement period, i.e. roughly 6,177 hours, 1,031,886 memory errors are observed on 12,721 Blue waters nodes, and 70.01% of them involve 1 bit error, 29.98% of them have 2-8 consecutive bits errors, only 28 errors are not correctable by SECDED EC-C/chipkill. Therefore, if there is only SECDED ECC deployed, the DUE MTBF

rate is 1.2s, and with x8 Chipkill, the DUE MTBF is 794,185s. The MTBF of the system is inversely proportional to the size of the system: if the system scales up by an order of magnitude, the MTBF decreases by an order of magnitude [22], assuming that similar device technology is used.

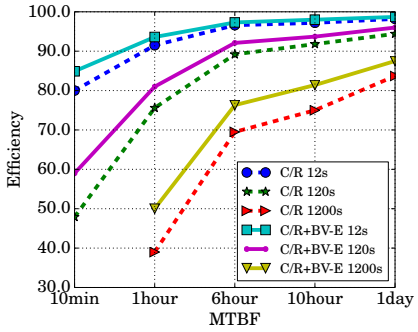
Checkpoint overhead. For system level checkpointing techniques, normally the entire memory (typically 32GB to 128GB on modern system node) is backed up to persistent storage. We assume three types of systems: a well-provisioned system with burst-buffer implemented with SSD (I/O bandwidth: average 1GB/s, peak 6GB/s), an average-provisioned system that has burst-buffer and spinning disk, and an under-provisioned system that only deploys spinning disks (I/O bandwidth 50MB/s to 500MB/s), and extrapolate the checkpointing overhead to be 12, 120 and 1200 seconds respectively.

Other parameters. We optimistically assume that the recovery overhead equals the checkpointing overhead (i.e., equal storage read and write speeds). We also assume application correctness checks at the end of checkpoint interval take 1% of checkpoint overhead, and that the application needs to spend 10% of checkpointing overhead to synchronize across all nodes as we assume synchronized checkpointing.

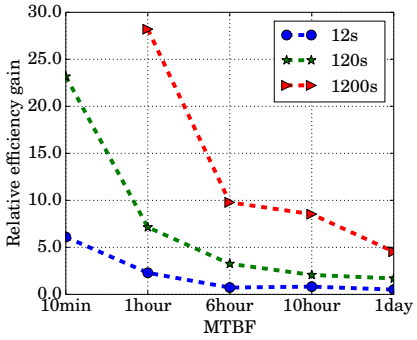
Results Figure 5.6 shows the efficiency comparison between the vanilla C/R system and the C/R system using BonVoision-E. Left plots present absolute application efficiency in various configurations while right plots present the relative improvement offered by BonVoision-E. We vary the system size in combination with different checkpointing overheads. We consider systems with 50k, 60k, 100k, 600k, and 3600k nodes - with MTBF scaled from data inferred from Martino et al. [21]. These have MTBFs of 1 day, 10 hours, 6 hours, 1 hour and 10 mins respectively. Overall, C/R+BonVoision-E achieves higher efficiency than C/R across all configurations, ranging from 76% to 1%, and in particular on average 8% for LULESH, and 15% for CLAMR, 15% for PENNANT, 3% for TENSORDECOMP and FLUID, and a marginal improvement for COMD.

We also find that there is a clear trend for the efficiency gain offered by BonVoision-E: when MTBF decreases, the gains accelerate across all configurations. We speculate that this enables an opportunity for future HPC systems to reduce energy consumption: the system could intentionally reduce the DRAM refresh rate [100,

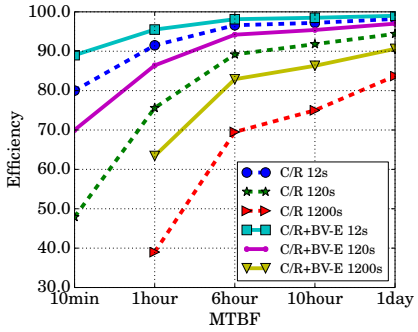
127, 135], resulting in increasing memory error rates. Applications running with C/R+BonVoision-E can potentially maintain the same level of efficiency e.g., as shown in Figure 5.6, the efficiency of C/R+BonVoision-E's is about 90% at 10 mins MTBF and 12 seconds checkpointing overhead, while C/R's efficiency is 91% at 1 hour MTBF and the same checkpointing overhead. This trend is consistent for other configurations across all benchmarks.



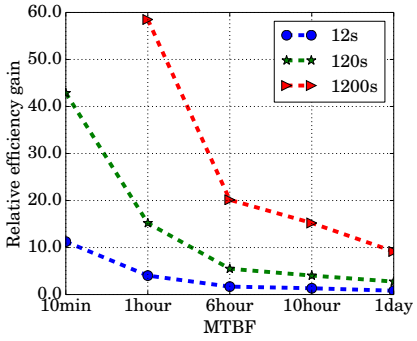
a Efficiency in the context of a typical C/R and C/R with BonVoision-E for LULESH



b Relative efficiency gain of C/R+BonVoision-E over typical C/R for LULESH



c Efficiency in the context of C/R and C/R with BonVoision-E for CLAMR



d Relative efficiency gain of C/R+BonVoision-E over typical C/R for CLAMR

Figure 5.6: Efficiency comparison between C/R and C/R+BonVoision-E for simulated 10-years of application time for LULESH and CLAMR. The MTBF decreases as the system scales up.

5.7 Discussion

While this paper demonstrates that BonVoision and BonVoision-E can effectively and efficiently recover applications from memory DUEs using the stride-based speculation heuristic that leverages space data smoothness, there are three issues that should be discussed, including *(i).* is the proposed heuristic superior to other repair approaches? *(ii).* can one use BonVoision on other architectures like GPUs? *(iii)*, how well would BonVoision work with more advanced ECC schemes like chipkill?

(i) Compare the stride-based approach with others The heuristic proposed by BonVoision assumes no information from the application and speculates the stride of the data structure with a lightweight dynamic analysis. While we show its usefulness, it is worth investigating if this approach works better than other possible solutions for devising the repairs for memory DUEs. Below, we discuss our perspective on this comparison:

- *speculating the stride differently*. A straightforward approach to speculate the stride of the data structure is to find the immediate neighbours using the size-based approach. To Obtain the size one only needs to parse the DUE-causing instruction at run-time, so it incurs less overhead than BonVoision's stride-based heuristic. However, as many HPC applications employ complex data structures to represent the physical space, the immediate neighbours may not necessarily reflect the spatial data smoothness.

- *estimating the impact of potential data similarity*. The program data might be similar across a large chunk of memory. In this case, the proposed heuristics might output incorrect strides while still offering appropriate repairs. To rule out this factor, we propose two approaches: *random with stride*, which reads elements from two random locations in the range of (i.e. addr-128*stride, addr+128*stride), and computes their average as the repair value; and similarly *random with size*, which reads elements from two random locations in the range of (i.e. addr-128*size, addr+128*size) and compute their average as the repair value.

