8-18-2017

A Cross-Layer Resilient Multicore Architecture

Qingchuan Shi
University of Connecticut - Storrs, qingchuan.shi@uconn.edu

A Cross-Layer Resilient Multicore Architecture

Qingchuan Shi, Ph.D.

University of Connecticut, 2017

ABSTRACT

The ever-increasing miniaturization of semiconductors has led to important advances in mobile, cloud and network computing. However, it has caused electronic devices to become less reliable and microprocessors more susceptible to transient faults induced by radiations. These intermittent faults do not provoke permanent damage, but may result in incorrect execution of programs by altering signal transfers or stored values. These transitory faults are also called soft errors. As technology scales, researchers and industry pundits are projecting that soft-error problems will become increasingly important. Today's processors implement multicores, featuring diverse set of compute cores and on-board memory sub-systems connected via networks-on-chip and communication protocols. Such multicores are widely deployed in numerous environments for their computational capabilities.

To protect multicores from soft-error perturbations, resiliency schemes have been developed with high coverage but high power and performance overheads. It is observed that not all soft-errors affect program correctness, some soft-errors only affect program accuracy, i.e., the program completes with certain acceptable deviations from error free outcome. Thus, it is practical to improve processor efficiency by trading off resiliency overheads with program accuracy. This thesis explains the idea of declarative resilience that selectively applies resiliency schemes to both crucial and non-crucial code. At the application level, crucial and non-crucial code is identified

based on its impact on the program outcome. A cross-layer architecture is developed, through which hardware collaborates with software support to enable efficient resilience with holistic soft-error coverage. Only program accuracy is compromised in the worst-case scenario of a soft-error strike during non-crucial code execution.

A Cross-Layer Resilient Multicore Architecture

Qingchuan Shi

M.S., Boston University, Boston, MA, United States, 2012

B.S., Purdue University Northwest, Hammond, IN, United States, 2010

A Dissertation

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

at the

University of Connecticut

2017

APPROVAL PAGE

Doctor of Philosophy Dissertation

A Cross-Layer Resilient Multicore Architecture

Presented by

Qingchuan Shi, M.S.

Major Advisor _____
Omer Khan

Associate Advisor _____
John Chandy

Associate Advisor _____
Marten Van Dijk

University of Connecticut

2017

ACKNOWLEDGMENTS

First I would like to express my special appreciation and thanks to my advisor, Prof. Omer Khan. It has been an honor to be one of his first Ph.D. students. I appreciate all his contributions of time, ideas, and funding to make my Ph.D. experience productive and stimulating. The joy and enthusiasm he has for research was contagious and motivational for me. I am really grateful for his support during the tough time in my Ph.D pursuit. I will always remeber the numerous all-nighters he spent with me for the breakthroughs. His guidance on both my research as well as career has been priceless.

I would like to thank my group mates, Farrukh Hijaz, Masab Ahmad, Hamza Omar, and Halit Dogan for all the stimulating discussions we had. They have always been patient to explain the questions I had, and provide feedbacks for many ideas, papers, and presentations.

Last but not least, I would like to thank my family. Words cannot express how grateful I am to my parents and grandparents for all the sacrifices they made. They supported me with their unconditional love throughout my life and made me the person I am today.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Soft-error in Multicore System

Moore's Law has enabled integration of multibillion transistors on a chip. However this has been accompanied by increased process and transistor variations, aging and reliability problems. Today processors need to deliver performance within tight power budgets, which exacerbates reliability concerns. *Soft errors* pose a serious challenge to the availability and reliability of future computing systems. Transient and intermittent perturbations in logic values can result in user-visible effects ranging from complete system failure, application/protocol/hardware level deadlocks, or mundane errors such as software bugs and mis-configurations. These errors cannot be pruned through pre-silicon or post-silicon (manufacturing-time) testing and must be dealt with using runtime resiliency methods. Several studies (e.g., [2]) have surveyed the impact of soft errors on various components of an integrated circuit and concluded that it is no longer just the data stored in memory cells that needs to be protected (e.g., using [3]), but mitigation techniques are needed to limit the impact of soft-errors in logic as well. Let's consider a scenario where a soft error in the logic can cause the

processor to fail. Say the processor pipeline reads an operand from the register file and the data is waiting on the bypass multiplexor. The soft error strike on the multiplexor highly attenuates certain data bits from 0 to 1. The incorrect data flows through the data-path, execution units and finally commits to the register file. Any further use of this stale program state in the register file can produce incorrect program output or lead to an uncorrectable exception.

