Compiler-Directed Error Resilience for Reliable Computing

Qingrui Liu

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Engineering

Changhee Jung, co-chair

Haibo Zeng, co-chair

Binoy Ravindran

Changwoo Min

Patrick R. Schaumont

June 27, 2018

Blacksburg, Virginia

Keywords: Reliability, Compiler Optimization, Computer Architecture

Copyright 2018, Qingrui Liu

Compiler-Directed Error Resilience for Reliable Computing

Qingrui Liu

(ABSTRACT)

Error resilience has become as important as power and performance in modern computing architecture. There are various sources of errors that can paralyze real-world computing systems. Of particular interest to this dissertation are single-event errors. They can be the results of energetic particle strike or abrupt power outage that corrupts the program states leading to system failures. Specifically, energetic particle strike is the major cause of *soft error* while abrupt power outage can result in *memory inconsistency in the nonvolatile memory systems*.

Unfortunately, existing techniques to handle those single-event errors are either resource-consuming (e.g., hardware approaches) or heavy-weight (e.g., software approaches). To address this problem, this dissertation identifies idempotent processing as an alternative recovery technique to handle the system failures in an efficient and low-cost manner. Then, this dissertation first proposes to design and develop a compiler-directed lightweight methodology which leverages idempotent processing and the state-of-the-art sensor-based detection to achieve soft error resilience at low-cost. This dissertation also introduces a lightweight soft-error tolerant hardware design that redefines idempotent processing where the idempotent regions can be created, verified and recovered from the processor's point of view. Furthermore, this dissertation proposes a series of compiler optimizations that significantly reduce

the hardware and runtime overhead of the idempotent processing. Lastly, this dissertation proposes a failure-atomic system integrated with idempotent processing to resolve another type of single-event error, i.e., failure-induced memory inconsistency in the nonvolatile memory systems.

Compiler-Directed Error Resilience for Reliable Computing

Qingrui Liu

(GENERAL AUDIENCE ABSTRACT)

Our computing systems are vulnerable to different kinds of errors. All these errors can potentially crash real-world computing systems. This dissertation specifically addresses the challenges of single-event errors. Single-event errors can be caused by energetic particle strikes or abrupt power outage that can corrupt the program states leading to system failures. Unfortunately, existing techniques to handle those single-event errors are expensive in terms of hardware/software. To address this problem, this dissertation leverages an interesting property called idempotence in the program. A region of code is idempotent if and only if it always generates the same output whenever the program jumps back to the region entry from any execution point within the region. Thus, we can leverage the idempotent property as a low-cost recovery technique to recover the system failures by jumping back to the beginning of the region where the errors occur. This dissertation proposes solutions to incorporate the idempotent property for resilience against those single-event errors. Furthermore, this dissertation introduces a series of optimization techniques with compiler and hardware support to improve the efficiency and overheads for error resilience. We believe that our proposed techniques in this dissertation can inspire researchers for future error resilience research.

Acknowledgments

I never realize how time flies until I reminisce about my five-year journey in Blacksburg. I am thankful that I have lived in this peaceful and gorgeous town. I will always treat Blacksburg as my hometown in US. Especially, there are many people I want to thank here.

I want to first thank my advisor, Dr. Changhee Jung for his patience and advices along my doctoral journey. He is the best advisor for me who is a highly responsible and considerate person. My success could not be possible without his guidance. It is my pleasure to publish a series of top-tier conference papers with him. He is very supportive and constantly provides valuable insights in our research projects. I will be always grateful for his generous advices for my research and career.

I want to thank Dr. Haibo Zeng, Dr. Binoy Ravindran, Dr. Changwoo Min and Dr. Patrick Schaumont for kindly serving my committee member. They are extremely helpful throughout my doctoral journey. I do appreciate their valuable time and suggestions.

I am grateful for all the people I have met and worked with. Particularly I am thankful to meet a lot of friends with great minds at VT and at church. They did inspire and affect me

in many ways. This journey becomes much more wonderful with all these people around.

I want to thank my family for their endless love and support. They have been extremely supportive from the moment I decided to study abroad. I owe them a huge debt of gratitude and I wish I can be their pride as always. After being away from home for five years and missing every holiday for family reunion, I realize how important my family is to me. I will cherish every moment with my family ever since.

Lastly, I want to thank my girlfriend Bingyao Sun, who is the most brilliant girl I've ever seen, for being with me all these years. We met at VT and began our doctoral journey at the same time. We support and learn from each other since then. There are many failure moments in my doctoral journey. She was always there, standing by my side and encouraging me. Without her, it is not possible for me make it through my journey. We are both graduating at the same time and will soon begin a new journey in our lives. I believe our journey will continue until the end of time.

Contents

1 Introduction **1**

1.1 Background and Challenges 2

1.2 Dissertation Contributions 4

1.3 Organization 7

2 Compiler-Directed Soft Error Resilience via Tail-DMR **9**

2.1 Challenges and Contributions 10

2.2 Preliminaries 14

2.2.1 Terminologies 14

2.2.2 Sensor-Based Soft Error Detection 14

2.2.3 Error Detection Latency Exploration 16

2.2.4 Idempotent Processing for Soft Error Recovery 17

2.2.5 Fault Model 19
2.3 Clover Approach 20
2.3.1 Tail-DMR 23
2.3.2 Tail-Wait 37
2.4 Evaluation 40
2.4.1 Region Characteristics 42
2.4.2 Performance Overhead and Code Size 43
2.4.3 Sensitivity Analysis 45
2.4.4 Comparison with Tail-Wait 48
2.5 Other Related Work 50
2.5.1 Soft Error Detection and Correction 50
2.5.2 Soft Error Recovery 53
2.6 Summary 54

3 Soft Error Resilience with Idempotent Processing from Hardware's View 56

3.1 Challenges and Contributions 57
3.2 Turnstile Overview 61
3.3 Turnstile Compiler 66

3.3.1 Checkpoint Set Identification 66

3.3.2 Region Formation 67

3.3.3 Optimization for Storeless Loops 72

3.4 Turnstile Hardware Support: Region Boundary Buffer (RBB) 73

3.4.1 Region Verification and Recovery Algorithm 74

3.5 Discussion and Limitations 79

3.6 Evaluation and Analysis 80

3.6.1 Experimental Methodology 80

3.6.2 Turnstile Hardware Effect 81

3.6.3 Turnstile Compiler Effect 83

3.6.4 Overall Overhead 85

3.6.5 Exploration of Region Boundary Buffer Size 87

3.7 Other Related Work 88

3.8 Summary 89

4 Soft Error Recovery with Lightweight Checkpointing 90

4.1 Challenges and Contributions 91

4.2 Fault Model 96

4.3 Overview of Bolt 96
4.3.1 Eager Checkpointing: Guaranteed Recovery without Prohibitive Hardware Support 97
4.3.2 Checkpoint Pruning: Minimizing Performance Overhead 98
4.3.3 Fault Recovery Model 99
4.4 Bolt Compiler 100
4.4.1 Region Formation 101
4.4.2 Eager Checkpointing 101
4.4.3 Checkpoint Pruning 103
4.4.4 Checkpoint Pruning for Loop 110
4.5 Implementation 111
4.5.1 Limiting the Backtrack Depth for Recovery Time 112
4.5.2 Just-in-time Recovery Slice Generation 112
4.6 Discussion 114
4.7 Evaluation and Analysis 116
4.7.1 Experimental Methodology 116
4.7.2 The Breakdown of Checkpoint Candidates 117

4.7.3 Overheads 118

4.7.4 Fault-Recovery Overhead 122

4.8 Other Related Work 123

4.9 Summary 124

5 Error Resilience for Nonvolatile Memory Systems 126

5.1 Challenges and Contributions 127

5.2 Preliminaries 132

5.2.1 System Model 132

5.2.2 Programming Model 134

5.2.3 Idempotence 136

5.3 iDO Failure Atomicity System 137

5.3.1 The iDO Log 138

5.3.2 Indirect Locking 140

5.3.3 iDO Recovery 141

5.4 Implementation Details 143

5.4.1 Compiler Implementation 143

5.4.2 Persist Coalescing 145

5.4.3 Persistent Region Support 145

5.5 Evaluation 145

5.5.1 Performance Overhead 148

5.5.2 Scalability 151

5.5.3 Region Characteristics 155

5.5.4 Recovery Overheads 156

5.5.5 Sensitivity to NVM Latency 158

5.6 Related Work 159

5.6.1 Nonvolatile Memory 159

5.6.2 Idempotence 162

5.7 Summary 163

6 Conclusions and Future Work **165**

6.1 Summary 165

6.2 Limitations and Future Work 167

Bibliography **170**

List of Figures

2.1 Soft error detection latencies varying the number of sensors under the configurations of ARM cortex-A9 out-of-order processors where the core part takes a quarter of total die size. 15

2.2 Problem of idempotent processing in the presence of the sensor-based soft error detection scheme and its worst-case detection latency (WCDL), and our tail-DMR solution. 21

2.3 Clover Compiler Workflow 26

2.4 Tail DMR-Frontier and Code Duplication Example (a) an example control-data flow graph with tail-DMR frontiers (b) code duplication performed by Clover 35

2.5 Example of Tail Wait, (a) is the original program (b) is the tail-wait approach. 37

2.6 The distribution of the original idempotent code regions (dynamic) 41

2.7 Performance overhead of Clover vs. Full-DMR 43

2.8 Sensitivity to the WCDL where pipeline width is 2 46

2.9 Sensitivity to the WCDL where pipeline width is 3 46

2.10 Sensitivity to the WCDL where pipeline width is 4 46

2.11 Tail-wait across different WCDLs where pipeline width is 2 49

2.12 Tail-wait across different WCDLs where pipeline width is 3 49

2.13 Tail-wait across different WCDLs where pipeline width is 4 49

3.1 The region verification/recovery idea (top) and the status change of region execution (bottom) 62

3.2 The high-level view of Turnstile 65

3.3 An example of Turnstile Region Partition Heuristic 71

3.4 The high-level view of Turnstile hardware scheme 73

3.5 An illustrating example of Turnstile's hardware support. 77

3.6 Turnstile hardware effect varying different detection latencies (10, 30, 100) . 82

3.7 The average region length when partitioned for different store queue size (40, 80, 160) 84

3.8 The normalized instruction counts when partitioned for different store queue size (40, 80, 160) 85

3.9 Turnstile overhead with 40 gated store queue entries varying different detection latencies (5, 10, 30) 86

3.10 Turnstile overhead with 10 cycles WCDL varying different gated store queue size (40, 80, 160) 86

3.11 Region boundary buffer dynamic entry number with 40 gated store queue entries and 30 cycles WCDL 87

4.1 Idempotent processing and vulnerability window 93

4.2 Bolt's recovery model. 98

4.3 Motivating example with *byte_reverse* code of `sha` in MiBench: only interesting part is shown 99

4.4 An eager checkpointing example 102

4.5 Examples of unsafe checkpoint pruning. 104

4.6 A data dependence backtracking example 107

4.7 A control dependence backtracking example 109

4.8 An illustrating example of ineliminable checkpoints for loop 110

4.9 Checkpoint Breakdown 116

4.10 Performance overhead in terms of execution time (cycles) considering the architectural effect of store buffering. 118

4.11 Architectural-neutral performance overhead in terms of total dynamic instruction count. 118

5.1 Hybrid architecture model in which a portion of memory is nonvolatile, but the core, caches, and DRAM are volatile. 133

5.2 FASEs with different interleaved lock patterns. 133

5.3 iDO log structure and management: the number of iDO logs matches the number of threads created. 139

5.4 iDO compiler overview. Starting with LLVM IR from dragonegg/clang, the compiler performs three iDO phases (indicated in bold) and then generates an executable. 142

5.5 Memcached throughput (millions of data structure operations per second) as a function of thread count. 148

5.6 Redis throughput for databases with 10K, 100K, and 1M-element key ranges. 149

5.7 Throughput (millions of data structure operations per second) as a function of thread count. 151

5.8 Benchmark region characteristics: cumulative distribution of stores (top) and live-in registers (bottom) per dynamic region. 154

5.9 Sensitivity to different NVM latency (ns). 157

List of Tables

2.1 Dynamic region characteristic with 5-Cycle-WCDL 41

2.2 Code size comparison: Full-DMR vs. Tail-DMR. 44

4.1 Distribution of the time to generate recovery slice. 122

5.1 Recovery time ratio (ATLAS/iDO) at different kill times. 156

5.2 Failure Atomic Systems and their Properties 160

Chapter 1

Introduction

Error resilience is becoming indispensable nowadays since various errors can crash our computing systems resulting in thousands of millions of economic losses. This dissertation specifically addresses the single-event errors. Examples of single-event errors are energetic particle strikes and abrupt power outage. Such errors are unpredictable and will not have permanent effect on the computing systems unlike hardware stuck-at faults. This dissertation focuses on resilience techniques to protect against such errors and specifically targets two types of single-event errors, i.e., *soft error* and *failure-induced memory inconsistency issues in the emerging nonvolatile memory systems*.

1.1 Background and Challenges

Soft error is one important type of the single-event errors. Resilience against soft errors is one of the key research challenges for current and future computing systems. Soft errors have been the cause of a significant number of failures in real-world systems, ranging from embedded systems to large-scale high performance computing (HPC) systems [1, 2, 3, 4, 5, 6]. Unfortunately, due to technology scaling, electronic circuits are likely to be more susceptible to radiation-induced soft errors (also known as transient faults). They are typically caused by cosmic rays and alpha particles from packaging material. Soft errors may lead to application crash or even worse, silent data corruptions (SDC) which are not caught by the error detection logic but may cause the program to produce incorrect output. Another type of results are detected unrecoverable errors (DUE) that often directly impact the reliability of the computer systems.

In the dark silicon era, soft errors are becoming an increasingly important concern for computer system reliability. Ever-growing power density due to the limited supply voltage scaling is leading toward the advent of near-threshold computing that can improve energy efficiency by an order of magnitude, but at the expense of near-threshold voltage and lower frequency [7, 8]. However, the near-threshold voltage and the process variation make it harder to predict the response of the circuits to a particle strike, thus making them much more susceptible to the soft errors. According to Shafique et al. [9], near-threshold voltage operation may cause up to 30x higher soft error rate than nominal voltage operation. Simi-

lar trends have been observed by other researchers as well [10, 11]. Consequently, soft error resilience is essential not just for guaranteeing program correctness, but also for realizing the full potential of near-threshold voltage computing to maximize energy efficiency, which is particularly important for energy constrained computing systems.

In the emerging nonvolatile memory (NVM) systems, abrupt power outage can result in another type of single-event error. In contrast to traditional computing system with volatile DRAM main memory and nonvolatile disk storage, NVMs enables programmer to manage the persistent data with ordinary LOAD and STORE byte-addressable instructions. However, nonvolatile main memory alone does not guarantee recoverability, since caches may write data back to memory out of order resulting in memory inconsistency. For example, a failure (e.g., power outage) in the middle of a linked-list insertion may lead to a post-crash dangling reference if the next pointer of the predecessor node is written back to memory before the inserted node itself. Therefore, practical techniques to ensure the memory consistency in such nonvolatile memory systems are necessary for guaranteeing the reliable computation.

As silicon technology scaling enables designers to pile up more components into our chips and a plethora of emerging nonvolatile memory technologies are coming out, it is inevitable to face the challenges of addressing the single-event errors. These trends have motivated researchers to devise effective resilience mechanisms to mitigate the side effects of the single-event errors. Unfortunately, existing techniques often suffer from high performance overhead [12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22] or require costly hardware support and resource consumption

(e.g., occupying entire cores or leveraging special microarchitecture) [23, 24, 25, 26, 27, 28]. Despite increased hardware, performance and power costs, these techniques cannot provide guaranteed protection against the single-event errors.

Addressing the above problems poses a daunting challenge, i.e., how to provide error resilience in a way that does not increase the performance overhead, power consumption, and complexity of underlying hardware. In light of this, *this dissertation endeavors to design and propose novel error resilience technologies that can guarantee system reliability at low cost.*

1.2 Dissertation Contributions

The overall goal of this dissertation is creating tools and technologies for low-cost realization of error resilient computing systems, covering from mobile devices to high-performance large-scale computing systems. Consequently, use of the proposed techniques will make the execution of current and emerging applications much more reliable.

In this dissertation, we first identify idempotent processing [29, 30, 31, 32, 33] as a promising alternative to traditional checkpoint-restart model [34, 35, 36] for error recovery. With the idempotent processing, we make the following four major contributions for low-cost and guaranteed error resilience.

Contribution 1: Compiler Directed Lightweight Soft Error Resilience We propose Clover, a compiler directed soft error detection and recovery scheme for lightweight soft error

resilience. Clover leverages the recent advance of acoustic sensor based soft error detection for error detection. A small number of acoustic wave detectors deployed in the processor can identify soft errors by sensing the wave made by a particle strike. For error recovery, the compiler carefully generates soft error tolerant code based on idempotent processing without explicit checkpoint. Once a soft error is detected by the sensor, Clover takes care of the error as in the case of exception handling. To recover from the error, Clover simply redirects program control to the beginning of the idempotent code region where the error is detected. To prevent DUE (detected unrecoverable errors) caused by the detection latency (sensing time), which makes it possible for soft errors to escape the faulty region where they occur, we propose a novel selective instruction duplication technique called tail-DMR (dual modular redundancy) ensuring region-level error containment. The experiment results demonstrate that the average runtime overhead is only 26%, which is a 75% reduction compared to that of the state-of-the-art soft error resilience technique.

Contribution 2: Compiler-Microarchitecture Interaction for Core-Level Error Containment We introduce Turnstile, a hardware/software cooperative technique for soft error resilience which redefines the idempotent processing from the processor's point of view to contain the errors within the core reducing the runtime overhead of Clover. The key idea is to regard the committed stores of each region as unverified, holding them in a gated store queue (GSQ) until they become error-free, i.e., no sensor raises the alarm during the detection latency period after the boundary (end) of the region; a new logic called region boundary

buffer (RBB) precisely gates/releases the GSQ by tracking the end time of regions. That way the processor core never merge unverified stores to cache, and the recovery can be made by flushing the GSQ on error and restarting from the most recently verified region boundary. The upshot is that the region-level error containment is relaxed to core-level containment. That is, unlike the traditional idempotent processing, Turnstile does not require the error to be detected within the faulty region where it occurs, thereby avoiding the tail-DMR overhead incurred by Clover. The experimental results demonstrate that Turnstile can offer guaranteed soft error recovery with low performance overhead (<8% on average).

Contribution 3: Compiler Optimizations for Lightweight Idempotent Processing Prior idempotent processing techniques require expensive hardware support (e.g. ECC to the Register file) and incur in-negligible performance overhead. To address these challenges, we propose Bolt, a compiler-directed soft error recovery scheme, that provides fine-grained and guaranteed recovery without excessive performance and hardware overhead. To get rid of expensive hardware support, the compiler protects the architectural inputs during their entire liveness period by safely checkpointing the last updated value in idempotent regions. To minimize the performance overhead, Bolt leverages a novel compiler analysis that eliminates those checkpoints whose value can be reconstructed by other checkpointed values without compromising the recovery guarantee. As a result, Bolt incurs only 4.7% performance overhead on average which is 57% reduction compared to the state-of-the-art scheme that requires expensive hardware support for the same recovery guarantee as Bolt.

Contribution 4: Compiler-Directed Failure Atomicity for Nonvolatile Memory We lastly propose solution to another type of upcoming single-event error, namely failure-induced memory inconsistency in the nonvolatile memory system. For error resilience in emerging nonvolatile memory systems, we present *iDO*, a compiler-directed approach to failure atomicity with nonvolatile memory. Unlike most prior work, which instruments each store of persistent data for redo or undo logging, the iDO compiler identifies *idempotent* instruction sequences, whose re-execution is guaranteed to be side-effect-free, thereby eliminating the need to log every persistent store. The compiler then arranges, during recovery from failure, to back up each thread to the beginning of the current idempotent region and re-execute to the end of the current failure-atomic section. As a result, iDO significantly outperforms state-of-the art persistence mechanisms on current hardware during normal execution, while preserving very fast recovery times.

1.3 Organization

The rest of the dissertation is organized as follows.

- Chapter 2 describes the a compiler-directed lightweight methodology which leverages low-cost detection and recovery schemes to protect computing systems against soft errors.
- Chapter 3 proposes a hardware/software co-design approach which includes a novel error verification logic to achieve practical soft error resilience.

- Chapter 4 discusses a series of compiler optimizations for idempotent processing to address both the hardware requirement and the runtime overhead issues without compromising the reliability guarantee.

- Chapter 5 demonstrates an idempotent processing based failure-atomicity system for nonvolatile memory to guarantee the crash consistency in a lightweight manner.

- Chapter 6 concludes this dissertation and proposes possible future extensions.

Chapter 2

Compiler-Directed Soft Error Resilience via Tail-DMR

This chapter introduces Clover, a compiler directed soft error detection and recovery scheme for lightweight soft error resilience. Clover first identifies idempotent processing as a promising alternative to traditional checkpoint-restart model for error recovery. Then, Clover leverages the recent advance of acoustic sensor based soft error detection for error detection. However, simply combining idempotent recovery and acoustic sensor based detection cannot guarantee the complete error resilience. Therefore, Clover further proposes tail-DMR (dual modular redundancy) to fill the missing gap and provides a guaranteed soft error resilience mechanism.

2.1 Challenges and Contributions

Existing techniques often suffer from high performance overhead [12, 13, 14] or require costly hardware support and resource consumption (e.g., occupying entire cores or leveraging special microarchitecture) [23, 24, 25, 26]. Despite increased hardware, performance and power costs, these techniques may not eliminate both the SDC and the DUE, or need for expensive checkpointing. To address these issues, we present Clover, a compiler directed lightweight resilience scheme that can detect and correct the soft errors without the need for checkpointing and high performance overhead.

Clover leverages recent advances on a sensor-based soft error detection technique. It detects a soft error by sensing the acoustic wave generated by a particle strike rather than the consequence (e.g., program crash), thereby causing no direct performance penalty [36, 37]. For soft error recovery, Clover combines this soft error detection technique with idempotent processing [33, 38, 32]. The compiler partitions and transforms the entire program into different idempotent code regions, the re-execution of which does not change the output of the regions. Such side-effect-free re-execution enables Clover to correct the errors occurred in a region by simply jumping back to its beginning without explicit checkpoints, provided they are detected within the same region. However, naively combining the sensor-based soft error detection and the idempotent processing does not automatically guarantee correct program execution.

Curse of DUE The crux of the problem is that the sensor-based soft error detection incurs a certain detection latency. It can be minimized at the expense of adding more sensors (i.e., chip area overhead). Therefore, in practice there would be non-negligible error detection latency. Unfortunately, this makes it possible for a soft error to be detected across idempotent regions, which leads to DUE. For example, an error occurring in one idempotent region ends up being detected in the next idempotent region. Thus, simply re-executing the idempotent region will not correct those errors that have occurred in the previous region but were detected in this idempotent region whose inputs (live-in) may have been corrupted by the errors. Worse, the idempotent regions generally have a small region size (see Section 4.1) leading to more DUE as more errors cannot be detected within the region by the sensors.

To overcome this challenge, Clover intelligently augments these techniques with the instruction level dual modular redundancy (DMR) where instructions are duplicated, verified and intertwined with the original instructions. In this approach (referred to DMR hereafter), the compiler inserts checks to determine if the original instructions and their duplicated copies have the same computed values at certain synchronization points in the combined code for error detection [12]. In general, this approach can achieve zero detection latency since the checks instantly identify the soft errors. However, such an advantage comes with the significant overhead in terms of performance and power, due to the increased instruction count.

Tail-DMR To achieve low overhead, Clover attempts to minimize the use of DMR by exploiting sensor-based soft error detection. The idea is that as long as the error is detected in the same idempotent region, its re-execution can correct the error. In light of this, we proposes tail-DMR where the compiler delineates a boundary in each region to break it into two parts: (head and tail). The first part (head) relies on soft error detection via the sensors while the second part (tail) on the DMR to detect the errors, i.e., tail-DMR. We calls such a boundary *tail-DMR frontier*. In particular, the compiler determines the frontier so that the DMR-enabled part (i.e., tail) has to be longer than the worst-case sensor-based error detection latency. This ensures that all the errors are detected in the same region, enabling re-execution of idempotent regions to guarantee correct execution. Since the detection latency is typically small as shown in Section 2.2, the length of the DMR-enabled part can also be small, and hence execution of the DMR-enabled part will incur only low overhead; Section 2.4 investigates the trade-off between the sensor area overhead and performance penalty caused by the DMR execution. Consequently, Clover can transparently provide soft error resilience without significant resource consumption and performance degradation.

The following are the contributions of this work:

- We propose a novel technique to detect and recover the soft errors with low performance overhead. This is the first technique to exploit the advantages of idempotent processing, dual-modular redundancy and sensor-based support for detecting the soft errors, toward achieving this goal. We show that Clover intelligently combines these techniques and offsets the drawback of each technique to provide a low-cost and low-

overhead mechanism against the soft errors.

- We explore and quantifies the trade-offs in exploiting sensor-based support for soft error detection, instruction-level DMR, and idempotent processing. We show that these trade-offs yield a practical design point for Clover to be applied in real-world scenarios.
- Clover can detect and recover from the soft errors without significant performance degradation. Clover incurs an average performance overhead of 26% for a range of Mediabench applications which is a 75% reduction compared to that of the state-of-the-art approach. Moreover, unlike prior work, Clover does not increase code size significantly, which is particularly important for embedded systems.
- Clover's tail-DMR turns out to be superior to a new alternative to it called tail-wait that waits at the end of each region for the time of the worst-case error detection latency. For example, the tail-DMR achieves 1.06∼3.49X speedup over the tail-wait.

2.2 Preliminaries

2.2.1 Terminologies

We refer the term *inputs* to the variables that are *live-in* to a region. Such a variable has a definition that reaches the region entry and thus has the corresponding use of that definition after the region entry. This paper also refers the term *anti-dependence* to a write-after-read (WAR) dependence where a variable is used and subsequently overwritten.

2.2.2 Sensor-Based Soft Error Detection

Recently, researchers have proposed a new approach that detects the actual particle strike rather than its consequence (i.e., the program crash, hang or incorrect output) [36, 37]. For example, Upasani *et al.* [36] deploy a set of acoustic wave detectors with cantilevers on silicon and propose techniques to precisely detect the particle strike without requiring redundant micro-architectural structure. Clover relies on this kind of sensor-based soft error detection scheme.

The error detection latency determines how long the tail part of idempotent regions (tail-DMR) should be to guarantee that its execution time is greater than the detection latency. Thus, the length of the DMR-enabled part is subject to the error detection latency, i.e., a lower soft error detection latency allows a shorter DMR-enabled part (lower performance overhead) but at the expense of more sensors on the chip (higher area overhead).

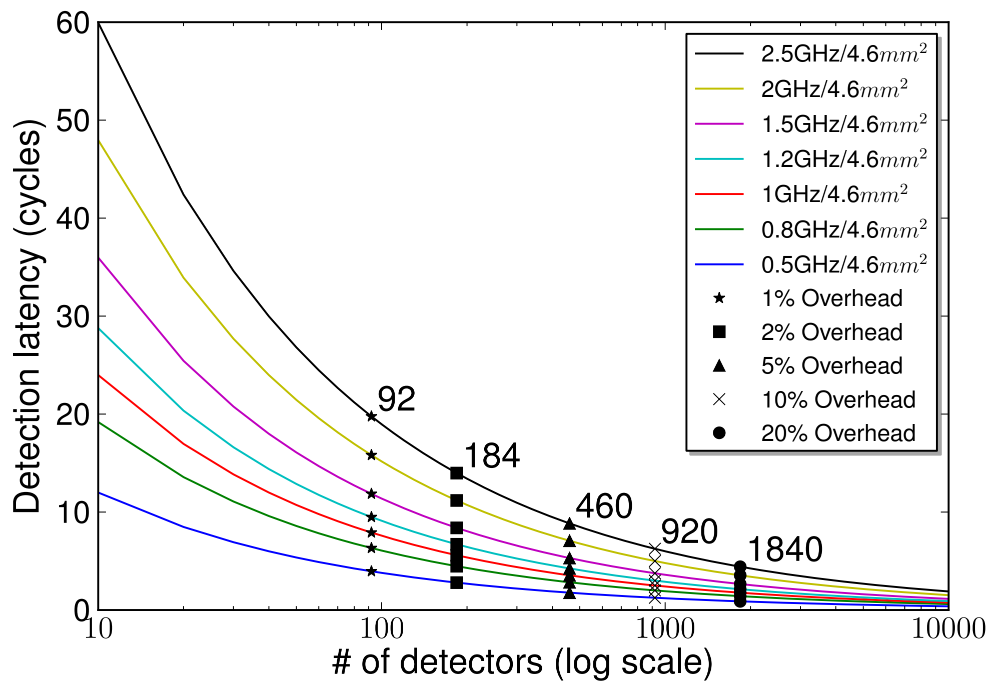


Figure 2.1: Soft error detection latencies varying the number of sensors under the configurations of ARM cortex-A9 out-of-order processors where the core part takes a quarter of total die size.

2.2.3 Error Detection Latency Exploration

To this end, we investigate possible detection latencies on various processor configurations to find an appropriate detection latency with an acceptable area overhead. Figure 2.1 shows different detection latencies for ARM cortex-A9 out-of-order processors. Leveraging data presented by Upasani et al. [36], the detection latency was calculated for a 25% core area ratio to the total die size [1]. Given a total die size (4.6 mm^2) across different clock frequencies (0.5~2.5 GHz), we vary the number of sensors to understand how the resulting detection latency changes. In the curves of Figure 2.1, we show several interesting points to represent how many sensors can be deployed within different area overhead budgets, i.e., 1% , 2%, 5%, 10% and 20%. We make two major observations:

- A short error detection latency can be achieved without increasing the area overhead significantly, e.g., detection latency of 5 cycles can be achieved only by increasing the die size by 1% with a 0.5 GHz frequency.

- As expected, lower clock frequency translates to shorter error detection latency, i.e., if NTV-like voltage scaling, which inevitably decreases the clock frequency, is used to improve energy efficiency, the resulting error detection latency will be much shorter. This exactly fits the philosophy of Clover, since it mainly targets NTV-enabled embedded systems that are particularly vulnerable to the soft errors due to the aggressive voltage scaling with NTV operation.

[1] The core part area excludes L1 and L2 caches.

Based on the exploration, we make the assumption that the sensor-based soft error detection can achieve the worst-case detection latency of 5 cycles, i.e., this is the default configuration of Clover. According to the recent work of Upasani *et al.* [37], it is possible to achieve much lower area overhead with a more careful placement of sensors on the chip. Section 2.4 later evaluates Clover across the different worst-case detection latencies varying the performance of the underlying processor, i.e., the pipeline commit-width.

2.2.4 Idempotent Processing for Soft Error Recovery

An idempotent region can be freely re-executed to generate the same output. Therefore, soft error recovery can be achieved by simply jumping back to the beginning of the faulty region and re-executing the region.

Idempotent processing partitions and transforms the entire control flow graph (CFG) of a program into different idempotent regions. That is, the entire program is protected by Clover. Each region contains a single entry basic block and zero or more exiting basic blocks where the entry block dominates all other blocks. Then, all those regions are transformed or instrumented to be idempotent. A region of code is idempotent if and only if its inputs (e.g. the values that are live-in to the region) are not overwritten, i.e., no anti-dependence on the inputs, during the execution of the region. Thus, the inputs to the entry of the region will remain the same within the region, making idempotent regions harmless to be re-executed many times. If some inputs are overwritten within the region, their values do not remain

the same as it were at the region entry. Therefore, this makes the re-execution of the region unsafe, i.e., ending up changing the expected output produced by the region. Consequently, it is a requirement for the idempotent execution that the inputs to the regions should never be overwritten during the execution of the region.

With that in mind, researchers propose different techniques for preserving the input as it is at the entry of the region. De Kruijf *et al.* [33, 38] leverages their region partition algorithm to place region boundaries to break the memory-level anti-dependence, and utilizes register renaming to eliminate the register anti-dependence (i.e., a new pseudo-register is allocated to break the dependence) on the inputs to the region. This enables the idempotence of the regions in an elegant manner without explicit checkpoint but at the expense of increasing the register pressure. Once the soft errors are detected in the idempotent region, it can be simply re-executed to recover from the errors.

On the other hand, Feng *et al.* [32] take a different approach to get around the anti-dependence without significant increase of the register pressure. They first identify all the non-idempotent regions and selectively protect some of them by explicitly checkpointing at the region entry those inputs that are overwritten within the region, i.e., all the regions are not protectable. For every protected region, a recovery block is generated to restore the checkpointed values from memory on a fault. Thus, the resulting code size increase might not be acceptable for embedded systems. Finally, the actual recovery process requires a rollback runtime that consults the recovery block.

For complete soft error recovery, Clover extends the technique of De Kruijf *et al.* because of

its simplicity (i.e., lack of explicit checkpoint and rollback), complete coverage (i.e. partition the entire program into different idempotent regions) and insignificant code size increase.

2.2.5 Fault Model

The fault model of Clover exactly follows that of idempotent processing. First, memory, caches, and register files are protected against the soft errors, e.g., using error correcting codes (ECC). Many commodity embedded processors have already integrated ECC protection to these components [39]. Second, execution of the program is guaranteed to follow its static control flow paths; we assume a low-cost, low-latency solution such as [40]. Third, instructions must write to the correct register destinations. Forth, stores in the region have to be safely buffered as with branch misprediction until the region is verified. For this purpose, a gated store buffer is often leveraged [41, 42, 43]. Finally, the address generation unit is protected for stores to write in the correct locations. Those five components are widely assumed in the literature on software-based error recovery [12, 33, 38], and there have been many solutions to realize them [26, 12, 44, 40]. All other microarchitectural units remain unchanged and can be protected by Clover. The takeaway is that Clover can protect the processor core including random logic state, that is hard to detect and correct the soft errors at low cost.

2.3 Clover Approach

The goal of Clover is to provide a low-cost hardware/software cooperative technique for soft error resilience. Given a reasonable amount of sensors and the resulting detection latency, Clover exploits a novel selective instruction duplication technique called tail-DMR (dual modular redundancy), to eliminate DUE (detected unrecoverable errors) caused by the sensing latency of error detection. For soft error recovery, Clover leverages idempotent processing. Once an error is detected, Clover recovers from it by re-executing the region where it is detected. This error recovery process is performed as in the case of an exception raised by either the sensors or tail-DMR, the exception handler of which simply redirects program control to the beginning of the region.

Achieving Complete Soft Error Recovery Although the merits of sensor-based soft error detection scheme and idempotence-based recovery scheme look complementary to each other, simply combining them together cannot always achieve correct soft error recovery. As we illustrate next, the soft errors may still corrupt the architectural state of the processor core and these schemes can not recover such soft errors correctly. We also show in Section 2.4 that such a naive combination of both the schemes ends up leaving considerable portion of dynamic instructions susceptible to the soft errors.

To illustrate this, Figure 2.2 describes the idempotence-based recovery scheme and highlights its limitation in the presence of the sensor-based soft error detection scheme and its worst-case detection latency (WCDL). Figure 2.2 (a) shows the original program execution timeline.

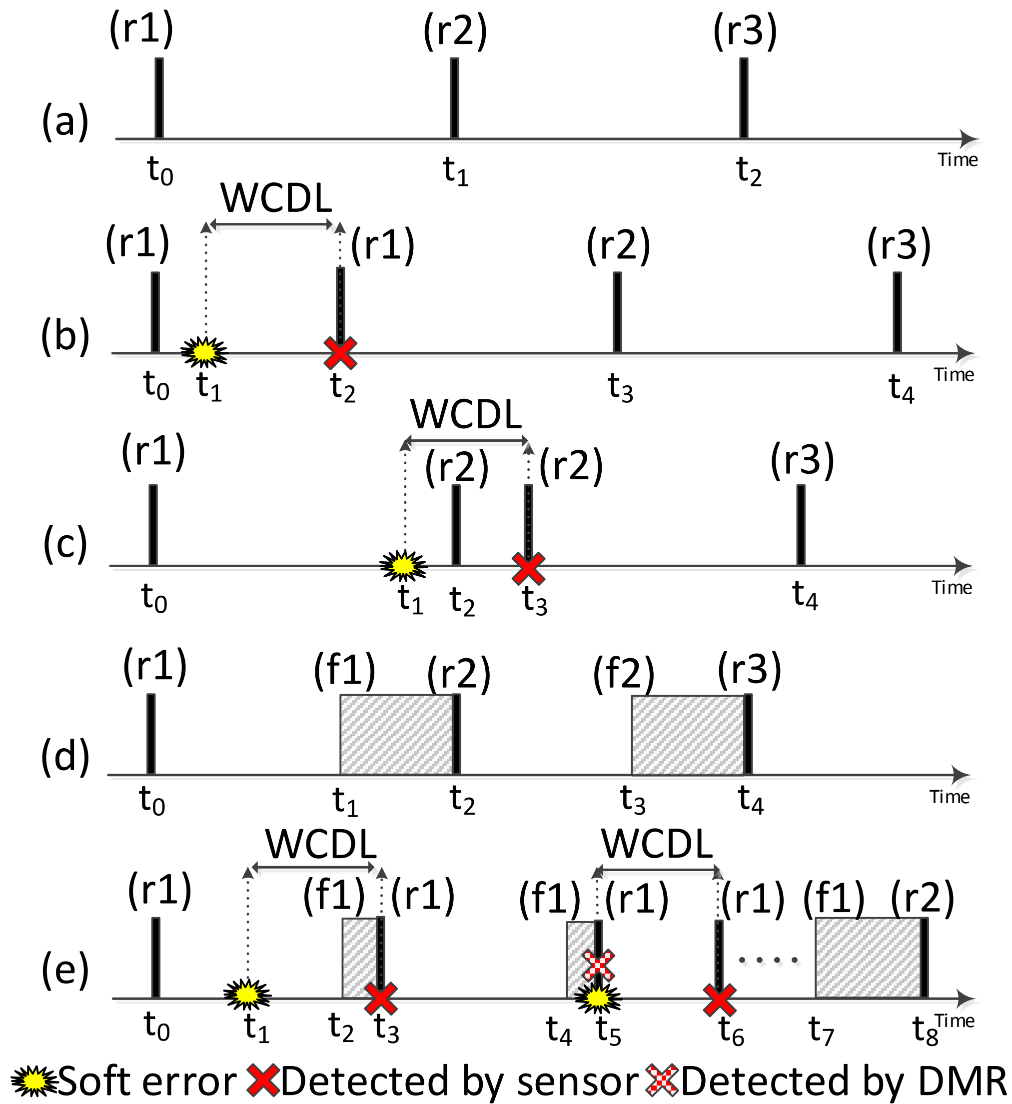


Figure 2.2: Problem of idempotent processing in the presence of the sensor-based soft error detection scheme and its worst-case detection latency (WCDL), and our tail-DMR solution.