We have executed a fault injection experiment similar to the one presented in

Section 5.6.1. For each error injection trial that leads to benign, SDC or detected case (we exclude crashes as applications terminate in these cases hence the repairs have no impact on the output's correctness), we compute the relative difference between the original data in the corrupted memory location and the repair values computed based on each of the approaches above. We find that, the stride-based approach (i.e. BonVoision's heuristic) outperforms other approaches significantly. In particular, BonVoision offers on average 41% more values that differ from the original value by less than 5%.

(ii) Portability The current implementation of BonVoision mainly focuses on supporting well-adopted architectures (i.e. X86-64) in HPC systems. As the trend in HPC moves towards heterogeneous computing, it is worth understanding the BonVoision's feasibility for diverse architectures like GPUs and ARM CPUs. The two fundamental ideas of BonVoision: leveraging spatial data smoothness in the application' data for repair, and attempting to infer the stride of the data structure based on how the index register is adjusted, are independent of the characteristics of a specific architecture. While extending BonVoision for diverse architectures is promising, there are certain issues that need attention. For example, BonVoision requires the just-in-time compiler tool for dynamic instrumentation, which may not be easily accessible on other platforms; and in particular to GPUs, additional complexity results from less mature interrupt functionality on current GPUs and GPU-specific optimization techniques like coalesced memory access.

(iii) Advanced ECC support Advanced ECC schemes like chipkill can largely improve the memory error resilience, and have been deployed in some HPC systems. For example, Blue Waters [21] feature both x4 (i.e. single symbol error correction and dual-symbol error detection, with 4 bit/symbol), and x8 chipkill ECC DRAM. It is interesting to understand how would BonVoision work with such schemes. In Section 5.4.3 we introduce the BonVoision's strategy for writing the repair back to the memory: BonVoision's repair overwrites the entire memory word that contains the error bits. Therefore, as long as the detected error bits (i.e. by chipkill) are in the same memory word, BonVoision should be able to handle it.

5.8 Summary

In this chapter, we show that the impact of memory DUEs on applications in the context of C/R schemes can be mitigated. We present BonVoision, a lightweight, automatic approach that leverages spatial data smoothness present in HPC applications to calculate the repair value of the corrupted memory location based on the neighboring data elements. We evaluate the corresponding performance gain (i.e. the second part of RQ2) with the proposed prototype. We find that when including our most effective techniques, on average, BonVoision-E is able to continue to completion 30% of the cases while decreasing the application's SDC probability to almost zero. With BonVoision-E enabled, we show that the efficiency of the standard C/R system can be potentially improved significantly.

Chapter 6

Conclusion

This chapter first summarizes this dissertation and highlights its expected impact on the design and implementation of error characterization and fault tolerance techniques. It concludes the dissertation with a discussion on research directions for future work.

6.1 Summary

The prevalence of transient hardware faults in modern HPC systems necessitates the design of efficient and effective fault characterization and tolerance approaches. Toward this goal, my dissertation first presents an improvement of the error characterization mechanism and showcases how one can use the inferred error resilience characteristics of applications to guide the selective error detection technique; next, the dissertation proposes approaches that introduce the roll-forward recovery scheme in addition to the standard roll-back (C/R) systems. A summary of the approaches includes the following:

- Chapter 3 describes ePVF, a methodology that extends the PVF analysis [131] by distinguishing crash-causing bits from the ACE bits so as to get a tighter bound on the estimation of the SDC rate. ePVF reduces the vulnerable bits estimated by the original PVF analysis by between 45% and 67%, depending on the benchmarks. To further reduce the overall analysis time, we propose partial ePVF that samples representative patterns from ACE bits and find

that the extrapolated ePVF values are a good approximation for the overall ePVF—on average the error is less than 1%. Finally, we present the following use-case for this methodology: an ePVF-informed selective duplication technique, which leads to 30% lower SDCs than hot-path instruction duplication.

- Chapter 4 targets crashes from errors affecting computational components in HPC systems. It presents LetGo, a technique that attempts to continue the execution of a HPC application when crashes would otherwise occur. It monitors the execution of an application and modifies its default behavior when a termination-causing OS signal is generated.

  When used in the context of a C/R scheme, LetGo enables sizable resource usage efficiency gains: for a set of HPC benchmarks, LetGo offers over 50% chance that the application can continue and produce correct results without performing a roll-back/recovery. We evaluate the end-to-end impact of LetGo in the context of a C/R scheme and find that LetGo offers significant efficiency gains (1.01x to 1.20x) in the ratio between the time spent for useful work and the total time cost compared to the standard C/R scheme.

- Chapter 5 targets crashes from errors affecting memory systems. It presents BonVoision, a run-time system that intercepts DUE events, analyzes the application binary at runtime to identify the data elements in the neighborhood of the memory location that generates a DUE and uses them to fix the corrupted data. Compared to other repair techniques, BonVoision is able to convert, on average, $1.2-2.8\times$ DUEs to benign cases, with a marginal increase in SDC rates. Using an online classifier, the enhanced BonVoision (i.e. BonVoision-E) eliminates almost all detected and SDC outcomes and produces repairs that lead to benign outcomes 30% more often than the default fail-stop modes, potentially increasing the efficiency of checkpoint/restart mechanisms. We find that when including our most effective techniques, on average, BonVoision-E is able to continue to completion 30% of the cases while decreasing the application's SDC to almost zero.

6.2 Expected impact

6.2.1 Efficient SDC probability estimation and a quantitative approach for SDC detection

The first impact made by this dissertation is demonstrating the existence of a different trade-off point between the effort to characterize the error resilience of an application and the accuracy of the estimate. The proposed instruction-level SDC-proneness estimates can be used to guide error detection. In particular, the ePVF methodology indicates the following:

- PVF-like analysis is a good vehicle to allow fine-grained understanding over the program states, thus vastly accelerates the error resilience characterization process and maintains an accurate estimate on the impact of errors.
- With fault injections, the application under test is usually treated as a black box, and it is difficult to link the error resilience characteristics of an application to error detection strategies. ePVF allows the error detection techniques to obtain a quantitative estimate of SDC-proneness at the instruction level.
- There are other use cases of ePVF. First, it can be used to determine which architectural structures are more likely to cause SDCs and selectively protect these structures through hardware techniques such as selective ECC. Second, the ePVF methodology can be used to determine the total number of crash-causing bits in the program and inform a fault-tolerance mechanism for crash-causing faults (e.g. checkpointing).

Highlights of impacted research contributions: ePVF shows that it is possible to combine error resilience characterization and error propagation analysis for more efficient and accurate SDC probability estimation and detection for the dependability community:

The ePVF methodology pioneers the approaches that refine the AVF/PVF analysis to classify architectural/program states based on their differing impacts on the application outcomes. These approaches have been extended lately. For example, Wibowo et. al. [146, 147] propose a study of the AVF that leads to DUE (i.e.