As technology scales, researchers and industry pundits are projecting that soft-error problems will become increasingly important [4]. Today's processors implement multicores, featuring diverse set of compute cores and on-board memory sub-systems connected via networks-on-chip and communication protocols. Such multicores are widely deployed in numerous environments for their computational capabilities, from traditional applications such as data centers; to emerging areas including unmanned aerial vehicles (UAVs) [5] and self-driving cars [6]. These cyber-physical systems require high resilience for safety-criticality [7, 8], yet high performance for their timing constraints. Applications running on such systems include graph analytics (e.g., path planning, motion detection), computer vision, and artificial intelligence (e.g., machine learning) [9, 10, 11, 12]. The challenge is to prevent the system running such safety-critical algorithms from failure due to soft-errors, while still meet real-time processing constraints.

While extensive research has been done on protecting single core processors from soft-errors, multicore systems introduce new challenges, especially when running parallel applications under complex shared memory protocols. Multicores integrate compute pipelines, cache hierarchy and interconnection networks on a single die, which introduces additional challenges such as heterogeneity, cache coherence, synchronization. Such challenges lead to complex logic interactions and make high soft-error coverage guarantee a hard problem. Moreover, shared memory complicates error detection and recovery mechanisms since data races among cores lead to false alarms, and make it harder to replay the program in a deterministic manner. The error detection-to-recovery latency suffers when many cores rollback and synchronize their redundant execution. Although

these rollbacks may happen rarely, with safety-critical systems' tight real-time constrains in mind, such recovery schemes may not be acceptable. The program execution must produce programmer acceptable result under soft-error perturbations, and meet real-time constraints.

1.2 Protection Schemes

Conventional hardware/software-only resiliency methods have been introduced which provide high soft-error coverage. Software-only resiliency mechanisms incur high performance overheads [13, 14, 15, 16, 17, 18], when compared to a system with no resiliency support. While hardware-only resiliency schemes exhibit less performance overheads compared to software-only schemes, the performance loss is still high (e.g., [19, 20]) along with more power consumption. To improve performance, researchers have explored selective resilience within applications [21, 22]. These schemes obtain efficiency by providing high resiliency for high vulnerability code; however, they tradeoff performance with soft-error coverage.

Resiliency solutions implement *error detection* followed by *system recovery* mechanisms. To deliver high coverage, practically all proposals rely on a checkpoint and rollback mechanism to recover from soft-errors. A Hardware Redundant Execution scheme [19] re-executes instructions at per-core granularity. It uses a resilient cache coherence protocol [23] to protect the on-chip communication (details in Section 3). The cost of high soft-error coverage is performance. With the cyber-physical systems' tight timing constraints in mind, works have been done on accuracy-performance tradeoffs [24, 25, 26]. However to my best knowledge, no prior works have considered resilience, accuracy, and performance, in a holistic way at the architecture level. Performance overhead can be reduced by eliminating unnecessary protection for certain code regions.

Holistic protection schemes for multicore (and GPU) systems have been developed using mech-

	Performance Overhead	Hardware Overhead	Selectively Trading-Off	Multicore	Complier Aid	Coverage			
						Crashing	Deadlock	Livelock	SDC
TLR [20]	High	High	None	Yes	Not Applicable	High	High	High	High
HaRE [19]	Mid	Mid	None	Yes	Not Applicable	High	High	High	High
dTune [27]	Mid	Mid	Vulnerability	Yes	Not Applicable	Mid	Mid	Mid	Mid
RASTER [28]	Mid	Mid	Vulnerability	No	Not Applicable	Mid	Mid	Mid	Mid
Khudia et.al. [29]	Low	None	Accuracy	No	Yes	Low	Low	Low	High
Proposed Scheme	Low	Mid	Accuracy	Yes	Possible	High	High	High	High

Table 1.2.1: Summary of different resilience schemes. "High/Low" in fault type refers to the possibility of protecting system from such faults.

anisms, such as thread-level redundancy (TLR) [20, 16]. These schemes deliver high coverage by performing cross checks in a duplicated thread. However, redundant computation sacrifices available parallelism, which leads to relatively large performance overheads. In general multithreading schemes operate at coarse granularity, and require complex checkpoint and global roll-backs. Thus, they require long detection-to-recovery latency, which is not ideal for systems with real-time constraints. To improve performance, the idea of selectively applying resilience protection in a program has been explored. Research has focused on applying certain n-Modular Redundancy (nMR) or symptom-based schemes selectively to a program [21, 22]. These schemes apply protection based on code's vulnerability, which makes the program less likely to be affected. However, they do not bound soft-error's impact. In similar context, the idea of lowering program accuracy for performance has been explored in approximate computing [30, 31, 32, 33]. These works do not focus on the soft-error resilience perspective. Not all code in an approximated program can tolerate errors, and still requires certain level of resilience protection. Most resent research [29], D.S.Khudia et.al. have explored this idea, and selects the instructions based on their program accuracy impact. Similar to other selective resilience schemes, it only protects certain instructions and leaves others unprotected, and focuses on Silent Data Corruption (SDC). Moreover, it is not specifically designed for multicores, and thus may have lower protection for such systems. These

state of the art schemes are summarized in Table 1.2.1, and contrasted to the proposed scheme along performance, hardware overhead, selective resiliency, and coverage tradeoffs.