Here, vertical bars indicate idempotent region boundaries during program execution, thus there are three regions (i.e., `r1, r2, r3`) on each timeline. Figure 2.2 (b) represents an ideal case where the idempotent region can recover correctly from a soft error. At time `t1`, an energetic particle strikes the processor and corrupts the architectural state. After the time of WCDL, the detection scheme causes an exception for the system to initiate the recovery process. Due to the idempotence of the region, the system can recover from the soft error by simply jumping back to the most recent region boundary, i.e., the beginning of the current region (i.e., `r1`) where the error is detected. Note that the region `r1` is restarted at time `t2` on the timeline.

In contrast, Figure 2.2 (c) demonstrates how the WCDL can make an error go uncorrected even in the presence of the idempotence-based recovery scheme. Suppose an energetic particle strikes the processor at time `t1`. After as much time has passed as the WCDL (i.e., at `t3`), the detection scheme causes an exception for the soft error. However, the system jumps back to the most recent region boundary (i.e., `r2`) instead of `r1` due to the worst-case detection latency. Hence, the error escapes from the former region thereby corrupting the architectural states of the processor and possibly causing a program crash/hang/silent data corruption. This is referred to as the detected unrecoverable error (DUE).

2.3.1 Tail-DMR

To overcome this challenge, we propose to utilize a reasonable amount of sensors (thereby, reducing the chip area overhead) and to selectively duplicate those instructions that are under a risk of DUE in the tail of an idempotent region (thereby, reducing the runtime overhead while maintaining the correct error recovery in all cases). we refer to this selective instruction duplication as *tail-DMR*. For a given small number of sensors and the resulting detection latency, the compiler delineates a boundary in each region to break it into two parts, head and tail; the sensor-based detector identifies the errors occurred in the head of region while the DMR identifies those occurred in the tail. We call such a boundary the *tail-DMR frontier*. Figure 2.2 (d) shows the program execution timeline after delineating the tail-DMR frontiers, where f1 and f2 represent the tail-DMR frontiers of r1 and r2, respectively. The shaded zones in the figure are protected by the tail-DMR for soft error detection. Figure 2.2 (e) describes how the proposed tail-DMR prevents the DUEs. While an error can take place outside the shaded zone at time `t1`, it can be detected still within the current region after WCDL (i.e., at time `t3`). Hence, the recovery scheme can safely redirect the program control to the beginning of the region (i.e., `r1`), thereby ensuring correct recovery from the error.

On the other hand, when an error occurs within the tail of a region (i.e., at time `t5` of Figure 2.2 (e)), the DMR immediately detects the error, and the re-execution of region `r1` can correctly recover from the error. After the time of WCDL (i.e., at `t6`), the error is detected once more by the sensor causing an exception. Thus, the program control is redirected to the beginning of the most recent region `r1` again. Note that this does not

harm the program correctness due to the side-effect-free nature of the idempotent region. Moreover, since a soft error occurs once in a while, the overhead of such redundant recovery will not have a negative impact on the performance (see Section 5.5.3).

Tail-DMR Frontier for a Region-Level Error Containment

To guarantee that each soft error occurred in each region must be detected within the same region, Clover carefully determines the *tail-DMR frontier* so that the execution time of the DMR-enabled part (i.e., tail of the region) is longer than the length (time) of the WCDL. This is required for to prevent the errors from escaping the region, where they occur, without being detected. As a result, along with such a region-level error containment, Clover's idempotence-based recovery scheme can always correctly recover from them by re-executing the region. Again, if the errors occurring in past regions remain uncorrected in the current region, re-executing it cannot achieve the recovery. However, we show that the design of Clover never allows such a case:

Theorem 1. *Given a tail-DMR frontier that makes the execution time of the DMR-enabled part longer than the time of WCDL, all the errors occurred in each region are detected in the same region.*

Proof. We provide the proof by contradiction. Suppose the argument is false, i.e., for an error occurred in the current region R_c, the error is not detected in the current region R_c . Since a region is divided into two parts by the tail-DMR, there are two possibilities for the

assumption.

- The error took place in the tail of R_c. This directly contradicts the assumption, since the DMR detects all the errors that occurs in the tail of the region. That is, if the error occurs in the DMR-enabled part (i.e., tail) of R_c, the error must be detected by R_c. This is a contradiction to the assumption.

- The error took place in the head of R_c. According to the tail-DMR frontier, the DMR-enabled part of (i.e., tail) R_c takes longer than the time of WCDL. Therefore, the error is to be identified by the sensor-based detector before the tail of R_c finishes. That is, it is impossible for the error to escape from R_c. This is another contradiction to the assumption.

Therefore, Theorem 1 must be true. □

Theorem 2. *Given idempotent processing, all the errors that take place in each idempotent region are corrected before the region finishes.*

Proof. We omit the proof due to the page limitation. It can be trivially proved by induction using Theorem 1. □

Intuitively, Theorem 1 means that all the errors occurred in a region should be detected before the region finishes, while Theorem 2 provides a strong guarantee that when program control enters a new region, all the errors that previously occurred must have already been correctly recovered. Thus, no error can escape from the region where they take place without

being detected and corrected. Clover exploits these theorems as a basis for the idempotent processing to successfully recover from all the errors occurring in each region by re-executing it.

In particular, if a region is so small that all its instructions should be protected by DMR (i.e., the tail-DMR frontier is set to the beginning of the region), the sensor may detect an error occurred in the region after it is finished. However, at this moment, the error must have already been corrected based on Theorem 2. Therefore, re-executing the most recent region (not the region where the error occurred) does not break the program correctness due to the side-effect-free nature of the idempotent region.

Consequently, the tail-DMR enables idempotent processing to correctly recover from all the errors detected in an arbitrary region by simply jumping back to the beginning of the most recent region boundary, i.e., the beginning of the current region where the error is detected. The takeaway is that Clover can eliminate detected unrecoverable errors (DUE).

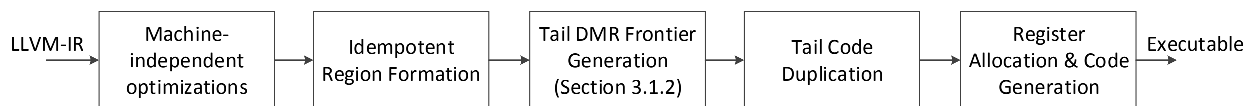


Figure 2.3: Clover Compiler Workflow

Clover Compiler Overview

Clover performs detailed compiler analyses to protect an entire idempotent code region against the soft errors. Clover introduces additional compiler backend passes to generate soft error tolerant code. Figure 2.3 shows the compilation workflow of Clover.

Once the compiler frontend translates source code into LLVM intermediate representation (IR), Clover applies traditional compiler optimizations on the IR. Then, the optimized IR goes through idempotent region formation passes which partition and transform the entire program into idempotent regions so that the regions become re-executable without any side effects. Note that, Clover applies De Kruijf's region partition algorithm [33] to partition and generate the idempotent regions. At the end of this stage, the LLVM IR is lowered to the machine-specific IR, i.e., instruction selection has already been done.

Then, the compiler computes the tail-DMR frontier of an idempotent region, and performs the tail-DMR to selectively duplicate necessary instructions and insert compulsory checking instructions for complete error recovery of the region with no DUE (detected unrecoverable error). Finally, the compiler performs register allocation and runs the rest of the backend passes to emit an executable. This section focuses on elaborating the tail-DMR frontier generation pass.

Safe Approximation of WCDL with Instruction Counting

The tail-DMR frontier pass is designed to recognize those instructions that are vulnerable to DUE in the tail of an idempotent region. As mentioned in Theorem 1, it is essential that the frontier must be properly set for the execution time of the DMR-enabled part to be longer than the time of WCDL cycles. However, static analysis cannot exactly model the execution time of the instructions.

To get around this problem, Clover conservatively represents the time in terms of the number of instructions to be executed. The time of WCDL is conservatively approximated as the product of the WCDL and the commit-width of processor's pipeline [2] , which is called $Threshold_{WCDL}$; if the WCDL is 5 cycles and the commit-width is 2, the tail-DMR forms the DMR-enabled part with only 10 instructions. Thus, Clover can match the time of WCDL by simply counting instructions starting from the end of an idempotent region.

Note that the instruction counting should be applied to the resulting instructions of the DMR, not to the original instructions. That is, during the backward traversal of a region, Clover should increase the count not just for the original instruction but also for the duplication and the check instructions to be inserted for DMR, as if the region has already been transformed by the next tail code duplication pass shown in Figure 2.3. For this purpose, the tail-DMR frontier pass leverages a cost model of DMR which categorizes the instructions of each region as presented below. Note that the unit of Clover's cost model is in terms of instruction as we approximate the time with the number of instructions discussed above.

Synchronization Instructions Instructions in this category require immediate verification [12]. Originally, store and control flow instructions fall into this category. They require equivalence verification to detect the soft errors, e.g., for store instructions, the compiler

[2]Such approximation is safe as the commit-width is the ideal IPC number for a processor. The number of instructions being committed in each cycle can never exceed the ideal IPC number. Further optimization such as best-case execution time (BCET) analysis can be applied to reduce the number of instructions to be duplicated. For example, SIMD instructions may need more than one cycle to be executed. However, this is out of the scope of this dissertation, and we leave it as future work along with a profile-guided region multi-versioning [45] based on adaptive execution techniques [46, 47, 48].

inserts check instructions to compare the value of original operands of the store instruction with their duplicated counterpart. If any mismatch detected, tail-DMR raises an alarm to invoke the recovery process. In particular, the tail-DMR considers a region boundary as a new synchronization point. This is necessary to prevent the errors occurring in the tail of a region from escaping to the following regions. That is, any live-out registers, that are defined in the tail of the region, are required to have check instructions before the region boundary. We define a synchronization instruction set as these three types of instructions, and denote the set and the cost of the instructions as $\mathbb{SYN}$ and C_{syn}, respectively. The value of C_{syn} is one instruction, thus it does not include the cost of check instructions which is modeled separately.

Duplication Instructions These instructions are supposed to be duplicated by the compiler thus generating one additional instruction in the region. Note that these instructions are only in the tail of the region. Unlike traditional DMR approaches, Clover duplicates all the instructions from the tail-DMR frontier to the end of the region. Again, all the synchronization instructions will not be duplicated. The cost of the duplication instructions is denoted as C_{dup}. In general, the cost is two instructions; one for the original instruction and the other for the duplication instruction. In particular, C_{dup} of PHI instructions is zero, since the compiler eliminates them in the step of static-single-assignment (SSA) deconstruction for register allocation.

Safe Instructions These instructions are in the head of the region, preceding the tail-DMR frontier. In particular, they are not vulnerable to DUE as long as the tail-DMR is correctly applied. Every error occurring in these instructions will be detected within the region (i.e., before it finishes). Thus, safe instructions are never duplicated, i.e., there is no cost associated with them.

Check Instructions These instructions are supposed to be inserted to verify the operands and the *live-out* value of the instructions or the region boundaries. These instructions are basically comparisons to check the equivalence of the values in the original instruction and the duplicated one; Section 2.3.1 illustrates the insertion of check instructions in more detail. The cost of check instructions is denoted as C_{ck}. We define a check instruction set as those instructions whose defined register needs to be verified at synchronization points, and denote the set and the cost of check instructions as $\mathbb{CK}$ and C_{ck}, respectively. Note that the C_{ck} depends on underlying architecture. For example, the C_{ck} would be two instructions for ARMv7-A instruction set; one instruction for the compare instruction that updates the condition register, and the other one for the branch instruction that transfers the control flow based on the result of the condition register.

The next section presents the detailed algorithm of the tail-DMR frontier computation leveraging the cost model.

Algorithm 1 Region-based Vulnerability Analysis

Inputs: **CDFG**, $\mathbb{SYN}, \mathbb{R}$ (i.e., a set of idempotent regions)
Outputs: $\mathbb{TF}, \mathbb{CK}$

1: **function** UPDATEVULREGSET($\mathbb{VR}$, I, $\mathbb{PATHI}$)
2: // DefR is a register defined by I
3: // $\mathbb{KILL}$[DefR] is the set of instructions that kill DefR.
4:
5: **if** $I \in \mathbb{SYN}$ **then**
6: $\mathbb{VR} \leftarrow \mathbb{VR} + I's$ use registers
7: **else if** $\mathbb{KILL}$[DefR] $\bigcap \mathbb{PATHI} = \emptyset$ **then**
8: $\mathbb{VR} \leftarrow \mathbb{VR}$+DefR
9: **end if**
10: Return $\mathbb{VR}$
11: **end function**
12:
13: **function** CALCULATECOST(C, I)
14: **if** $I \in \mathbb{SYN}$ **then**
15: $C \leftarrow C + C_{syn}$
16: **else**
17: $C \leftarrow C + C_{dup}$
18: **end if**
19: **if** DefR $\in \mathbb{VR}$ **then** // DefR is the register defined by I
20: $C \leftarrow C + C_{ck}$
21: $\mathbb{CK} \leftarrow \mathbb{CK} + I$
22: **end if**
23: Return C
24: **end function**
25:
26: **for** each region boundary $R \in \mathbb{R}$ *in* **CDFG do**
27: **for** each path P in reverse-DFS order from R **do**
28: $\mathbb{PATHI} \leftarrow \emptyset$
29: $\mathbb{VR} \leftarrow \emptyset$
30: $C_{path} \leftarrow 0$
31: **for** each Instruction $I \in P$ **do**
32: $\mathbb{PATHI} \leftarrow \mathbb{PATHI} + I$
33: $\mathbb{VR} \leftarrow$ UPDATEVULREGSET($\mathbb{VR}, I, \mathbb{PATHI}$)
34: $C_{path} \leftarrow$ CALCULATECOST(C_{path}, I)
35: **if** $C_{path} \geq Threshold_{WCDL} \vee I \in \mathbb{R}$ **then**
36: $\mathbb{TF} \leftarrow \mathbb{TF} + I$
37: Terminate path P
38: **end if**
39: **end for**
40: **end for**
41: **end for**

Region-based Vulnerability Analysis: Computing the Tail-DMR Frontier on SSA

Clover iterates the instructions of an idempotent region backward from its end by traversing the control data flow graph (CDFG). For counting the instructions to match the time of WCDL for each path, Clover consults the cost model of each visited instruction to appropriately increase the count (i.e., path cost) depending on how the tail-DMR treats the instruction as discussed above. If the count reaches a threshold that represents the time of WCDL (i.e., $Threshold_{WCDL}$), then Clover adds the last visited instruction to the tail-DMR frontier of the region. Before discussing the details, we define the following terms and notations that are used throughout this section.

- $\mathbb{VR}$: Vulnerable register set includes the registers whose value may corrupt the architectural state if it is not verified.
- $\mathbb{CK}$: Check instruction set denotes the instructions whose definition register needs to be verified with checking instructions during the tail-DMR duplication.
- $\mathbb{TF}$: A set of instructions that belong to tail-DMR frontier.
- $\mathbb{PATH}$: A set of visited instructions along one path during the reverse depth-first-search traversal.
- C_{path}: Accumulated cost of the visited instructions on the current path. If the cost reaches $Threshold_{WCDL}$, then the last-visited instruction is added to the tail-DMR frontier.

- $\mathbb{KILL}$: A function that maps each defined register to a set of instructions that *kill* the register.

Algorithm 1 describes how to compute the tail-DMR frontier. Starting from each region boundary in the CDFG, Clover traverses all the paths in a reverse depth-first-search (RDFS) order (line 26 $\sim$ 41). Each path is first initialized and keeps track of its own set of visited instructions ($\mathbb{PATHI}$), vulnerable registers ($\mathbb{VR}$) and path cost (C_{path}) during the RDFS traversal (line 28 $\sim$ 30).

For each visited instruction in the path, Clover updates $\mathbb{PATHI}$, $\mathbb{VR}$, and C_{path}, correspondingly (line 32 $\sim$ 34). To update $\mathbb{PATHI}$, Clover simply inserts the visited instruction I into $\mathbb{PATHI}$. Keeping track of visited instructions is beneficial for analyzing the liveness of a register in static-single-assignment (SSA) form which will be used in computing the $\mathbb{VR}$ set. Then, to update the $\mathbb{VR}$, Clover leverages a heuristic shown in line 1 $\sim$ 11. If the visited instruction is a synchronization instruction, Clover adds all its use registers to the $\mathbb{VR}$ set since the value in those use registers may corrupt the architectural state if it is not verified. For example, if the operands of a store instruction are corrupted, the store may stores value to some other memory location which may break the idempotent property. In the next pass (i.e., the tail code duplication in Figure 2.3), necessary check instructions are therefore inserted to the original CDFG for verifying these registers.

In particular, Clover does not need to protect those registers that are live-in at the tail-DMR frontier. As discussed in the proof of Theorem 1, all the errors occurring before the tail-DMR

frontier should be dealt with by the sensor-based soft error detection within the region, i.e., its re-execution can correctly recover from the errors. This allows Clover to safely assume that all the live-in registers at the tail-DMR frontier are resilient against the soft errors. To this end, the next code duplication pass does not insert check instructions for such live-in registers even if they belong to the $\mathbb{VR}$ set.

Recall that Clover considers the region boundary as a synchronization point. That is, every live-out register at the end of the region must be verified by the region if the live-out register is defined after the tail-DMR frontier. Line 7 in Algorithm 1 shows how Clover can easily compute such a live-out register on the SSA form. Suppose $\mathbb{KILL}$[DefR] is the set of instructions that kill the definition register DefR. Then, the intersection of $\mathbb{KILL}$[DefR] and $\mathbb{PATH}$ represents whether DefR is killed before the region ends. If the intersection is empty, i.e., the DefR is not killed which means it is live-out across the end of the region, the DefR is added to $\mathbb{VR}$ for verification.

After the $\mathbb{VR}$ is updated, Clover calculates the path cost (i.e., C_{path}) based on the $\mathbb{VR}$ set and the instruction type of current visiting instruction. Clover accumulates the cost of visited instructions which is determined differently depending on whether the instruction is a synchronization instruction or it is a duplication instruction (line 14 $\sim$ 18). In addition, if the register defined by the visiting instruction I belongs to $\mathbb{VR}$ set, a check instruction cost (C_{ck}) is added to C_{path} (line 19 $\sim$ 22). Accordingly, the instruction I is added into $\mathbb{CK}$ set to inform the next code duplication pass of where check instruction needs to be placed.

In line 35 ∼ 38, Clover terminates one path if C_{path} reaches the $Threshold_{WCDL}$, i.e., the time of WCDL is matched. Thus, Clover adds the last-visited instruction to the tail-DMR frontier (line 36). Currently, the default values of the WCDL and the commit-width are 5 and 2, respectively, i.e., the $Threshold_{WCDL} = 10$. In addition, Clover also terminates the path cost calculation process when another idempotent region boundary is encountered, i.e., the region is too short. In this case, Clover simply considers the boundary instruction as the frontier, thus all the instructions of such a short region are protected by DMR.

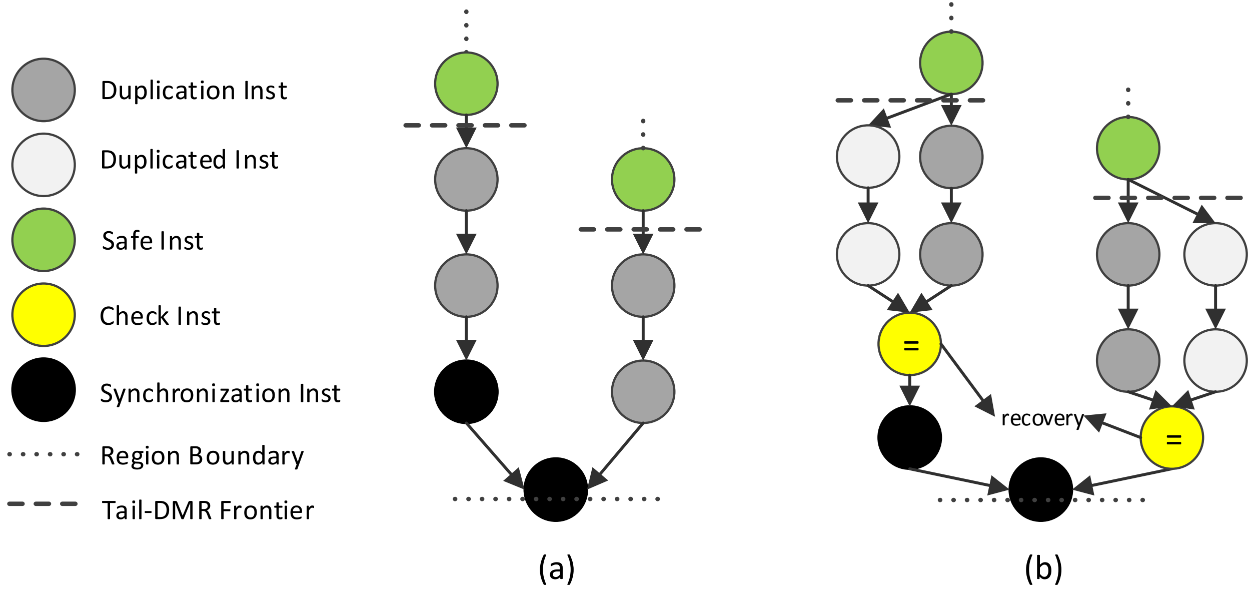


Figure 2.4: Tail DMR-Frontier and Code Duplication Example (a) an example control-data flow graph with tail-DMR frontiers (b) code duplication performed by Clover

Example Figure 2.4 demonstrates an example of identifying the tail-DMR frontiers and performing code duplication to the program with tail-DMR frontiers. Figure 2.4 (a) shows an example control-data flow graph. Clover recognizes the tail-DMR frontiers on a path-sensitive basis which means Clover will iterate all the paths beyond the end of the region.

Starting from the region boundary (one of synchronization instructions), Clover traverses the paths in a reverse depth first search order. It accumulates the cost of each path based on Algorithm 1 and categories the instructions (Section 2.3.1) until it meets the threshold or other region boundaries. As we can see, the instructions between the tail-DMR frontiers and region boundary are categorized as duplication instructions except for the synchronization instructions. After that, Clover thus performs code duplication to the duplication instructions and inserts check instructions right before the synchronization instruction as needed (Figure 2.4 (b)). In this way, Clover protects the tail part of the region with DMR.

Discussion and Limitation

SDC Even if an energetic particle strike is the major source of soft errors, they can also be induced by other sources, e.g., random noise such as inductive/capacitive crosstalk and power supply noise. Since these sources are not covered by the sensor-based soft error detection, Clover might generate silent data corruption (SDC).

Multiple Soft Errors Occurring in One Region They can be easily handled by Clover. As stated in Section 2.3.1, all the errors that happen in the same region are guaranteed to be detected and corrected by Clover.

Multi-threaded Applications Although we do not evaluate Clover in the context of multi-threaded applications, Clover is capable of handling multi-threaded programs. Both

DMR [12] and idempotent processing [33, 38, 32] have well addressed the problems with multi-threaded programs. In addition, there are no conflicts if we extend both DMR and idempotent processing to support multi-threaded applications. Therefore, Clover can be easily extended to support multi-threaded programs.

Exception Handling Soft error may lead to exception events such as divide by zero or segmentation fault. We advocate postponing the exception handling service by waiting for WCDL cycles until all potential errors have been detected as with prior work [36]. In this way, Clover can achieve the region-level error containment even in the presence of such exceptions.

2.3.2 Tail-Wait

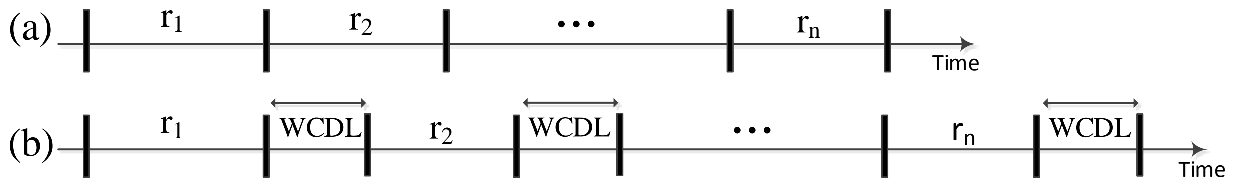


Figure 2.5: Example of Tail Wait, (a) is the original program (b) is the tail-wait approach.

Tail-wait is a straight-forward alternative to Clover. In order to avoid DUE in the tail region, the system can just wait for WCDL cycles at the end of each region to detect any potential errors as shown in Figure 2.5. In fact, it is possible for tail-wait to outperform Clover for some situations. This is because Clover's way to estimate the WCDL time is very conservative, i.e., assuming the ideal processor performance to determine the tail-DMR

frontier for the region-level error containment. For example, even if the SIMD instruction may take more than one cycle to execute, Clover calculates the execution time as one over the *commit_width* of the underlying processor. The rest of this section presetns the analysis of the situations where Clover (tail-DMR) outperforms tail-wait in theory and vice versa.

First, Clover's performance overhead can be represented as additional cycles due to the instruction duplication, using equation 2.1.

$$Overhead_{Clover} = \frac{WCDL \times commit_width}{2 \times IPC} \tag{2.1}$$

where IPC (Instructions Per Cycles) is the actual performance number delivered by the processor. The equation assumes that half of the $WCDL \times commit_width$ instructions are those that are additionaly inserted for Clover to duplicate the original instructions. Here, Clover's IPC is assumed to be the same as tail-wait's for ease of presentation. Note that this rather underestimates the performance of Clover as the DMR is known to increase the IPC significantly [12], i.e., Clover's IPC is likely to be higher than tail-wait's.

As the tail-wait's performance overhead is just WCDL for each region, we can derive equation 2.2 from equation 2.1 to analyze the situation where Clover outperforms tail-wait.

$$\frac{WCDL \times commit_width}{2 \times IPC} \leq WCDL \tag{2.2}$$

Therefore, we have:

$$\frac{commit_width}{2} \leq IPC \tag{2.3}$$

From equation 2.3, we can draw the conclusion that if the IPC number of Clover is greater or equal to half of the commit width of the underlying processor, Clover can always perform better than tail-wait. For example, when the commit width is 2, Clover can outperform tail-wait provided Clover's IPC is greater than 1. In contrast, if the processor suffers from many long latency instructions within a short amount of time, such that the IPC drops down below 1, then tail-wait can be a better choice over Clover.

Note that if the region size is smaller than $WCDL \times commit_width$, Clover can be superior over tail-wait, even though the IPC is less than the half of the commit width violating the inequality in equation 2.3. As an extreme example, when a region consists of only a few instructions, Clover duplicates them all since its execution time is too short to match the WCDL. On the contrary, after such a short region ends, tail-wait fully waits for WCDL cycles that would be longer than the execution time of the additional instructions inserted by Clover.

In Section 2.4.4, we empirically evaluate the tail-DMR compared to the tail-wait to support our analyses on their performance overhead varying WCDL and underlying processor configurations.

2.4 Evaluation

We implement the compiler passes of Clover on top of LLVM Compiler Infrastructure [49]. The idempotent region formation algorithm is also integrated in LLVM. We perform the experiments with 17 applications from Mediabench [50] and MIbench [51] benchmarks in different categories. All the applications were compiled with standard -O3 optimization. We conduct our simulations on Gem5 [52] with system call emulation mode for a modern 2-issue out-of-order 0.5 GHz processor whose L1 (2-way/2-cycles) and L2 (8-way/20-cycles) LRU caches are 32KB and 2MB, respectively. The pipeline widths are all 2 including commit-width, and the ROB and physical integer RF have 128 and 256 entries, respectively.

We first analyze the length of idempotent regions, since it is a critical factor that affects the performance of Clover; in general, the longer region, the better performance. For example, in longer regions, the portion of the DMR-enabled part is relatively small, whereas in short regions, the majority of their instructions have to be duplicated by DMR. Then, we analyze the execution time overhead of Clover comparing it to the state-of-the-art technique, i.e., combination of idempotent processing and full-DMR [33]. Finally, we provide sensitivity analysis results to understand the trade-off between the sensor area overhead and the resulting performance of Clover.

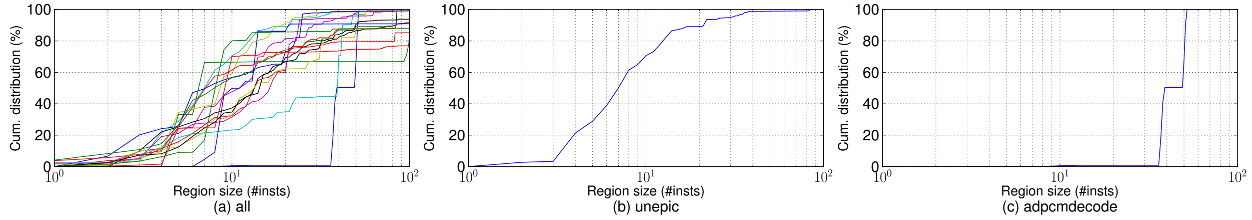


Figure 2.6: The distribution of the original idempotent code regions (dynamic)

Application	#Total insts (10^3)	#Total regs (10^3)	Aver. leng.	#Vulner. insts (10^3)	Vulner. insts ratio
adpcmdecode	6557	149	44	1486	22.66%
adpcmencode	8346	223	37	1231	14.75%
epic	59759	1186	50	8370	14.00%
unepic	9898	941	10	6847	69.17%
jpegdecode	4382	280	15	2252	51.39%
jpegencode	17335	1205	14	9335	53.85%
mesatexgen	204820	8497	24	68748	33.56%
pegwitencrypt	35616	2600	13	18586	52.18%
g721decode	512016	14239	35	94027	18.36%
g721encode	268789	7766	34	51155	19.03%
gsmdecode	68406	2270	30	19625	28.68%
gsmencode	110750	3350	33	27756	25.06%
mpeg2decode	165491	5792	28	49946	30.18%
mpeg2encode	1320760	17867	73	143323	10.85%
sha	120338	1121	107	9530	7.91%
susanedges	78967	2190	36	15526	19.66%
susancorners	27265	455	59	2823	10.35%
geomean	58368	1822	31	13736	23.53%

Table 2.1: Dynamic region characteristic with 5-Cycle-WCDL

2.4.1 Region Characteristics

Figure 2.6 (a) shows a cumulative distribution of dynamically executed idempotent regions of all the applications listed in Table 2.1. The x-axis (in log scale) represents the number of instructions in regions. We highlight *unepic* and *adpcmdecode*. As shown in Figure 2.6 (b), the majority of regions in *unepic* are comprised of less than 10 instructions, and they occupy considerable amount of the total execution time. The implication is that the tail-DMR will cause significant performance overhead for *unepic*. In contrast, *adpcmdecode* has many long regions as shown in Figure 2.6 (c). That is, most of the regions are long enough to hide the performance penalty caused by the tail-DMR, thus it will cause negligible performance overhead for *adpcmdecode*.

Table 2.1 further details on the dynamic region characteristics of the benchmark applications. Column 2 and 3 show the dynamic instruction count and the number of idempotent regions executed, respectively. Column 4 presents the average region length, i.e., (2nd column/3rd column). The geometric mean of the average region length is 31. Such fine-grain recovery is beneficial because it guarantees the continuity of program execution in the event of soft errors. The system just needs to rollback to the beginning of the faulty region and re-execute the region, whose average length is 31, making the recovery overhead negligible to the users. In contrast, restarting the program will disrupt the program continuity and affect the users. In the extreme case, the sensor could raise an alarm every 1.3 minutes [36], the program can never finish if it takes more than 1.3 minutes. However, Clover just needs to pay the negligible recovery overhead of re-executing 31 instructions per fault.

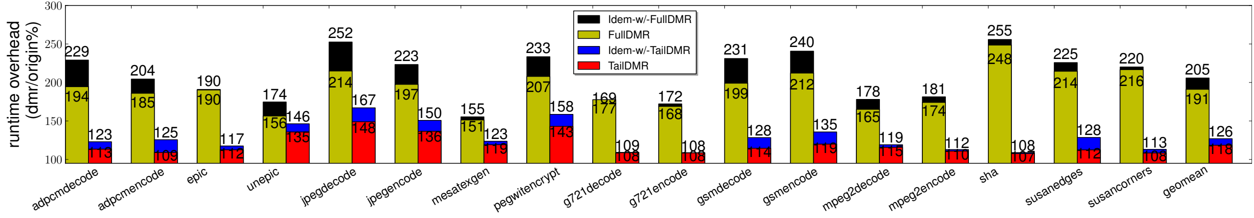


Figure 2.7: Performance overhead of Clover vs. Full-DMR

Column 5 and 6 represent the total number of vulnerable instructions and the ratio (i.e., 5nd column/2th column), respectively. Note that simply relying on the naive combination of the idempotent processing and sensor-based detection scheme leave all the regions potentially vulnerable to DUE. On average, a total 23% of dynamic instructions are vulnerable to the soft errors. This indicates that only a small portion of instructions need to be protected by the tail-DMR. More precisely, the portion is almost cut in half in that the number of instructions are doubled after the DMR is performed, i.e., not all the vulnerable instructions are protected by Clover. This is because the tail-DMR inserts duplication and check instructions to the original region, which allows some vulnerable instructions to be placed beyond the tail-DMR frontier. Consequently, Clover can achieve much lower performance overhead compared to full DMR.

2.4.2 Performance Overhead and Code Size

Figure 2.7 represents the runtime overhead of different soft error resilience schemes, which is normalized to the baseline execution time with no resilience scheme. For each application, the first bar corresponds to the runtime of the state-of-the-art scheme [33] where full-DMR

Application	#Full DMR insts	#Tail DMR insts	Insts reduction ratio	Full DMR binary size increase ratio	Tail DMR binary size increase ratio	Binary size reduction ratio
adpcmdecode	408	146	64.21%	47.24 %	1.34 %	97.17%
adpcmencode	411	146	64.47%	47.24 %	1.34 %	97.17%
epic	9314	5443	41.56%	106.54%	58.96 %	44.66%
unepic	7179	5210	27.42%	108.46%	69.70 %	35.74%
jpegdecode	52209	34728	33.48%	124.79%	82.13 %	34.19%
jpegencode	50282	33326	33.72%	130.77%	79.86 %	38.93%
mesatexgen	183345	104805	42.83%	128.74%	68.18 %	47.04%
pegwitencrypt	12297	7144	41.90%	96.63 %	50.01 %	48.24%
g721decode	2524	1393	44.80%	66.87 %	34.81 %	47.95%
g721encode	2659	1481	44.30%	68.02 %	35.88 %	47.26%
gsmdecode	10687	5490	48.62%	68.65 %	31.20 %	54.55%
gsmencode	10687	5490	48.62%	68.65 %	31.20 %	54.55%
mpeg2decode	15884	9580	39.68%	107.01%	58.29 %	45.53%
mpeg2encode	23680	11528	51.31%	112.55%	49.24 %	56.25%
sha	857	407	52.50%	61.02 %	26.05 %	57.30%
susanedges	7187	2522	64.90%	102.50%	33.03 %	67.77%
susancorners	7187	2522	64.90%	102.50%	33.03 %	67.77%
geomean	7552	3858	46.23%	86.60 %	30.42 %	53.02%

Table 2.2: Code size comparison: Full-DMR vs. Tail-DMR.
gsmencode and *susancorners* share the same binaries with *gsmdecode* and *susanedges*, respectively. Therefore, they have the exact same data.

is combined with idempotent processing, while the second bar to the runtime of Clover. In the figure, each bar is broken into two parts; the bottom and the top represent the overhead of error detection and recovery, respectively. For example, the top parts of the first and the second bars (i.e., *Idem-W/-FullDMR* and *Idem-w/-TailDMR*) represents the overhead due to idempotence-based error recovery in the presence of full-DMR and Clover's tail-DMR, respectively. As shown in Figure 2.7, the idempotence-based recovery is not that significant i.e., on average 14% (*Idem-w/-FullDMR*) and 8% (*Idem-w/-TailDMR*. Thus, most of the overhead is caused by the error detection schemes, i.e., on average, 91% (*FullDMR*)

and 18%(*TailDMR*). Overall, the full-DMR with with idempotent processing incurs 105% runtime overhead on average. In contrast, Clover incurs only 26% runtime overhead on average, which is a 75% reduction, at the expense of only 1% chip area overhead. Figure 2.7 also confirms that the length of regions is critical to Clover's performance overhead (i.e., 2nd bar). The general trend is that the higher ratio of vulnerable instructions shown in Table 2.1 translates to higher performance overhead.

Table 2.2 summarizes the code size increase of the full-DMR with idempotent processing versus Clover. The number of additional static instructions inserted to the original program is represented in Column 2 (the full-DMR approach) and Column 3 (Clover). Clover achieves on average a 46% static instruction reduction when compared to the full-DMR approach as shown in Column 4. With the importance of binary size in embedded systems in mind, we also show the ratio of the binary size increase to the original binary size for the full-DMR approach and Clover in Column 5 and Column 6, respectively. Overall, the average binary size increase of the full-DMR approach is 86%, whereas that of Clover is only 30%. Clover achieves on average a 53% binary size reduction when compared to the full-DMR approach as shown in Column 7.