DUE_{avf}) and the AVF that leads to SDC (i.e. SDC_{avf}) over the phases of execution of all major memory structures of a modern superscalar processor and correlates the SDC_{avf} with the SDC_{pvf} of the Architectural Register File. The SDC_{pvf} approach can work in combination with the ePVF methodology to enhance the SDC_{pvf} estimation and reduce the cost of evaluating a program's error resilience.

The ePVF methodology inspires the idea of using quantitative instruction-level SDC-proneness to guide the design of error detection techniques. Trident [96] follows this idea, and shares a similar goal and high-level methodology with ePVF. It leverages the static and dynamic dependence graph analysis and offers quantitative estimates for both the overall SDC probability and the per instruction SDC probability. To conduct instruction-based selective duplication, Trident computes the SDC proneness for each instruction and duplicates the instructions that have higher chances to lead to SDCs for SDC detection.

6.2.2 A new direction to improve C/R efficiency

The second impact made by my dissertation is that our research is the first to introduce the roll-forward recovery scheme in standard C/R systems. As described in Chapters 4 and 5, such roll-forward C/R scheme creates the opportunity to trade confidence in results for efficiency (time-to-solution or energy-to-solution). Certainly, for some applications—or for some operational situations—confidence in results is the user's primary concern, and LetGo and BonVoision will hesitantly be used. We believe, however, that there are many situations that make this trade-off attractive, as follows:

- First, since SDCs can occur anyway (due to bit-flips regardless of whether LetGo or BonVoision are used), HPC users are already taking the risk of obtaining incorrect results and have developed techniques to validate their results. For example, using application-specific checks to diminish this risk is an active research area [80] and a roll-forward C/R system will benefit from all these efforts.

- Second, some applications, as we show with LetGo and BonVoision, can be successfully continued after failures and produce no additional SDCs; here, LetGo and BonVoision definitely represent appealing solutions.

- Third, the proposed approach offers users additional information that can be used to reason about the confidence in results.

Highlights of impacted research contributions: The proposed roll-forward recovery methodology offers a promising strategy in failure recovery domains: it is possible to trade confidence in the recovery data for better recovery efficiency while making a minimal impact on the outcomes of the recovery.

One example that exercises this idea after LetGo is EasyCrash [124], an application-level solution to handle application failures on non-volatile memory (NVM). Similar to LetGo and BonVoision, EasyCrash attempts to continue the application's execution after the crash with an online repair strategy. The heuristic used by EasyCrash is specific to NVM: it loads the persistent data available in NVM at the time of failure to repair the corrupted program state for the application. Due to the inconsistency that might occur between the cache and persistent memory, EasyCrash also relies on the natural error resilience observed among typical HPC workloads (i.e. convergent computational patterns and the application-specific correctness checks) to reduce the risk of introducing SDCs.

Another example of the roll-forward recovery system that is currently drawing attention in the HPC community after LetGo is CARE [31]. CARE attempts to recover applications from segmentation faults, which is in line with one of the main target failure types of LetGo. CARE assumes that the dependent registers to calculate the corrupted memory address remain error-free when the segmentation fault occurs, and it proposes a repair heuristic that re-computes the correct memory address based on values in those registers and reloads data from the intended memory location as the repair. The repair is applied on-the-fly after the application encounters the failure, while the information needed for the repair is obtained through compiler-assisted techniques.

6.2.3 Practicality

Finally, the approaches proposed in this dissertation are practical for HPC applications. HPC applications generally favor a conservative environment, where changing the application code or runtime environment may introduce risks to the HPC practitioners. Both LetGo and BonVoision can be applied directly to the applica-

tion's executable and transparently track and alter the program's state/data. We believe our design choices increase the chance of our techniques being adopted in future HPC systems.

6.3 Future work

We propose four potential future research directions:

6.3.1 Direction *(i)*: Predicting the impact of a roll-forward failure recovery

Chapters 4 and 5 reveal that the roll-forward recovery scheme implicitly trades faster time to solution for higher risk—as it usually performs a probabilistic repair, which may cause further failures, such as SDCs, which are the main type of outcomes to be eliminated. We believe that answering the following two questions can make the roll-forward recovery more effective and more likely to be accepted by practitioners:

1. A runtime decision: Assuming that a failure has occurred and assuming a specific repair heuristic can be used, can one predict the likelihood of a specific outcome (re-crash, termination with the correct result, or termination with SDC) ? Answering this question is useful to optimize the scheme by making the following two decisions: (a) to inform which repair heuristic to use, and (b) to decide whether to employ a roll-forward strategy or fall-back on the traditional roll-back C/R scheme.

2. A decision after application termination: Assuming that a failure has occurred, a specific repair heuristic has been used, and the application terminates successfully, can one predict the likelihood that the result is correct, i.e., not an SDC? Answering this question is useful for making an informed decision as to confidence in the result(s).

6.3.2 Direction *(ii)*: Semantic-based data recovery for memory DUEs

In Chapter 5 we observe the spatial data smoothness in HPC applications' data space, and we take a basic heuristic (i.e. arithmetic average on the values of the

neighbouring data) to compute the repair value for recovery. Our approach of BonVoision assumes a general data displacement scheme across applications, while in many cases of stencil computations, data can be placed in an N-dimensional format. A refinement over the current approach would require an understanding of the following questions: what does the data space look like for an application, and does the simple repair operator always work? Hence, a potential research direction would be to investigate whether such spatial data smoothness exists for a particular data storage format and verify whether more sophisticated repair operators such as the Lorenzo predictor [81], could lead to a higher likelihood of accurate/acceptable recovery for the application.

6.3.3 Direction *(iii)*: Reducing energy consumption on memory systems

This dissertation shows that HPC applications can benefit from the roll-forward recovery scheme when facing memory DUEs; BonVoision allows the application to attempt to continue execution and produce correct results rather than crash right away. Since the application instrumented with BonVoision fails less often due to memory DUEs, it can afford to run on the systems where the DRAM failure rate is higher. A future study could explore the opportunity of a system design regarding the interplay among energy consumption, the error resilience of memory systems and the application correctness. For example, purposely lowering the DRAM refresh rate trades the memory's reliability for more efficient energy consumption without hurting the correctness of the applications. This idea would possibly lead to a new generation of memory technologies [100, 127].

6.3.4 Direction (iv): Data compression

Data compression is an effective technique to reduce the size of the data for more efficient storage usage. On HPC systems, since scientific applications can generate massive amounts of simulation data, periodically storing the snapshots of data becomes a bottleneck for the entire simulation. To mitigate this challenge, error-controlled lossy compression techniques [6, 14, 48, 136] have been proposed to reduce the data size significantly while guaranteeing that the distortion of the

compression data is acceptable by the applications. As BonVoision makes the observation that many scientific applications exhibit inherent continuities in their modeled physical space and that such smoothness is well-preserved in the logical space, leveraging data smoothness to compress data in lossy compression techniques presents a promising direction.