2.4.3 Sensitivity Analysis

We investigate the factors that affect Clover in this section. As the overhead of Clover mostly comes from the tail-DMR, the fraction of the tail of each region protected by DMR

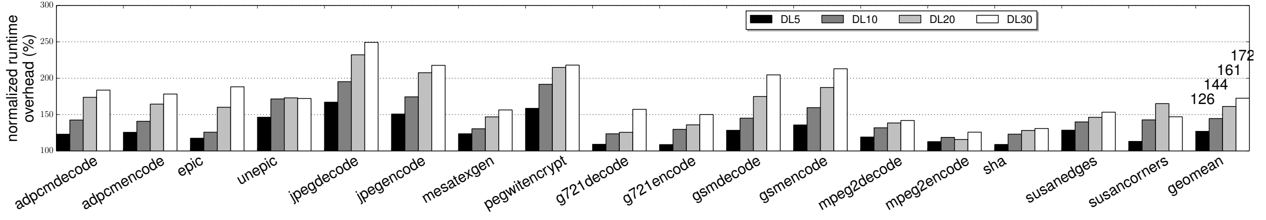


Figure 2.8: Sensitivity to the WCDL where pipeline width is 2

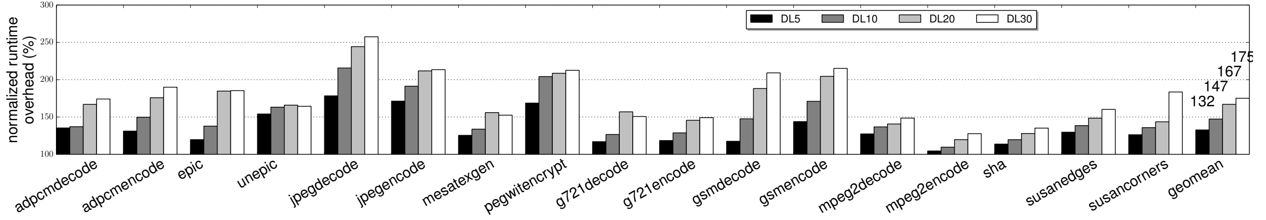


Figure 2.9: Sensitivity to the WCDL where pipeline width is 3

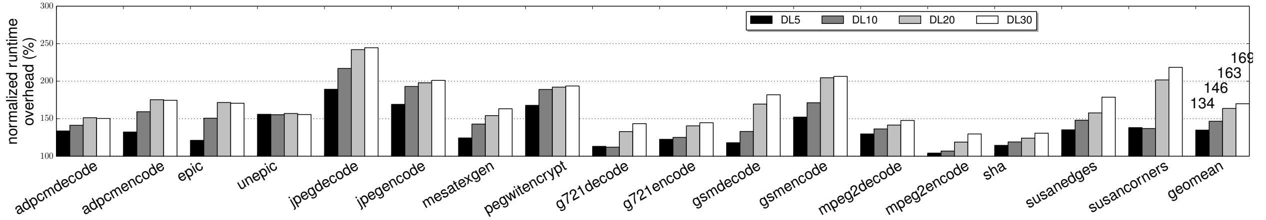


Figure 2.10: Sensitivity to the WCDL where pipeline width is 4

will impact the resulting performance of Clover. The WCDL and pipeline width are two critical factors that determine the fraction of tail region with tail-DMR frontiers. As shown in Figure 2.8∼2.10, we perform sensitivity analysis by varying the pipeline width and WCDL to see the impact of those two factors to the performance. Figure 2.8, 2.9 and 2.10 shows the normalized runtime overhead with respect to the baseline without instrumentation with different pipeline width from 2, 3, to 4, respectively. For each configuration, we vary the WCDL from 5, 10, 20, to 30.

For all cases, the performance overhead of Clover generally increases (from 26∼34% to 69∼75%) as the worst-case detection latency increases (from 5 to 30). The results were

expected because the number of instructions that need to be duplicated for tail-DMR becomes larger with longer WCDL. Note that the sensor-based detection scheme can adjust its WCDL by varying the amount of sensors being deployed and their placement on the processor core. However, as illustrated in section 2.2.3, a larger amount of sensors will result in higher area overhead. We observe that a few applications such as susancorners and mpegencode deviate from this general trend. The abnormal L1 instruction cache access (including hits and miss) increase in these applications can account for these aberrant cases. For example, we observe around 20% increase of L1 instruction cache access cycles in susancorners when the detection latency is 20 which in turn increase the performance overhead making it an exception. We suspect that DMR affects the code layout triggering such abnormality.

When the detection latency becomes large enough (e.g. 20), the performance overhead does not increase significantly as the detection latency keeps growing (from 20 to 30). The excessive number of small regions can account for such performance overhead saturation phenomenon. As shown in Figure 2.6 (a), most of the application have more than 50% of total dynamic regions whose region size is less than 10 instructions. This implies that when the detection latency becomes large enough, most of the regions need to be fully protected by DMR because the sensor cannot detect the error within the region in the worst case. Therefore, the performance overhead becomes saturated when most of the region are fully protected by the DMR. However, Clover is still effective to reduce the performance overhead as large regions still dominate in the total program execution time. For example, we assume a program comprised of 100 regions where 99 regions' size is 10 and only 1 region's size is

10000. The program spends more than 90% of the time in the large region. Therefore, Clover can effectively reduce the performance overhead with tail-DMR compared to full DMR in those large regions.

On the other hand, we also observe that the performance overhead of Clover does not degrade significantly when the pipeline width increases (from 2 to 4) even though the number of vulnerable instructions increases as the pipeline width grows. Sometimes, the performance overhead even decreases as the pipeline width increase. Such performance variants were generally expected. This is because as the pipeline width increases, more instruction-level parallelism can be exposed to the processor thus increasing the IPC as the duplicated instructions generally do not depend on the original instructions. Prior work (e.g. Swift [12]) also observe the same phenomenon.

2.4.4 Comparison with Tail-Wait

Tail-wait is a straight-forward way to guarantee all the errors to be detected within their regions. In this section, we compare the overhead of Clover with that of tail-wait. We implemented the tail-wait by inserting nops to the end of each region. The number of nops to be inserted is determined as the product of WCDL and pipeline width. We normalize the performance overhead of tail-wait to the same baseline similar to the previous section.

Figure 2.11∼2.13 show that Clover is consistently better than tail-wait across different configurations. We confirm that the IPC numbers in Clover are always greater than half of the

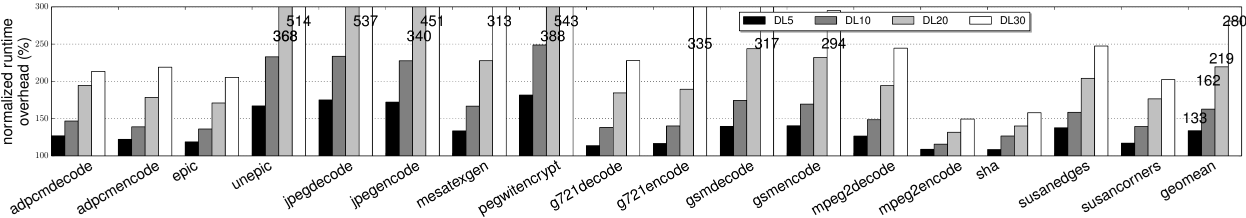


Figure 2.11: Tail-wait across different WCDLs where pipeline width is 2

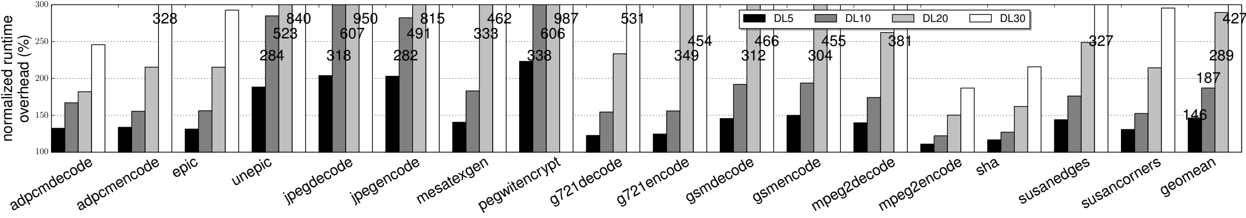


Figure 2.12: Tail-wait across different WCDLs where pipeline width is 3

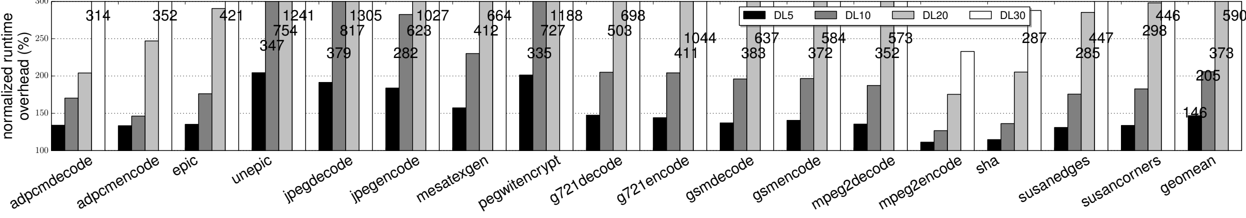


Figure 2.13: Tail-wait across different WCDLs where pipeline width is 4

commit width (not shown in the graph) which supports our theoretical performance analyses in Section 2.3.2. Interestingly, the performance overhead of Tail-wait degrades much more significantly with longer WCDL and/or wider pipeline width, which shows the benefits of tail-DMR compared to naive tail-wait strategy. The large number of small regions can account for such trend as the performance overhead of tail-wait is proportional to WCDL and pipeline width while that of Clover will be saturated as discussed in Section 5.5.5. In all, Clover achieves 1.06∼3.49X speedup over the tail-wait approach on average.

2.5 Other Related Work

This section describes the prior works related to soft-error detection, correction and recovery. We also explain how our proposal Clover advances the state-of-art and differs from previous approaches in this domain.

2.5.1 Soft Error Detection and Correction

Soft error detection relies on either hardware or software instruments to identify the errors. Software-based detection schemes often refer to N-modular redundancy execution. SWIFT is one of the state-of-the-art single-threaded software detection schemes [12]. It checks the value of registers with their duplication counterpart at certain synchronization points, i.e., memory and control flow instructions. Rotenberg takes the advantage of simultaneous multithreading (SMT) to run a trailing thread that verifies the leading thread [53]. Although these methods achieve a high fault coverage, they suffer from significant performance overhead or occupying one more processor core.

With that in mind, researchers explore the program characteristics to find opportunities for reducing the performance overhead [54, 55, 56, 57, 58, 59, 60, 61, 54, 62]. They all exploit some heuristics or statistics to identify the instructions that are critical to the program output and selectively protect these instructions with DMR. Hari et al. [56] also investigated approaches where the outcome of the soft errors for specific applications could be predicted. Hari et al. [58] also proposed novel program-level detectors to detect silent data corruptions

and demonstrate that it reduces the overhead imposed by redundancy based techniques. Li et al. [57] proposed online methods to estimate the architecture vulnerability factor to the soft errors for various micro-architecture factors. On the other hand, SWAT uses program invariants to detect the hardware errors [59]. mSWAT, an extension of SWAT framework, applies symptom based detection and diagnosis for faults in multicore architectures [60]. Epipe [62] identifies the vulnerable instructions that lead to SDC and selectively protects a subset of those vulnerable instructions considering the worst-case execution time (WCET) constraints in the fault-free execution. Similarly, Rehman et al. [63, 64, 65] explore the varying vulnerability property of different applications along with different compilation versions to significantly reduce the application failures.

Khudia et al. leverage profile information to reduce the cost of duplication and checking [66, 54]. They propose profile-based instruction-level DMR to focus on those instructions that are likely to affect the important program output [66]. Recently, Khudia and Mahlke exploit the value-locality of instruction results during the profile run to achieve selective DMR [54]. Other techniques estimate the soft error vulnerability of instructions executed in the underlying processor [67] , and use statistical models to measure the probabilities of the error masking and the propagation in the processor for enabling selective protection [61]. However, all those approaches achieve low performance overhead at the expense of reduced fault coverage. In contrast, Clover selectively protects some of the instructions (i.e., only those instructions vulnerable to DUE) without sacrificing the fault coverage. Chen and Yang propose a technique, that identifies the minimum set of instruction results being compared

and checkpointed for the error resilience, to reduce the performance overhead while achieving full coverage [68]. However, the resulting runtime overhead reduction is not stated in their paper.

Hardware-based detection schemes introduce redundant hardware to verify the execution in the processor. DIVA [25] relies on a simple in-order core to verify the program execution while Argus [26] leverages invariant checking to ensure correctness. However, these approaches often introduce excessive hardware complexity increase which is not acceptable in embedded systems. ReStore [34] advocates to utilize symptoms of the soft errors to detect them without significant overhead. Shoestring [35] enhances ReStore by selectively duplicating some vulnerable instructions with simple heuristic. However, both ReStore and Shoestring incur long detection latency which may result in DUE. Similar to the recent work of Khudia and Mahlke [54], Racunas *et al* [24] proposes to make use of the value-locality to detect the soft errors. However, the issues of false positives/negatives [69, 70, 71] in the locality-based approaches prevent us from adopting their methods.

Several recent works have proposed to reduce the ECC performance and power overhead and increase the correction capabilities, such as Bamboo ECC [72], FreeFault [73], and Virtualized ECC [74]. On the other hand, proposals such as Containment domains argue for a new programming construct for applications to express resilience requirements, and tune error detection, state preservation and recovery schemes [75]. More recently, Upasani *et al.* propose to detect the soft errors with configurable amount of acoustic wave sensors [76, 77, 36, 37, 78]. Their sensor-based detection scheme achieves low soft error detection latency

with reasonable hardware overhead. According to the recent work of Upasani *et al.* [37], it is possible to detect all the soft errors in an old ARM cortex-A5 core within one cycle by meticulously deploying only 17 sensors on the core. However, their technique requires the core size and the frequency to be extremely small, which is not realistic for modern processors. We takes advantage of such sensor-based detection scheme and selectively duplicates only the instruction vulnerable to DUE in the tail of an idempotent region for guaranteed soft error recovery.

2.5.2 Soft Error Recovery

Checkpointing the whole program states (memory and registers) guarantees recovery from the soft errors by allowing programs to roll back to the previous safe checkpoint[34, 35, 36]. However, full checkpointing often comes with significant performance loss and high power consumption. With that in mind, researchers propose techniques that can reduce the checkpointing overhead, but they require costly hardware support and resource consumption. For example, the recent work of Upasani *et al.* [36] keeps two copies of the register file and the register allocation table (RAT) to achieve low performance overhead. Jeyapaul *et al.* [79] explore multicore CMP architecture to recover from the soft errors with an efficiently modified cache structure. However, they rely on only parity checking to sequential logic for detecting a soft error, i.e., combinational logic is still vulnerable to the soft errors thus they may generate SDC. Flushing the pipeline to recover from a soft error [24, 37] is another alternative. This approach is expected to be very efficient in term of runtime

overhead. However, this approach is often based on the assumption that detection can be done before the faulty instruction is committed, i.e., the error detection latency should be zero. Such low detection latency inevitably requires high performance/hardware overhead as stated in Section 2.5.1. In particular, Clover avoids such high overhead by integrating idempotent processing that recovers from the soft errors by simply re-executing the region in which they occur. That is, even if the soft errors have already corrupted architectural states, Clover can recover from the errors, and the detection latency does not need to be zero. However, idempotent processing requires the soft errors to be detected within the same region as stated in Section 2.3. Clover overcomes such a challenge with a novel tail-DMR technique in the presence of sensor-based soft error detectors.

2.6 Summary

We present Clover, a compiler directed soft error detection and recovery scheme. This is a fundamentally new approach to achieving lightweight soft error resilience with no DUE (detected unrecoverable error). It can also achieve almost zero SDC (silent data corruption) as long as the soft errors come from the energetic particle strike which is the major source of soft errors. Clover is a low-cost hardware/software cooperative scheme. On the hardware side, Clover relies on a small number of acoustic wave detectors deployed in the processor to identify the soft errors by sensing the wave made by the particle strike. On the software side, Clover leverages a novel selective instruction duplication technique called tail-DMR

(dual modular redundancy) that offers a region-level error containment, to cope with DUE caused by the sensing latency of the error detection. In addition, Clover generates soft error tolerant code based on idempotent processing for soft error recovery. Once a soft error is detected, Clover recovers from it by re-executing the idempotent region where it is detected. This error recovery process is performed as in the case of an exception raised by either the sensor or the tail-DMR, the exception handler of which simply redirects program control to the beginning of the region. In this way, Clover can achieve soft error resilience with low runtime and negligible area overheads. The experimental results demonstrate that the runtime overhead of Clover is only 26%, which is a 75% reduction compared to that of the state-of-the-art soft error resilience technique (i.e. idempotent processing + full DMR). Finally, this chapter evaluates the proposed tail-DMR technique compared to a new alternative to it called tail-wait. Evaluating the techniques with the different processor configurations and the various error detection latencies confirms that the tail-DMR is a superior technique, achieving 1.06∼3.49X speedup over the tail-wait.

Chapter 3

Soft Error Resilience with Idempotent Processing from Hardware's View

This chapter presents Turnstile, a hardware/software cooperative technique for low-cost soft error resilience. Turnstile further reduces the runtime overhead of Clover by removing the tail-DMR overhead. In order to achieve that, Turnstile redefines the idempotent processing in a way that the idempotent regions can be created, verified and recovered from the processor's point of view. In this way, Turnstile can allow the error to escape the faulty region and thus avoid the tail-DMR overhead. At the same time, Turnstile still offers guaranteed soft error recovery because Turnstile's hardware support can rollback to the program state of an arbitrary region boundary by buffering the program state of each region until it becomes verified.

3.1 Challenges and Contributions

State-of-the-art fine-grained recovery schemes [80, 32, 33, 38, 81, 82, 41] follow the checkpoint, rollback and re-execution model, but at a much finer granularity (<30 instructions on average [33]). They divide the control flow graph into different regions and correct a soft error by jumping back to the beginning of the faulty region. To achieve this, the compiler guarantees the inputs to the region boundary are not overwritten, i.e., no anti-dependence (write-after-read) on the inputs, during the execution of the region. That way the region inputs remain the same within the region, making the regions harmless to be re-executed multiple times. If some inputs are overwritten within the region, their values do not remain the same as they were at the region boundary making the re-execution of the region unsafe, i.e., ending up with unexpected output. Thus, prior fine-grained recovery schemes require that the region inputs are never overwritten during the execution of the region. For example, De Kruijf *et al.* place region boundaries to break the memory-level anti-dependence and leverage register renaming to eliminate the register anti-dependence on the inputs to the region [33, 38, 81].

Despite the benefit of false-positive tolerance and region-level error containment, prior fine-grained recovery schemes expose three critical challenges that hinder their pervasive use.

- *Challenge 1*: Prior fine-grained recovery schemes contain the error within the region rather than the core. If the destination address of a store is corrupted and the store is performed, the memory state might be corrupted. Therefore, those recovery schemes

assume a large store buffer to hold all the stores in a region. Besides, prior region partition algorithms [33, 32] fail to constrain the number of stores in a region which might overflow the store buffer leaving the possibility of failure to recovery.

- *Challenge 2*: Prior fine-grained recovery schemes have to pay a high overhead to protect the region inputs from being corrupted (overwritten) by the soft errors (anti-dependence). For example, [33] needs to protect the register files (RF) with error correcting code (ECC) as well as hardening logic for RF's controller which is on the processor's critical path. Besides, [33] leverages register renaming to eliminate the register anti-dependence which leads to performance degradation. Even worse, [32] gives up the protection of some regions due to the high overhead caused by preserving all the their inputs.

- *Challenge 3*: An error must be detected within the region, which either requires an expensive detector to ensure the correctness of region inputs or sacrifices error coverage. If an error occurring in one region is detected in the next region, simply re-executing it cannot achieve the recovery since its inputs may have been corrupted by the error. In general, a majority of regions in the state-of-the-art schemes are very short (e.g., $\approx$ 10 instructions [33, 81]) which requires an expensive error detector with almost zero latency. Thus, acoustic sensors cannot work with prior fine-grained recovery schemes.

To solve the above challenges, we present Turnstile, a lightweight hardware/software cooperative technique for soft error resilience that leverages the acoustic sensors. The design ob-

jective of Turnstile is to provide low-cost and fine-grained checkpoint/rollback/re-execution mechanisms. Turnstile is motivated by the insight that all the committed instructions before a program point p are verified only if the acoustic sensors raise no alarm during the time of the error detection latency since p. Therefore, Turnstile partitions the program into different verifiable fine-grained regions and proposes a simple hardware support to verify those regions one by one, preventing any errors from escaping the core. In case of an error, Turnstile rollbacks the program to the most recently verified region boundary rb, restores rb's architectural and memory states, and re-executes from rb to recover from the error.

For the region based verification and recovery, Turnstile needs to verify and preserve the program state (registers and memory) with regard to the beginning of each region (i.e., rb). As for the memory state, Turnstile leverages gated store queue (GSQ) to buffer all the committed yet unverified stores before rb until its last preceding region (i.e., the region before rb) ends and spends the time of the error detection latency [83, 84, 43]. Once the last preceding region is verified, its committed stores can be drained to the L1 cache [1]. If a fault happens since then, all the unverified stores after rb are flushed out (squashed) from the GSQ, thus the memory state with regard to rb is always preserved.

As for the register state, Turnstile proposes unified data verification which leverages a novel compiler analysis to automatically identify the minimal register state necessary for restoring a recovery point in case of a fault, and inserts stores to checkpoint their value in a reserved checkpoint location of the memory. This approach allows Turnstile to convert the problem

[1]Stores are merged to L1 when the bandwidth of the bus between the GSQ and the cache is available

of register data verification into that of memory data verification, which is very cheap since stores are not on the critical path in general. Therefore, the stored data of the not-yet-verified registers are eventually verified in the same way of memory data verification. The error recovery process remains the same as well, i.e., reading from verified (checkpointed) register data from memory.

In particular, to realize the proposed verification and recovery, Turnstile introduces a simple hardware support called RBB (region boundary buffer) that is off the critical path of the pipeline and interacts with the ROB (reorder buffer) and the GSQ (gated store queue). Taking into account the error detection latency, the RBB precisely controls the GSQ for timely verification as well as directs the program control to the most recently verified recovery point on a fault. The ROB notifies the RBB of each committed recovery point, and the RBB notifies the GSQ of unverified stores to be squashed during the error recovery.

Finally, for the proposed hardware support, Turnstile introduces another new compiler analysis to statically partition the whole program into different verifiable fine-grained regions considering the capacity of the GSQ in the core. For Turnstile to buffer all the stores in an unverified region, its compiler guarantees that the number of stores in a single region never exceeds the size of the GSQ to contain the errors within the core. Besides, to avoid performance degradation, Turnstile forms the regions such that the stores in the neighboring regions can be verified in a pipelined fashion, i.e., overlapping the verification of one region with the execution of the next region.

The following are the contributions of this work:

- A low-cost soft error resilient solution that offers guaranteed, fine-grained recovery with core-level error containment. Turnstile does not require expensive hardware support (e.g., large store buffer and ECC-protected register file) for recovery. The runtime overhead is only $\sim$8% on average.
- A new hardware support called RBB that interacts with GSQ and ROB to efficiently and precisely realize the region verification and the recovery considering the latency of the sensor-based soft error detection.
- A novel compiler analysis framework, that requires no source code, to automatically partition the whole program into verifiable fine-grained regions taking into account Turnstile's hardware support, as well as to checkpoint the minimal architectural state with regard to the region boundaries.

3.2 Turnstile Overview

The goal of Turnstile is to provide a low-cost fine-grained hardware/software cooperative technique for soft error resilience with guaranteed error recovery which can work with detectors with detection latency, e.g., acoustic sensors.

Containing The Errors Within The Core Turnstile proposes a software/hardware co-design scheme to contain the errors within the core. Turnstile first partitions the entire program into different fine-grained regions preventing the number of stores in each region

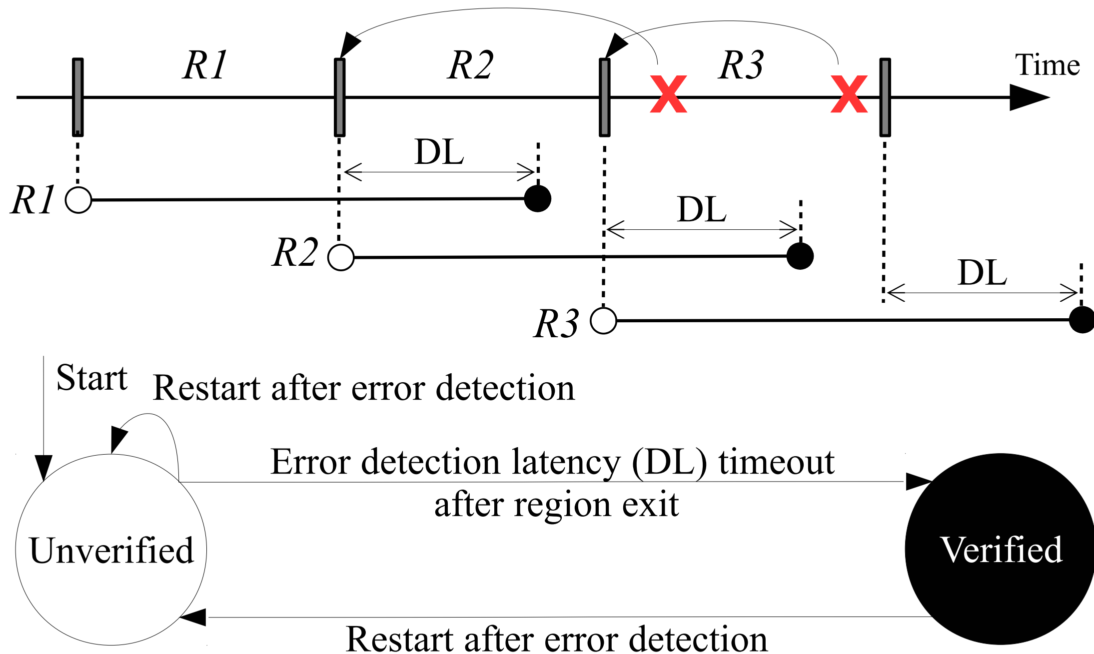


Figure 3.1: The region verification/recovery idea (top) and the status change of region execution (bottom)

from overflowing the gated store queue (GSQ) (Section 3.3). Then, Turnstile introduces the region boundary buffer (RBB) to control the GSQ to hold the stores of a region until the region ends and spends WCDL cycles (Section 3.4). Thus, Turnstile can verify the entire region as fault-free after WCDL cycles, by carefully exploiting the acoustic sensor-based detector and the GSQ. Figure 3.1 describes the status change of each region during its execution in the presence of soft errors. Initially, Turnstile regards all the committed stores in the region as unverified, thus holding their write-back to memory until the region is verified, i.e., if there is no error detected during the time of WCDL after the region exit. For regions R1 and R2, they can drain the stores in their region only after the time of WCDL elapses from when they reach the next region boundaries, i.e., R2 and R3, respectively. Consequently, no faulty store of unverified regions can be drained to memory system (L1

cache), That is, Turnstile can contain the error occurred in a region within the core.

Unified Data Verification Premised on Turnstile's hardware support (Section 3.4), Turnstile enables unified data verification. Each region generates some or all of the inputs to the later regions by defining the variables that can reach the later regions. The region input is two-fold: (1) the values stored in memory, simply memory inputs. They can be verified by leveraging the Turnstile's RBB and GSQ; and (2) the values of register, i.e., register inputs. Turnstile leverages a novel compiler analysis [41] to insert checkpoints right after the instructions that define the registers as long as they are used as the inputs of later regions (Section 3.3). Note that the checkpoints are essentially store instructions that save the checkpointed register value in reserved memory location. In effect, Turnstile transforms the verification of register region inputs into that of their store instructions for checkpointing. That way Turnstile can unify the verifications of memory and register inputs.

Detection Latency Aware Rollback and Re-execution Based Recovery Turnstile can successfully recover the system even if the error is not detected within the faulty region, i.e. Turnstile can tolerate the error detection latency of acoustic sensors. If an error is detected during the time of WCDL after a region (R2) encounters its next region boundary (R3), e.g., the first error in R3, Turnstile's recovery handler takes the following three actions to recover the error:

- First, Turnstile discards all the unverified stores in the GSQ, i.e., all the stores and

checkpointing stores after R2's entry point. Note that all the entries in the internal structures (e.g. instruction queue etc.) are invalidated and drained from the pipeline as well.

- Second, it restores the region input of R2 next to most recently verified region R1. Since all the stores after region R2's entry are discarded, the entire memory remains intact, that includes the memory inputs to R2 as well as the reserved memory location for the register checkpointing. Turnstile can thus safely restore the register inputs to R2 by loading the values from the location.

- Finally, Turnstile jumps back to R2's entry and continue execution from there to correct the error.

As shown in Figure 3.1, the following regions (R2, R3) are then executed and verified once again. Note that to recover from the second error detected in R3, Turnstile redirects the program control to the beginning of R3, since R2 has already been verified.

Addressing the Previous Challenges Turnstile solves the three challenges in the previous section. *Challenge 1* is solved with Turnstile's compiler and hardware support to contain the error within the core without sacrificing error recovery capability. *Challenge 2* is also solved with unified data verification. Turnstile does not require any hardware protection for register files since it handles the register verification in a unified way as discussed above. Besides, as the checkpointing stores are generally off-the-critical-path of the processor, we

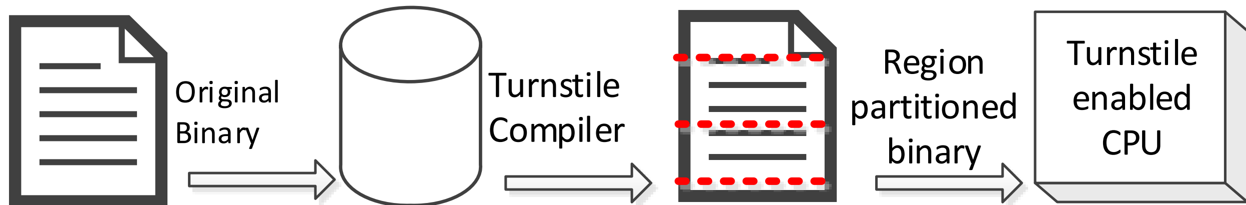


Figure 3.2: The high-level view of Turnstile

expect the performance degradation is small. *Challenge 3* is solved, since Turnstile does not require an error to be detected within the region. This is because Turnstile can always guarantee to recover the inputs of the most recently verified region boundary, i.e., the inputs to the faulty region.

Figure 3.2 shows a high-level view of Turnstile. An application binary is fed into the compiler, in this case a binary rewriter. The resulting binary has special instructions to represent a region boundary as well as store instructions to checkpoint registers. During program execution, Turnstile holds all the unverified data stored in a region using a gated store queue in the first place. At the same time, the verification controller of the Turnstile processor verifies each region execution by intelligently counting down the detection latency time after the region exit. Once a region is verified, the controller signals the GSQ to release those data stored by the region, and keeps the PC of the first instruction of the next region. For later errors, Turnstile uses the PC as a recover point once flushing (squashing) every unverified data in the GSQ and restoring the region inputs.

3.3 Turnstile Compiler

Turnstile compiler primarily performs two tasks on the binary before it runs on the Turnstile-enabled processor. First, Turnstile partitions the binary into different verifiable fine-grained regions considering the capacity of the gated store queue (GSQ) in the processor to contain the error within the core. Then, the compiler inserts checkpointing store instructions to preserve the register inputs to the regions so that Turnstile can achieve unified data verification with simple hardware support called RBB (region boundary buffer) shown in Section 3.4 that controls and interacts with the GSQ and the ROB (reorder buffer).

As Turnstile's region partition algorithm requires the knowledge of checkpoints to be inserted, we first introduce the checkpoint set identification in Section 3.3.1 before discussing the region partition algorithm in Section 3.3.2. Finally, we present our loop optimization technique in Section 3.3.3.

3.3.1 Checkpoint Set Identification

Turnstile leverages a novel compiler analysis to identify the minimal register state [41] necessary for restoring a recovery point (the most recently verified region boundary) in case of a fault. Then, Turnstile inserts stores to checkpoint those register value in the reserved checkpoint locations in the memory. The checkpoint-set analysis investigates register-updating instructions to checkpoint the resulting value being used in later regions as their *live-in* registers [85]. That is, Turnstile is interested in the last among those instructions that update the

same register. For a given region r, Turnstile thus cares about only the instruction d whose register definition is *downward-exposed* [85]. To determine if the updated register needs to be checkpointed, Turnstile evaluates the following boolean function.

$$Ckpt(r, d) = \begin{cases} 1 & \text{if } (Reg(d) \in LiveOut(r)) \\ \\ 0 & otherwise \end{cases}$$

where Reg maps an instruction to its first operand (i.e., the destination register) while $LiveOut$ maps a region to a set of *live* registers at the end of the region. If the output of $Ckpt(r, d)$ is set, Turnstile simply inserts a checkpoint (i.e., a store instruction) immediately following those instructions d to store the updated value $Reg(d)$ in a reserved memory location. Note that for each register being checkpointed, Turnstile reserves a specific region in the stack frame. Thus, the worst-case memory overhead due to the checkpoints is bound to the register file size times the maximum stack depth at runtime.

3.3.2 Region Formation

At a first glance, forming regions appears to be as simple as counting the store instructions while traversing the control flow graph (CFG) and placing boundaries whenever a threshold (i.e., half size of GSQ) is reached. However, before determining the region boundaries, Turnstile cannot determine the locations where checkpoints (i.e., stores) are inserted, making the region formation non-trivial. That is, the region formation problem is circularly dependent on itself: determining region boundaries requires the resulting instrumented binary containing checkpoints, which in turn requires identification of the regions.

To solve the problem, Turnstile first considers all the basic blocks in the CFG as initial regions, and calculates the number of checkpoints to be instrumented in each region. Traversing the CFG, Turnstile then attempts to combine those initial regions into larger regions as much as possible. By combining them, Turnstile can eliminate many checkpoint instructions (i.e., stores). This is because the registers being checkpointed are no longer live-out to the later regions. The key of our heuristic is to ensure the number of stores during the execution of each region will not overflow the threshold (i.e., half size of GSQ) even after the checkpointing stores are inserted. Note that we set our threshold to be half the size of the store queue so that Turnstile can overlap the verification of one region (i.e., waiting for WCDL at the end of the region) with the execution of the next region. Therefore, Turnstile can greatly reduce the pipeline stalls due to the store queue overflow.

Algorithm 2 shows the heuristic. First, Turnstile considers all the entry and exit points of functions as region boundaries (line 1), as prior works do [33, 38, 81]. Second, Turnstile places a boundary at the beginning of each loop header (line 2) [2]. Third, Turnstile treats memory fences and atomic operations as region boundaries (line 3). The ISA enforces an ordering constraint on memory operations before and after the memory fences (e.g, x86 TSO consistency model requires that queued stores must be merged into the memory system at a fence instruction). In addition, atomic operations (e.g., atomic compare-and-swap) should complete, i.e., obtain the write permission and become visible to other processors [3]. As

[2] Since the loop trip count is not always statically known, GSQ may overflow in some number of iterations even with a single store in the loop body. Section 3.3.3 proposes our optimization to remove such a mandatory region boundary for certain loops.

[3] TSO model also requires draining the store queue at an atomic operation as it should ensure the program order between stores, whereas RMO model only requires the store of the atomic operation to be completed

Algorithm 2 Turnstile Region Formation Algorithm

1: Place a boundary at the beginning of functions and at the end of each callsite.
2: Place a boundary at the beginning of the loop header of each loop.
3: Place a boundary at a memory fence and an atomic operation.
4: **for** each basic block bb_i in CFG **do**
5: $Str_{bb_i} \leftarrow Str_ori_{bb_i} + Str_ckpt_{bb_i}$
6: $IncomeStr_{bb_i} \leftarrow 0$
7: **end for**
8: **for** each basic block bb_i in program topological order **do**
9: **if** bb_i starts with region boundary **then**
10: accum_str $\leftarrow Str_{bb_i}$
11: **else**
12: accum_str $\leftarrow Str_{bb_i} + IncomeStr_{bb_i}$
13: **end if**
14: **while** accum_str > threshold **do**
15: place boundary and split bb_i into bb_i' and bb_i
16: recalculate $Str_ori_{bb_i}$ and $Str_ckpt_{bb_i}$
17: accum_str $\leftarrow Str_ori_{bb_i} + Str_ckpt_{bb_i}$
18: **end while**
19: **for** each successor basic block bb_j of bb_i **do**
20: $IncomeStr_{bb_j} \leftarrow \max(IncomeStr_{bb_j}$, accum_str)
21: **end for**
22: **end for**

these are critical to guarantee correctness of synchronization operations for multithreaded programs, Turnstile places a boundary at a memory fence and an atomic operation, similar to [33]. Then, Turnstile identifies the basic block that has region boundaries in the middle of it, and splits it into different basic blocks. This guarantees that region boundaries always start at the beginning of basic blocks, thus helping the next step to compute the initial checkpoint instructions.

After placing the first set of region boundaries, Turnstile further analyzes the program and, if necessary, places additional boundaries to prevent the GSQ from overflowing. In line 4∼7,

as it may employ unordered store queue [86]

as each basic block bb_i is initially regarded as a region, Turnstile computes the total number of stores (Str_{bb_i}) in bb_i as the sum of original stores ($Str_ori_{bb_i}$) and checkpointing stores ($Str_ckpt_{bb_i}$).

Based on the checkpoint set identification (Section 3.3.1), Turnstile conservatively calculates the number of checkpointing stores in each basic block bb_i with the following equation:

$$Str_ckpt_{bb_i} = Def_{bb_i} \bigcap LiveOut_{bb_i} \tag{3.1}$$

where Def_{bb_i} is the set of registers defined in bb_i and $LiveOut_{bb_i}$ is the set of live-out registers of bb_i. Then, Turnstile assigns zero to $IncomeStr_{bb_i}$ which will be updated with the maximum of number of incoming stores accumulated from bb_i's predecessors during the later analyses.

And then, Turnstile traverses the CFG in topological order, consulting Str_{bb_i}, i.e., the total number of stores of each basic block bb_i (line 8∼22). In line 9∼13, if bb_i already has a region boundary at the entry, Turnstile updates the accumulating store number (accum_str) with Str_{bb_i} since we have started a new region. Otherwise, Turnstile keeps combining each basic block, updating accum_str with the sum of Str_{bb_i} and $IncomeStr_{bb_i}$, i.e., the incoming store number of bb_i. If accum_str is greater than the threshold which is half the size of the GSQ (line 14∼18), Turnstile tries to place a new boundary after a store where the number of stores in the region prior to the new boundary is less than the threshold, if one exists, and splits the bb_i (line 14) into head ($bb_i{}'$) and tail (bb_i) basic blocks. If there is no such store, Turnstile puts the new boundary at the basic block entry. Then, Turnstile updates the accumulating

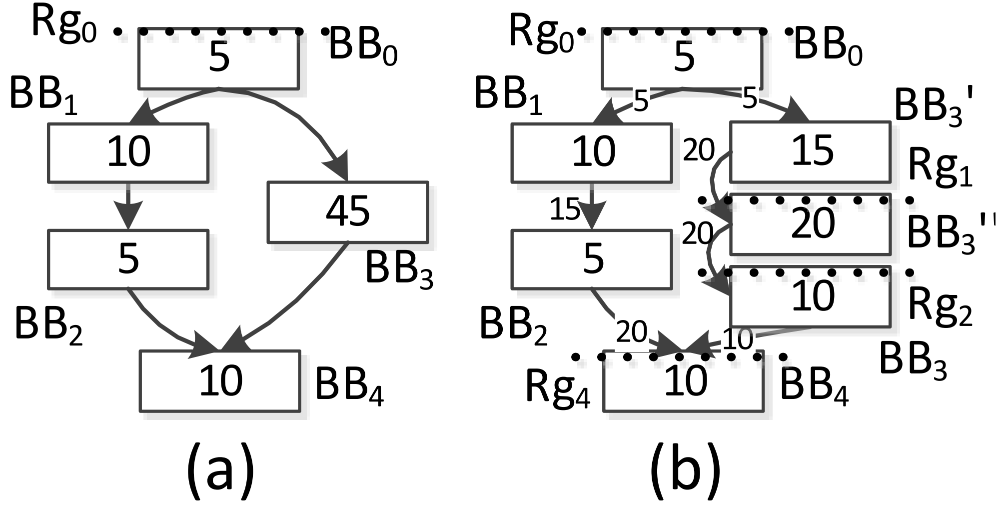


Figure 3.3: An example of Turnstile Region Partition Heuristic

store number (accum_str) with the total store number Str_{bb_i} of the new (tail) basic block bb_i (line 17). The above process keeps repeated as long as accum_str is greater than the threshold. Finally, for each successor basic bb_j of bb_i, Turnstile updates the incoming store number of bb_j ($IncomeStr_{bb_j}$) with the remaining accum_str. $IncomeStr_{bb_j}$ is updated iff the remaining accum_str is greater than $IncomeStr_{bb_j}$'s original value (line 19∼21). Note that, as Turnstile traverses the CFG in program topological order, the incoming store number of each basic block is always guaranteed to be the maximum incoming store number from all its predecessor basic blocks when Turnstile visits the basic block.

Example: Figure 3.3 presents an example of Turnstile's region formation heuristic. Figure 3.3 (a) shows the original CFG where the number in each basic block is the total store number. Assuming the basic block BB_0 starts with a region boundary Rg_0 (shown as a dotted line), Turnstile traverses the basic blocks in a topological order ($BB_0 \rightarrow BB_1 \rightarrow BB_2 \rightarrow BB_3 \rightarrow BB_4$). We also assume a store queue with 40 entries, thus the

threshold is 20 in this case. When Turnstile visits BB_3, as the calculated accumulating store number (accum_str) is already 50 thus greater the threshold.

Turnstile then keeps placing a region boundary thus splitting BB_3 until the accum_str becomes smaller than the threshold. For ease of presentation, we assume that the sum of each recalculated total store number for every split blocks remain the same (45) as shown in Figure 3.3 (b). The figure shows the partitioned program where the numbers on the edges are the incoming store numbers. Note that the maximum incoming store number of BB_4 (i.e., $IncomeStr_{BB_4}$) is 20. Thus, we place a boundary at the entry of BB_4 as its accum_str is greater than the threshold.

3.3.3 Optimization for Storeless Loops

In order to improve the performance, Turnstile identifies those loops that have no store in the loop body, i.e., storeless loops. Here, Turnstile can eliminate the region boundaries in the header of such loops, since there is no chance of the store queue overflow during the execution of the loops. As a result, Turnstile can avoid inserting checkpoints (i.e., store) right after the instructions that update the register region inputs in the loop. However, the updated registers in the loop may be able to reach the later regions once the loop terminates.

To preserve the live-out registers, Turnstile places a region boundary at the end of loop preheader, and creates new basic blocks before the loop's exit basic blocks. Note that the new blocks exist outside the loop. Then, Turnstile inserts checkpoints into those new basic

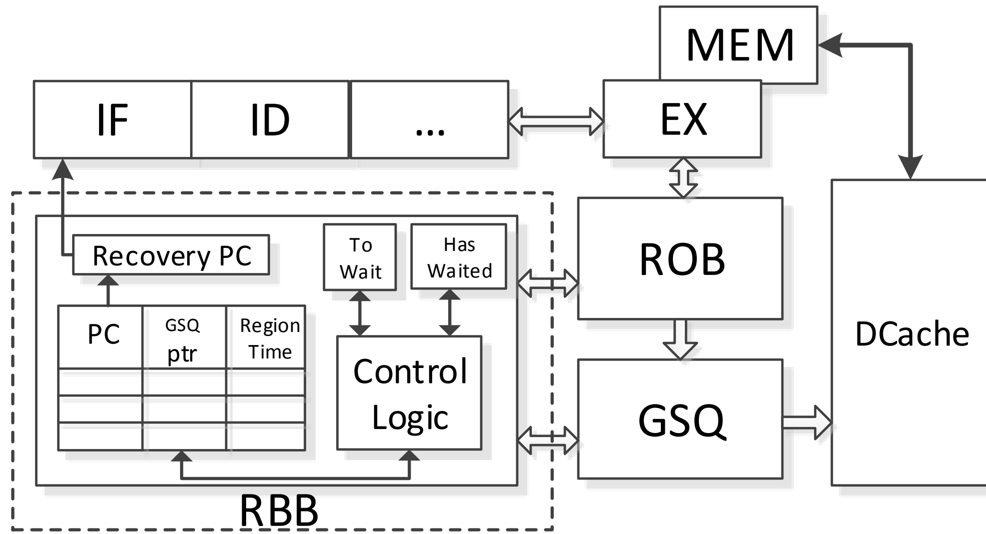


Figure 3.4: The high-level view of Turnstile hardware scheme

blocks, and places a region boundary at the end of the new blocks. With this optimization, Turnstile still guarantees 100% error recovery while significantly improving the performance.

3.4 Turnstile Hardware Support: Region Boundary Buffer (RBB)

Turnstile leverages special hardware logic to realize the region verification and the recovery for soft error resilience. As highlighted with the dashed box in Figure 3.4, the hardware logic interacts with the ROB and the gated store queue (GSQ) to recognize the boundary of executed regions and to write back their verified data, respectively. Note that Turnstile honors the original semantics of the store queue in an Out-of-Order processor (e.g. CAM search, Non-snoopable *etc.*).

To keep track of those regions that have not been verified yet, the hardware logic leverages the region boundary buffer (RBB) whose entry is allocated whenever the boundary instruction is executed. The RBB entry is a tuple of three values: (1) the PC of the boundary instruction; once the region ending at the boundary is verified, the PC is used as a recovery point when an error is detected, (2) the tail pointer of GSQ; this is necessary to write back only the data written in the fault-free region once it is verified, (3) the execution cycles of the region; this is important information for Turnstile's efficient region verification in the next section. Whenever a new region is verified, the logic writes the PC of the boundary instruction that finishes the region to a special register called RP (recovery PC) which will be used in case of an error.

3.4.1 Region Verification and Recovery Algorithm

This section first introduces a simple but an inefficient region verification/recovery algorithm and the lessons from it. Then, Turnstile's efficient algorithm is presented with an example.

Naive Algorithm: On a region boundary, one would write the TSC (time stamp counter) in the tail index of RBB, checking it from the head to see if there is an entry whose TSC value is less than the difference between the current TSC and the error detection latency time (cycles). Obviously, this naive approach is expensive due to the access to multiple entries of RBB on each region boundary, the check cost, and the area overhead (64-bit-TSC) in RBB entries. Even worse, for the region that has already spent the detection latency cycles

since it is finished, the GSQ cannot release its verified data until the next region boundary is reached. Note that this makes GSQ become full more frequently than usual, thus causing pipeline stalls leading to performance degradation.

Turnstile's Efficient Algorithm: The lesson from the previous algorithm is three-fold. First, TSC based approach is expensive, e.g., 64 bits per each RBB entry; Turnstile only uses $Log_2 DL$ bits where DL is the error detection latency in cycles. Second, the verification check should be done in a real time manner, i.e., it should not be delayed to the next region boundary. Third, as soon as a region is verified, Turnstile should be able to write back all its data buffered in GSQ without unnecessarily holding them. In light of this, Turnstile leverages a timer based approach to efficiently achieve region verification based on the following insight.

Axiom 1. *If a given region, R_n, is verified at a time T, then the very next region, R_{n+1}, will be verified at a later time T', $T' = T + ElapsedTime(R_{n+1})$ where $ElapsedTime$ maps a region to its execution cycles.*

The takeaway is that Turnstile can achieve the region verification by simply tracking the execution cycles of each executed region. For this purpose, Turnstile requires only two metadata for tracking every region. (1) a **watchdog timer** called $ToWait$ represents the number of cycles for which the region in the head of RBB should wait more before being verified. (2) an auxiliary **counter** called $HasWaited$ represents the number of cycles for which a region has already waited to be verified once the region becomes the head of RBB. Thus, $ToWait$ and $HasWaited$ both track the head entry of RBB.

When a region boundary instruction is executed, Turnstile allocates a new RBB entry for the

region that has just finished at the boundary. At the same time, Turnstile reads the watchdog timer ($ToWait$) and the counter ($HasWaited$) to calculate the elapsed time of the region, i.e., $RegionTime$. Note that unlike the watchdog timer that automatically counts down each cycle, Turnstile here reads the value of $HasWaited$ which was written at the previous region boundary. Therefore, the $RegionTime$ can be obtained by subtracting the sum of $HasWaited$ and $ToWait$ from the error detection latency (DL). Then, Turnstile records the resulting $RegionTime$ in the new RBB entry allocated, and updates the $HasWaited$ counter as "$DL - ToWait$".

When $ToWait$ becomes zero, i.e., RBB's head entry region is now verfied, it is removed by updating the head pointer of RBB; the removed entry is referred to as the old head entry hereafter. Thus, the PC of the old head entry is written in RP that will be used for recovery in case of an error. Then, GSQ marks as verified those entries preceding the GSQ pointer of the old head entry, so that they can be written back to L1 data cache when the bandwidth between the store queue and the cache is available. Note, having the old head entry tracked by $ToWait$ and $HasWaited$ removed from RBB requires updating them for a new head entry region. $ToWait$ is reset to the $RegionTime$ of the new head entry of RBB (See the Axiom 1), and the value of $RegionTime$ is subtracted from $HasWaited$. Recall that $HasWaited$ means the time for which a region has waited to verify itself since it became the head entry of RBB. Therefore, excluded is the time between the end of the old head region and the end of the new head region, which is the execution cycles of the new head region, i.e., its $RegionTime$.

If an error is detected, Turnstile's recovery handler discards all unverified data in the GSQ and empties the RBB. Then, Turnstile restores the checkpointed register values from memory. At this point, all the program status is guaranteed to be the same as it were before the error occurs. Finally, Turnstile jumps back to the end of the most recently verified region whose address is available in RP, the recovery point register.

Summary of Hardware Cost: Turnstile's hardware support is lightweight and non-intrusive to the critical path. It only needs a 5 bit timer and 5 bit counter. The modification to store queue for gating the stores requires one bit to flag verification status for each entry. The RBB is also trivially small (<14 entries) as evaluated in Section 3.6.5.

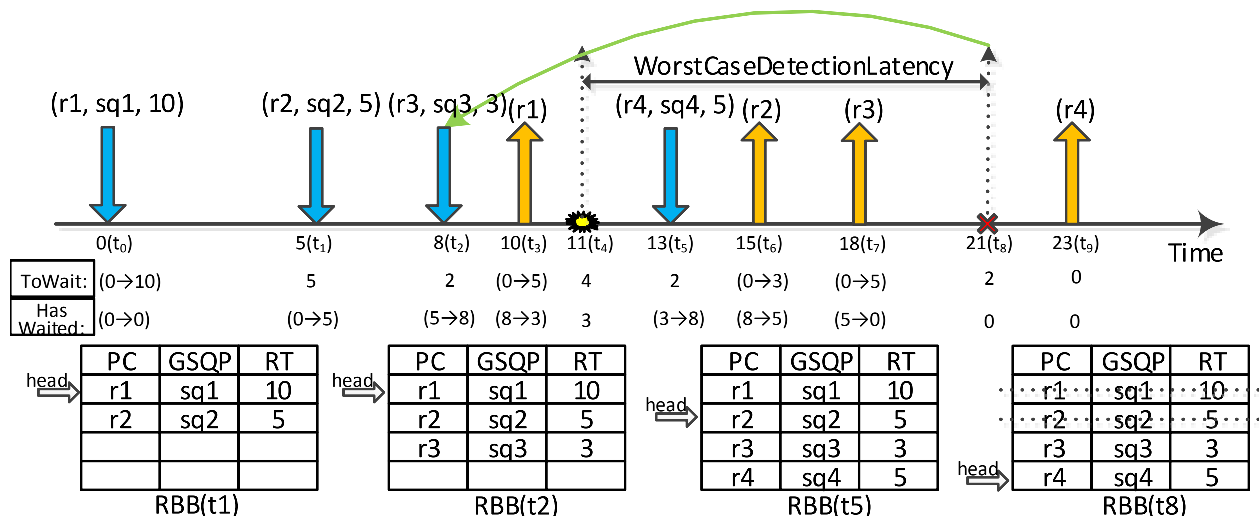


Figure 3.5: An illustrating example of Turnstile's hardware support.

Example: Figure 3.5 presents an example to show how RBB verification works. We assume the worst-case error detection latency (WCDL) to be 10 cycles. Along the timeline, the blue arrow corresponds to the time points on which a region boundary instruction is executed, while the orange arrow to the time points on which a region is verified.

When a region boundary instruction is executed, i.e., a blue arrow is reached in the timeline, the RBB is updated with the tuple above the arrow comprising the PC of the boundary instruction, the tail of pointer the GSQ, and RegionTime shown as RT in Figure 3.5. Below the timeline, there are two lines of numbers which represent the $ToWait$ timer and $HasWaited$ counter at each time point, respectively. When the $ToWait$ timer expires reaching 0, RBB first pops out the head entry, and updates the timer with the RegionTime in the new head entry. At the same time, RBB also updates the $HasWaited$ accordingly.

Below the two lines of numbers, there are four tables showing the status of RBB at different time points, i.e., t1, t2, t5, and t8. At t1, there are two region entries sitting in the table waiting to be verified. At t2, both the $ToWait$ timer and $HasWaited$ counter are not zero in the RBB where the RT is calculated by subtracting the sum of $HasWaited$ and $ToWait$ from the WCDL. At t3, the region r1 has already been verified. Therefore, the head pointer of RBB comes to point to r2 at t3, and the $ToWait$ timer is updated with r2's RT value. At the same time, r2's RT value is subtracted from $HasWaited$ to update the $HasWaited$ counter.

At t4, a soft error occurs, and it is detected at t8 after the WCDL. At t8, r3 has already been verified since t7 and popped out from the RBB, which means the instructions before r3 has been verified. Therefore, at t8, Turnstile redirects program control to the end of the most recently verified region (r3 in this case), thus re-executing r4 from the beginning to recover from the error. The takeaway is that Turnstile can achieve the region verification by simply tracking the execution cycles of each executed region.

3.5 Discussion and Limitations

Turnstile Hardware Protection/Fault Model: We assume that both the gated store queue (GSQ) and the region boundary buffer (RBB) are protected against soft errors. Note that there will not be any timing delay as they are off the critical path. Besides, we also assume that our watchdog timer and counter are also hardened by some protection schemes.

Tolerating Multiple-bit Errors: Multiple-bit errors can be handled easily by Turnstile. As discussed in Section 5.3, all the errors are guaranteed to be detected by the sensor-based detection scheme, and any un-verified stores are discarded during recovery.

Handling I/O Operations: Turnstile takes a conventional way, i.e., holding the I/O stores during the detection-latency cycles in an ECC-protected buffer to ensure error-free I/O stores as well as replaying the I/O loads upon recovery as with [36, 53]. Turnstile can have a gated I/O buffer, which is similar to our gated store queue (GSQ), to verify I/O requests in case they bypass the GSQ.

Silent Data Corruption (SDC): Even if an energetic particle strike is the major source of soft errors, they can also be induced by other sources, e.g., transistor variability or power supply noise *etc.* [87, 6]. Since these sources are not covered by the sensor-based soft error detection, Turnstile might generate silent data corruption (SDC). However, under aggressive transistor scaling and near-threshold computing, high-energy particle strikes are considered a critical source of soft errors [4, 88, 89].

3.6 Evaluation and Analysis

Our evaluation answers the following research questions:

- What is the performance impact of our hardware support and how sensitive is it to the worst-case detection latency (WCDL)?
- How does our compiler transformation affect the application performance and region characteristics?
- What is the impact on overall performance when Turnstile functions as a whole? How is the overhead affected by varying the size of the gated store queue (GSQ) and the time of the WCDL?
- What is the right size of the region boundary buffer (RBB)?

3.6.1 Experimental Methodology

We implemented the compiler analysis and checkpoint instrumentation passes described in Section 3.3 using the LLVM Compiler Infrastructure [49]. As with the common practice in the literature [35, 12], We used SPEC2006 [90], MediaBench [50, 91] and SPLASH2 [92] benchmark suites targeting different computing areas for our experiments, and all applications were compiled with standard -O3 optimization.

We conduct our simulations on the Gem5 simulator [52] with the ARMv7 ISA, modeling a modern two-issue out-of-order 2 GHz (1∼4 cores) processor with private L1-I/D (32KB, 2-

way, 2-cycle latency, LRU), and shared L2 (2MB, 8-way, 20-cycle latency, LRU) caches. The ROB, physical integer register file, and load queue have 192, 256, and 40 entries, respectively. The simulator was modified to accurately implement the effect of the gated store queue (GSQ) and the region boundary buffer (RBB) as discussed in Section 3.4. We vary the size of the GSQ (i.e., 40, 80, and 160 entries) to perform the sensitivity analysis.

For SPEC2006 applications, in order to show the performance impact of the executables generated by Turnstile's compiler versus the original executables, we synchronize the simulation length by measuring the number of functions executed which is constant between two versions as in the case of the prior work [33]. All the benchmarks in SPEC2006 are fast-forwarded the number of function calls to execute at least 5 billion instructions on the original executables, and then we simulate 1 billion additional instructions on the original executables. For the MediaBench and SPLASH2, we simulate the entire program for all the applications.

3.6.2 Turnstile Hardware Effect

Figure 3.6 examines the effect of Turnstile's hardware support by varying the WCDL. We partitioned the original binaries into different regions. However, we do not inject the checkpointing stores. We use the partitioned binaries as input and run the experiments on our modified simulator to observe how Turnstile's hardware support affects the performance of original binaries. We normalize the overhead with the baseline where the original binaries run on an un-modified simulator.

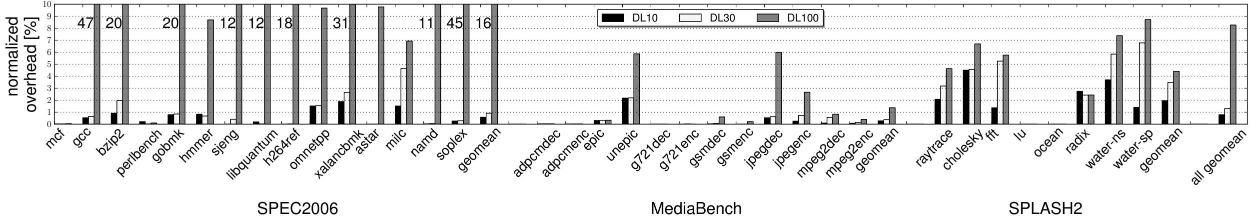


Figure 3.6: Turnstile hardware effect varying different detection latencies (10, 30, 100)

From left to right, each application has three bars representing different worst-case error detection latencies (i.e., 10, 30, 100) which are the time needed to verify the region. According to Upasani's work [36], having a 30 or 100 WCDL cycles of sensor-based detection scheme will cost less than 1% area overhead. However, as we assume that the memory system is protected by other schemes, e.g., ECC, there is no need to protect the caches, etc. That is, it is possible to deploy those detectors on only the core parts to achieve a lower WCDL, thus we also explore the effect of 10 WCDL cycles.

It is interesting to observe that the applications in three different benchmark suites show significantly different degrees of sensitivity to different WCDL. As the applications in MediaBench are mostly computation-intensive, they are rarely affected by Turnstile's hardware even when the WCDL is 100 cycles. On the other hand, the applications in SPLASH-2 benchmark show relatively higher overhead, and are more sensitive to WCDL than the others. The reason is that Turnstile treats fence operations and atomic operations, which are the basic ingredients in implementing synchronization between threads, as region boundaries. This implies that the cost of such operations become more expensive than before as Turnstile hardware have to wait for WCDL, affecting other threads. However, our experiment shows that the overall overhead is about 3.5% for WCDL of 30 and 4.5% for WCDL of 100.

Not surprisingly, with longer WCDL, the resulting performance overhead is higher. With 100-cycles-of-WCDL, Turnstile can slow down the original program up to 47% with 16% performance degradation on average. This is because holding the stores for a long time may result in the pipeline stalls when the gated store queue (GSQ) become full. As expected, smaller WCDL of 10 does little harm to the program performance which is less than 1% degradation in average. It is interesting to find that 30-cycles-of-WCDL also have a trivial impact on the performance (around 1%). To sum up, Turnstile can spend less than 1% hardware overhead to achieve 30-cycles-of-WCDL detection scheme, and the runtime overhead of Turnstile's hardware is not significant. Thus, in order to achieve low-cost, the following sections evaluate Turnstile with 10, and 30 cycles of WCDL configuration. In addition, we also evaluate Turnstile with 5 cycles WCDL assuming an aggressive sensor placement to reduce the WCDL [36].

3.6.3 Turnstile Compiler Effect

This section shows how Turnstile's compiler affects the original binaries. We report 1) the average number of instructions per region, and 2) the number of increased instructions for checkpointing, when different region formation thresholds are used.

Region Length

Figure 3.7 shows the average dynamic region length by varying the gated store queue (GSQ) size (i.e., 40, 80, 160) during the region formation. Note that we use the half GSQ size as the region formation threshold. However, for most of the applications, increasing the threshold

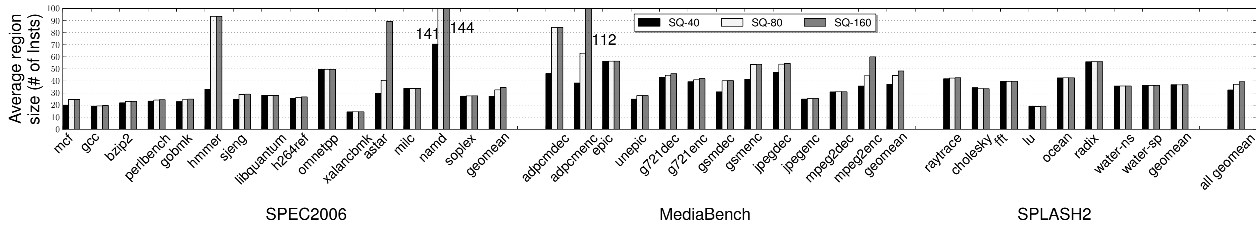


Figure 3.7: The average region length when partitioned for different store queue size (40, 80, 160)

does not help improve the dynamic region length. This is because we have to place the region boundary at the beginning of the header for most loops that have stores in the loop body to avoid the GSQ overflow. As such loops dominate the execution time, we can expect that the dynamic region length should not vary a lot even with different thresholds during the region formation.

A few applications (e.g., hmmer, astar and adpcm etc.) show great improvement in dynamic region length after increasing the threshold. This is because these applications have few loops with store instruction in the loop body or have a large loop body where increasing the threshold improves the dynamic region length.

As Turnstile does not require the error to be detected within the each region. The region length actually has no impact on the error coverage. In fact, a longer region may be less preferable during recovery as we may waste a lot of execution time. However, shorter regions are not always good for Turnstile, as Turnstile need to preserve the register region inputs with checkpointing store instruction. The shorter the region length is, the more performance overhead we need to pay for the checkpointing. Therefore, mediocre region length may be more preferable to Turnstile. The average dynamic region length is around 35 instructions

across different GSQ sizes which is beneficial for both tolerating false-positives and performance.

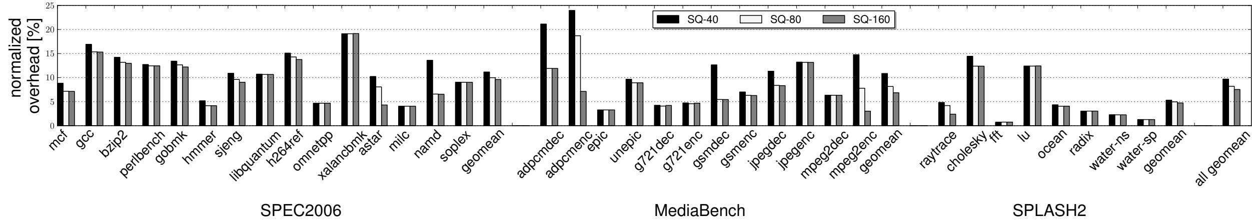


Figure 3.8: The normalized instruction counts when partitioned for different store queue size (40, 80, 160)

Checkpointing Instructions

Figure 3.8 shows the number of increased instructions for checkpointing normalized to the original binary. The average instruction overhead of Turnstile is around 10% when regions are formed for the gated store queue (GSQ) of size 40. The general trend is that when increasing the GSQ sizes (region formation threshold) which in turn generate less regions, the checkpoint instruction overhead gets smaller as fewer region inputs need to checkpoint.

3.6.4 Overall Overhead

This section evaluates the overall overhead by running our Turnstile compiler generated binaries on top of Turnstile's hardware support. We vary the factors (e.g. size of GSQ and WCDL) to see how it can affect the overall performance. Our baseline is the original binaries that run on an un-modified simulator.

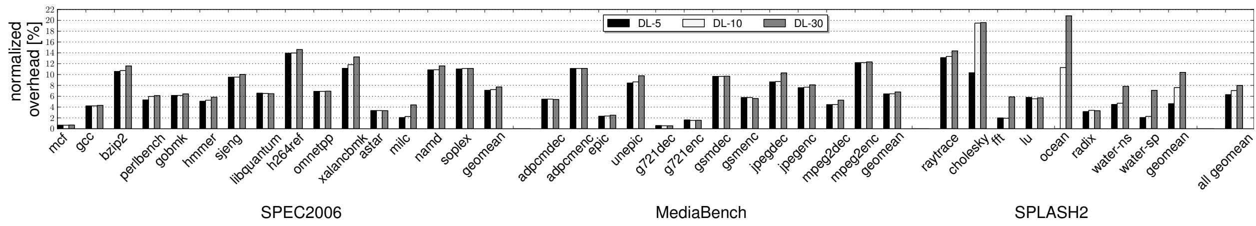


Figure 3.9: Turnstile overhead with 40 gated store queue entries varying different detection latencies (5, 10, 30)

Sensitivity to WCDL

Figure 3.9 show the the normalized runtime overhead with the gated store queue (GSQ) of size 40 and different WCDL from 5, 10, to 30, compare to the baseline. Increasing the WCDL can slow down some applications' performance by up to 20%, but the geometric mean (6∼8%) shows trivial differences between different WCDL.

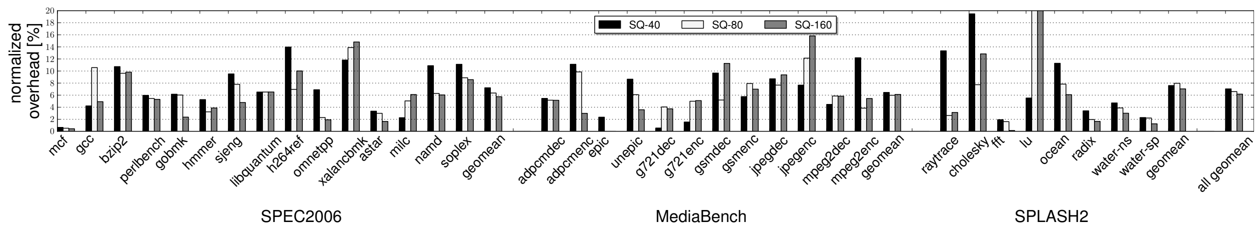


Figure 3.10: Turnstile overhead with 10 cycles WCDL varying different gated store queue size (40, 80, 160)

Sensitivity to the Size of Gated Store Queue

Figure 3.10 show the normalized runtime overhead compared to the baseline with varying the size of the gated store queue (GSQ). The overall overhead across different GSQ sizes stay around 6% up to 7.2%. With a large GSQ, Turnstile compilers can create a longer region, leading to less checkpoint instructions (Section 3.6.3). The general trend observed here is

that the increase of GSQ's size helps applications reduce the runtime.

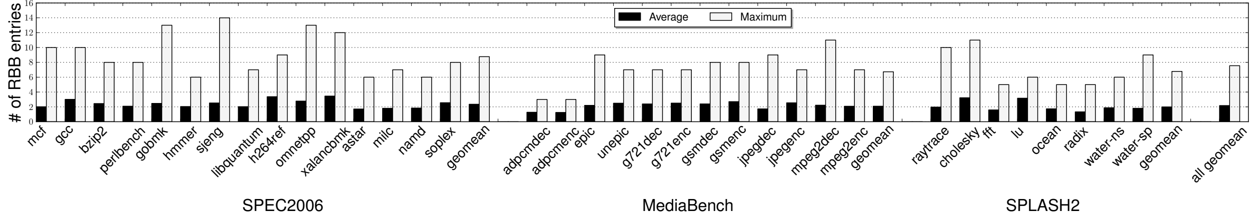


Figure 3.11: Region boundary buffer dynamic entry number with 40 gated store queue entries and 30 cycles WCDL

3.6.5 Exploration of Region Boundary Buffer Size

Lastly, we explore the design choice of the size of the region boundary buffer (RBB). Figure 3.11 shows the dynamic number of entries occupied during the execution. We use a gated store queue (GSQ) with 40 entries and a WCDL of 30 cycles to perform our profiling simulation. The left bar shows the dynamic average number of entries used by Turnstile while the right bar shows the maximum number of RBB entries occupied. As we can see, the maximum number across all the applications is 14 while the average number is 2. The number of bits needed in an entry of RBB can be calculated as follows:

$$\#RBB_Entry = \#(PC) + Log_2\#(GSQ) + Log_2\#(WCDL)$$

where #(PC) represents instruction size (32), #(GSQ) is the number of entries in GSQ (40) and #(WCDL) is the cycle number of WCDL (30).

Therefore, one entry in the RBB requires 43 bits. Even if we design the RBB with the maximum number (i.e. 14). We only need 602 bits for the RBB. Besides, Turnstile also requires a 5 bit timer and 5 bit counter. The overall hardware overhead is trivial. More

importantly, Turnstile's hardware implementation is off the critical path of the processor.

3.7 Other Related Work

In addition to the recovery schemes discussed previously, this section compares Turnstile with other existing recovery schemes and shows how Turnstile differs from those recovery schemes.

Coarse-Grained Recovery: Prior coarse-grained recovery schemes [36, 93] generally cannot contain the error within the core and requires expensive hardware support to relieve the performance overhead which relegates their use to high-end server systems. For example, to preserve the register states, those recovery schemes usually leverage two additional copies of ECC-protected register file (RF) and register alias table (RAT). As for the memory state preservation, previous coarse-grained recovery schemes either restructure the caches (e.g., new coherence) to incrementally checkpoint the memory state [36] or maintain a large buffer for memory logging [93]. Besides, Jeyapaul et al. [94] explores multicore CMP architecture to recover from soft errors with an efficiently modified cache structure. In contrast, Turnstile does not require such expensive hardware support and provides a low-cost alternative.

Finer-Grained Recovery: There are several recovery schemes that can also contain the error within the core. Flushing the pipeline to recover from a soft error is another alternative [24]. However, such schemes requires a near-zero detection latency detector and cannot be applied to low-cost acoustic sensor based detector with detection latency. Triple-Modular-Redundancy [13] can correct the error at the instruction-level at the cost of significant per-

formance degradation. On the contrary, Turnstile leverages the low-cost acoustic sensors and imposes low performance overhead making it a practical choice for soft error resilience.

3.8 Summary

We introduce Turnstile, a lightweight hardware/software cooperative soft error resilience technique that leverages low-cost acoustic sensors. Turnstile can provide guaranteed soft-error recovery that can contain the error within the core, incurring only ∼8% runtime overhead without expensive hardware support for checkpointing. Leveraging Turnstile's novel compiler analysis and lightweight hardware support, we can preserve the program state effectively and practically. Turnstile neither requires expensive RF protection to preserve the register state nor a extra large store buffer to maintain the memory state. To the best of our knowledge, Turnstile is the first technique for soft error resilience, that does not impose expensive hardware overhead, with guaranteed recovery. We believe that Turnstile can lay the foundation for the soft error resilience of future computing systems.

Chapter 4

Soft Error Recovery with Lightweight Checkpointing

This chapter presents Bolt, a compiler-directed soft error recovery scheme that further optimizes idempotent processing in the context of soft errors. Bolt performs two effective compiler optimization techniques, i.e., *eager checkpointing* and *checkpoint pruning*. To remove the need of the expensive hardware support, Bolt compiler protects the register inputs of each idempotent region during their entire liveness period by eagerly checkpointing the input values right after they are defined. To minimize the checkpoint overhead, the compiler performs a novel program analysis that identifies and eliminates those checkpoints whose value can be safely reconstructed by other values of existing checkpoints. The beauty of this approach is that it shifts the runtime overhead of the soft-error free execution to that of the error recovery execution without compromising the recovery guarantee.

4.1 Challenges and Contributions

The general idea behind soft error recovery is that when a fault is detected, the processor takes the recovery procedure to rollback to a fault-free state and continues execution. For example, traditional periodic checkpointing, an industrial-strength recovery paradigm, periodically checkpoints processor states [95, 36, 93, 96]. Upon an error, the system triggers a rollback to a fault-free snapshot and continues execution. However, periodic checkpointing is notoriously expensive due to its coarse-grained checkpoint-interval which is the period between two neighbouring checkpoints. First, coarse-grained checkpoint-interval means that a large number of states need to be checkpointed, incurring substantial performance/area overhead. Further, the longer the checkpoint-interval is, the more executed instructions are wasted upon recovery, imposing significant recovery overhead [97].

Instead, emerging idempotence-based recovery schemes become promising alternatives due to their fine-granularity (<100 instructions) and simple recovery mechanism [33, 32, 81, 82, 38, 98, 99, 100]. The compiler partitions and transforms the entire program into idempotent regions [33, 32].

An idempotent region is a SEME (single-entry, multiple-exits) subgraph of the control flow graph (CFG) of the program. It can be freely re-executed without loss of correctness. More precisely, a region is *idempotent* if and only if it always generates the same output whenever the program jumps back to the region entry from any execution point within the region. To achieve this, the region inputs must not be overwritten, i.e., no anti-dependence on the

inputs, during the execution of the region. Otherwise, re-executing the region can generate unexpected output since the inputs do not remain the same when the program jumps back to the beginning of the region.

In light of this, researchers propose different kinds of techniques to preserve the inputs. Any recovery schemes must preserve both the *memory* and *register* inputs with regard to the region boundary for correct recovery. Interestingly, previous techniques [33, 38] have developed simple algorithms to elegantly dismiss the overhead for preserving the memory inputs by partitioning the regions such that the memory inputs will never be overwritten in the regions (i.e., no anti-dependence to the memory input). Therefore, preserving the register inputs becomes the only source of cost.

De Kruijf *et al.* [33] (renamed as Idem hereafter) leverages register renaming to eliminate the anti-dependence on the register inputs, thus achieving idempotence at the expense of increasing the register pressure. Figure 4.1 (b) shows how Idem renames X to Z at S_2 to eliminate the anti-dependence on register X in the bottom region of the original code in Figure 4.1 (a). In contrast, Feng *et al.* [32] (renamed as Encore) preserve the register inputs by logging at the region entry only the register inputs that have anti-dependence. Figure 4.1 (c) shows how Encore preserves the register inputs by checkpointing only X at the region entry. Once a fault is detected during the execution of the region, Encore consults the checkpointed value to restore the inputs to the region for recovery.

However, prior idempotence-based recovery schemes neither guarantee recovery without expensive hardware support. nor achieve insignificant performance overhead [33, 38, 32, 101].