Bibliography

[1] Mpi standard 3.0. http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf. Accessed: 2015-06-04. → page 27

[2] Samsung electronics develops world's first eight-die multi-chip package for multimedia cell phones. http://www.samsung.com. Samsung Electronics Corporation. → page 86

[3] International technology roadmap for semiconductors./ model for assessment of cmos technologies and roadmaps (mastar). http://www.itrs.net. Semiconductor Industries Association. → page 86

[4] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of Programming Language Design and Implementation (PLDI)*, New York, NY, USA, 1990. ISBN 0-89791-364-7. → page 33

[5] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. Goofi: generic object-oriented fault injection tool. In *DSN*, pages 83–88, 2001. → page 5

[6] M. Ainsworth, S. Klasky, and B. Whitney. Compression using lossless decimation: Analysis and application. *SIAM Journal on Scientific Computing*, 39(4):B732–B757, 2017. doi:10.1137/16M1086248. URL https://doi.org/10.1137/16M1086248. → page 120

[7] H. Ando, Y. Yoshida, A. Inoue, I. Sugiyama, T. Asakawa, K. Morita, T. Muta, T. Motokurumada, S. Okada, H. Yamashita, Y. Satsukawa, A. Konmoto, R. Yamashita, and H. Sugiyama. A 1.3 ghz fifth generation sparc64 microprocessor. In *2003 IEEE International Solid-State Circuits Conference, 2003. Digest of Technical Papers. ISSCC.*, pages 246–491 vol.1, Feb 2003. doi:10.1109/ISSCC.2003.1234286. → page 18

[8] J. Arlat, Y. Crouzet, and J. . Laprie. Fault injection for dependability validation of fault-tolerant computing systems. In *[1989] The Nineteenth*

International Symposium on Fault-Tolerant Computing. Digest of Papers, pages 348–355, June 1989. doi:10.1109/FTCS.1989.105591. → page 17

[9] R. Ashraf, R. Gioiosa, G. Kestor, R. F. DeMara, C.-Y. Cher, and P. Bose. Understanding the propagation of transient errors in HPC applications. *SC*, pages 72–12, 2015. → page 87

[10] B. Atkinson, N. DeBardeleben, Q. Guan, R. Robey, and W. M. Jones. Fault injection experiments with the clamr hydrodynamics mini-app. In *2014 ISSREW*. → page 27

[11] G. Aupy, A. Benoit, T. Hérault, Y. Robert, F. Vivien, and D. Zaidouni. On the combination of silent error detection and checkpointing. In *Proceedings of the 2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing*, PRDC '13, pages 11–20, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-5130-2. doi:10.1109/PRDC.2013.10. URL http://dx.doi.org/10.1109/PRDC.2013.10. → page 21

[12] T. M. Austin. Diva: a reliable substrate for deep submicron microarchitecture design. In *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 196–207, Nov 1999. doi:10.1109/MICRO.1999.809458. → page 18

[13] F. Ayatolahi, B. Sangchoolie, R. Johansson, and J. Karlsson. A study of the impact of single bit-flip and double bit-flip errors on program execution. In *Computer Safety, Reliability, and Security*, volume 8153, pages 265–276. 2013. ISBN 978-3-642-40792-5. doi:10.1007/978-3-642-40793-2_24. → page 32

[14] R. Ballester-Ripoll, P. Lindstrom, and R. Pajarola. Tthresh: Tensor compression for multidimensional visual data. *IEEE transactions on visualization and computer graphics*, 2018. → page 120

[15] L. Bautista-Gomez and F. Cappello. Exploiting spatial smoothness in hpc applications to detect silent data corruption. In *2015 IEEE 17th International Conference on High Performance Computing and Communications*, 2015. → page 87

[16] A. Benso, S. Di Carlo, G. Di Natale, and P. Prinetto. A watchdog processor to detect data and control flow errors. In *9th IEEE On-Line Testing Symposium, 2003. IOLTS 2003.*, pages 144–148, July 2003. doi:10.1109/OLT.2003.1214381. → page 17

[17] W. Bhimji, D. Bard, M. Romanus, D. Paul, A. Ovsyannikov, B. Friesen, M. Bryson, J. Correa, G. K. Lockwood, V. Tsulaia, et al. Accelerating science with the nersc burst buffer early user program. *Proceedings of Cray Users Group.[Online]. Available: https://cug. org/proceedings/cug2016 proceedings/includes/files/pap162. pdf*, 2016. → page 80

[18] R. Bianchini and T. A. El-Ghazawi. System resilience at extreme scale white paper. 2009. → page 55

[19] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011. → page 101

[20] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. S. Mukherjee, and R. Rangan. Computing architectural vulnerability factors for address-based structures. *SIGARCH Comput. Archit. News*, 33(2), May . ISSN 0163-5964. → page 19

[21] B. Bode, M. Butler, T. Dunning, W. Gropp, T. Hoe-fler, W.-m. Hwu, and W. Kramer. The blue waters super-system for super-science. contemporary hpc architectures, jeffery vetter editor, 2012. → pages 1, 80, 86, 108, 109, 112

[22] G. Bosilca, A. Bouteiller, E. Brunet, F. Cappello, J. Dongarra, A. Guermouche, T. Herault, Y. Robert, F. Vivien, and D. Zaidouni. Unified model for assessing checkpointing protocols at extreme-scale. *Concurr. Comput. : Pract. Exper.*, 26(17):2772–2791, Dec. 2014. ISSN 1532-0626. doi:10.1002/cpe.3173. URL http://dx.doi.org/10.1002/cpe.3173. → pages 108, 109

[23] G. e. a. Bosilca. Unified model for assessing checkpointing protocols at extreme-scale. *Concurr. Comput. : Pract. Exper.*, pages 2772–2791, 2013. ISSN 1532-0626. → pages 8, 75, 77, 80

[24] G. Bronevetsky, B. de Supinski, and M. Schulz. A foundation for the accurate prediction of the soft error vulnerability of scientific applications. In *SELSE*, 2009. → page 19

[25] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1), 2014. ISSN 2313-8734. → page 1

[26] M. Carbin, S. Misailovic, M. Kling, and M. C. Rinard. Detecting and escaping infinite loops with jolt. In *Proceedings of the 25th European*

Conference on Object-oriented Programming, ECOOP'11, 2011. → page 23

[27] J. Carreira, H. Madeira, and J. Silva. Xception: a technique for the experimental evaluation of dependability in modern computers. *Software Engineering, IEEE Transactions on*, Feb 1998. → page 5