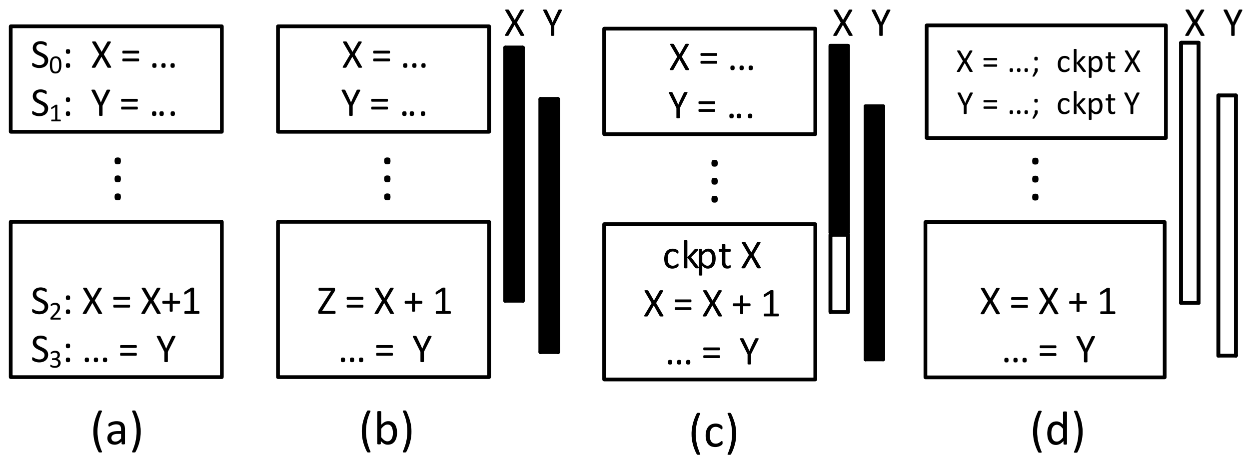


Figure 4.1: Idempotent processing and vulnerability window
(a) original code (partitioned into regions in code boxes), (b) Idem[33], (c) Encore[32], (d) Bolt

Lack of Recovery Guarantee

To provide guaranteed recovery, previous schemes must assume the following expensive hardware supports:

First, the contents in the internal structures (e.g., register file, instruction queue etc.) must remain intact. To illustrate, the vertical bars in Figure 4.1 (b) and (c) show the vulnerability windows of input X and Y for the bottom region; each time point of the window represents whether the input value in the register file (RF) is recoverable (white) or unrecoverable (black) from a fault at that time point. For example, if input Y is corrupted in the RF after defined at S1, then both Idem and Encore fail to recover from the soft error because the re-execution starts from the corrupted states. Thus, previous schemes assume ECC protection to the RF and other internal structures which is excessively expensive in terms of power, delay, and area for low-cost systems. It is reported that ECC protection to RF can incur

an order of magnitude larger power consumption [102, 103], up to 3X the delay of ALU operations [103] and 22% area overhead [104].

Second, all the writes events must write to the correct destination. Thus, the read/write combinational logic to those internal structure must be hardened, which is exorbitant in commodity processors.

Significant Performance Overhead

Previous techniques incur significant performance overhead due to register renaming [33] or register logging [32]. We observe up to 40% performance overhead in our experiments. Taking into account that soft errors rarely occur (1/day in 16nm [35]), programmers are reluctant to use idempotence for such rare error correction at the cost of paying the high performance overhead all day.

However, the existing idempotence-based recovery includes the following limitations. First, prior idempotence-based recovery schemes cannot provide guaranteed recovery without expensive hardware support (Section 5.2), which greatly undermine the benefits brought by the fine-grained recovery [33, 32, 81, 82, 38]. Second, even with the expensive hardware support, prior schemes introduce a prohibitive performance overhead due to their instrumentation/-transformation.

In light of these challenges, we present Bolt, a practical compiler-directed soft error recovery scheme that provides 1) guaranteed recovery without expensive hardware support, 2) negligible performance overhead for fault-free execution, and 3) fast and fine-grained error

recovery on a fault. Bolt leverages the following two key insights: First, it is still possible to achieve the guaranteed recovery by checkpointing only the necessary architectural states for idempotent region boundaries without expensive hardware support. To this end, we propose **eager checkpointing** to preserve the value of the registers that are *live-in* to the regions as soon as those registers are defined, obviating expensive hardware support.

Second, there are correlations among the checkpoints created by the eager checkpointing, That is, some checkpointed value can be reconstructed by other checkpointed values, thereby being removable without compromising the recovery guarantee. This insight enables Bolt's **checkpoint pruning**, a compiler technique to achieve negligible performance overhead. Bolt explores the program dependence graph (PDG) [105] among these checkpoints and identifies the subset of the checkpoints, which is essential for soft error recovery, to minimize the performance overhead.

Following are the major contributions of this chapter:

- To the best of our knowledge, Bolt is the first fine-grained soft error recovery scheme without expensive hardware support and significant performance overhead.
- Bolt can correct soft errors even if the internal structures (e.g., register file, instruction queue) are corrupted by incorrect destination write events or multi-bit flips. Note that unlike previous schemes Bolt requires no hardware protection such as ECC for these internal structures.
- Bolt incurs only 4.7% runtime overhead across a large set of applications which benefits

from Bolt's novel compiler analysis to eliminate unnecessary checkpoints.

- To better understand the performance of Bolt, we also implemented two state-of-the-art fine-grained recovery schemes that require expensive hardware support for recovery guarantee. Bolt outperforms these schemes achieving 57% and 49% runtime overhead reduction on average.

4.2 Fault Model

Except the aforementioned hardware support in the internal structures, Bolt shares the other assumptions in prior idempotent recovery schemes [33, 32, 101, 38, 81]: (1) Store queue, caches, and main memory are protected with ECC which has already existed in current commodity processors [39, 106]. (2) As with branch misprediction, all the stores must be verified. They are buffered until the region reaches the end with no error detected. This is called store verification. For this purpose, gated store queue [107, 43, 42] is often used, and we evaluated its buffering overhead in our experiments (Section 4.7.3). (3) PC and SP are protected as in prior schemes. However, we argue that only PC needs parity checking while in fact all other special registers can be handled by our scheme (see Section 4.6). (4) All the faults should be detected within the regions (see Section 4.6),

4.3 Overview of Bolt

Bolt proposes two novel compiler techniques to address the above challenges and offers a practical fine-grained recovery scheme. Eager checkpointing provides guaranteed recovery

without expensive hardware support. Checkpointing pruning minimizes the performance overhead by eliminating unnecessary checkpoints.

4.3.1 Eager Checkpointing: Guaranteed Recovery without Prohibitive Hardware Support

Bolt preserves the register inputs to the regions throughout their entire liveness period. To achieve this, Bolt eagerly checkpoints the value of *register inputs* to a region as soon as they are defined (Figure 4.1(d)). Such define-time checkpointing guarantees recovery of all the inputs to each idempotent code region. In particular, Bolt checkpoints once for each register input by tracking the last write. Even if an input is defined multiple times in one region, Bolt checkpoints only one time the value of the last definition.

Artificial Define-Checkpoint Vulnerable Window: One may be concerned about a vulnerable window where the register is defined and subsequently checkpointed. However, such vulnerable window is considered artificial because *in eager checkpointing, the checkpoints in one region are for the subsequent regions, not the current one.* That is, even if a checkpoint is corrupted during the define-checkpoint window in a region r, the checkpoint will not affect the recovery of the current region r and will be recreated (corrected) during the re-execution of r upon recovery. Thus, such a vulnerable window is implicitly eliminated in our eager checkpointing scheme. In case the checkpointing store may corrupt other memory locations, Bolt simply follows the aforementioned fault model and buffers those stores until they are verified as with branch misprediction.

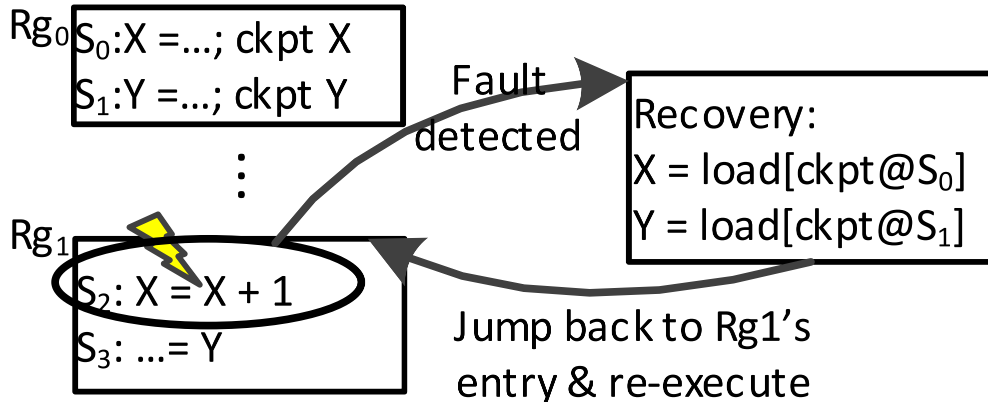


Figure 4.2: Bolt's recovery model.

4.3.2 Checkpoint Pruning: Minimizing Performance Overhead

The overhead of idempotent processing is proportional to the number of checkpoints executed at runtime. With that in mind, we propose a novel compiler analysis that can identify unnecessary checkpoints in those eager checkpoints based on the following insight. *For a value corrupted due to a soft error, the original value can be restored without checkpointing it, as long as it can be recomputed by leveraging other checkpoints.* Without compromising the recovery capability, Bolt formulates the problem of checkpoint pruning as that of finding a *recovery slice*, which can recompute the value of the pruned checkpoints (Section 4.4.3). Such a slice is similar to traditional *backward slices* [108], however, with more constraints. If the recovery slice is successfully built, Bolt removes the corresponding checkpoint. On a fault, Bolt's recovery runtime will simply execute the slice to reconstruct the original value.

The takeaway is that checkpoint pruning enables Bolt to effectively offload the runtime overhead of fault-free execution to the fault-recovery, which is indispensable taking into account the low soft error rate(1 error/day [35]).

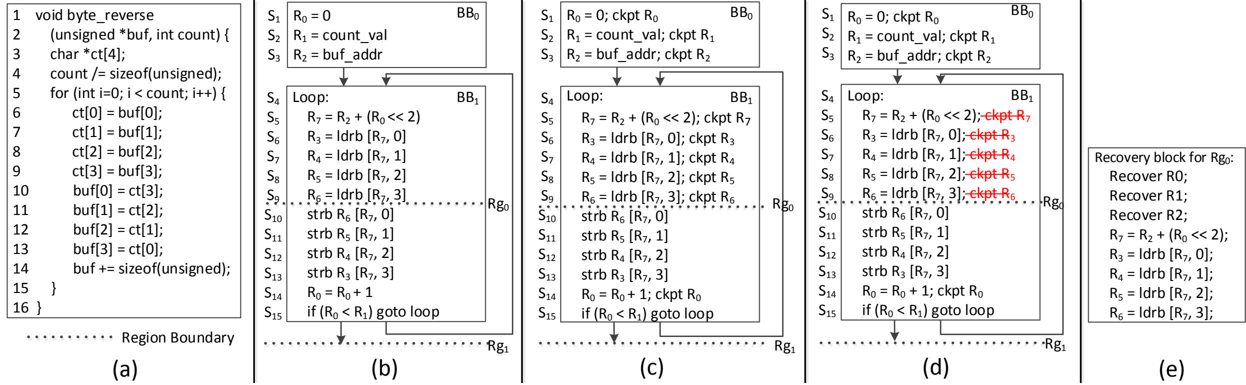


Figure 4.3: Motivating example with *byte_reverse* code of `sha` in MiBench: only interesting part is shown

4.3.3 Fault Recovery Model

Once a soft error is detected during the execution of the region, Bolt's runtime system first discards the buffered stores in the faulty region. Then, it takes the control to execute a *recovery block* that restores all the inputs to the faulty region, i.e., the same live-in registers as they were at the beginning of the region before the fault occurred. Bolt can generate the recovery block either statically or dynamically (Section 4.5.2). Lastly, Bolt redirects the program control to the entry of the faulty region and re-starts from it. Figure 4.2 describes such a recovery model.

Motivating Example: Figure 4.3 shows how Bolt works as a whole for the *byte_reverse* function of `sha` in MiBench [51]. (a) shows the code snippet of the *byte_reverse* function. (b) illustrates the control flow graph divided into idempotent regions (Rg_0,Rg_1) by using an adapted region partitioning algorithm based on Idem [33] where dashed lines show the region boundaries. (c) shows Bolt's eager checkpointing to provide guaranteed recovery where the

register inputs to Rg_0 are $R_0 \sim R_7$. (d) minimizes the performance overhead with checkpoint pruning where (e) is the resultant recovery block for Rg_0. For example, the value of R_7 can be reconstructed by executing the recovery block that consults the checkpointed values of R_0 and R_2. In this example, Bolt can achieve fine-grained guaranteed recovery without expensive hardware and over 80% performance improvement by pruning the checkpoints in the loop.

Algorithm 3 The High-level Bolt Algorithm

Inputs: CFG PDG

Outputs: Minimal Checkpoint Set $\mathbb{MIN_CKPT}$

1: $\mathbb{REGION} \leftarrow$ region_formation(**CFG**)

2: $\mathbb{BASE_CKPT} \leftarrow$ eager_checkpoint($\mathbb{REGION}$, **CFG**)

3: $\mathbb{MIN_CKPT} \leftarrow$ checkpoint_pruning($\mathbb{BASE_CKPT}$, **CFG**, **PDG**)

4.4 Bolt Compiler

Algorithm 3 shows a high-level Bolt algorithm which takes the control flow graph (CFG) and program dependence graph (PDG) as inputs. Bolt first partitions the entire CFG into different regions (Section 4.4.1). Then, it performs the eager checkpointing, that inserts checkpoints right after the last-updated registers in each region, to preserve the register inputs for guaranteed recovery without expensive hardware support (Section 4.4.2). Finally, Bolt prunes those checkpoints that can be reconstructed by other checkpointed value to minimize the performance overhead (Section 4.4.3, 4.4.4).

4.4.1 Region Formation

Bolt is versatile in that it is applicable to many region formation schemes [33, 32, 81, 82]. As discussed in Section 5.2.3, previous idempotence-base recovery schemes [33, 38] have developed a simple region partition algorithm to guarantee no memory anti-dependence in the regions, making preserving register inputs the only cost.

For comparison of Bolt and other idempotence-based recovery schemes, this paper intentionally uses Idem's region formation algorithm [33] to partition the entire program (CFG) into different idempotent regions. By doing so, we can fairly compare Bolt with the other schemes that leverage different methodologies for register input preservation.

Bolt also treats memory fences as region boundaries to obey underlying memory models and handles the I/O instructions as single instruction region as with [33, 32].

In particular, Bolt checks if the original idempotent regions overflow the store buffer, in which case such regions are split. This is particularly important for the store verification (See fault model in Section 4.2). Note that prior schemes do not prevent the overflow, which is another reason why they cannot achieve the guaranteed error recovery[1].

4.4.2 Eager Checkpointing

To achieve guaranteed recovery in the absence of expensive hardware support, Bolt employs eager checkpointing that preserves register inputs to a region right after their definition.

[1] Bolt's technique to avoid the overflow is implemented on top of Idem's region formation and used for other schemes in experiments for comparison.

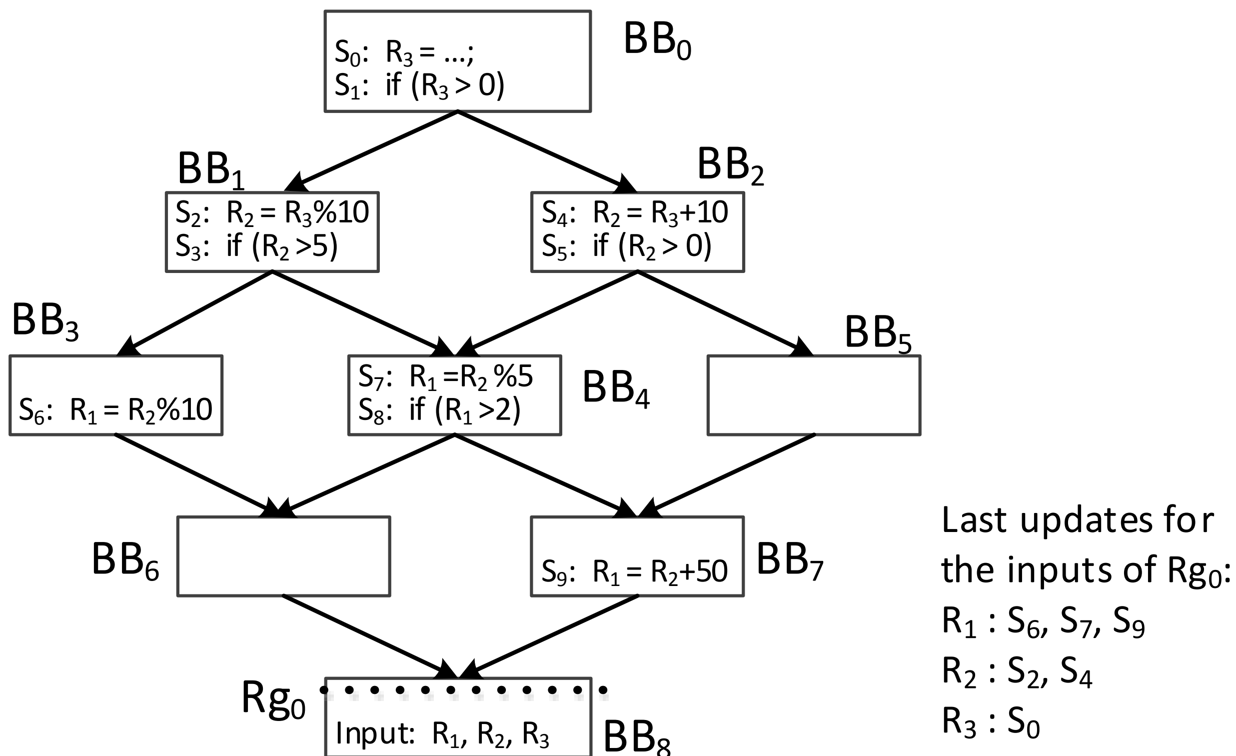


Figure 4.4: An eager checkpointing example

That is, Bolt is interested in the last update instructions that define the register inputs prior to the region entry. The identification of register inputs (live-ins) to a region is a standard analysis in modern compilers and is omitted due to space constraints. Given the partitioned region $\mathbb{REGION}$, for each register input r of a region $Rg \in \mathbb{REGION}$, Bolt reverses the edges of CFG and traverses it starting from the entry of Rg in a depth first order, to search for the last update instructions of r. Figure 4.4 shows an example of the last updates to the inputs of region Rg_0 where the inputs are R_1, R_2, R_3.

All those identified last update instructions form the baseline checkpoint set $\mathbb{BASE_CKPT}$ where some checkpoints in the set might be eliminated by Bolt's checkpoint pruning techniques. Therefore, in the worst case, Bolt can just instruments right after these last update

instructions in $\mathbb{BASE_CKPT}$ with checkpointing stores to achieve the guaranteed recovery, but at a worse performance overhead. Upon recovery, Bolt's runtime system simply recovers the checkpointed input as in Figure 4.2. Hereafter, we refer the last updates in $\mathbb{BASE_CKPT}$ as *checkpoints* in ease of illustration.

4.4.3 Checkpoint Pruning

To reduce the runtime overhead, Bolt prunes the checkpoints in $\mathbb{BASE_CKPT}$ without compromising the recovery capability. The problem of checkpoint pruning is to find an minimal subset $\mathbb{MIN_CKPT}$ out of $\mathbb{BASE_CKPT}$ that still allows Bolt's recovery runtime to restore all the register inputs of a faulty region in the event of a soft error. To address this, Bolt leverages the following axiom:

Axiom 2. *Given a register input r of a region Rg, if r's value can be* **safely** *reconstructed, the checkpoints for r's last updates are unnecessary for Rg.*

At the first glance, the checkpoint pruning problem simply seems like a program slicing problem [108, 109] by exploring the backward slice of the register inputs. However, traditional backward slicing cannot guarantee the value of the register inputs to be **safely** reconstructed, i.e. restoring the register inputs to their original value. Note that such a guarantee is required according to axiom 2. Therefore, eliminating checkpoints with the traditional backward slicing is unsafe. For instance, Figure 4.5 shows examples of unsafe checkpoint pruning with the traditional backward slicing. (a) tries to eliminate the checkpoint for input R_2 of region Rg_1 at S_3 by leveraging S_1 and S_2. However, the checkpoint for R_1 at S_1 is unsafe as it will be overwritten by the checkpoint at S_5. Note, each register has one checkpointing location

in a specially reserved region of the stack frame. Thus, the value of R_2 cannot be recovered upon recovery. (b) attempts to eliminate the checkpoints for input R_1 of region Rg_1 at S_5 and S_9 leveraging backward slicing to recover input R_1. However, the checkpoint of R_3 at S_1 is unsafe as it might be overwritten by S_7, thus failing to recover R_1 due to the lack of control flow consideration.

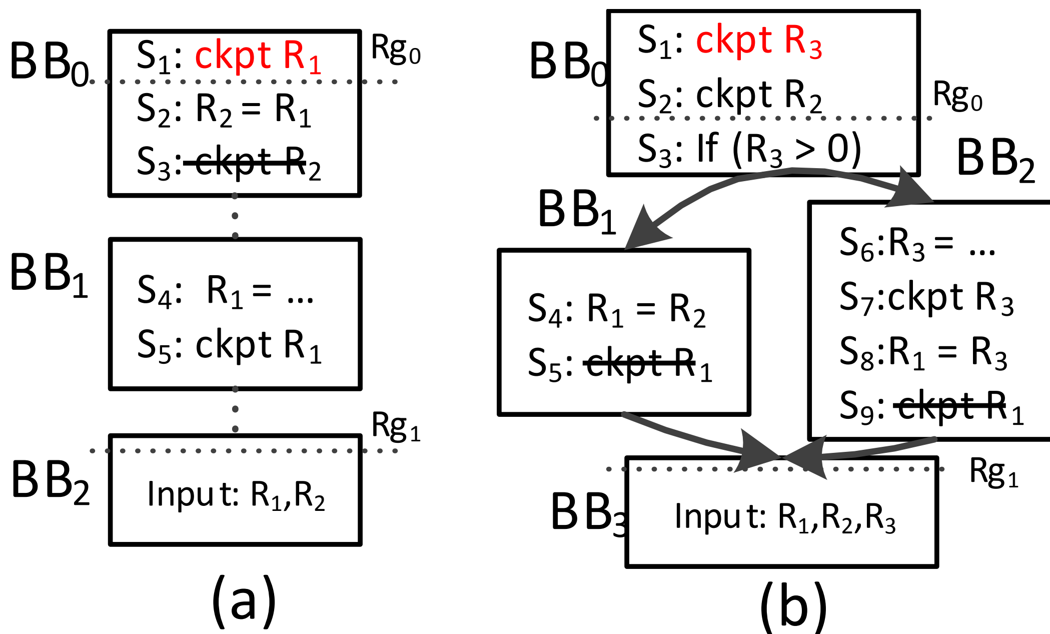


Figure 4.5: Examples of unsafe checkpoint pruning.

Therefore, Bolt introduces **recovery slice** which guarantees to restore all the register inputs upon recovery. To determine whether it is safe to eliminate the checkpoints for a register input, we must guarantee that the integrity of both the data flow and control flow of the recovery slice such that the recovery slice can precisely reconstruct the value of register inputs without eagerly checkpointing them.

To construct a recovery slice from the program dependence graph (PDG), Bolt must depend

on (1) *Data dependence backtracking* to ensure that the resulting slice recomputes the value of register input from only **safe** checkpoints and statements; and (2) *Control dependence backtracking* to guarantee the right control flow in the recovery slice. Note, in a PDG, all the statements are represented as vertices and connected with edges annotating control/data dependence relationship.

Data Dependence Backtracking

To reconstruct the data flow of register inputs, Bolt traverses backwards through the vertices of the PDG via the data-dependent edges in a depth-first search manner. We use the notation $v \xrightarrow[\delta^d]{r} v'$ to denote v is data dependent on v', i.e., v' defines the register r that can reach v, and v uses r. Given a register input r of a region Rg, Bolt backtracks along a sequence of vertices ($RgE \xrightarrow[\delta^d]{r_1} v_1 \xrightarrow[\delta^d]{r_2} \dots \xrightarrow[\delta^d]{r_n} v_n$), where RgE represents the entry of region Rg which is data-dependent on r_1 and v_n is the last node in a PDG path. In particular, the backtracking terminates along the path when one of the following is met:

- The vertex v_n has no data-dependent edge;
- The vertex v_n has already been in $\mathbb{MIN_CKPT}$ set;
- The vertex v_n is an unsafe statement.

First, if there is no vertex on which v_n depends, it is in form of $r = const$, On a fault, the register can be re-assigned with the constant value, thus Bolt can recover r without checkpointing it.

Second, if v_n has already been in in $\mathbb{MIN_CKPT}$ set, it means v_n fails in previous data/control-dependence backtracking. Therefore, Bolt terminates backtracking along the path and validates whether v_n is a safe checkpoint to ensure data flow integrity. That is, the checkpoint must not be overwritten along all the **reachable control flow paths (RCFP)**. We use the notation $v \xrightarrow[\Delta]{r} v'$, where v defines r used by v', to denote the control flow paths on which v can reach v' without intervening definition of r along the path. To validate a checkpoint (v_n) along the path, $v_n \xrightarrow[\Delta]{r_n} v_{n-1} \xrightarrow[\Delta]{r_{n-1}} \ldots \xrightarrow[\Delta]{r_1} RgE$, Bolt simply traverses the path to ensure there is no other checkpoints for the same register r_n. If the validation succeeds, Bolt terminates backtracking along this path. Otherwise, Bolt return to the most recent vertex (v_i) that are in the $\mathbb{BASE_CKPT}$ set and validate the vertex until Bolt find a safe checkpoint and place the checkpoint to $\mathbb{MIN_CKPT}$. Then, Bolt terminates backtracking along this path. Note, whenever Bolt puts a checkpoint into $\mathbb{MIN_CKPT}$, Bolt needs to verifiy whether the checkpoint breaks any existing recovery slices, i.e. the checkpoint overwrites the checkpoints depended by the existing recovery slices. In such cases, Bolt needs to invalidate those broken recovery slices and re-construct them.

Lastly, Bolt also terminates backtracking if v_n is an unsafe statement, e.g., *load, call.* Same with validating checkpoints, if the value in the memory location of the *load* is overwritten along RCFP, Bolt cannot rely on the value to build the recovery slice. Thus, Bolt must validate the *load* same as validating checkpoints to ensure the no stores overwrite the same memory location. As Bolt limits itself to intra-procedural in its current form, Bolt also stops backtracking from call instructions and applies the same procedure to call instructions as

with dealing unsafe checkpoints.

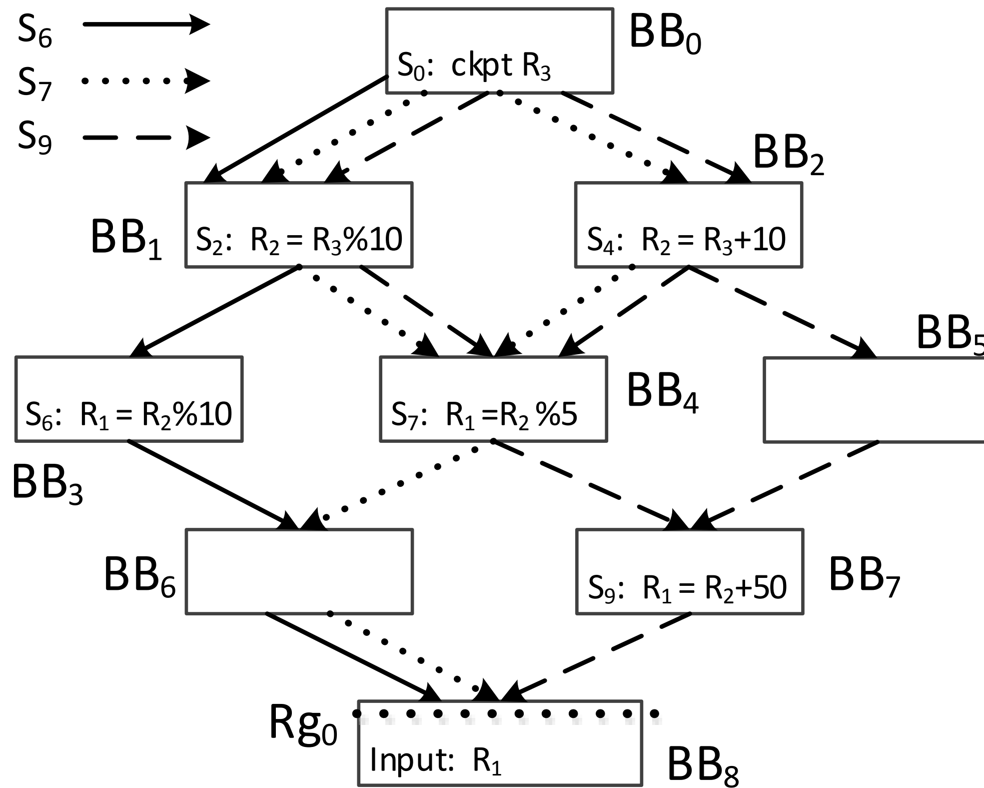


Figure 4.6: A data dependence backtracking example

Figure 4.6 shows a data dependence backtracking example following the example in Figure 4.4 for a live-in register R_1 with respect to region Rg_0. Assuming the checkpoint at S_0 is already in $\mathbb{MIN_CKPT}$, all data-dependent paths terminate at S_0 during the data dependence backtracking. The reachable control flow paths (RCFP) via R_1's last update instructions (i.e., S_6, S_7, and S_9) are shown in with different type of line. The checkpoint for R_3 at S_0 is not overwritten along all the RCFP. Thus, Bolt ensures the integrity of the data flows for register input R_1.

Control Dependence Backtrack

Suppose a vertex v_i has a set of data-dependent vertices ($\mathbb{V}$) for register r, i.e., $\forall v'_i \in \mathbb{V}$, $v_i \xrightarrow[\delta^d]{r} v'_i$. Once all the data-dependent paths via v_i successfully finish data dependent backtracking, Bolt should ensure the control flow integrity so that the recovery slice can produce the expected value of r at v_i. We consider the following 3 cases according to the size of $\mathbb{V}$, i.e., the number of data-dependent vertices of v_i, and whether all these dependent vertices are checkpointed or not:

- If $|\mathbb{V}|$ is 1, the control always reaches v_i after its singular data-dependent vertex.
- If $|\mathbb{V}|$ is greater than 1 and all the data-dependent vertices are checkpointed, there is no need to distinguish the checkpoints as they store the checkpointed value to the same location reserved in the stack frame. Thus, we can restore the value of r from the location.
- If $|\mathbb{V}|$ is greater than 1 but not all of the data-dependent vertices are checkpointed, Bolt needs to distinguish them by tracking their control flow.

Only for the third case, Bolt backtracks control dependence to ensure that only one of the values produced by the data-dependent vertices in $\mathbb{V}$ can reach the vertex v_i. To achieve this, Bolt first computes the nearest common dominating predicate $Pred_{ncd}$ of the data-dependent vertices based on a *dominator tree* [85]. Then, it traverses control-dependent edges backwards from each vertex in $\mathbb{V}$ up to $Pred_{ncd}$ validating the visited control predicates.

To simplify presentation, our discussion below assumes that each basic block in CFG contains at most one predicate. We use the notation $v \underset{\delta^c}{\longrightarrow} v'$ to represent v is control dependent on v'. For each vertex v'_i in $\mathbb{V}$, Bolt inspects the sequence of vertices is $v'_i \underset{\delta^c}{\longrightarrow} v_{c0} \underset{\delta^c}{\longrightarrow} \ldots \underset{\delta^c}{\longrightarrow} v_{cn}$ where $v_{c0} \ldots v_{cn}$ are the control dependence predicates, and v_{cn} is the $Pred_{ncd}$ vertex. Then, Bolt validates the control dependence predicates, e.g., $v_{c0} \ldots v_{cn}$ by exploring the recovery slice of each predicate $\{v_{ci} | i \in \{0 \ldots n\}\}$. Note that every vertex in the recovery slice of v_{ci} should be validated by traversing the reachable control flow paths not only to v_{ci}, but also to the region entry RgE. In other words, every vertex in the recovery slice of v_{ci} should also be validated along the paths: $v_{ci} \underset{\Delta}{\longrightarrow} v_i \overset{r_i}{\underset{\Delta}{\longrightarrow}} \ldots\ RgE$. Specifically, $v_{ci} \underset{\Delta}{\longrightarrow} v_i$ represents all the control flow paths that can reach v_i after v_{ci}.

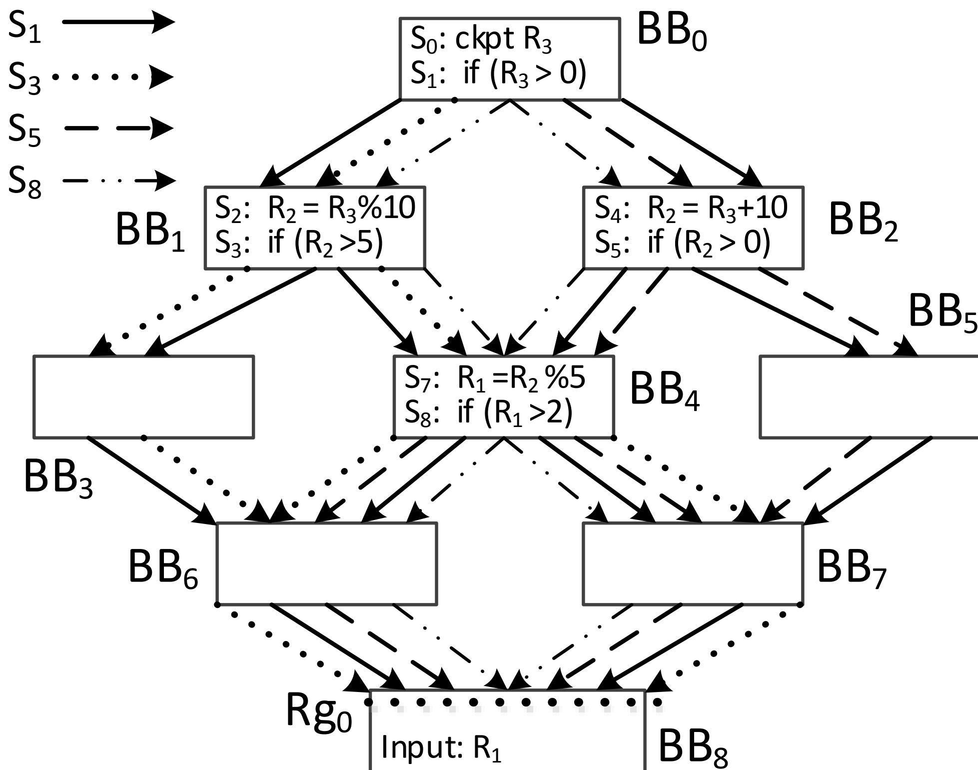


Figure 4.7: A control dependence backtracking example

As shown in Figure 4.7, once all the paths via the data dependent vertices (e.g., S_6, S_7, S_9 in Figure 4.4) of register input R_1 are successfully backtracked, Bolt starts the control dependence backtracking. First, Bolt determines the nearest common dominating predicate of S_6, S_8, S_9 as S_1. Then, from each data dependent vertex (i.e., S_6, S_7, S_9) to the nearest common dominating predicate, Bolt validates each control-dependent vertex, i.e., S_1, S_3, S_5 and S_8. In this example, Bolt can validate the checkpoint S_0 in the predicate S_3's recovery slice along the reachable control flow path (RCFP) to S_3 ($BB_0 \rightarrow BB_1$) as well as every RCFP from S_3 to the region entry. In Figure 4.7, as the checkpoint for R_3 is not overwritten along all the RCFP via each control dependence predicate. Therefore, Bolt consider the control flow of the recovery slice of register input R_1 are well-formed and the checkpoints for the last updates of R_1 (S_6, S_7, S_9) can be safely eliminated.

4.4.4 Checkpoint Pruning for Loop

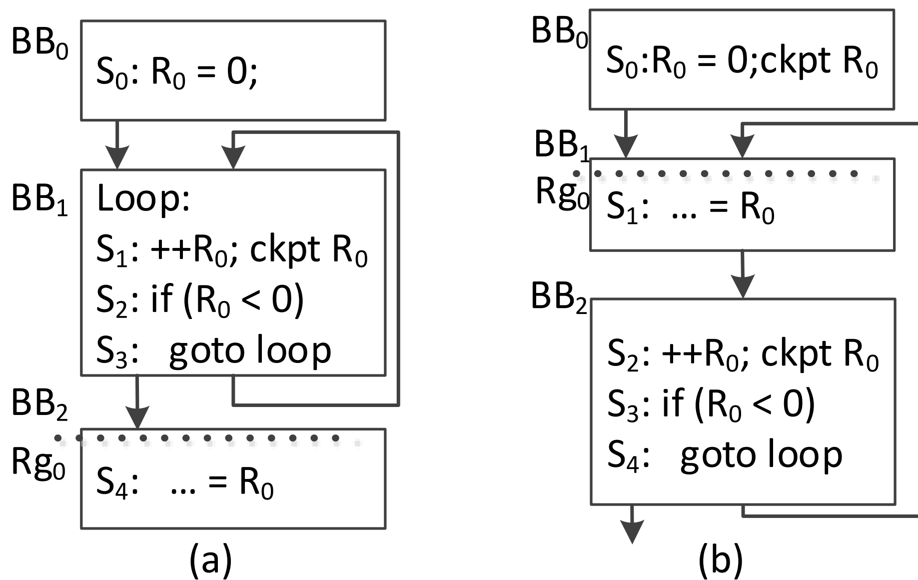


Figure 4.8: An illustrating example of ineliminable checkpoints for loop

Incorporating loops into the checkpoint pruning complicates the analysis, since they cause issues in the data/control dependence backtracking. Figure 4.8 (a) corresponds to the case where the data dependence backtracking fails in the loop while Figure 4.8 (b) to the case where the control dependence backtracking fails. Suppose that R_0 is live-in to a region, Rg_0 starts at BB_2 in Figure 4.8 (a). The last update of R_0 happens in the instruction S_1. As Bolt tries to eliminate the checkpoint for S_1, it runs into infinite backtracking as S_1 is data dependent on itself. To solve the problem, Bolt tracks the history of visited vertices and treats any of them as unsafe statements during the backtracking.