[28] M. Casas, B. R. de Supinski, G. Bronevetsky, and M. Schulz. Fault resilience of the algebraic multi-grid solver. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 91–100. ISBN 978-1-4503-1316-2. → pages 56, 60

[29] 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 *IISWC*, IISWC '09. → pages 34, 43

[30] S. Cheemalavagu, P. Korkmaz, K. V. Palem, B. E. S. Akgul, and L. N. Chakrapani. A probabilistic cmos switch and its realization by exploiting noise. In *the Proceedings of the IFIP international*, 2005. → page 23

[31] C. Chen, G. Eisenhauer, S. Pande, and Q. Guan. Care: Compiler-assisted recovery from soft failures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC âĂŹ19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362290. doi:10.1145/3295500.3356194. URL https://doi.org/10.1145/3295500.3356194. → page 118

[32] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan. Analysis and characterization of inherent application resilience for approximate computing. In *Design Automation Conference (DAC), 2013*. → page 23

[33] H. Cho, S. Mirkhani, C.-Y. Cher, J. Abraham, and S. Mitra. Quantitative evaluation of soft error injection techniques for robust system design. In *DAC, 2013 50th ACM/EDAC/IEEE*, pages 1–10, May . → pages 17, 31, 32

[34] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra. Quantitative evaluation of soft error injection techniques for robust system design. *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pages 1–10, 2013. → pages 2, 3

[35] P. Cicotti, S. M. Mniszewski, and L. Carrington. An evaluation of threaded models for a classical md proxy application. In *Hardware-Software Co-Design for High Performance Computing (Co-HPC), 2014*, Nov . → pages 71, 101

[36] P. Cicotti, S. M. Mniszewski, and L. Carrington. Comd: A classical molecular dynamics mini-app, 2013. URL http://exmatex.github.io/CoMD/doxygen-mpi/index.html. → pages 71, 102

[37] C. Constantinescu, S. Krishnamoorthy, and T. Nguyen. Estimating the effect of single-event upsets on microprocessors. In *2014 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 185–190, Oct 2014. doi:10.1109/DFT.2014.6962059. → page 1

[38] J. Cook and C. Zilles. A characterization of instruction-level error derating and its implications for error detection. In *DSN*, 2008. → page 53

[39] J. J. Cook and C. Zilles. A characterization of instruction-level error derating and its implications for error detection. In *2008 DSN*, June . doi:10.1109/DSN.2008.4630119. → page 63

[40] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 161–174, Berkeley, CA, USA, 2008. USENIX Association. ISBN 111-999-5555-22-1. URL http://dl.acm.org/citation.cfm?id=1387589.1387601. → page 25

[41] M. Dadashi, L. Rashid, K. Pattabiraman, and S. Gopalakrishnan. Hardware-software integrated diagnosis for intermittent hardware faults. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 363–374, June 2014. doi:10.1109/DSN.2014.1. → page 99

[42] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener. Comput. Syst.*, 22(3):303–312, Feb. 2006. ISSN 0167-739X. doi:10.1016/j.future.2004.11.016. URL http://dx.doi.org/10.1016/j.future.2004.11.016. → pages 25, 56, 75, 90, 108

[43] T. Davies and Z. Chen. Correcting soft errors online in lu factorization. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 167–178, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1910-2. doi:10.1145/2462902.2462920. URL http://doi.acm.org/10.1145/2462902.2462920. → page 22

[44] M. de Kruijf, S. Nomura, and K. Sankaralingam. Relax: An architectural framework for software recovery of hardware faults. In *ISCA 14*, pages 497–508. ISBN 978-1-4503-0053-7. → page 29

[45] N. DeBardeleben, J. Laros, J. Daly, S. Scott, C. Engelmann, and B. Harrod. High-end computing resilience: Analysis of issues facing the hec community and path-forward for research and development. *Whitepaper, Dec*, 2009. → page 25

[46] T. J. Dell. A white paper on the benefits of chipkill-correct ecc for pc server main memory by. 1997. → page 86

[47] J. W. Demmel. *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997. ISBN 0-89871-389-7. → page 71

[48] S. Di, D. Tao, X. Liang, and F. Cappello. Efficient lossy compression for scientific data based on pointwise relative error bound. *IEEE Transactions on Parallel and Distributed Systems*, 30(2):331–345, Feb 2019. ISSN 2161-9883. doi:10.1109/TPDS.2018.2859932. → page 120

[49] N. El-Sayed and B. Schroeder. Checkpoint/restart in practice: When âĂŸsimple is betterâĂŹ. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 84–92. IEEE, 2014. → pages 75, 76, 80

[50] N. El-Sayed and B. Schroeder. Checkpoint/restart in practice: When simple is better;. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 84–92, Sept 2014. doi:10.1109/CLUSTER.2014.6968777. → page 25

[51] N. El-Sayed and B. Schroeder. Checkpoint/restart in practice: When simple is better;. In *2014 Cluster*, Sept 2014. → pages 8, 90

[52] A. C. et al. Versioned distributed arrays for resilience in scientific applications: Global view resilience. *Procedia Computer Science*, 51:29 – 38, 2015. ISSN 1877-0509. doi:http://dx.doi.org/10.1016/j.procs.2015.05.187. URL http://www.sciencedirect.com/science/article/pii/S1877050915009953. → pages 21, 22

[53] A. P. et al. Improving application resilience by extending error correction with contextual information. In *IEEE/ACM 8th FTXS@SC 2018*, 2018. → page 24

[54] G. B. et al. Mpich-v: Toward a scalable fault tolerant mpi for volatile nodes. In *Supercomputing, ACM/IEEE 2002 Conference*, Nov . → pages 8, 56, 89

[55] B. Fang, J. Chen, K. Pattabiraman, M. Ripeanu, and S. Krishnamoorthy. Towards predicting the impact of roll-forward failure recovery for hpc applications. In *DSN 2019*, Fast-abstract. IEEE. → page 104

[56] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi. GPU-Qin: a methodology for evaluating the error resilience of gpgpu applications. In *Performance Analysis of Systems and Software (ISPASS), 2014*, 2014. → pages 2, 4, 5

[57] B. Fang, Q. Lu, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi. epvf: An enhanced program vulnerability factor methodology for cross-layer resilience analysis. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), acceptance rate = 21%*, pages 168–179, June 2016. doi:10.1109/DSN.2016.24. → pages 59, 65, 99

[58] B. Fang, P. Wu, Q. Guan, N. DeBardeleben, L. Monroe, S. Blanchard, Z. Chen, K. Pattabiraman, and M. Ripeanu. Sdc is in the eye of the beholder: A survey and preliminary study. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*, pages 72–76, June 2016. doi:10.1109/DSN-W.2016.46. → page 4