Figure 4.8 (b) demonstrates the infeasible case of the control dependence backtracking. Here, Rg_0 starts at BB_1, and R_0 is live-in to Rg_0. Even if the data dependence backtracking may be successful in this case (i.e., checkpoint at S_0 can be eliminated), Bolt cannot differentiate definitions of R_0 at S_0 and S_2 by the control dependence backtracking. Fortunately, these two cases only happen for basic induction variables and other non-linear induction variables [110]. To get around the problem, Bolt places the last updates of such variables into $\mathbb{MIN_CKPT}$.

4.5 Implementation

The Bolt compiler described in Section 4.4 is a series of passes for the LLVM compiler framework [49]. After checkpoint pruning, Bolt instruments each last update instruction in $\mathbb{MIN_CKPT}$ with corresponding checkpointing stores. In addition, Bolt also considers the following implementation details.

4.5.1 Limiting the Backtrack Depth for Recovery Time

In the limit, the exploration of recovery slice for a checkpoint explores all the way to the initial input of the program. That is, one can always recover by restarting the entire execution. Besides, this also influences scalability for compilation. Therefore, it is important for Bolt to set a reasonable backtrack depth which limits the depth of the dependence backtracking. For example, a backtrack depth of 5 prevents Bolt from backtracking more than 5 data/control-dependent vertices in the program dependence graph (PDG). If the dependence backtracking does not terminate by itself within the depth, Bolt simply treats the last-visited vertex as if it is unsafe, applying the same procedure to deal with invalid statements. In particular, we discover that a small backtrack depth (10) is enough to achieve the significant reduction of checkpoint candidates compared with that of a high backtrack depth (100). The implication is two-fold: First, our compiler is scalable; Second, the fault-recovery execution time is reasonably small. Section 4.7 evaluates different backtrack depths in more details.

4.5.2 Just-in-time Recovery Slice Generation

Before jumping back to the beginning of a faulty region where the error is detected, Bolt's runtime system needs to executes a recovery slice to restore the inputs to the region (See Section 5.3). For this purpose, Bolt can generate the recovery slice either statically or dynamically.

Bolt can statically generate the recovery slice during the exploration to prune the checkpoints. However, two problems limit a static approach: (1) It significantly increases the code size

by generating the recovery block for each region, which is prohibited for low-end embedded systems. (2) More importantly, the static slice generation cannot exploit the opportunity to prune the checkpoints for the inputs to a function boundary which is also a region boundary. Recall that Bolt's checkpoints store the register value to a reserved location in the stack frame of each function. Therefore, the checkpoints of the callee do not overwrite that of the caller, even if they save the same register for checkpointing. That is, we can leverage the checkpoints in the caller to reconstruct the checkpoints in the callee. However, it is very expensive to determine the calling contexts statically.

In contrast, a dynamic approach is much more preferable due to the lack of these problems. Code size will not increase, because the recovery slice is built dynamically. For the second problem of the static approach, Bolt's runtime system first builds the program dependence graph by analyzing the binary. Then, it generates the recovery slice using the algorithm in Section 4.4.3. Note that since all required analyses are performed after the register allocation, there is no technical problem in achieving such just-in-time slice generation without relying on source code. By looking at the return address of the callee's stack, Bolt can determine the caller function and continue the recovery slice exploration from the call site. In essence, Bolt can "inter-procedurally" generate the recovery slice. Section 4.7.4 further investigates the overhead of Bolt's just-in-time recovery slice generation.

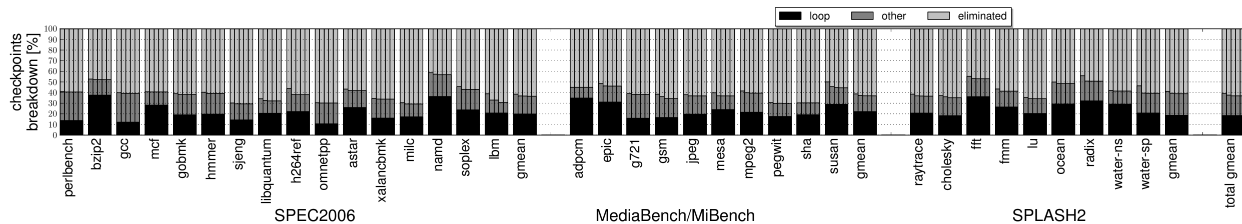
4.6 Discussion

Error Detection Detection schemes are orthogonal to our proposed recovery scheme. As with other region-based recovery schemes [33, 101, 32], soft errors must not escape the region where they occur to achieve full recoverability. Existing software [81, 82, 12, 35] and hardware [36, 26, 53] approaches can be employed for the previous work [33, 101, 32] to achieve full detection coverage within the region. For example, Idem [33] leverages dual-modular-redundancy (DMR) [12, 53] to detect the errors before leaving the faulty region. However, DMR-based detection schemes come with expensive performance/area overhead which might overwhelm the benefits brought by the fine-grained recovery schemes.

Fortunately, Clover [81, 82] proposes an efficient error detection scheme to contain the errors within the idempotent-region while incurring negligible area and moderate performance overheads. Clover detects all the soft errors before leaving the region where they occurred by using the acoustic sensors [36] and partial instruction duplication. In fact, Clover can further reduce their overhead: 1) The area overhead and detection latency of the sensors can be significantly reduced with careful placement of the sensors on top of the processor-die rather than naive mesh-like placement [78, 36]. 2) Extending the idempotent-region length for less instruction-duplication can dramatically reduce the performance overhead; previous work [38] shows that the region can be lengthen by orders of magnitudes with precise points-to analysis thus tolerating a much higher detection latency. Thus, we believe said scheme would serve well for Bolt.

Special Register Protection There are various special register in the register file such as stack pointer (SP), condition status register (e.g. EFLAGS) and program counter PC, etc. While all other special registers can be checkpointed in the stack frame as with the regular registers, Bolt checkpoints SP in a global array in case of multithreaded program. Thus, upon recovery, we first leverage instruction like *rdtscp* to retrieve the thread ID. Then, Bolt safely reloads SP from the element in the global array with the thread ID. Last, using the recovered SP, Bolt can retrieve all other checkpointed register value safely from the stack frame.

Impact on HPC applications As soft-errors are becoming more dominant in large scale HPC systems [111], it is critical to innovate new schemes that can reduce the overhead of checkpointing and soft-error recovery. Unfortunately, pure software-based schemes incur very high overhead and often have high barrier to entry for adoption since it may require changing the application source code. On the other hand, hardware-based schemes impose high chip area and performance overheads. Therefore, a compiler-based scheme, such as proposed in this study, is likely to positively impact scientific applications running on HPC systems, since it does not require expensive hardware support for register file protection and manual code changes. As shown in our evaluation section, Bolt reduces the soft-error recovery overhead by 95% for a number of pthread applications taken from SPLASH benchmark suites, and relieves chip designers from providing expensive ECC protection for register file and other internal structures such as instruction queue – potentially reducing the chip design and



From left to right, the columns demonstrate the fraction of necessary (ineliminable and other) checkpoints after applying our checkpoint pruning optimization to prune the checkpoint candidates with different backtrack depths (5, 10, 20, 50, 100).

Figure 4.9: Checkpoint Breakdown

testing cost as well.

4.7 Evaluation and Analysis

To evaluate Bolt, we first analyze the checkpoint pruning optimization and how different backtrack depths impact this optimization. In particular, how this affects the number of checkpoints to be removed. Then, we evaluate Bolt's overhead and compare with the state-of-the-art idempotence-based soft error recovery schemes. Finally, we evaluate our fault-recovery execution overhead after fault occurrence.

4.7.1 Experimental Methodology

We conduct our simulations on top of the Gem5 simulator [52] with the ARMv7 ISA, modeling a modern two-issue out-of-order 2 GHz processor with L1-I/D (32KB, 2-way, 2-cycle latency, LRU), and L2 (2MB, 8-way, 20-cycle latency, LRU) caches. The pipeline width is two; and the ROB, physical integer register file, and load/store buffer have 192, 256, and 42 entries, respectively.

For the experiments, we use three sets of benchmarks: SPEC2006 [90] for general-purpose

computation, MediaBench/MiBench [50, 51, 91] for embedded systems, and SPLASH2 [92] for parallel systems. All the applications are compiled with a standard `-O3` optimization and fully simulated with appropriate inputs.

4.7.2 The Breakdown of Checkpoint Candidates

Figure 4.9 shows how Bolt's checkpoint pruning works across different backtrack depths with the breakdown of *checkpoint candidates* set, i.e., $\mathbb{BASE_CKPT}$ (See Section 4.4.2). In the breakdown of each bar, the top portion corresponds to the checkpoints that can be eliminated while the rest correspond to the necessary checkpoints that must be instrumented. We classify the necessary checkpoints as *loop* and *other*. The *loop* checkpoints are the ones Bolt cannot prune due to limitations for loop discussed in Section 4.4.4, thus they remain the same regardless of the backtrack depth. In contrast, ***other*** checkpoints are affected by different backtrack depths because they are the ones identified by the recovery slice exploration. For each application, we show such five breakdown bars corresponding to the backtrack depths of 5, 10, 20, 50, and 100, respectively (from left to right in Figure 4.9). We make the following observations:

- Bolt's checkpoint pruning technique is effective at decreasing the number of checkpoints. It can eliminate on average more than 60% of the checkpoint candidates ($\mathbb{BASE_CKPT}$).
- For most of the applications, the portion of the eliminated checkpoints start to saturate when the backtrack depth is greater than 10. Such small backtrack depth is beneficial in

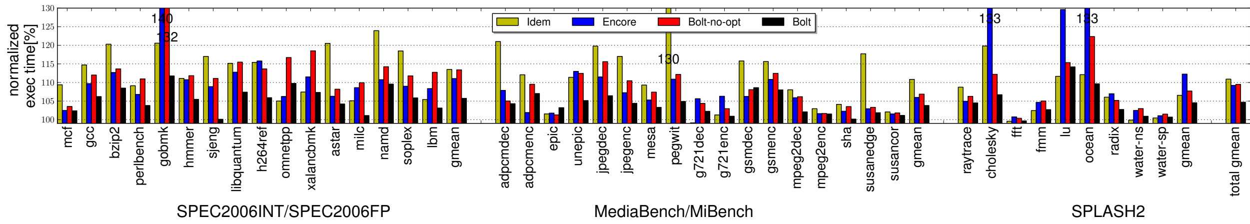


Figure 4.10: Performance overhead in terms of execution time (cycles) considering the architectural effect of store buffering.

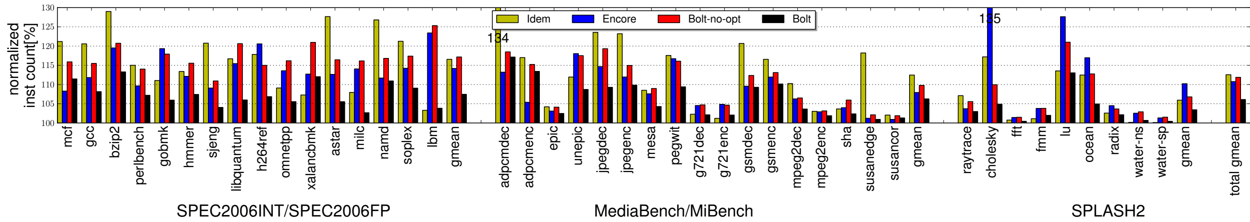


Figure 4.11: Architectural-neutral performance overhead in terms of total dynamic instruction count.

two-fold: (1) the compilation will be scalable as the number of backtrack is dramatically reduced, (2) The recovery time is reasonably small after fault occurrence because at most 10 instructions will be executed to recover one register input.

With those in mind, we empirically determine the backtrack depth as 10 for the remaining experiments.

4.7.3 Overheads

We compare performance overhead with that of the state-of-the-art's recovery techniques. For comparison, all the techniques employ the same region construction algorithm described in Idem [33]. We set the baseline to the original application binary without any recovery support. Before presenting more detailed discussion, we categorize and summarize each technique as follows:

- **Idem** [33] uses register renaming to preserve the live-in registers that have anti-dependence. Idem may increase the register pressure, thus degrading the performance due to the resulting register spillings and reloadings. It requires expensive RF protection to guarantee soft error recovery.

- **Encore** is our version of Encore [32]. It checkpoints the live-in register, that have anti-dependence, at the region entry. As with Idem, Encore also assumes RF protection.

- **Bolt-no-opt** checkpoints all the live-in registers as soon as they are defined (Section 4.4.2), offering guaranteed soft error recovery without RF protection.

- **Bolt** is equipped with the checkpoint pruning, that eliminates unnecessary checkpoints, offering guaranteed soft error recovery at low overhead without RF protection.

We present performance overhead in two forms. The first one reflects the architectural effect of store buffering which buffers the unverified stores until their region ends (See fault model in Section 5.2). For the other form, we provide a architecture-neutral performance overhead in terms of total dynamic instruction count [33, 32, 38].

Runtime Overhead with Architectural Effect

Figure 4.10 shows the architectural performance overhead in terms of execution time (cycles) where the y-axis represents the overhead as percentage compared to that of the original program. We modified the simulator to model the effect of store buffering, i.e. the processor holds the stores in one region until the region ends. Then, those stores will be drained to

the caches if there is available bandwidth between the store queue and caches. For those regions (<0.001%) that contain stores more than the size of store queue (42 entry in intel i7 haswell), we can place additional region boundaries to break those regions into smaller ones so that they won't overflow the store queue.

Store buffering might adversely degrade the performance. If the stores in the previous region cannot be drained to cache due to the bandwidth congestion, those stores have to stay in the store queue. Thus, later stores in the current region cannot get executed if the store queue is full, causing pipeline stall. However, we found out that such situation is rare and store buffering trivially affects the performance by $< 1\%$ (not shown in the figure).

Idem incurs more overhead in most of the applications compared to other schemes. Idem can introduce on average an 11% performance loss and up to a 30% performance loss. This is reasonable as Idem preserves the region inputs by register renaming which may ends up with more spillings and reloadings. Worse, reloading instructions (load) are on the critical path of processor which might greatly degrade the performance. In contrast, other schemes just need to pay the overhead of checkpointing stores which are off the critical path. Besides, the performance overhead of Idem will further degrade given the effect of ECC protection to the register file. It's interesting to observe that Idem outperforms Encore and Bolt-no-opt in the SPLASH2 applications. This is because the regions in the applications of SPLASH2 generally have larger region sizes which hide Idem's overhead as register pressure is higher in larger regions. This phenomenon is corroborated by the previous work [33, 38].

Encore in general performs better than Bolt-no-opt as it only checkpoints the live-in register

with anti-dependence in the region resulting in 9.2% overhead on average. In a few applications (e.g. *gobmk*, *cholesky*, *lu* etc.), Encore incurs much more overhead (upto 40%) than Bolt-no-opt. This is because Bolt-no-opt employ eager checkpointing and thus avoid putting some checkpoints in the loops. As with Idem, Encore also have to pay the overhead of RF protection.

Without RF protection, Bolt-no-opt achieves a comparable performance overhead with Encore and Idem resulting in 9.5% on average. Since the effect of adding ECC to a RF is not considered in our experiments, the power efficiency and performance hits to Encore and Idem are not visible. But, with good reason Bolt-no-opt should dominate both Encore and Idem respectively.

To further reduce and shift the runtime overhead of fault-free execution to that fault-recovery execution, Bolt applies the checkpoint pruning optimization. This pruning greatly shrinks overall overhead leading to an average overhead of 4.7%, which is 57% and 49% reduction compared to Idem and Encore respectively. Again, the improvements are under-estimated since our simulation does not reflect the ECC delay of the prior work. In summary, Bolt provides guaranteed soft error recovery at low overhead without expensive RF hardware protection schemes.

Architecture-Neutral Performance Overhead

Figure 4.11 shows architecture-neutral performance overhead across different schemes. A similar trend as in Figure 4.10 is observed in the dynamic instruction count overhead. As

Generation time (ms)	$\leq$5	$\leq$10	$\leq$20	$\leq$50
Ratio in all regions (%)	95.72	3.02	0.97	0.28

Table 4.1: Distribution of the time to generate recovery slice.

expected, Bolt executes much less instructions than Idem and Encore. Note that a checkpoint approach is in general preferable over a register renaming approache as stores are off the critical path in modern out-of-order processors. The downside of the checkpoint approach is that it needs to restore checkpointed data for recovery making it slightly slower. However, as soft errors happen once in a while, it is much more desirable to make the common fault-free case faster.

4.7.4 Fault-Recovery Overhead

After a fault occurrence, Bolt invokes the exception handler to generate the recovery slice and execute the recovery slice to recover the region input before releasing the control to the entry of faulty region. Thus, fault-recovery time in Bolt contains two parts: (1) time to generate recovery slice and; (2) time to execute the slice and the faulty region.

As we limit the backtrack depth to 10, the maximum number of instructions in the slice is less than 10 $\times$#RF, where #RF is the size of register file. Since the region sizes are $<$ 30 instructions on average, the time to execute the slice and faulty region are trivially small relative to the time to generate recovery slice.

We perform dynamic recovery slice generation to examine Bolt's practicality. For each recovery slice, we first generate the program dependence graph (PDG) and dominator tree

(DT) information. Then, we generate the recovery slice based on the PDG and DT information with our adapted algorithm. Then the total cycles are recorded. Table 4.1 shows the recovery time distribution for all the regions generated from all the applications reported. We use a 2GHz processor frequency to calculate the time in millisecond (ms) scale. As we can see, over 95.72% of regions can generate their recovery slice within 5 ms., and 99.99% of the region can generate their recovery slice within 50 ms, which is negligible for user as soft error happens rather infrequently.

4.8 Other Related Work

This section describes the prior works related to soft error recovery. We also explain how our proposed scheme advances the state-of-art and differs from previous approaches in this domain.

There exists a large body of work on soft-error recovery with software/hardware approaches. For commercial systems, hardware/software recovery schemes [95, 36, 93, 96] involve taking a snapshot of the system status including register file and memory. To achieve that, they usually maintain multiple copies of register files and a memory log to checkpoint the whole system status. For example, Upasani *et al.* [36] requires two additional copies of the architectural state units (register files, RAT, etc.) with their ECC protection. Besides, they modify the cache structure and its coherency protocol for memory logging. Such recovery schemes usually introduce exorbitant performance/energy/area overhead making them only viable in high-end commercial server systems.

Flushing the pipeline to recover from a soft error [24, 37] is another alternative. However, such recovery schemes require the errors to be detected before the instruction is committed in the pipeline implying a high-cost detection scheme. Other techniques also explore simultaneous multithreading (SMT) [53] to recover the leading thread from the trailing thread which occupy the computing resource leading to performance degradation. Chang *et al.* enable fault recovery at the granularity of a single instruction by incorporating triple-modular-redundancy (TMR) [13]. TMR essentially copies the original execution to two more redundant executions and recovers the error by majority voting among those three versions. However, they also introduce significant performance overhead preventing their adoption in commodity systems.

In contrast, idempotent-based recovery is a promising recovery approach. Our proposed schemes eliminate the performance/hardware overhead problem with our novel compiler analysis making our idempotence-based recovery scheme realistic to be applied in low-cost commodity systems.

4.9 Summary

We present Bolt, a lightweight soft error recovery scheme. Bolt guarantees 100% recovery without expensive register file protection. It can recover from soft errors even in the case when the RF is corrupted. To the best of our knowledge, Bolt is the first compiler-directed recovery solution that does not require expensive RF protection mechanisms for idempotence.

We also demonstrate that Bolt can effectively shift the runtime overhead of fault-free ex-

ecution to that of fault-recovery execution for lightweight idempotent processing. The experiment results show that Bolt incurs only 4.7% performance overhead on average which is 57% and 49% reduction compared to two state-of-the-art schemes that require expensive hardware support for the same recovery guarantee as Bolt.

Chapter 5

Error Resilience for Nonvolatile Memory Systems

The previous chapters leverage idempotent processing to provide low-cost recovery for soft error resilience. This chapter dedicates to solve another type of single-event error with idempotent processing in the nonvolatile memory system. To this end, this chapter presents *iDO*, a compiler-directed approach to failure atomicity with nonvolatile memory. Unlike most prior work, which instruments each store of persistent data for redo or undo logging, the iDO compiler leverages idempotent recovery thus eliminating the need to log every persistent store. The compiler then arranges, during recovery from failure, to back up each thread to the beginning of the current idempotent region and re-execute to the end of the current failure-atomic section. As a result, iDO significantly outperforms state-of-the art persistence mechanisms on current hardware during normal execution, while preserving very fast recovery times.

5.1 Challenges and Contributions

With the emergence of fast, byte-addressable nonvolatile memory such as commercial 3D XPoint, ReRAM, and STT-RAM, we can now conceive of systems in which main memory, accessed with ordinary loads and stores, is simply "always available," and need not be flushed to the file system to survive a crash. The obvious use case of such a technology, and the one we focus on here, is to allow programmers to store heap objects persistently in memory, bypassing the expensive serialization of those objects onto traditional storage devices. This use case is widely applicable: we envision applications using persistent heap objects as an alternative to disk-resident local databases or as a way (e.g., on energy-harvesting devices with frequent crashes) to enable fast restarts.

Unfortunately, from the perspective of crash recovery, nonvolatile main memory is compromised by the fact that traditional caches can write data back to memory in arbitrary order, leading to inconsistent values in the wake of a crash [27, 18]. A failure in the middle of a linked-list insertion, for example, may lead to a post-crash dangling reference if the `next` pointer of the predecessor node is written back to memory before the inserted node itself. Moreover even in the absence of reordering, failure during an operation that is meant to be atomic can leave the contents of memory in an inconsistent intermediate state, rendering it unusable.

In order to avoid such errors and ensure post-crash consistency of persistent data, researchers have developed failure-atomicity systems that allow programmers to delineate failure-atomic

operations on the persistent data—typically in the form of transactions[1] [15, 16, 17, 18, 19] or *failure-atomic sections* (FASEs) protected by outermost locks [20, 21, 22]. Given knowledge of where operations start and end, the failure-atomicity system can ensure, via logging or some other approach, that all operations within the code region happen atomically with respect to failure and maintain the consistency of the persistent data. Transactions have potential advantages with respect to ease of programming and (potentially) performance during normal operation (at least in comparison to coarse-grain locking), but can be difficult to retrofit into existing code, due to idioms like hand-over-hand locking and limitations on the use of condition synchronization or irreversible operations. Transactions also tend to perform more poorly than well tuned fine-grain locking. Our own work is based on locking.

The principal challenge of FASE-based recovery, compared to transactional recovery, stems from the lack of isolation in critical sections. In a lock-based program, FASEs that involve more than one lock, even when data-race–free, may be able to see each other's changes while both are still in progress; in fact, correct execution may *depend* on them seeing each other's changes (e.g., for condition synchronization). UNDO logging, which makes updates "in place," avoids hiding the FASE's updates, but must address the possibility that a completed FASE will depend on values written in some other FASE that was interrupted by a crash. Systems like Atlas [20], which incorporate UNDO logging, must therefore track cross-FASE dependences and be prepared to roll back even completed FASEs during post-crash recovery. A similar problem arises with REDO logging for FASEs, as in the NVThreads system [21]:

[1] In this dissertation, we refer the transaction system to the software transactional memory system. Exploration of hardware transactional memory system can be a possible future work.

if an incomplete FASE releases a lock, it must share its locally buffered changes with any thread that subsequently acquires the lock; it must also track the dependence. If the earlier FASE fails, the dependent must fail as well. This implies that when a thread reaches the end of a dependent FASE during normal execution, it must wait until the earlier FASE has completed before replaying its own log and proceeding.

To simplify the management of logs for FASE-based persistence, and, in particular, to avoid the need for dependence tracking, Izraelevitz et al. introduced the notion of JUSTDO logging [22]. Rather than rolling back a FASE during recovery (as one would with UNDO logging) or replaying a FASE's writes (as one would with REDO logging), the JUSTDO system logs enough information to *resume* a FASE during recovery and execute it to completion ("recovery via resumption"). Immediately prior to each store instruction in a FASE, the JUSTDO system logs (in persistent memory) the program counter, the to-be-updated address, and the value to be written. During recovery, the system uses the code of the crashed program to complete the remainder of each interrupted FASE, beginning with the most recent log entry. Future program runs can then be assured that the recovered data is consistent, much as conventional programs can be ensured of the integrity of data in a journaled file system.

The problem with JUSTDO logging is its requirement that the log be written *and made persistent* before the related store—a requirement that is very expensive to fulfill on conventional machines with volatile caches. Current ISAs provide limited support for ordering write-back from cache to persistent memory, and these limitations seem likely to continue into the foreseeable future [112]. To ensure that writes reach memory in a particular order, the program

must typically employ a sequence of instructions referred to as a *persist fence*. On an Intel x86, the sequence is ⟨`sfence, clwb, clwb, ..., sfence`⟩. This sequence initiates and waits for the write-back of a set of cache lines, ensuring that they will be persistent before any future writes. Unfortunately, the wait incurs the cost of round-trip communication with the memory controller.

Given the cost of persistence ordering, JUSTDO assumes—unlike Atlas and NVThreads—that it will run on a machine in which caches are persistent, due either to implementation in STT-RAM or to capacitor-driven flushing in the event of power failure. On a more conventional machine with persist fences, JUSTDO is 2–3× slower than Atlas [22, 21].

JUSTDO logging also imposes a restricted programming model within FASEs, with no use of volatile data and no caching of values in registers [22, 21]. These restrictions would seem to preclude the use of SIMD instructions, widely regarded as essential to data-intensive applications [113, 114] and in-memory databases [115, 116].

The key contribution of our work is to demonstrate that recovery via resumption can in fact be made efficient on conventional machines, with volatile caches and expensive persist fences. The key is to arrange for each log operation (and in particular each persist fence) to cover multiple store instructions of the original application. We achieve this coverage via compiler-based identification of *idempotent* instruction sequences. Because an idempotent region of code can safely be re-executed an arbitrary number of times without changing its output, the recovery procedure in the wake of a crash can resume execution at the beginning of the current region, eliminating the need to log each individual store instruction of the

original program.

We present iDO, a practical compiler-directed failure-atomicity system. Like JUSTDO logging, iDO supports fine-grained concurrency through lock-based FASEs, and avoids the need to track dependences by executing forward to the end of each FASE during post-crash recovery. Unlike JUSTDO, iDO allows the use of registers in FASEs, and persists its stores at coarser granularity. While these advantages should allow iDO to outperform prior systems on hypothetical machines with nonvolatile caches, experiments confirm that it also outperforms them—by substantial margins—on conventional machines with volatile caches.

Instead of logging information at every store instruction, iDO logs (and persists) a slightly larger amount of program state (registers, live stack variables, and the program counter) at the beginning of every idempotent code region within the overall FASE. In practice, idempotent sequences tend to be significantly longer than the span between consecutive stores—tens of instructions in our benchmarks; hundreds or even thousands of instructions in larger applications [33]. As iDO is implemented in the LLVM tool chain [49], our implementation is also able to implement a variety of important optimizations, logging significantly less information—and packing it into fewer cache lines—than one might naively expect. We also introduce a new implementation for FASE-boundary locks that requires only a single memory fence, rather than the two employed in JUSTDO.

Our principal contributions can be summarized as follows:

- We introduce iDO logging, a lightweight strategy that leverages idempotence to ensure

both the atomicity of FASEs and the consistency of persistent memory in the wake of a system crash. Rather than log individual memory stores, iDO logs a lightweight summary of live program state at the beginning of each idempotent region.

- We describe an implementation of iDO in the LLVM toolchain [49].
- We compare the performance of iDO to that of several existing systems, demonstrating up to an order of magnitude improvement over Atlas in run-time speed, and dramatically better scaling than transactional systems like Mnemosyne [18].
- We verify that recovery time in iDO is also very fast—one to two orders of magnitude faster than Atlas in long-running programs.

This chapter is organized as follows. Section 5.2 gives additional background on failure-atomicity systems and idempotence. Section 5.3 discusses the high-level design of iDO logging; Section 5.4 delves into system details. Performance results are presented in Section 5.5. We discuss related work in Section 5.6 and conclude in Section 5.7.

5.2 Preliminaries

5.2.1 System Model

iDO assumes a near-term hybrid architecture (Fig. 5.1), in which some of main memory has been replaced with nonvolatile memory, but the rest of main memory, the caches, and the processor registers remain volatile. Data in the core and caches are therefore transient

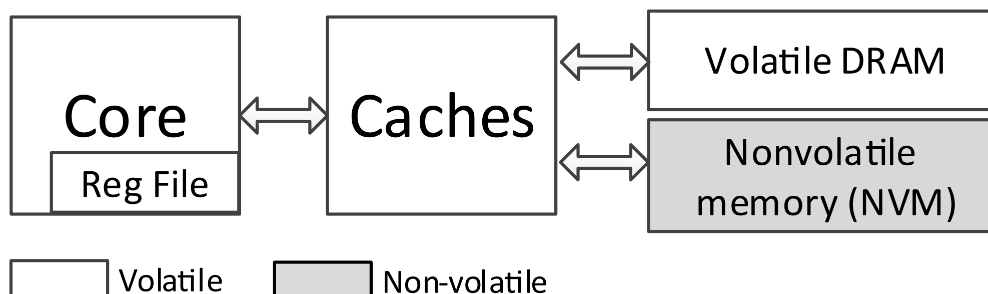


Figure 5.1: Hybrid architecture model in which a portion of memory is nonvolatile, but the core, caches, and DRAM are volatile.

and will be lost on system failure.[2] Portions of main memory are likely to continue to be implemented with DRAM in the short term, due to density, cost, and/or endurance issues with some NVM technologies. As in other recent work, we assume that read and write latencies of NVM are similar to those of DRAM [20, 21] and that writes are atomic at 8-byte granularity [117]. Our failure model encompasses (only) fail-stop errors that arise outside the running application. These include kernel panics, power outages, and various kinds of hardware failure.

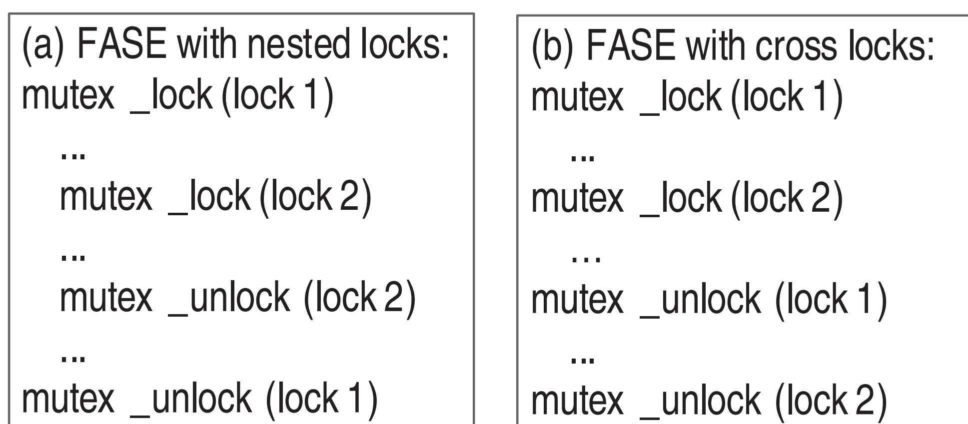


Figure 5.2: FASEs with different interleaved lock patterns.

[2]In general, we refer to physical memory as *volatile* or *nonvolatile*, and to program memory (data) as *transient* or *persistent*.

5.2.2 Programming Model

As noted in Section 5.1, iDO employs a programming model based on lock-delineated failure-atomic sections (FASEs), primarily because of their ubiquity in existing code. A FASE is defined as a maximal-length region of code beginning with a lock (mutex) acquire operation and ending with a lock release, in which at least one lock is always held [118, 20, 21, 22]. Note that the outermost lock and unlock pairs do not necessarily need to be the same. Figure 5.2 shows examples of FASEs with two possible interleaved lock patterns. The left-hand side shows nested locks; the right has a cross-locking (hand-over-hand locking) pattern.

For each FASE, iDO provides a variant of the classic ACID transaction properties [119]:

atomicity means that updates to persistent data performed in a FASE complete in an "all or nothing" manner. A FASE that is interrupted by a crash is completed as part of the recovery procedure.

consistency is typically defined by program semantics. We assume that every FASE transitions memory from a state in which all program invariants hold to another in which they still hold. By completing an interrupted FASE during recovery, we preserve consistency even in the presence of failures by pushing persistent data to a state in which no locks are held.

isolation requires that a transaction never see other threads' changes during its execution and, likewise, that its own changes be invisible until commit time. In a FASE-based programming model, isolation is a consequence of mutual exclusion, but only for

properly-nested FASEs (Fig. 5.2(a)) with the same outermost locks.

durability (persistence) means that the results of FASEs survive crashes. More specifically, if the results of one FASE are visible to a second, and the second survives a crash, the first survives as well.

For single threaded programs or code that accesses privatized variables, iDO also supports programmer-delineated *durable code regions*. These code regions are defined by the programmer to be failure atomic but lack the isolation guarantees of lock-delineated FASEs. From here on, we use the term "FASE" to denote both lock- and programmer-delineated failure atomic code regions.

FASE-based failure-atomicity systems based on UNDO and REDO logging typically prohibit thread communication outside of critical sections [20, 21]. This prohibition prevents a happens-before dependence between critical sections from being created without the system's knowledge. An advantage we gain from recovery via resumption is that thread communication outside of critical sections can occur without compromising correctness.

Despite its strengths, recovery via resumption has some pitfalls. In order for recovery to succeed, the failure atomic code region must be allowed to be run to completion. For this reason, resumption is infeasible for speculative transactions, which must be able to abort and roll back all updates made so far when a conflict is detected late in their execution (possibly during recovery). Consequently, iDO logging is vulnerable to software bugs within FASEs—on recovery, reexecuting the buggy code will not restore consistency.

In this work, we assume a programming model that expects all writes to persistent locations to occur within lock- or programmer-delineated FASEs. This model ensures that program state after a crash corresponds to a cut across the store order aligned with either a lock release or the end of a durable code region in every thread. Like other FASE-based systems [20, 21, 22], we disallow not only conventional data races within FASEs but also races on atomic variables, to avoid the possibility that the order in which the race is resolved may be inverted at recovery time. Provided they do not cause a race, persistent reads are allowed outside FASEs.

5.2.3 Idempotence

An *idempotent region* is a single-entry, (possibly) multiple-exit subgraph of the control flow graph of the program. In keeping with standard terminology, we use the term *inputs* to refer to variables that are *live-in* to a region. An input has a definition that reaches the region entry and a use of that definition after the region entry. Similarly, we use the term *outputs* to refer to variables that are updated in a region and *live-out* at the end of the region. That is, an output of a region is a variable written in the region that serves as an input to some following region. We also use the term *antidependence* to refer to a write-after-read dependence, in which a variable is used and subsequently overwritten. A region is *idempotent* if and only if it would generate the same output if control were to jump back to the region entry from any execution point within the region (assuming isolation from other threads). To enable such a jump-back, the region inputs must not be overwritten—i.e., there must be

no antidependence on the inputs—during the execution of the region.

Idempotent regions have been used for a variety of purposes in the literature, including recovery from exceptions, failed speculation, and various kinds of hardware faults [33, 41, 82, 42]. For any of these purposes—and for iDO—inputs must be preserved to enable re-execution.

5.3 iDO Failure Atomicity System

Unlike UNDO or REDO logging, iDO logging provides failure atomicity via *resumption* and requires no log for individual memory stores. Once a thread enters a FASE, iDO must ensure that it completes the FASE, even in the presence of failures. At the beginning of each idempotent code region in the body of a FASE, all inputs to the region are known to have been logged in persistent memory. Since the region is idempotent, the thread never overwrites the region's inputs before the next log event. Consequently, if a crash interrupts the execution of the idempotent region, iDO can re-execute the idempotent region from the beginning using the persistent inputs.

More precisely, at the end of each (compiler-delineated) idempotent region, iDO logs the *output* data of the region—the data that were modified by the region and that serve as input to some following region. In data flow terms, we define the output of region r as the live-out data defined in the region—the values that are written and *downward-exposed* [85]:

$$OutputSet_r = Def_r \bigcap LiveOut_r \tag{5.1}$$

where Def_r is the set of values defined in r and $LiveOut_r$ is the set of live-out values of r. The iDO compiler persists the values in $OutputSet_r$ at the end of region r.

Successful recovery requires additional care. In particular, if we re-execute a FASE using a *recovery thread*, this thread must hold the same locks as the original crashed thread. Tracking this information is the responsibility of the thread's local *lock_array* (Sec. 5.3.1), which is updated at every lock acquisition and release.

The following subsections consider the structure of the iDO log, the implementation of FASE-boundary locks, and the recovery procedure. Additional compiler details—and in particular, the steps required to identify FASEs and transform the FASEs into a series of idempotent regions—are deferred to Section 5.4.

5.3.1 The iDO Log

For each thread, the iDO runtime creates a structure called the `iDO_Log`. We manage the per-thread iDO logs using a global linked list whose `iDO_head` is placed in a persistent memory location to be found by the recovery procedure (Sec. 5.4.3). Log structures are added to the list at thread creation. As shown in Figure 5.3, each `iDO_log` structure comprises four key fields. The `recovery_pc` field points to the initial instruction of the current idempotent region. The `intRF` and `floatRF` fields hold live-out register values; each register has a fixed location in its array. The `lock_array` field holds *indirect lock*

addresses for the mutexes owned by the thread—more on this in Section 5.3.2.

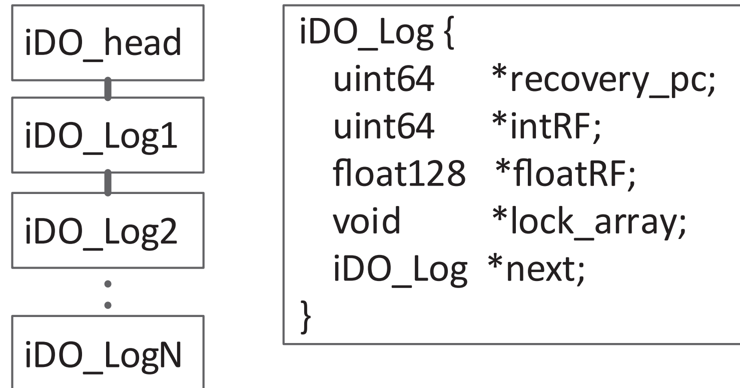


Figure 5.3: iDO log structure and management: the number of iDO logs matches the number of threads created.

Here then is the series of steps required, within a FASE, to complete the execution of idempotent region r and begin the execution of region s:

1. Issue write-back instructions for all output registers of r (saving them to `intRF` and `floatRF`) and for all output values in the stack. Together, these comprise $OutputSet_r$. Note that live-out values that were not written in r are already sure to have persisted; no additional action is required for them.

2. Update `recovery_pc` to point to the beginning of s. Once this step is finished, s can be re-executed to recover from failures that occur during its execution.

3. Execute the code of s, generating the values in $OutputSet_s$. These values will be persisted at the end of s—i.e., the boundary between s and its own successor t, as described in step 1. Note that by definition an idempotent region will never overwrite its own input.

iDO continues in this fashion until the end of the FASE. To enforce the order of these steps, the iDO compiler inserts a single *persist fence* between the first step and the second, and again between the second and the third. After completing the steps, a thread moves on to the next idempotent region. Output registers are written to `intRF` and `floatRF` immediately after their final modification in the current region. Writes-back of output values in the stack are likewise initiated immediately after the final write of the current region, though we do not wait for completion until the fence between steps 1 and 2. In the absence of precise pointer analysis, we cannot always identify the final writes to variables accessed via pointers; these are therefore tracked at run time and then written back at the end of each idempotent region.

Recovery in the wake of a crash is described in Section 5.3.3.

5.3.2 Indirect Locking

Our discussion thus far has talked mostly about idempotent regions. To obtain failure atomicity for entire FASEs, we must introduce lock recovery. In particular, in the wake of a crash, we must reassign locks that were held at the time of the crash to the correct recovery threads, ensure that those locks are held before re-executing the interrupted FASEs, and guarantee that no other locks are accidentally left locked from the previous program execution (else deadlock might occur). Previous approaches [15, 22] persist each mutex. Then, during recovery, they unlock each held mutex to release it from a failed thread before assigning it to a recovery thread. In JUSTDO logging, this task requires updating a *lock intention log* and a *lock ownership log* before and after the lock operation. Each lock or unlock operation then

entails two persist fence sequences—a significant expense.

iDO introduces a novel approach that avoids the need to make mutexes persistent. The key insight is that all mutexes must be unlocked after a system failure, so their values are not really needed. We can therefore minimize persistence overhead by introducing an *indirect lock holder* for each lock. The lock holder resides in persistent memory and holds the (immutable) address of the (transient) lock. During normal execution, immediately after acquiring a lock, a thread records the address of the lock holder in one of `lock_array` entries of the `iDO_Log`. It also sets a bit in an initial index slot in the array to indicate which array slots are live. Immediately before releasing a lock, the thread clears both the `lock_array` entry and the bit. Finally, the iDO compiler inserts an idempotent region boundary immediately after each lock acquire and before each lock release.

Upon system failure, each transient mutex will be lost. The recovery procedure, however, will allocate a new transient lock for every indirect lock holder, and arrange for each recovery thread to acquire the (new) locks identified by lock holders in its `lock_array`. An interesting side effect of this scheme (also present in JUSTDO logging), is that if one thread acquires a lock and, before recording the indirect lock holder, the system crashes, another thread may steal the lock in recovery! This effect turns out to be harmless: the region boundaries after lock acquire ensure that the robbed thread failed to execute any instructions under the lock.

5.3.3 iDO Recovery

Building on the preceding subsections, we can now summarize the entire recovery procedure:

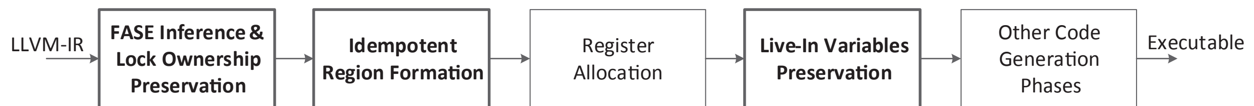


Figure 5.4: iDO compiler overview. Starting with LLVM IR from dragonegg/clang, the compiler performs three iDO phases (indicated in bold) and then generates an executable.

1. On process restart, iDO detects the crash and retrieves the `iDO_Log` linked list.

2. iDO initializes and creates a recovery thread for each entry in the log list.

3. Each recovery thread reacquires the locks in its `lock_array` and executes a barrier with respect to other threads.

4. Each recovery thread restores its registers (including the stack pointer) from its iDO log, and jumps to the beginning of its interrupted idempotent region.

5. Each thread executes to the end of its current FASE, at which point no thread holds a lock, recovery is complete, and the recovery process can terminate.

It should be emphasized that, as with all failure atomicity systems, iDO logging does not implement full checkpointing of an executing program, nor does it provide a means of restarting execution or of continuing beyond the end of interrupted FASEs. Once the crashed program's persistent data is consistent, further recovery (if any) is fully application specific.

5.4 Implementation Details

5.4.1 Compiler Implementation

Figure 5.4 shows an overview of the iDO compiler, which is built on top of LLVM. It takes the generated LLVM-IR from the frontend as input. It then performs three phases of instrumentation and generates the executable. We discuss the three phases in the paragraphs below.

FASE Inference and Lock Ownership Preservation In its first instrumentation phase, the iDO compiler infers FASE boundaries in lock-based code, and then instruments lock and unlock operations with iDO library calls, on the assumption that each FASE is confined to a single function. As in the technical specification for transactions in C++ [120], one might prefer in a production-quality system to have language extensions with which to mark FASE boundaries in the program source, and to identify functions and function pointers that might be called from within a FASE.

Idempotent Region Formation In its second instrumentation phase, the iDO compiler identifies idempotent regions. Previous idempotence-based recovery schemes have developed a simple region partition algorithm to guarantee the absence of memory antidependences, making the preservation of live-in variables the only run-time cost. We use the specific scheme developed by De Kruijf et al. [33]. The iDO compiler first computes a set of cutting points for antidependent pairs of memory accesses using LLVM's basicAA alias analysis,

then applies a hitting set algorithm to select the best cutting strategy. We report region characteristics in Section 5.5.3.

Preserving Inputs and Persisting Outputs In its third and final instrumentation phase, the iDO compiler performs two key analyses. First, it guarantees that the inputs to each idempotent region are not overwritten during the region's execution. For registers, we artificially extend the live interval of each live-in register to the end of the region [38], thereby preventing the register allocator from assigning other live intervals in the region to the same register and reintroducing an antidependence. For stack variables, we perform a similar live interval extension, preventing them from being shared in LLVM's stack coloring phase [49].

In a second, related analysis, the iDO compiler ensures that outputs of the current idempotent region have persisted at the end of the region. As noted in Section 5.3.1, registers that are live-out but were not written in the region (i.e., are being passed through from a previous region) are already known to have persisted. If a register is written multiple times, only the final value is logged. The log entries are then persisted (written back) at the end of the idempotent region. Similarly, writes-back of output values in the stack are initiated at the final write of the idempotent region. Writes-back of variables accessed via pointers (e.g., in the heap) are tracked at run time and then written back at the end of the region.

5.4.2 Persist Coalescing

As a further optimization, the iDO compiler takes advantage of the fact that register values are small, and do not need to persist in any particular order. A system like Atlas, which logs 32 bytes of information for every store, can persist at most two contiguous log entries in a single 64-byte cache line write-back. In iDO, as many as eight register values can be persisted with a single write-back (`clflush`). This *persist coalescing* [27] is always safe in iDO, even though registers are grouped by name rather than by order of update at run time, because the registers logged in the current region are used only in later regions. If, for example, a running program updates registers A, C, and B, in that order, it is still safe to persist the logged values of A and B together, followed by C.

5.4.3 Persistent Region Support

iDO requires mechanisms to enable processes to allocate regions of persistent memory and make those regions visible to the program. We leverage Atlas's implementation for this purpose. Atlas's region manager represents persistent memory regions as files, which processes incorporate into their address space via `mmap`. The mapped regions then support memory allocation methods such as `nv_malloc` and `nv_calloc`.

5.5 Evaluation

For our evaluation of iDO logging we compared against several alternative failure atomicity runtimes. We employ real-world applications to explore iDO logging's performance impact

during normal (crash-free) execution. We also employ microbenchmarks to measure scalability. For all our benchmarks, we report statistics on the idempotent regions as a guide to understanding performance. Separately, we measure recovery time. Finally, we assess the sensitivity of our results to changes in NVM latency.

Where applicable, we compare against the following failure atomic runtimes, which guarantee crash consistency on a persistent memory machine.

Atlas is an UNDO-logging system that uses locks for synchronization. Like iDO logging, Atlas equates failure-atomic regions with outermost critical sections. The use of UNDO logging allows Atlas to delay a FASE's writes-back (though not those of its UNDO log) until the end of the FASE. At the same time, the lack of isolation, combined with the rollback-based recovery model, forces Atlas to track dependences across critical sections and to be prepared to roll back even a completed FASE if it depends on some other FASE that failed to complete before a crash [20].

Mnemosyne is a REDO-based transactional system integrated into the language-level transactions of C and C++ [18]. We used the updated version included in the recently published WHISPER benchmark suite [121].

JUSTDO is a recovery-via-resumption system, originally designed for machines with persistent caches, that logs recovery information at every store [22]. Unlike the version from the original paper, our JUSTDO implementation adopts the iDO strategy of placing the program stack in nonvolatile memory. This change leads to a significant performance

improvement by avoiding the need to manually copy stack variables into the heap on FASE initialization.

NVML is Intel's UNDO-logging system. It tracks information on persistent objects and separates persistence from synchronization using programmer delineated FASEs [16]. A library-based system, NVML requires the programmer to annotate persistent accesses in each FASE.

NVThreads is a REDO-logging, lock-based system that operates at the granularity of pages using OS page protections. Critical sections maintain copies of dirty pages and release them upon lock release [21].

Origin indicates the uninstrumented (and thus crash-vulnerable) code, provided as a baseline for performance comparisons.

Atlas, iDO, and JUSTDO use the LLVM [49] backend. Mnemosyne uses the gcc 4.8 backend due to its reliance on C++ transactions, a feature not yet implemented in LLVM. For all experiments, all runtimes use the same FASEs (but Mnemosyne, as a transactional system, loses concurrency).

For testing, we used an otherwise-idle machine with four AMD Opteron 6276 processors, each of which has 16 single-threaded cores, for a total of 64 hardware threads. Each core has access to private L1 and shared L2 caches (totaling 1 MB per core); the L3 cache (12 MB) is shared across all cores of a single processor. The machine runs CentOS 7.4. In the absence of actual nonvolatile DIMMs, we placed our "persistent" data structures in ordinary

memory (DRAM). We assume that `clflush` instructions followed by an `sfence` roughly approximate the overhead of persistence on machines (e.g., those with Intel's ADR [122]) in which the on-chip memory controller is part of the persistence domain. We explore the sensitivity of our results to this assumption in Section 5.5.5.

5.5.1 Performance Overhead

To understand iDO's performance on real-world benchmarks, we integrated it, along with several other failure atomicity libraries, with **Memcached** [123] and **Redis** [124], two production-quality key-value stores.

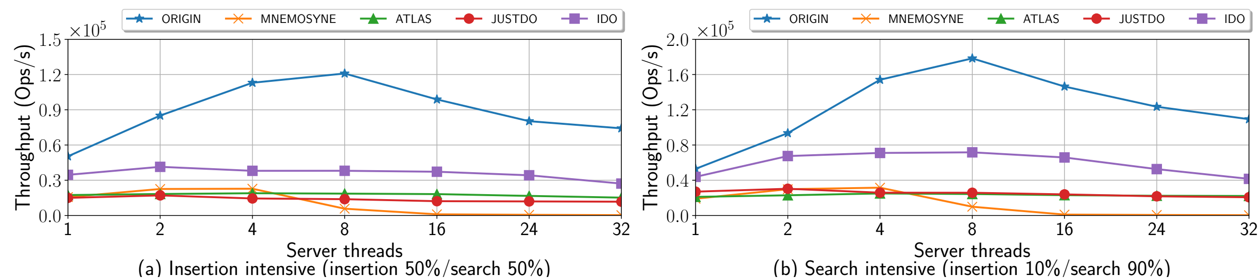


Figure 5.5: Memcached throughput (millions of data structure operations per second) as a function of thread count.

Memcached [123] is used—typically to cache query results—by a wide variety of commercial enterprises, including Facebook, Wikipedia, and Flickr. It has been in active development since 2003. We took advantage of the fact that Mnemosyne was already integrated into an older version of the software (1.2.4) in the WHISPER benchmark [121] and further integrated iDO, Atlas, and JUSTDO into that same, lock-based code. For our experiments, we ran both a Memcached server and client on our AMD Opteron server and followed the methodology of Dice et al. [125] to maximize throughput. We used the tool `memaslap` [126] as the

client to generate a stream of Memcached requests according to a desired distribution. We used 32 client threads, which generated requests with uniformly distributed 16-byte keys and 8-byte values. We experimented with two types of workloads: insertion-intensive (50% insertion / 50% search) and search-intensive (10% insertion / 90% search).

We display throughput in Figure 5.5. In general, iDO logging outperforms all competitors by a factor of two or more. Notably, none of the systems manages to scale particularly well, and even the original version scales only to eight threads. Older versions of Memcached were notorious for exhibiting poor scaling due to coarse-grain locking [125] and the synchronization framework has been reworked since 1.2.4. At its peak, iDO throughput reaches 25–33% of that of the original code, imposing significant but arguably tolerable overhead in return for persistence and crash consistency.

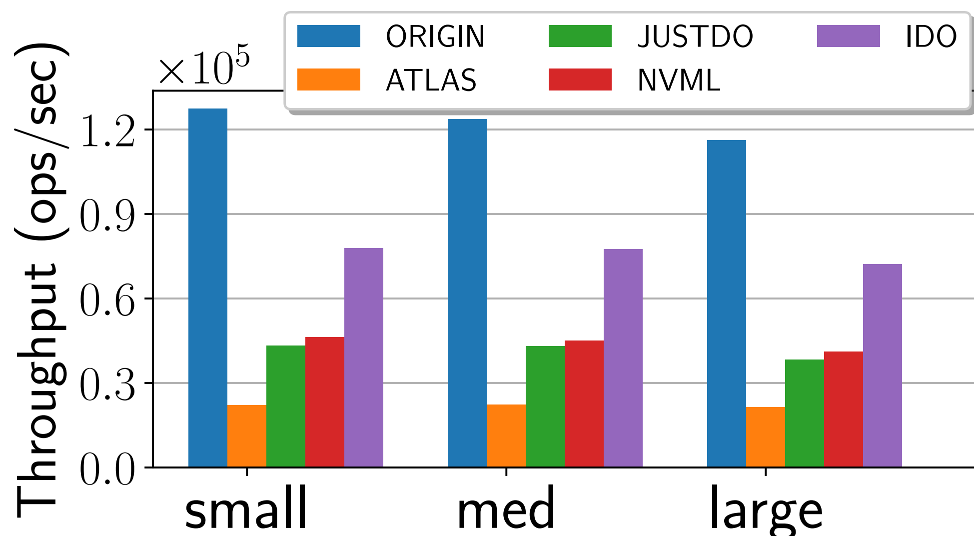


Figure 5.6: Redis throughput for databases with 10K, 100K, and 1M-element key ranges.

Redis [124] is an object-based key-value store that supports a wide variety of data struc-

tures as values. Unlike Memcached, Redis is single threaded, so we relied on programmer-annotated FASEs (rather than outermost locks) to delineate failure-atomic regions. As in our Memcached experiments, we took advantage of the fact that Redis has already been adapted for persistent memory [121]—in this case using NVML. Building on this prior work, we integrated iDO, JUSTDO, and Atlas into the code base. We ran both server and client on our AMD Opteron machine, using Redis's included `lru` test as the client. This client queries the server with a mix of 80% `gets` and 20% `puts`, with a power-law key distribution over fixed key range (10K, 100K, or 1M) for one minute.

As shown in Figure 5.6, iDO outperforms existing persistence systems on Redis by significant margins for all key ranges, with overhead of 30–50% relative to the crash-vulnerable code. As Redis has rather long FASEs with relatively few persistent writes, iDO can take significant advantage of idempotent regions. Notably, as the database grows, the performance difference between iDO and the uninstrumented code shrinks. This effect occurs because the benchmark spends more time searching for keys in the larger database, and iDO logging imposes minimal costs on read paths, since they are idempotent. Also of note is the performance of the two UNDO logging systems—Atlas and NVML. While both provide UNDO logging, NVML has neither compiler integration nor synchronization; programmers must manually annotate every persistent store in a FASE and insert necessary synchronization. Atlas, on the other hand, achieves substantially greater ease of use (for multithreaded code) through compiler-based detection of persistent accesses and automatic tracking of cross-FASE dependences. These additional features in Atlas become performance overheads in a

single-threaded benchmark like Redis.

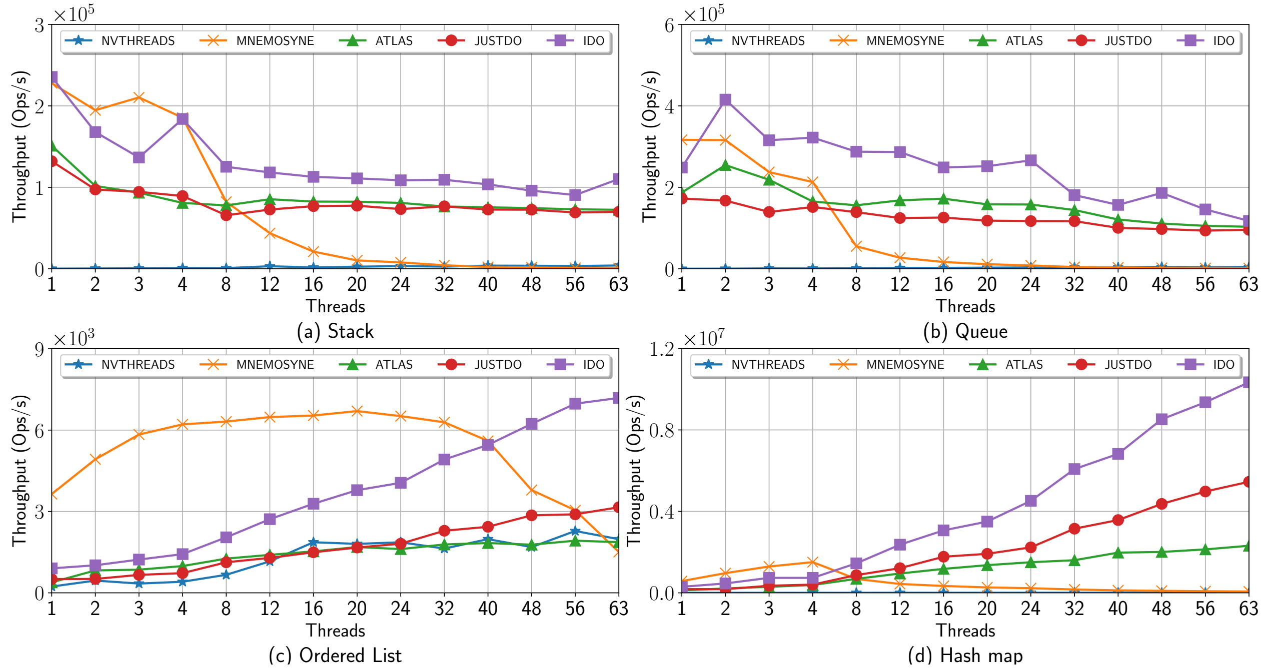


Figure 5.7: Throughput (millions of data structure operations per second) as a function of thread count.

5.5.2 Scalability

For scalability experiments, we used the same data structure microbenchmarks used in the evaluation of JUSTDO logging [22]. These microbenchmarks perform repeated accesses to a shared data structure across a varying number of threads. The data structures we implemented were:

Stack A locking variation on the Treiber Stack [127].

Queue The two-lock queue implementation of Michael and Scott [128].

Ordered List A sorted list traversed using hand-over-hand locking. This implementation allows for concurrent accesses within the list, but threads cannot pass one another.

Map A fixed-size hash map that uses the ordered list implementation for each bucket, obviating the need for per-bucket locks.

These data structures allow varying degrees of parallelism. The stack, for example, serializes accesses in a very small critical section. At the other extreme, the hash map allows concurrent accesses both across and within buckets. We expect low-parallelism data structures to scale poorly with worker thread count whereas high-parallelism data structures should exhibit nearly linear scaling. Our performance results are conservative in that they present the maximum possible stress-test throughput of the structure. In real code, these data structures may not be the overall bottleneck.

At each thread count, tests are run for a fixed time interval using a low overhead hardware timer, and total operations are aggregated at the end. For the duration of microbenchmark execution, each thread repeatedly chooses a random operation to execute on the structure. For our evaluations of the queues and stacks, threads choose randomly between `insert` or `remove`. For the ordered list and hash maps, threads choose randomly between `get` or `put` on a random key within a fixed range. Threads were pinned to cores in a consistent order for all experiments: we entirely fill a single 16-core processor before moving to the next.

During each test, threads synchronize only through the tested data structure. Variables within the data structures are appropriately padded to avoid false sharing. To generate

random numbers, threads use thread-local generators to avoid contention. To smooth performance curves, pages are prefaulted to prevent soft page faults. Performance of the microbenchmarks is up to 10× better without persistence; we elided this result for clarity.

Scalability results appear in Figure 5.7. As in the Memcached and Redis experiments, iDO logging matches or outperforms Atlas in all configurations, especially at higher thread counts. In general, iDO logging also scales better than Mnemosyne, showing near perfect speedup on the hash map. This scaling demonstrates the absolute lack of synchronization between threads in the iDO runtime—all thread synchronization is handled through the locks of the original program. In contrast, both Atlas and Mnemosyne quickly saturate their runtime's synchronization and throttle performance.

The only case in which iDO logging is beaten by Mnemosyne is the ordered list, which uses hand-over-hand locking for traversal. iDO and Atlas support this style of concurrency, but they require ordered writes to persistent memory at every lock acquisition and release in order to track lock ownership. Mnemosyne, as a transaction system, cannot support hand-over-hand locking, so the entire traversal is done in a single transaction and data is written to persistent memory only once. iDO and Atlas extract more concurrency from the benchmark, but per-thread execution is slowed relative to Mnemosyne. Consequently, at very high thread counts, iDO outperforms Mnemosyne due to extracted parallelism, despite its single thread performance being about 4× slower.

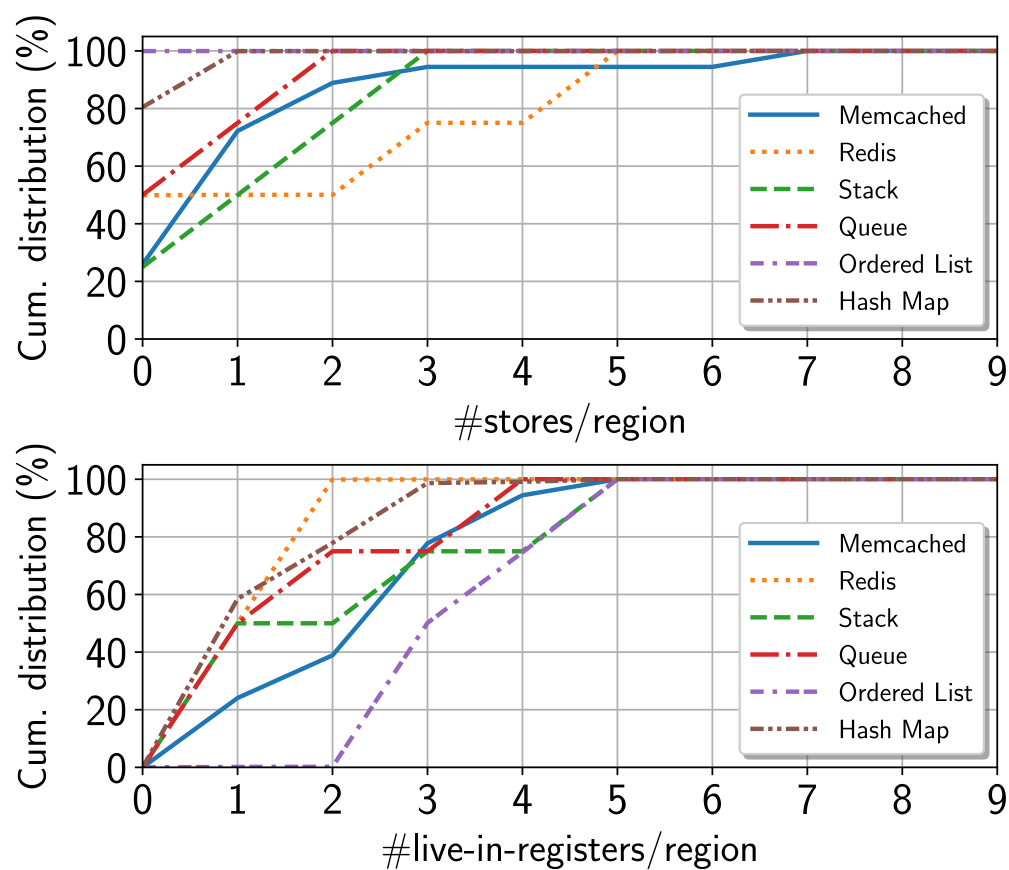


Figure 5.8: Benchmark region characteristics: cumulative distribution of stores (top) and live-in registers (bottom) per dynamic region.

5.5.3 Region Characteristics

To better understand performance differences, we used Intel's Pin tool [129] to collect statistics on idempotent regions. For each of our applications and microbenchmarks, the upper half of Figure 5.8 displays the cumulative dynamic distribution of stores per idempotent region. Any number larger than one indicates a savings in logging operations relative to REDO, UNDO, or JUSTDO logging.

For the microbenchmarks, most regions contain zero or one stores. The Ordered List, in particular, spends much of its time searching the list with hand-over-hand locking, performing no data updates. This can explain why Mnemosyne, which avoids the need to log lock operations, outperforms all other schemes in this microbenchmark for low and moderate thread counts. Note that even with very small regions, iDO still outperforms JUSTDO by substantial margins, largely because of its indirect locking mechanism.

For more realistic applications, we observe that roughly 30% (Memcached) to 50% (Redis) of all regions have multiple stores, allowing iDO to consolidate log operations, leading to higher throughput even at low thread counts. We believe the average region size could be improved with better alias analysis in the compiler. We currently rely on LLVM's basic-AA algorithm, which is quite conservative.

The lower half of Figure 5.8 displays the cumulative dynamic distribution of live-in registers per idempotent region. Significantly, more than 99% of the dynamic regions in all the benchmarks have fewer than five live-in registers, indicating that the typical log operation

Kill Time	1 s	10 s	20 s	30 s	40 s	50 s
Stack	0.7	6.6	14.0	20.7	28.7	34.9
Queue	0.8	9.0	20.1	31.6	43.3	56.1
OrderedList	4.1	72.1	162.2	260.9	301.8	424.8
HashMap	0.3	1.5	2.7	4.2	5.2	6.2

Table 5.1: Recovery time ratio (ATLAS/iDO) at different kill times.

requires only a single cache line flush for the register input.

5.5.4 Recovery Overheads

We evaluate the speed and correctness of recovery by running the microbenchmarks of Section 5.5.2 and killing the process. We interrupt the applications by sending an external SIGKILL signal after the applications have run for 1, 10, 20, 30, 40 and 50 seconds. For the recovery, iDO follows the recovery procedure in Section 5.3.3. As summarized before, iDO needs to first initialize the recovery threads. Then iDO recovers the live-in variables for the interrupted region, jumps back to the entry of the interrupted region, and continues execution until the end of the FASE. Interestingly, the recovery time for iDO with 64 threads is always about one second. Since most of the FASEs in the benchmarks are short (generally on the order of a microsecond), the main overhead for iDO recovery comes from mapping the persistent region into the process's virtual address space and creating the recovery threads, all of which is essentially constant overhead. In contrast, for Atlas, recovery needs to first traverse the logs and compute a global consistent state following the happens-before order recorded in the logs, then undo any stores in the interrupted FASEs.

Table 5.1 shows the ratio of recovery times for ATLAS and iDO. When the applications run for only a short time (1 second) before "crashing," ATLAS can quickly traverse the small number of logs and compute a consistent state, while iDO still has to pay the overhead of creating and initializing recovery threads. However, when the applications run for a longer time (> 10 seconds), ATLAS must traverse a much larger number of logs and compute a consistent state. We can observe up to 400× faster recovery for iDO in this case.

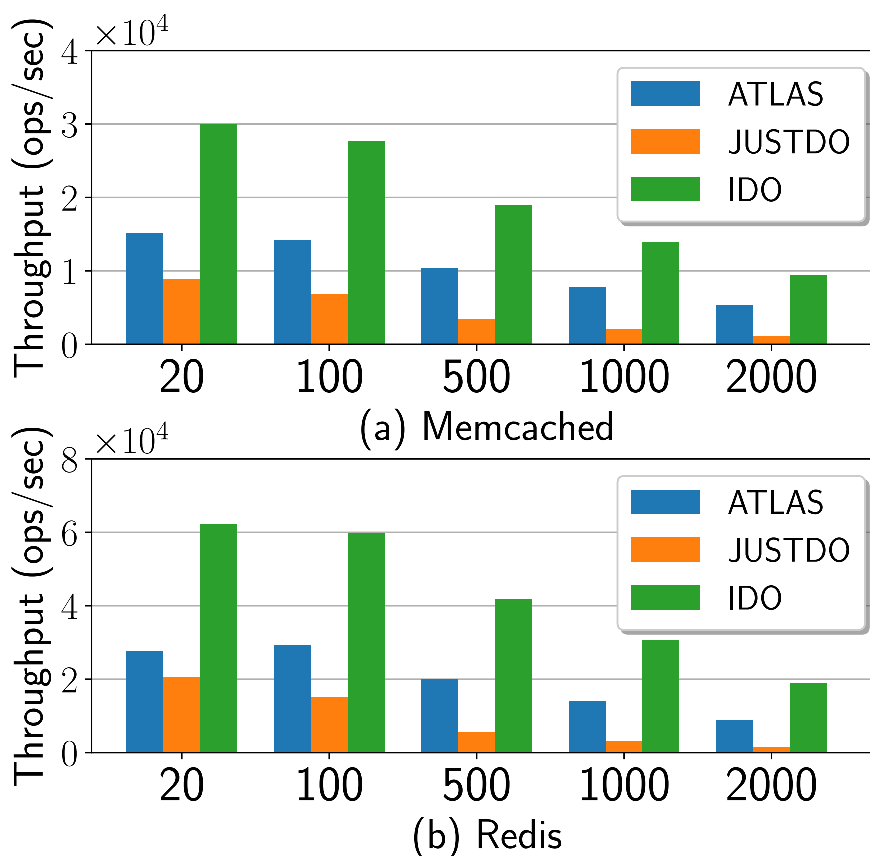


Figure 5.9: Sensitivity to different NVM latency (ns).

5.5.5 Sensitivity to NVM Latency

In the experiments of Sections 5.5.1 through 5.5.4, we relied on `sfence` instructions to capture the cost of waiting for previous writes-back (`clflushes`) to reach the on-chip memory controller. Given that some machines may implement this controller with, say, STT-MRAM instead of capacitor-backed SRAM, while others may require a handshake across the memory bus, we re-ran our Memcached and Redis experiments with additional delays to emulate the cost of nonvolatile writes (which are typically more expensive than reads) or of traversing a long data path. Specifically, we augmented our compiler to insert a configurable delay (looping with `nops`) after each `sfence` that follows one or more persistent stores. We also leveraged a similar capability in Atlas. Results, for delays ranging from 20–2000 ns, appear in Figure 5.9. The Memcached result reprises the 32-server, insertion-intensive data point of Figure 5.5(a); the Redis result reprises the "large" data point of Figure 5.6.

Both iDO logging and Atlas maintain their performance up to a delay of around 100 ns; beyond this point, significant slowdown occurs. JUSTDO logging, by contrast, sees significant (1.5–2×) slowdown relative to Figures 5.5 and 5.6 with even 20 ns of addition `pfence` delay. We attribute this difference to the relatively frequent logging of JUSTDO relative to Atlas and iDO. While very preliminary, we take these results both as a strong endorsement of ADR and as a suggestion that it may be reasonable, *with an appropriate runtime*, to replace capacitor-backed SRAM with physically nonvolatile memory in an ADR memory controller.

5.6 Related Work

5.6.1 Nonvolatile Memory

With the impending end of DRAM scaling, several device technologies are competing to provide inexpensive, dense, and nonvolatile storage in the hopes of becoming the next dominant main memory technology. Candidates include PCM [130, 131], Memristors [132], and spin-transfer torque magnetic memory (STT-MRAM) [133]. In building these technologies, researchers are attempting to maximize density, endurance, economy, and speed, resulting in various compromises across these variables.

At the level of the microarchitecture, architects are trying to give programmers fast and fine-grained control over the ordering and timing of writes-back from volatile caches into nonvolatile main memory; the semantics surrounding this ordering comprise the *memory persistency model* [27] analogous to traditional memory consistency [134]. While existing processors provide rough control over write back (e.g., Intel's `clflush` and nontemporal stores), future designs may track thread-local orderings and buffering to reduce the penalty of ordering writes into persistence. Various schemes and their hardware include epoch persistence [130], buffered epoch persistence [27, 28], explicit epoch persistence [135], DPO [136], and HOPS [121]. Intel has also released ISA extensions for its persistency model [112] including the new `clwb` and `clflushopt` instructions.

On top of these persistency models, several research groups have built high performance software for persistent applications. Example projects include concurrent data struc-

tures [137, 138, 139, 140] transactional key-value stores [141, 142, 19, 143], file systems [117], and databases [144, 145, 146].

Table 5.2: Failure Atomic Systems and their Properties

System	Failure-atomic region semantics	Recovery Method	Logging Granularity	Dependency tracking needed?	Designed for transient caches?
iDO Logging	Lock-inferred FASE	Resumption	Idempotent Region	No	Yes
Atlas[20]	Lock-inferred FASE	UNDO	Store	Yes	Yes
Mnemosyne[18]	C++ Transactions	REDO	Store	No	Yes
NVThreads[21]	Lock-inferred FASE	REDO	Page	Yes	Yes
JUSTDO[22]	Lock-inferred FASE	Resumption	Store	No	No
NVHeaps[15]	Transactions	UNDO	Object	No	Yes
NVML[147]	Programmer Delineated	UNDO	Object	No	Yes
SoftWrAP[148]	Programmer Delineated	REDO	Contiguous data blocks	No	Yes

In contrast with these high-performance and specialized applications, a growing body of work, of which iDO logging is a member, addresses run-time libraries and compiler support to enable programmers to more easily write crash-resistant code. Table 5.2 summarizes the differences among several of these systems. Mnemosyne [18], NV-Heaps [15], SoftWrAP [148], and NVML [147] extend transactional memory to provide durability guarantees for ACID transactions on nonvolatile memory. Mnemosyne emphasizes performance; its use of REDO logs postpones the need to flush data to persistence until a transaction commits. SoftWrAP, also a REDO system, uses shadow paging and Intel's now deprecated `pcommit` instruction [112] to efficiently batch updates from DRAM to NVM. NV-heaps, an UNDO log system, emphasizes programmer convenience, providing garbage collection and strong type checking to help avoid pitfalls unique to persistence—e.g., pointers to transient data inadvertently stored in persistent memory. NVML, Intel's persistent memory transaction system, uses UNDO logging on persistent objects and implements several highly optimized procedures that bypass transactional tracking for common functions.

Other failure atomic run-time systems [20, 22, 21], like iDO logging, use locks for synchronization and delineate failure atomic regions as outermost critical sections, as discussed in Section 5.1.

Extensions to several of these systems explore how to compose operations on concurrent persistent data structures into larger failure atomic sections, thereby eliminating fine-grained write tracking within the persistent data structure. For data structures that meet *detectable execution* [149], query-based logging [150] allows UNDO and JUSTDO based systems to support this optimized composition (analogous to "boosting" in software transactional memory [151]). It seems clear that similar optimizations could work in iDO logging.

All of these specialized persistent applications and runtimes can be seen as nonvolatile memory analogues of traditional failure atomic systems for disk/flash, and they borrow many techniques from the literature. Disk-based database systems have traditionally used write-ahead logging to ensure consistent recoverability [152]. Transactional file updates have been explored in research prototypes [153, 154] and commercial implementations [155]. User-space implementations of persistent heaps supporting failure-atomic updates have been explored in research [156]. Logging-based systems have historically ensured consistency by discarding changes from any update interrupted by failure (even in the REDO case, an update will not be completed on recovery unless it recorded everything it wanted to do before the failure occurred). In contrast, for idempotent updates, an update cut short by failure can simply be re-executed rather than discarding changes, reducing required logging [157, 30].