[59] B. Fang, Q. Guan, N. Debardeleben, K. Pattabiraman, and M. Ripeanu. Letgo: A lightweight continuous framework for hpc applications under failures. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '17, acceptance rate = 18%, pages 117–130, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4699-3. doi:10.1145/3078597.3078609. URL http://doi.acm.org/10.1145/3078597.3078609. → pages 100, 103, 107, 108

[60] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: Probabilistic soft error reliability on the cheap. *SIGPLAN Not.*, 45(3), Mar. . ISSN 0362-1340. → pages 6, 20, 29, 31, 32, 50, 59

[61] T. Gaitonde, S.-J. Wen, R. Wong, and M. Warriner. Component failure analysis using neutron beam test. In *Physical and Failure Analysis of Integrated Circuits (IPFA), 2010 17th IEEE International Symposium on the*, pages 1–5, 2010. doi:10.1109/IPFA.2010.5531992. → pages 5, 17

[62] M. Gamell, K. Teranishi, M. A. Heroux, J. Mayo, H. Kolla, J. Chen, and M. Parashar. Local recovery and failure masking for stencil-based applications at extreme scales. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 70:1–70:12, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3723-6. doi:10.1145/2807591.2807672. URL http://doi.acm.org/10.1145/2807591.2807672. → page 22

[63] R. Gioiosa, J. C. Sancho, S. Jiang, F. Petrini, and K. Davis. Transparent, incremental checkpointing at kernel level: A foundation for fault tolerance for parallel computers. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, pages 9–, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 1-59593-061-2. doi:10.1109/SC.2005.76. URL https://doi.org/10.1109/SC.2005.76. → page 25

[64] M. Gottscho. Opportunistic memory systems in presence of hardware variability. → page 24

[65] M. Gottscho, C. Schoeny, L. Dolecek, and P. Gupta. Software-defined error-correcting codes. In *2016 DSN-W*. → page 24

[66] B. Grigorian and G. Reinman. Accelerating divergent applications on simd architectures using neural networks. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*. → page 23

[67] W. Gu, Z. Kalbarczyk, and R. Iyer. Error sensitivity of the linux kernel executing on powerpc g4 and pentium 4 processors. In *DSN 2003*, . → pages 27, 32

[68] W. Gu, Z. Kalbarczyk, and R. K. Iyer. Error sensitivity of the linux kernel executing on powerpc g4 and pentium 4 processors. In *Dependable Systems and Networks, 2004 International Conference on*, June . → page 65

[69] W. Gu, Z. Kalbarczyk, R. Iyer, and Z. Yang. *Characterization of Linux Kernel Behavior under Errors*, pages 459–468. 2003. → page 2

[70] L. Guanpeng. *Understanding and modeling error propagation in programs*. PhD thesis, University of British Columbia, 2019. → page 2

[71] V. Gupta, D. Mohapatra, A. Raghunathan, and K. Roy. Low-power digital signal processing using approximate adders. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(1): 124–137, Jan 2013. ISSN 0278-0070. → page 23

[72] J. Han and M. Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. In *2013 18th IEEE European Test Symposium (ETS)*. → page 23

[73] S. Hari, R. Venkatagiri, S. Adve, and H. Naeimi. Ganges: Gang error simulation for hardware resiliency evaluation. In *ISCA 2014*, 2014. → page 29

[74] S. K. S. Hari, S. V. Adve, and H. Naeimi. Low-cost program-level detectors for reducing silent data corruptions. In *2012 42nd DSN*, pages 1–12. IEEE, 2012. → page 50

[75] S. K. S. Hari, S. V. Adve, and H. Naeimi. Low-cost program-level detectors for reducing silent data corruptions. In *2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE, 2012. → pages 4, 6, 20

[76] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran. Relyzer: exploiting application-level fault equivalence to analyze application resiliency to transient faults. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012. → pages 20, 29, 59

[77] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition. ISBN 0-89871-521-0. → page 56

[78] C. hsing Hsu and W. chun Feng. A power-aware run-time system for high-performance computing. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 1–1, Nov 2005. doi:10.1109/SC.2005.3. → page 25

[79] M.-C. Hsueh, T. Tsai, and R. Iyer. Fault injection techniques and tools. volume 30, pages 75 –82, apr 1997. doi:10.1109/2.585157. → pages 4, 17

[80] K.-H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, 1984. → page 117

[81] L. Ibarria, P. Lindstrom, J. Rossignac, and A. Szymczak. Out-of-core compression and decompression of large n-dimensional scalar fields. *Computer Graphics Forum*, 22(3):343–348, 2003. doi:10.1111/1467-8659.00681. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/1467-8659.00681. → page 120

[82] I.Karlin. Lulesh programming model and performance ports overview, 2012. URL https://codesign.llnl.gov/pdfs/lulesh_Ports.pdf. → pages 71, 102

[83] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. Still. Exploring traditional and emerging parallel programming models using a proxy application. In *IEEE IPDPS 2013*, Boston, USA. → pages 43, 71, 101, 102

[84] I. Karlin, J. Keasler, and R. Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, August 2013. → page 43

[85] J. Karlsson and P. Folkesson. Application of three physical fault injection techniques to the experimental assessment of the mars architecture. pages 267–287. IEEE Computer Society Press, 1995. → page 17

[86] D. S. Khudia and S. A. Mahlke. Harnessing Soft Computations for Low-Budget Fault Tolerance. *MICRO*, pages 319–330, 2014. → page 29

[87] A. Kleen. mcelog: memory error handling in user space. → page 89

[88] L. A. N. Laboratory. Apex 2020, 2016. URL http://www.lanl.gov/projects/apex/_assets/docs/2.4-RFP-Technical-Requirements-Document.doc. → page 80

[89] L. A. N. Laboratory. The pennant mini-app v0.9, 2016. URL https://github.com/losalamos/PENNANT. → pages 71, 101

[90] L. A. N. Laboratory. Snap: Sn (discrete ordinates) application proxy v107, 2016. URL https://github.com/losalamos/SNAP. → page 71

[91] L. A. N. Laboratory. Snap - sn application proxy summary, 2016. URL https://asc.llnl.gov/CORAL-benchmarks/Summaries/SNAP_Summary_v1.3.pdf. → page 71

[92] I. Laguna, M. Schulz, D. F. Richards, J. Calhoun, and L. Olson. Ipas: Intelligent protection against silent output corruption in scientific applications. CGO 2016, 2016. → page 19

[93] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *CGO*, CGO '04, 2004. → pages 28, 31

[94] S. Levy, K. B. Ferreira, and P. G. Bridges. Improving application resilience to memory errors with lightweight compression. In *SC '16*, 2016. → page 24

[95] G. Li, Q. Lu, and K. Pattabiraman. Fine-grained characterization of faults causing long latency crashes in programs. In *DSN*, pages 450–461, June 2015. doi:10.1109/DSN.2015.36. → page 56