5.6.2 Idempotence

Over the years, many researchers have leveraged *idempotence* for various purposes. Mahlke et al. were the first to exploit the idea, which they used to recover from exceptions during speculative execution in a VLIW processor [29]. Around the same time, Bershad et al. proposed *restartable atomic sequences* for a uniprocessor based on idempotence [30].

Kim et al. leveraged idempotence to reduce the hardware storage required to buffer data in their compiler-assisted speculative execution model [31]. Hampton et al. used idempotence to support fast and precise exceptions in a vector processor with virtual memory [158]. Tseng et al. used idempotent regions for data-triggered thread execution [159].

Recently, researchers have leveraged idempotence for recovery from soft errors—e.g., ECC faults [32, 33]. Also, Liu et al. [41] advanced the state of the art with *checkpoint pruning*, which serves to remove logging operations that can be reconstructed from other logs in the event of a soft run-time error. Liu et al. [82, 42, 81] also extend the original idempotent processing in the context of sensor-based soft error detectors to ensure complete recovery.

More recently, the energy-harvesting system community has started using idempotent processing to recover from the frequent power failures that occur in systems without batteries. Xie et al. [160] use idempotence-based recovery and heuristics to approximate minimal checkpoints (logs) to survive power failures. Their design revolves around the idea of severing anti-dependences by placing a checkpoint between a load-store pair, in a manner reminiscent of Feng et al. [32] and de Kruijf et al. [33]. Lately, their techniques were used by Woude et

al. [161] to highlight both the promise and the limitations of using idempotence to ensure forward progress when multiple power failures occur within a span of microseconds. In a similar vein, Liu et al. [43] highlight the limitations of anti-dependence based idempotence analysis in terms of additional power consumption due to unnecessary checkpoints. Significantly, all of these projects target embedded processors in which out-of-order execution and caches do not exist.

Despite this wealth of related work, iDO is, to the best of our knowledge, the first system to use idempotence to achieve lightweight, fault-tolerant execution of failure-atomic sections in general-purpose programs.

5.7 Summary

Fault tolerance is one of the most exciting applications of emerging nonvolatile memory technologies. Existing approaches to persistence, however, suffer from problems with both performance and usability. Transactional approaches are generally incompatible with existing lock-based code, and tend not to scale to high levels of concurrency. Failure-atomic regions (FASEs), by contrast, are compatible with most common locking idioms and introduce no new barriers to scalability. Unfortunately, prior FASE-based approaches to persistence incur significant run-time overhead.

To address these limitations, we have introduced iDO logging, a compiler-directed approach to failure atomicity. The iDO compiler divides each FASE into idempotent regions, arranging on failure recovery to restart any interrupted idempotent region and execute forward to the

end of the FASE. Unlike systems based on UNDO or REDO logging, iDO avoids the need to log individual program stores, thereby achieving a significant reduction in instrumentation overhead. Across a variety of benchmark applications, iDO significantly outperforms the fastest existing persistent systems during normal execution, even on machines with conventional volatile caches, while preserving very fast recovery times.

Chapter 6

Conclusions and Future Work

In this chapter, we first conclude the dissertation with a summary of the contributions of this dissertation. Then, we discuss the limitations in this dissertation and propose possible future research directions.

6.1 Summary

Error resilience is one of the key research challenges in modern computing systems. This dissertation specifically addresses the challenges of single-event errors, i.e., soft errors and memory inconsistency in the emerging nonvolatile memory systems. This dissertation takes advantages of idempotent processing and provides the challenges and insights of incorporating idempotent processing for resilience against single-event errors. Then, this dissertation proposes a series of techniques leveraging idempotent processing to achieve guaranteed error resilience in a low-cost and efficient manner. We briefly summaries the contributions in different chapters as follows.

In Chapter 2, We propose Clover, a compiler directed soft error detection and recovery scheme for lightweight soft error resilience. Clover leverages the recent advance of acoustic sensor based soft error detection for low-cost error detection and idempotent processing for low-cost error recovery. However, simply combining idempotent recovery and acoustic sensor based detection cannot guarantee the complete error resilience. In order to achieve guaranteed error resilience, Clover further proposes tail-DMR (dual modular redundancy) to contain the error within the region. The experiment results demonstrate that the average runtime overhead is only 26%, which is a 75% reduction compared to the traditional soft error resilience technique that leverages idempotent processing.

In Chapter 3, We introduce Turnstile, a hardware/software cooperative technique for soft error resilience which redefines the idempotent processing from the processors point of view to contain the errors within the core reducing the runtime overhead of Clover. We leverage a gated store queue and a new logic called region boundary buffer to buffer and verify the program state with regard to each region boundary. Therefore, region-level error containment is relaxed to core-level containment. That is, unlike the traditional idempotent processing, Turnstile does not require the error to be detected within the faulty region where it occurs, thereby avoiding the tail-DMR overhead incurred by Clover.

In Chapter 4, We further develop Bolt, a compiler-directed soft error recovery scheme that further optimizes idempotent processing in the context of soft errors. Bolt proposes two effective compiler optimization techniques, i.e., eager checkpointing to eliminate the need of the expensive hardware support and checkpoint pruning to significantly reduce the check-

point overhead. The beauty of this approach is that it shifts the runtime overhead of the soft-error free execution to that of the error recovery execution without compromising the recovery guarantee.

In Chapter 5, We present *iDO*, a compiler-directed approach to failure atomicity with non-volatile memory leveraging idempotent processing. Unlike most prior work, which instruments each store of persistent data for redo or undo logging, the iDO compiler leverages idempotent recovery thus eliminating the need to log every persistent store. In case of an error, iDO can back up each thread to the beginning of the interrupted idempotent region and re-execute to the end of the interrupted failure-atomic section. As a result, iDO significantly outperforms state-of-the art persistence mechanisms on current hardware during normal execution, while preserving very fast recovery times.

We believe the proposed techniques in this dissertation can lay the foundations for future reliability research.

6.2 Limitations and Future Work

This section discusses the current limitations in this dissertation and proposes future work to enhance our proposed techniques.

First, idempotent processing can be improved in many ways. For example, improving the size of the idempotent region can be beneficial as the overhead per region will be amortized along with the increased region size. Since idempotent processing requires alias analysis for

region partitioning, a better alias analysis can significantly improve the size of the regions. Thus, it would be an interesting future research direction to investigate how the different alias analyses impact our proposed techniques.

Besides, a hybrid approach for preserving the memory input can be a possible topic to further improve the size of region and possibly achieve the better performance. The compiler can checkpoint the memory input, that is to be overwritten because of a WAR dependence, rather than forcing a region boundary to break the dependence. It is important to note that there is a tradeoff between forcing a region boundary and checkpointing the memory input. A trade-off exists between forcing a region boundary and checkpointing the memory input. We believe that a good heuristic can find the sweet spot and further reduce the overhead of idempotent processing.

In addition, while we only explored the failure atomicity system with software transactional memory, it might also be a valuable future direction to incorporate the hardware transactional memory system into the state-of-the-art failure atomic system for nonvolatile memory.

Moreover, novel hardware design can be further proposed to improve the performance of our proposed techniques. For example, we can leverage a hardware support for delegating the persistence ordering among the different regions without the current cacheline flush and fence instructions. Thus, the resulting system will be able to achieve the lightweight persistent stores without breaking the memory consistency.

We believe that our proposed techniques are applicable to many other applications that

require the error resilience. For example, in the energy harvesting systems that suffer from frequent power failures, leveraging our proposed technique, i.e., restarting the interrupted region in the wake of power outage, can save the harvested energy that would otherwise have been wasted by restarting the program from the beginning.

Bibliography

[1] Y. Luo, S. Govindan, B. Sharma, M. Santaniello, J. Meza, A. Kansal, J. Liu, B. Khessib, K. Vaid, and O. Mutlu, "Characterizing application memory error vulnerability to optimize datacenter cost via heterogeneous-reliability memory," in *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pp. 467–478, IEEE, 2014.

[2] X. Li, M. C. Huang, K. Shen, and L. Chu, "A realistic evaluation of memory hardware errors and software system susceptibility," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, 2010.

[3] G. P. Saggese, N. J. Wang, Z. Kalbarczyk, S. J. Patel, and R. K. Iyer, "An experimental study of soft errors in microprocessors.," *IEEE Micro*, vol. 25, no. 6, pp. 30–39, 2005.

[4] C. Constantinescu, "Trends and challenges in vlsi circuit reliability," *Microarchitecture, 2003. MICRO-36. Proceedings. 36nd Annual International Symposium on*, vol. 23, July 2003.

[5] I. S. Haque and V. S. Pande, "Hard data on soft errors: A large-scale assessment of real-world error rates in gpgpu," in *Proceedings of the 2010 10th IEEE/ACM International*

Conference on Cluster, Cloud and Grid Computing, CCGRID '10, pp. 691–696, 2010.

[6] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, "The soft error problem: An architectural perspective," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pp. 243–247, 2005.

[7] L. Wang and K. Skadron, "Implications of the power wall: Dim cores and reconfigurable logic," *IEEE Micro*, pp. 40–48, 2013.

[8] M. B. Taylor, "Is dark silicon useful?: Harnessing the four horsemen of the coming dark silicon apocalypse," in *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pp. 1131–1136, 2012.

[9] M. Shafique, S. Garg, J. Henkel, and D. Marculescu, "The eda challenges in the dark silicon era: Temperature, reliability, and variability perspectives," in *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, DAC '14, pp. 185:1–185:6, 2014.

[10] J. Henkel, L. Bauer, N. Dutt, P. Gupta, S. Nassif, M. Shafique, M. Tahoori, and N. Wehn, "Reliable on-chip systems in the nano-era: Lessons learnt and future trends," in *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pp. 99:1–99:10, 2013.

[11] H. Kaul, M. Anders, S. Hsu, A. Agarwal, R. Krishnamurthy, and S. Borkar, "Near-threshold voltage (ntv) design-ppportunities and challenges," in *DAC*, pp. 1153–1158, ACM.

[12] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: Software implemented fault tolerance," in *Proceedings of the international symposium on Code generation and optimization*, pp. 243–254, IEEE Computer Society, 2005.

[13] J. Chang, G. A. Reis, and D. I. August, "Automatic instruction-level software-only recovery," in *International Conference on Dependable Systems and Networks (DSN'06)*, pp. 83–92, June 2006.

[14] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 200–209, 1962.

[15] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," in *16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, (Newport Beach, CA), pp. 105–118, 2011.

[16] Intel, "Intel NVM Library." `https://github.com/pmem/nvml/`.

[17] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, "DudeTM: Building durable transactions with decoupling for persistent memory," in *22nd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, (Xi'an, China), pp. 329–343, 2017.

[18] H. Volos, A. J. Tack, and M. M. Swift, “Mnemosyne: Lightweight persistent memory,” in *16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, (Newport Beach, CA), pp. 91–104, 2011.

[19] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, “High-performance transactions for persistent memories,” in *21st Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, (Atlanta, GA), pp. 399–411, 2016.

[20] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, “Atlas: Leveraging locks for non-volatile memory consistency,” in *Intl. Conf. on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, (Portland, OR), pp. 433–452, 2014.

[21] T. C.-H. Hsu, H. Bruegner, I. Roy, K. Keeton, and P. Eugster, “NVthreads: Practical persistence for multi-threaded applications,” in *12th ACM European Systems Conf. (EuroSys)*, (Belgrade, Republic of Serbia), pp. 468–482, 2017.

[22] J. Izraelevitz, T. Kelly, and A. Kolli, “Failure-atomic persistent memory updates via JUSTDO logging,” in *21st Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, (Atlanta, GA), pp. 427–442, 2016.

[23] J. Carretero, P. Chaparro, X. Vera, J. Abella, and A. Gonzlez, “End-to-end register data-flow continuous self-test.,” in *ISCA* (S. W. Keckler and L. A. Barroso, eds.), pp. 105–115, ACM, 2009.

[24] P. Racunas, K. Constantinides, S. Manne, and S. S. Mukherjee, "Perturbation-based fault screening," in *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pp. 169–180, IEEE, 2007.

[25] T. M. Austin, "Diva: A reliable substrate for deep submicron microarchitecture design," in *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*, pp. 196–207, IEEE, 1999.

[26] A. Meixner, M. E. Bauer, and D. J. Sorin, "Argus: Low-cost, comprehensive error detection in simple cores," in *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pp. 210–222, IEEE, 2007.

[27] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *41st Intl. Symp. on Computer Architecuture (ISCA)*, (Minneapolis, MN), pp. 265–276, 2014.

[28] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Efficient persist barriers for multicores," in *48th Intl. Symp. on Microarchitecture (MICRO)*, (Waikiki, HI), pp. 660–671, 2015.

[29] S. A. Mahlke, W. Y. Chen, W.-m. W. Hwu, B. R. Rau, and M. S. Schlansker, "Sentinel scheduling for vliw and superscalar processors," in *5th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, (Boston, MA), pp. 238–247, 1992.

[30] B. N. Bershad, D. D. Redell, and J. R. Ellis, "Fast mutual exclusion for uniprocessors," in *5th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, (Boston, MA), pp. 223–233, 1992.

[31] S. W. Kim, C.-L. Ooi, R. Eigenmann, B. Falsafi, and T. N. Vijaykumar, "Exploiting reference idempotency to reduce speculative storage overflow," *ACM Trans. on Programming Languages and Systems (TOPLAS)*, vol. 28, pp. 942–965, Sep. 2006.

[32] S. Feng, S. Gupta, A. Ansari, S. A. Mahlke, and D. I. August, "Encore: low-cost, fine-grained transient fault recovery," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 398–409, ACM, 2011.

[33] M. A. de Kruijf, K. Sankaralingam, and S. Jha, "Static analysis and compiler design for idempotent processing," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pp. 475–486, 2012.

[34] N. J. Wang and S. J. Patel, "Restore: Symptom-based soft error detection in microprocessors," *Dependable and Secure Computing, IEEE Transactions on*, vol. 3, no. 3, pp. 188–201, 2006.

[35] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: probabilistic soft error reliability on the cheap," in *ACM SIGARCH Computer Architecture News*, vol. 38, pp. 385–396, ACM, 2010.

[36] G. Upasani, X. Vera, and A. Gonzalez, "Avoiding core's due & sdc via acoustic wave detectors and tailored error containment and recovery.," in *ISCA*, pp. 37–48, 2014.

[37] G. Upasani, X. Vera, and A. Gonzalez, "Framework for economical error recovery in embedded cores," in *On-Line Testing Symposium (IOLTS), 2014 IEEE 20th International*, pp. 146–153, IEEE, 2014.

[38] M. de Kruijf and K. Sankaralingam, "Idempotent code generation: Implementation, analysis, and evaluation," in *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pp. 1–12, IEEE, 2013.

[39] ARM, "Cortex-a57 technique reference manual," 2015.

[40] D. S. Khudia and S. Mahlke, "Low cost control flow protection using abstract control signatures," in *Proceedings of the 14th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '13, (New York, NY, USA), 2013.

[41] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Compiler-directed lightweight checkpointing for fine-grained guaranteed soft error recovery," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2016.

[42] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Low-cost soft error resilience with unified data verification and fine-grained recovery," in *Proceedings of the 49th International Symposium on Microarchitecture (MICRO)*, Oct 2016.

[43] Q. Liu and C. Jung, "Lightweight hardware support for transparent consistency-aware checkpointing in intermittent energy-harvesting systems," in *Proceedings of the IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, 2016.

[44] G. A. Reis, J. Chang, N. Vachharajani, S. S. Mukherjee, R. Rangan, and D. August, "Design and evaluation of hybrid fault-detection systems," in *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on*, pp. 148–159, IEEE, 2005.

[45] M. Zhou, X. Shen, Y. Gao, and G. Yiu, "Space-efficient multi-versioning for input-adaptive feedback-driven program optimizations," in *Proceedings of the 29th International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pp. 763–776, ACM, 2014.

[46] C. Jung, D. Lim, J. Lee, and S. Han, "Adaptive execution techniques for SMT multiprocessor architectures," in *PPoPP'05*, pp. 236–246, 2005.

[47] J. Lee, J.-H. Park, H. Kim, C. Jung, D. Lim, and S. Han, "Adaptive execution techniques of parallel programs for multiprocessors," *J. Parallel Distrib. Comput.*, vol. 70, pp. 467–480, May 2010.

[48] C. H. Jung, D. S. Lim, J. J. Lee, and S. Y. Han, "Adaptive execution method for multithreaded processor-based parallel system," 2009. US Patent No. 7,526,637.

[49] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Genera-*

tion and Optimization, CGO '04, (Washington, DC, USA), pp. 75–, IEEE Computer Society, 2004.

[50] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems," in *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pp. 330–335, IEEE Computer Society, 1997.

[51] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pp. 3–14, IEEE, 2001.

[52] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.

[53] E. Rotenberg, "Ar-smt: A microarchitectural approach to fault tolerance in microprocessors," in *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*, pp. 84–91, IEEE, 1999.

[54] D. Khudia and S. Mahlke, "Harnessing soft computations for low-budget fault tolerance," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pp. 319–330, Dec 2014.

[55] J. Cong and K. Gururaj, "Assuring application-level correctness against soft errors," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 150–157, IEEE Press, 2011.

[56] 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 SIGPLAN Notices*, vol. 47, pp. 123–134, ACM, 2012.

[57] X. Li, S. V. Adve, P. Bose, J. Rivers, *et al.*, "Online estimation of architectural vulnerability factor for soft errors," in *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*, pp. 341–352, IEEE, 2008.

[58] S. K. S. Hari, S. V. Adve, and H. Naeimi, "Low-cost program-level detectors for reducing silent data corruptions," in *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pp. 1–12, IEEE, 2012.

[59] S. K. Sahoo, M.-L. Li, P. Ramachandran, S. V. Adve, V. S. Adve, and Y. Zhou, "Using likely program invariants to detect hardware errors," in *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pp. 70–79, IEEE, 2008.

[60] S. K. Sastry Hari, M.-L. Li, P. Ramachandran, B. Choi, and S. V. Adve, "mswat: low-cost hardware fault detection and diagnosis for multicore systems," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 122–132, ACM, 2009.

[61] M. Shafique, S. Rehman, P. V. Aceituno, and J. Henkel, "Exploiting program-level masking and error propagation for constrained reliability optimization," in *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, (New York, NY, USA), pp. 17:1–17:9, ACM, 2013.

[62] J. Li, J. Xue, X. Xie, Q. Wan, Q. Tan, and L. Tan, "Epipe: A low-cost fault-tolerance technique considering wcet constraints," *J. Syst. Archit.*, vol. 59, pp. 1383–1393, Nov. 2013.

[63] S. Rehman, K.-H. Chen, F. Kriebel, A. Toma, M. Shafique, J.-J. Chen, and J. Henkel, "Cross-layer software dependability on unreliable hardware," *Computers, IEEE Transactions on*, vol. 65, pp. 80–94, Jan 2016.

[64] S. Rehman, F. Kriebel, M. Shafique, and J. Henkel, "Reliability-driven software transformations for unreliable hardware," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 33, pp. 1597–1610, Nov 2014.

[65] S. Rehman, F. Kriebel, D. Sun, M. Shafique, and J. Henkel, "dtune: Leveraging reliable code generation for adaptive dependability tuning under process variation and aging-induced effects," in *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, (New York, NY, USA), pp. 84:1–84:6, ACM, 2014.

[66] D. S. Khudia, G. Wright, and S. Mahlke, "Efficient soft error protection for commodity embedded microprocessors using profile information," in *ACM SIGPLAN Notices*, vol. 47, pp. 99–108, ACM, 2012.

[67] S. Rehman, M. Shafique, F. Kriebel, and J. Henkel, “Reliable software for unreliable hardware: embedded code generation aiming at reliability.,” in *CODES+ISSS* (R. P. Dick and J. Madsen, eds.), pp. 237–246, ACM, 2011.

[68] H. Chen and C. Yang, “Boosting efficiency of fault detection and recovery throughapplication-specific comparison and checkpointing,” in *Proceedings of the 14th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES ’13, pp. 13–20, ACM, 2013.

[69] C. Jung, S. Lee, E. Raman, and S. Pande, “Automated memory leak detection for production use,” in *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[70] S. Lee, C. Jung, and S. Pande, “Detecting memory leaks through introspective dynamic behavior modelling using machine learning,” in *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[71] C. Jung, *Effective techniques for understanding and improving data structure usage.* Ph.D. Dissertation, Georgia Institute of Technology, Atlanta, GA, 2013.

[72] J. Kim, M. Sullivan, and M. Erez, “Bamboo ecc: Strong, safe, and flexible codes for reliable computer memory,” in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pp. 101–112, IEEE, 2015.

[73] D. W. Kim and M. Erez, "Balancing reliability, cost, and performance tradeoffs with freefault," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pp. 439–450, IEEE, 2015.

[74] D. H. Yoon and M. Erez, "Virtualized and flexible ecc for main memory," in *ACM SIGARCH Computer Architecture News*, vol. 38, pp. 397–408, ACM, 2010.

[75] J. Chung, I. Lee, M. Sullivan, J. H. Ryoo, D. W. Kim, D. H. Yoon, L. Kaplan, and M. Erez, "Containment domains: A scalable, efficient and flexible resilience scheme for exascale systems," *Scientific Programming*, vol. 21, no. 3-4, pp. 197–212, 2013.

[76] G. Upasani, X. Vera, and A. Gonzalez, "Setting an error detection infrastructure with low cost acoustic wave detectors.," in *ISCA*, pp. 333–343, 2012.

[77] G. Upasani, X. Vera, and A. Gonzalez, "Reducing due-fit of caches by exploiting acoustic wave detectors for error recovery.," in *IOLTS*, pp. 85–91, 2013.

[78] G. Upasani, X. Vera, and A. Gonzlez, "A case for acoustic wave detectors for soft-errors," *IEEE Transactions on Computers*, vol. 65, pp. 5–18, Jan 2016.

[79] R. Jeyapaul, A. Risheekesan, A. Shrivastava, and K. Lee, "Unsync-cmp: Multicore cmp architecture for energy efficient soft error reliability," *Transactions on Parallel and Distributed Systems*, vol. 25, pp. 254–263, January 2014.

[80] X. Li and D. Yeung, "Application-level correctness and its impact on fault tolerance," in *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pp. 181–192, IEEE, 2007.

[81] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Clover: Compiler directed lightweight soft error resilience," in *LCTES*, (New York, NY, USA), pp. 2:1–2:10, ACM, 2015.

[82] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Compiler directed soft error detection and recovery to avoid due and sdc via tail-dmr," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. XX, no. X, 2016.

[83] M. J. Wing and G. P. D'Souza, "Gated store buffer for an advanced microprocessor," *U.S. Patent 6,011,908 A*, January 2000.

[84] M. S. Gupta, K. K. Rangan, M. D. Smith, G.-Y. Wei, and D. Brooks, "Decor: A delayed commit and rollback mechanism for handling inductive noise in processors," in *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pp. 381–392, Feb 2008.

[85] S. Muchnick, *Advanced Compiler Design Implementation.* Morgan Kaufmann Publishers, 1997.

[86] C. Blundell, M. M. Martin, and T. F. Wenisch, "Invisifence: Performance-transparent memory ordering in conventional multiprocessors," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pp. 233–244, 2009.

[87] W. Jang, *Soft-error tolerant quasi delay-insensitive circuits.* PhD thesis, Pasadena, CA, USA, 2011.

[88] N. DeBardeleben, S. Blanchard, V. Sridharan, S. Gurumurthi, J. Stearley, K. Ferreira, and J. Shalf, "Extra bits on sram and dram errorsmore data from the field," in *IEEE Workshop on Silicon Errors in Logic-System Effects (SELSE)*, 2014.

[89] J. Wadden, A. Lyashevsky, S. Gurumurthi, V. Sridharan, and K. Skadron, "Real-world design and evaluation of compiler-managed gpu redundant multithreading," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, (Piscataway, NJ, USA), pp. 73–84, IEEE Press, 2014.

[90] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[91] Q. Liu, X. Wu, L. Kittinger, M. Levy, and C. Jung, "Benchprime: Effective building of a hybrid benchmark suite," *ACM Trans. Embed. Comput. Syst.*, vol. 16, pp. 179:1–179:22, Sept. 2017.

[92] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The splash-2 programs: characterization and methodological considerations," in *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*, pp. 24–36, June 1995.

[93] D. Sorin, M. Martin, M. Hill, and D. Wood, "Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery," in *ISCA*, pp. 123–134, 2002.

[94] R. Jeyapaul, F. Hong, A. Rhisheekesan, A. Shrivastava, and K. Lee, “Unsync-cmp: Multicore cmp architecture for energy-efficient soft-error reliability,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 1, pp. 254–263, 2014.

[95] P. Ramachandran, S. K. S. Hari, M. Li, and S. V. Adve, “Hardware fault recovery for i/o intensive applications,” *ACM Trans. Archit. Code Optim.*, vol. 11, pp. 33:1–33:25, Oct. 2014.

[96] T. Slegel, I. Averill, R.M., M. Check, B. Giamei, B. Krumm, C. Krygowski, W. Li, J. Liptay, J. MacDougall, T. McPherson, J. Navarro, E. Schwarz, K. Shum, and C. Webb, “Ibm’s s/390 g5 microprocessor design,” *Micro, IEEE*, vol. 19, pp. 12–23, Mar 1999.

[97] D. Tiwari, S. Gupta, and S. S. Vazhkudai, “Lazy checkpointing: Exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems,” in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pp. 25–36, June 2014.

[98] G. Gupta, S. Sridharan, and G. S. Sohi, “Globally precise-restartable execution of parallel programs,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’14, pp. 181–192, 2014.

[99] W. Zhang, M. de Kruijf, A. Li, S. Lu, and K. Sankaralingam, “Conair: Featherweight concurrency bug recovery via single-threaded idempotent execution,” in *Proceedings*

of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, pp. 113–126, 2013.

[100] D. A. Popescu, E.-D. Tirsa, M. I. Andreica, and V. Cristea, "An application-assisted checkpoint-restart mechanism for java applications.," in *International Symposium on Parallel and Distributed Computing (ISPDC)* (N. Tapus, D. Grigoras, R. Potolea, and F. Pop, eds.), pp. 190–197, IEEE, 2013.

[101] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An architectural framework for software recovery of hardware faults," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, (New York, NY, USA), pp. 497–508, ACM, 2010.

[102] ARM, "Developer suite," 2003. Version 1.2.

[103] G. Memik, M. T. Kandemir, and O. Ozturk, "Increasing register file immunity to transient errors.," in *DATE*, pp. 586–591, 2005.

[104] D. H. Yoon and M. Erez, "Memory mapped ecc: Low-cost error protection for last level caches," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pp. 116–127, 2009.

[105] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[106] Intel., “Xeon e7 processor - ras servers white paper.”

[107] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, “The transmeta code morphing™ software: Using speculation, recovery, and adaptive retranslation to address real-life challenges,” in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, pp. 15–24, 2003.

[108] M. Weiser, “Program slicing,” in *Proceedings of the 5th International Conference on Software Engineering*, ICSE ’81, (Piscataway, NJ, USA), pp. 439–449, IEEE Press, 1981.

[109] S. Horwitz, T. Reps, and D. Binkley, “Interprocedural slicing using dependence graphs,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 1, pp. 26–60, 1990.

[110] M. Wolfe, “Beyond induction variables,” in *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI ’92, (New York, NY, USA), pp. 162–174, ACM, 1992.

[111] M. Snir, R. W. Wisniewski, J. A. Abraham, V. Adve, S. Bagchi, P. Balaji, J. Belak, F. C. P. Bose, B. Carlson, A. A. Chien, P. Coteus, N. A. Debardeieben, P. Diniz, M. E. C. Engelmann, S. Fazzari, A. Geist, R. Gupta, F. Johnson, Krishnamoorthy, S. Leyffer, T. M. D. Liberty, Mitra, R. Schreiber, J. Stearley, and E. V. Hensbergen,

"Addressing failures in exascale computing," *"International Journal of High Performance Computing Applications"*, vol. 28, no. 2, 2014.

[112] Intel Corporation, "Intel architecture instruction set extensions programming reference," Tech. Rep. 3319433-029, Intel Corporation, Apr. 2017.

[113] O. Polychroniou, A. Raghavan, and K. A. Ross, "Rethinking SIMD vectorization for in-memory databases," in *Intl. Conf. on Management of Data (SIGMOD)*, (Melbourne, Victoria, Australia), pp. 1493–1508, 2015.

[114] T. Willhalm, I. Oukid, I. Müller, and F. Faerber, "Vectorizing database column scans with complex predicates.," in *Intl. Wkshp. on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS@VLDB)*, (Riva del Garda, Trento, Italy), pp. 1–12, 2013.

[115] K. Keeton, "The machine: An architecture for memory-centric computing," in *5th Intl. Wkshp. on Runtime and Operating Systems for Supercomputers (ROSS)*, (Portland, OR), p. 1, 2015.

[116] W. Wei, D. Jiang, J. Xiong, and M. Chen, "Exploring opportunities for non-volatile memories in big data applications," in *Wkshp. on Big Data Benchmarks, Performance Optimization, and Emerging Hardware*, pp. 209–220, Springer, 2014.

[117] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory," in *22nd Symp. on Operating Systems Principles (SOSP)*, (Big Sky, MT), pp. 133–146, 2009.

[118] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm, "Makalu: Fast recoverable allocation of non-volatile memory," in *Intl. Conf. on Object-Oriented Programming, Systems, Languages, & Applications (OOPSLA)*, (Amsterdam, The Netherlands), pp. 677–694, 2016.

[119] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Computing Surveys*, vol. 15, pp. 287–317, Dec. 1983.

[120] M. Wong, V. Luchangco, *et al.*, "SG5 transactional memory support for C++," Oct. 2014. Document number N4180, Programming Language C++, Evolution Working Group, Intl. Organization for Standardization.

[121] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with WHISPER," in *22nd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, (Xi'an, China), pp. 135–148, 2017.

[122] A. M. Rudoff, "Deprecating the pcommit instruction." software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction, Sept. 2016.

[123] memcached.org, "memcached – a distributed memory object caching system." http://memcached.org/. Accessed: 2017.

[124] RedisLabs, "Redis," 2015. http://redis.io.

[125] D. Dice, V. J. Marathe, and N. Shavit, "Lock cohorting: A general technique for designing NUMA locks," *ACM Trans. on Parallel Computing*, vol. 1, no. 2, p. 13, 2015.

[126] libMemcached.org, "libMemcached." http://www.libMemcached.org, 2011.

[127] R. K. Treiber, "Systems programming: Coping with parallelism," Tech. Rep. RJ 5118, IBM Almaden Research Center, Apr. 1986.

[128] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *1996 ACM Symp. on Principles of Distributed Computing (PODC)*, (Philadelphia, PA), pp. 267–275, 1996.

[129] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *26th Conf. on Programming Language Design and Implementation (PLDI)*, (Chicago, IL), pp. 190–200, 2005.

[130] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable DRAM alternative," in *36th Intl. Symp. on Computer Architecture (ISCA)*, (Austin, TX), pp. 2–13, 2009.

[131] G. W. Burr, M. J. Breitwisch, M. Franceschini, D. Garetto, K. Gopalakrishnan, B. Jackson, B. Kurdi, C. Lam, L. A. Lastras, A. Padilla, B. Rajendran, S. Raoux, and R. S. Shenoy, "Phase change memory technology," *Journal of Vacuum Science and Technology*, vol. 28, no. 2, pp. 223–262, 2010.

[132] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," *Nature*, vol. 453, pp. 80–83, May 2008.

[133] D. Apalkov, A. Khvalkovskiy, S. Watta, V. Nikitin, X. Tang, D. Lottis, K. Moon, X. Luo, E. Chen, A. Ong, A. Driskill-Smith, and M. Krounbi, "Spin-transfer torque magnetic random access memory (STT-MRAM)," in *ACM Journal on Emerging Technologies in Computing Systems (JETC)—Special issue on memory technologies*, pp. 13:1–13:35, 2013.

[134] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *Computer*, vol. 29, pp. 66–76, Dec. 1996.

[135] J. Izraelevitz, H. Mendes, and M. L. Scott, "Linearizability of persistent memory objects under a full-system-crash failure model," in *30th Intl. Conf. on Distributed Computing (DISC)*, (Paris, France), pp. 313–327, 2016.

[136] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, "Delegated persist ordering," in *49th Intl. Symp. on Microarchitecture (MICRO)*, (Taipei, Taiwan), pp. 1–13, 2016.

[137] D. Schwalb, M. Dreseler, M. Uflacker, and H. Plattner, "NVC-hashmap: A persistent and concurrent hashmap for non-volatile memories," in *3rd VLDB Wkshp. on In-Memory Data Mangement and Analytics (IMDM)*, (Kohala Coast, HI), pp. 4:1–4:8, 2015.

[138] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner, "FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory," in *Intl. Conf. on Management of Data (SIGMOD)*, (San Francisco, CA), pp. 371–386, 2016.

[139] S. Chen and Q. Jin, "Persistent B+-trees in non-volatile main memory," *Proc. of the VLDB Endowment*, vol. 8, pp. 786–797, Feb. 2015.

[140] F. Nawab, J. Izraelevitz, T. Kelly, C. B. Morrey, D. Chakrabarti, and M. L. Scott, "Dalí: A periodically persistent hash map," in *31st Intl. Symp. on Distributed Computing*, (Vienna, Austria), 2017.

[141] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *9th USENIX Conf. on File and Stroage Technologies (FAST)*, (San Jose, CA), p. 5, 2011.

[142] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "NV-tree: Reducing consistency cost for NVM-based single level systems," in *13th USENIX Conf. on File and Storage Technologies (FAST)*, (Santa Clara, CA), pp. 167–181, 2015.

[143] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh, "Wort: Write optimal radix tree for persistent memory storage systems," in *15th USENIX Conf. on File and Storage Technologies (FAST)*, (Santa Clara, CA), pp. 257–270, 2017.

[144] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge, "Storage management in the NVRAM era," *Proc. of the VLDB Endowment*, vol. 7, pp. 121–132, Oct. 2013.

[145] T. Wang and R. Johnson, "Scalable logging through emerging non-volatile memory," *Proc. of the VLDB Endowment*, vol. 7, pp. 865–876, June 2014.

[146] J. DeBrabant, J. Arulraj, A. Pavlo, M. Stonebraker, S. Zdonik, and S. Dulloor, "A prolegomenon on OLTP database systems for non-volatile memory," in *Intl. Wkshp. on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS@VLDB)*, (Hangzhou, China), pp. 57–63, 2014.

[147] A. Rudoff, "Persistent memory programming." http://pmem.io/. Accessed: 2017-04-21.

[148] E. R. Giles, K. Doshi, and P. Varman, "Softwrap: A lightweight framework for transactional support of storage class memory," in *31st Symp. on Massive Storage Systems and Technology (MSST)*, (Santa Clara, CA), pp. 1–14, 2015.

[149] M. Friedman, M. Herlihy, V. Marathe, and E. Petrank, "A persistent lock-free queue for non-volatile memory," in *23rd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, (Vienna, Austria), 2018.

[150] J. Izraelevitz, "Concurrency implications of nonvolatile byte-addressable memory," 2018. Ph.D. Thesis.

[151] M. Herlihy and E. Koskinen, "Transactional boosting: A methodology for highly-concurrent transactional objects," in *13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, (Salt Lake City, UT, USA), 2008.

[152] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Trans. on Database Systems (TODS)*, vol. 17, pp. 94–162, Mar. 1992.

[153] R. P. Spillane, S. Gaikwad, M. Chinni, E. Zadok, and C. P. Wright, "Enabling transactional file access via lightweight kernel extensions," in *7th USENIX Conf. on File and Storage Technologies (FAST)*, (San Francisco, CA), pp. 29–42, 2009.

[154] S. Park, T. Kelly, and K. Shen, "Failure-atomic `msync()`: A simple and efficient mechanism for preserving the integrity of durable data," in *8th ACM European Systems Conf. (EuroSys)*, (Prague, Czech Republic), pp. 225–238, 2013.

[155] R. Verma, A. A. Mendez, S. Park, S. Mannarswamy, T. Kelly, and C. B. M. III, "Failure-atomic updates of application data in a linux file system," in *13th USENIX Conf. on File and Storage Technologies (FAST)*, (Santa Clara, CA), pp. 203–211, 2015.

[156] S. Yoo, C. Killian, T. Kelly, H. K. Cho, and S. Plite, "Composable reliability for asynchronous systems," in *USENIX Annual Technical Conf. (ATC)*, (Boston, MA), pp. 27–40, 2012.

[157] M. de Kruijf and K. Sankaralingam, "Idempotent processor architecture," in *44th Intl. Symp. on Microarchitecture (MICRO)*, (Porto Alegre, Brazil), pp. 140–151, 2011.

[158] M. Hampton and K. Asanović, "Implementing virtual memory in a vector processor with software restart markers," in *20th Intl. Conf. on Supercomputing (ICS)*, (Cairns, Queensland, Australia), pp. 135–144, 2006.

[159] H.-W. Tseng and D. M. Tullsen, "CDTT: Compiler-generated data-triggered threads," in *20th Intl. Symp. on High Performance Computer Architecture (HPCA)*, (Orlando, FL), pp. 650–661, 2014.

[160] M. Xie, M. Zhao, C. Pan, J. Hu, Y. Liu, and C. Xue, "Fixing the broken time machine: Consistency-aware checkpointing for energy harvesting powered non-volatile processor," in *52nd IEEE/ACM Design Automation Conf. (DAC)*, (San Francisco, CA), pp. 184:1–184:6, 2015.

[161] J. Van Der Woude and M. Hicks, "Intermittent computation without hardware support or programmer intervention," in *USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, (Savannah, GA), pp. 17–32, 2016.