[96] G. Li, K. Pattabiraman, S. K. S. Hari, M. Sullivan, and T. Tsai. Modeling soft-error propagation in programs. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 27–38, June 2018. doi:10.1109/DSN.2018.00016. → pages 21, 117

[97] J. Li, J. Sun, and R. Vuduc. Hicoo: Hierarchical storage of sparse tensors. In *SC '18*. → page 101

[98] S. Li, K. Chen, M. Hsieh, N. Muralimanohar, C. D. Kersey, J. B. Brockman, A. F. Rodrigues, and N. P. Jouppi. System implications of memory reliability in exascale computing. In *SC '11*, 2011. → page 86

[99] D. Libes. expect: Curing those uncontrollable fits of interaction. In *PROCEEDINGS OF THE SUMMER 1990 USENIX CONFERENCE*, pages 183–192, 1990. → page 66

[100] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. Flikker: Saving dram refresh-power through critical data partitioning. *SIGPLAN Not.*, 46 (3):213–224, Mar. 2011. ISSN 0362-1340. doi:10.1145/1961296.1950391. URL http://doi.acm.org/10.1145/1961296.1950391. → pages 109, 120

[101] F. Long, S. Sidiroglou-Douskos, and M. Rinard. Automatic runtime error repair and containment via recovery shepherding. *SIGPLAN Not.*, 2014. → pages 22, 23, 103

[102] Q. Lu, K. Pattabiraman, M. S. Gupta, and J. A. Rivers. Sdctune: A model for predicting the sdc proneness of an application for configurable protection. In *CASE*, 2014. → pages 19, 20, 59

[103] Q. Lu, K. Pattabiraman, M. S. Gupta, and J. A. Rivers. Sdctune: A model for predicting the sdc proneness of an application for configurable protection. In *CASE 2014*, pages 1–10, New York, New York, USA, 2014. ACM Press. → pages 32, 50

[104] A. Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors-a survey. *IEEE Transactions on Computers*, 37(2): 160–174, Feb 1988. doi:10.1109/12.2145. → page 17

[105] A. Meixner, M. Bauer, and D. Sorin. Argus: Low-cost, comprehensive error detection in simple cores. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, Dec 2007. → page 29

[106] S. E. Michalak, W. N. Rust, J. T. Daly, A. J. DuBois, and D. H. DuBois. Correctness field testing of production and decommissioned high performance computing platforms at los alamos national laboratory. SC '14, pages 609–619, Piscataway, NJ, USA, 2014. ISBN 978-1-4799-5500-8. → pages 56, 87

[107] J. S. Miguel, J. Albericio, N. E. Jerger, and A. Jaleel. The bunker cache for spatio-value approximation. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, Oct 2016. doi:10.1109/MICRO.2016.7783746. → page 23

[108] S. Mittal. A survey of techniques for approximate computing. *ACM Comput. Surv.*, 2016. doi:10.1145/2893356. URL http://doi.acm.org.ezproxy.library.ubc.ca/10.1145/2893356. → page 23

[109] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin. Measuring architectural vulnerability factors. In *IEEE MICRO*, volume 23, . → page 5

[110] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin. Measuring architectural vulnerability factors. In *IEEE MICRO*, volume 23, . → page 30

[111] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multi-threading alternatives. In *Proceedings 29th Annual International Symposium on Computer Architecture*, pages 99–110, May 2002. doi:10.1109/ISCA.2002.1003566. → page 18

[112] M. Müller, D. Charypar, and M. Gross. Particle-based fluid simulation for interactive applications. In *SCA '03*. → page 101

[113] V. P. Nelson. Fault-tolerant computing: fundamental concepts. *Computer*, 23(7):19–25, July 1990. doi:10.1109/2.56849. → page 1

[114] D. Nicholaeff, N. Davis, D. Trujillo, and R. W. Robey. Cell-based adaptive mesh refinement implemented with general purpose graphics processing units. 2012. → pages 60, 71, 101, 102

[115] E. Normand. Single event upset at ground level. *IEEE Transactions on Nuclear Science*, 1996. → page 89

[116] M. S. R. M. V. Olga Goloubeva, Maurizio Rebaudengo. *Fundamental Principles of Optical Lithography*. Springer, 2006. → page 3

[117] K. Palem and A. Lingamneni. What to do about the end of moore's law, probably! In *Design Automation Conference (DAC), 2012*. doi:10.1145/2228360.2228525. → page 23

[118] H. e. a. Patil. Pinpointing representative portions of large intel itanium programs with dynamic instrumentation. In *MICRO-37*. → pages 65, 102

[119] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. Hpl - a portable implementation of the high-performance linpack benchmark for distributed-memory computers, 2008. URL http://www.netlib.org/benchmark/hpl. → pages 56, 60, 71

[120] A. Rahimi, L. Benini, and R. K. Gupta. Spatial memorization: Concurrent instruction reuse to correct timing errors in simd architectures. *IEEE Transactions on Circuits and Systems II: Express Briefs*, Dec 2013. ISSN 1549-7747. → page 23

[121] L. Rashid, K. Pattabiraman, and S. Gopalakrishnan. Characterizing the impact of intermittent hardware faults on programs. *IEEE Transactions on Reliability*, 64(1):297–310, March 2015. ISSN 0018-9529. doi:10.1109/TR.2014.2363152. → page 56

[122] M. Rebaudengo, M. Sonza Reorda, and M. Violante. Analysis of seu effects in a pipelined processor. In *Proceedings of the Eighth IEEE International On-Line Testing Workshop (IOLTW 2002)*, pages 112–116, July 2002. doi:10.1109/OLT.2002.1030193. → page 17

[123] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software Implemented Fault Tolerance. In *International Symposium on Code Generation and Optimization*, pages 243–254. IEEE, 2005. → pages 6, 20, 29, 31, 32

[124] J. Ren, K. Wu, and D. Li. Easycrash: Exploring non-volatility of non-volatile memory for high performance computing under failures, 2019. → page 118

[125] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, Jr. Enhancing server availability and security through failure-oblivious computing. OSDI'04. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251254.1251275. → page 22

[126] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, and T. Leu. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In *Computer Security Applications Conference, 2004. 20th Annual*, pages 82–90. IEEE, 2004. → pages 22, 23

[127] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: Approximate data types for safe and general low-power computation. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 164–174, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi:10.1145/1993498.1993518. URL http://doi.acm.org/10.1145/1993498.1993518. → pages 110, 120

[128] N. R. Saxena and E. J. McCluskey. Dependable adaptive computing systems-the roar project. In *Systems, Man, and Cybernetics, 1998. 1998 IEEE International Conference on*, volume 3, pages 2172–2177 vol.3, Oct 1998. doi:10.1109/ICSMC.1998.724977. → page 21

[129] C. Schoeny, F. Sala, M. Gottscho, I. Alam, P. Gupta, and L. Dolecek. Context-aware resiliency: Unequal message protection for random-access memories. In *ITW'17*, 2017. → page 24

[130] N. Spurrier and contributors. Pexpect is a pure python expect-like module, 2013. URL https://pexpect.readthedocs.io/en/stable/index.html. → page 65

[131] V. Sridharan and D. Kaeli. Eliminating microarchitectural dependency from architectural vulnerability. In *HPCA 2009.*, pages 117–128, 2009. → pages 5, 27, 30, 114

[132] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi. Memory Errors in Modern Systems: The Good, The Bad, and The Ugly. *ASPLOS*, 43(1):297–310, 2015. → pages 1, 86

[133] V. Sridharan, N. DeBardeleben, a. K. F. S. Blanchard, J. Stearley, J. Shalf, and S. Gurumurthi. Memory errors in modern systems: The good, the bad, and the ugly. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015. → page 67

[134] D. Stott, B. Floering, D. Burke, Z. Kalbarczpk, and R. Iyer. Nftape: a framework for assessing dependability in distributed systems with lightweight fault injectors. In *IPDPS 2000*, pages 91 –100, 2000. → page 5

[135] L. Tan, S. L. Song, P. Wu, Z. Chen, R. Ge, and D. J. Kerbyson. Investigating the interplay between energy efficiency and resilience in high performance computing. In *Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 786–796. IEEE, 2015. doi:http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7161565. → page 110

[136] D. Tao, S. Di, Z. Chen, and F. Cappello. Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1129–1139, May 2017. doi:10.1109/IPDPS.2017.115. → page 120

[137] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. Hybrid checkpointing for mpi jobs in hpc environments. In *2010 IEEE 16th International Conference on Parallel and Distributed Systems*, pages 524–533, Dec 2010. doi:10.1109/ICPADS.2010.48. → pages 25, 89

[138] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. Hybrid checkpointing for mpi jobs in hpc environments. In *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, pages 524–533, Dec 2010. doi:10.1109/ICPADS.2010.48. → pages 8, 56

[139] L. Wang, K. Pattabiraman, Z. Kalbarczyk, R. K. Iyer, L. Votta, C. Vick, and A. Wood. Modeling coordinated checkpointing for large-scale supercomputers. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 812–821, June 2005. doi:10.1109/DSN.2005.67. → pages 25, 56

[140] L. Wang, Z. Kalbarczyk, R. K. Iyer, and A. Iyengar. Vm-checkpoint: Design, modeling, and assessment of lightweight in-memory vm

checkpointing. *IEEE Transactions on Dependable and Secure Computing*, 12(2):243–255, March 2015. ISSN 1545-5971. doi:10.1109/TDSC.2014.2327967. → page 25

[141] N. Wang, M. Fertig, and S. Patel. Y-branches: when you come to a fork in the road, take it. *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on*, 2003. → page 87

[142] N. J. Wang, A. Mahesri, and S. J. Patel. Examining ace analysis reliability estimates using fault-injection. In *ISCA '07*, 2007. → pages 19, 53

[143] J. Wei and K. Pattabiraman. Blockwatch: Leveraging similarity in parallel programs for error detection. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12, June 2012. doi:10.1109/DSN.2012.6263959. → pages 21, 32

[144] J. Wei, A. Thomas, G. Li, and K. Pattabiraman. Quantifying the accuracy of high-level fault injection techniques for hardware faults. In *DSN*, June 2014. → pages 5, 28, 31, 43

[145] J. Wei, A. Thomas, G. Li, and K. Pattabiraman. Quantifying the accuracy of high-level fault injection techniques for hardware faults. In *2014 DSN*, 2014. → page 91

[146] B. Wibowo. *Cross-Layer Approaches for Architectural Vulnerability Estimation to Improve the Reliability of Superscalar Microprocessors*. PhD thesis, North Carolina State University, 2017. → page 116

[147] B. Wibowo, A. Agrawal, and J. Tuck. Characterizing the impact of soft errors across microarchitectural structures and implications for predictability. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pages 250–260, Oct 2017. doi:10.1109/IISWC.2017.8167782. → page 116

[148] P. Wu, Q. Guan, N. DeBardeleben, S. Blanchard, D. Tao, X. Liang, J. Chen, and Z. Chen. Towards practical algorithm based fault tolerance in dense linear algebra. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, pages 31–42, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4314-5. doi:10.1145/2907294.2907315. URL http://doi.acm.org/10.1145/2907294.2907315. → page 22

[149] Z. Yang, A. Jain, J. Liang, J. Han, and F. Lombardi. Approximate xor/xnor-based adders for inexact computing. In *Nanotechnology (IEEE-NANO)*, 2013. doi:10.1109/NANO.2013.6720793. → page 23

[150] K. S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, and R. Iyer. Hauberk: Lightweight silent data corruption error detector for gpgpu. In *IPDPS*, 2011. → pages 27, 32

[151] J. W. Young. A first order approximation to the optimum checkpoint interval. *Commun. ACM*, 17(9):530–531, Sept. 1974. ISSN 0001-0782. doi:10.1145/361147.361115. URL http://doi.acm.org/10.1145/361147.361115. → pages 25, 56, 75, 77, 90, 108

[152] L. Yu, D. Li, S. Mittal, and J. S. Vetter. Quantitatively modeling application resilience with the data vulnerability factor. In *SC*, 2014. ISBN 978-1-4799-5500-8. → page 19

Appendix A

Related Publications

- Lucas Palazzi, Guanpeng Li, **Bo Fang**, and Karthik Pattabiraman, "Improving the Accuracy of IR-level Fault Injection", IEEE Transactions on Dependable and Secure Computing (TDSC), accept date: Feb 2020
- Lucas Palazzi, Guanpeng Li, **Bo Fang**, and Karthik Pattabiraman, "A Tale of Two Injectors: End-to-End Comparison of IR-level and Assembly-Level Fault Injection", the IEEE International Symposium on Software Reliability Engineering (ISSRE), 2019 (Acceptance Rate: 31.4%)
- **Bo Fang**, Jieyang Chen, Karthik Pattabiraman, Matei Ripeanu and Sriram Krishnamoorthy, "Towards Predicting the Impact of Roll-Forward Failure Recovery for HPC Applications", the 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2019), Fast abstract
- **Bo Fang**, Panruo Wu, Qiang Guan, Nathan Debardeleben, Laura Monroe, Sean Blanchard, Zhizong Chen, Karthik Pattabiraman and Matei Ripeanu, "SDC is in the Eye of the Beholder: A Survey and Preliminary Study", 3rd IEEE International Workshop on Reliability and Security Data Analysis (co-located with DSN 2016), June 2016.
- **Bo Fang**, Karthik Pattabiraman, Matei Ripeanu, and Sudhanva Gurumurth, "A Systematic Methodology for Evaluating the Error Resilience of GPGPU Applications", IEEE Transactions on Parallel and Distributed Systems (TPDS), accept date: December 2015