1.3 Cross-Layer Resilience

In this thesis, I propose a cross-layer resilience framework, which is referred as *declarative resilience*. It applies different resilience schemes to different code regions based on their criticality. Strong schemes for the code regions (crucial) that affect program correctness; Lightweight schemes for the code regions (non-crucial) where compromising program accuracy is acceptable in case of soft-errors (details in Section 4).

To further motivate the idea, I perform a prospective experiment using a commercial off-the-shelf Intel core-i7 4790 system executing a multi-threaded implementation of a popular machine learning benchmark, AlexNet [34]. Only the input and output layers are considered crucial. They are redundantly executed at thread level (similar to [20]), and verified through software level checksums. All other AlexNet layers are considered non-crucial since they can be protected with lightweight resiliency to ensure program correctness. However, due to soft errors, program accuracy for the non-crucial layers can be compromised. The program analysis (as described in Section 4) shows that in the worst-case scenario the impact of program accuracy is within 7% (at 0.1% error rate). Considering that the above loss of accuracy is acceptable, for the non-crucial code regions, (1) control flow variables (such as the loop index ”i”) are duplicated and checked within each thread; (2) certain data variables are bound checked, and out of range values are discarded. When executed under the this framework, much better performance is achieved compared to thread level redundancy for the whole program. The results show that declarative resilience achieves a $\sim$1.2$\times$ slowdown over the system without resilience protection, however, thread level redundant

implementation is $\sim$4.6$\times$ worse. This setup shows the potential of trading off accuracy with resilience overhead. To guarantee high coverage, I propose novel multicore architectural schemes on top of the proposed framework, and demonstrate the applicability for graph analytics and machine learning workloads.

The following contributions are introduced in this thesis.

(1) I propose a cross-layer resilience framework (declarative resilience), which introduces program accuracy as a new trade-off to achieve efficient resiliency.

(2) I developed a state-of-the-art resilient multicore architecture with high soft-error coverage yet low overhead.

(3) The framework is applied to the emerging application domains of machine learning and graph analytics.

(4) I deployed software/hardware level resilience schemes. Together with the resilience framework, they are able to deliver high soft-error coverage with minimum overhead, while only compromise program accuracy in case of soft-errors.

Chapter 2

Soft-error Effects to Program Control and Data Flow

As technology scales below 22*nm*, researchers and industry pundits are projecting one undetected soft-error in processor logic or memory every day [13]. This failure rate is unacceptable and must be dealt with using resiliency methods. Today's large processors implement multicores, featuring diverse set of compute cores and on-board memory sub-systems connected via networks-on-chip and communication protocols. The key challenge is reducing resilience overhead while providing holistic protection. Redundancy is a well known resiliency concept and computer designers have explored information, time and space redundancy mechanisms from circuit to various system layers. Those mechanisms can achieve high soft-error coverage, but with high overheads. State-of-the-art resiliency approaches for multicores and GPUs perform redundant execution of program instructions [19, 20, 16, 21]. However, high soft-error coverage incurs high performance overheads (e.g., [19] reported $>1.5\times$ slowdown in completion time compared to a multicore with no resiliency). Although the overhead can be reduced, it compromises coverage [21, 13]. I introduce a new tradeoff to achieve *efficient resiliency*, where soft-error coverage is guaranteed and only

program accuracy is allowed to be affected.

The key idea is to protect different code regions with different resilience schemes that tradeoff program accuracy with efficiency. The novelty comes from two factors: the way code regions are identified, and the resilience schemes to ensure no side-effects due to soft-errors. Based on the notion of trading off resilience overheads with program accuracy, crucial and non-crucial code is defined as follows. Crucial code affects program correctness, which means the program should be able to complete without crashing, deadlocking, etc., due to soft-errors, while its outcome is explicable. Non-crucial code only affects program accuracy, which refers to how much the result is off compared to error-free scenario. For example, in an image processing application, the calculation of certain pixel's color value can be considered non-crucial, while the flow control of image stream is crucial for program correctness. Another example is the "repetitive calculation" in Monte Carlo method.

2.1 Control and Data Flow Protection

Soft-errors can cripple a program's execution. In this thesis, I consider soft-errors that impact the program control flow or data flow. The OS (operating system) code is not emulated and I assume that complementary resiliency mechanisms protect the program from soft-errors in the OS. In the proposed declarative resilience, the programmer (or compiler, profiler e.g., a loop perforation compiler [31]) identifies code that can tolerate soft-errors, and classifies them as crucial/non-crucial. For this purpose, the soft-error effects to program control and data flows are evaluated.

The control flow of a program may include branches, loops, jumps and function calls, as shown in Figure 2.1.1. A soft-error can affect the control flow instruction in two ways: wrong target/return address, or wrong branch condition. Wrong target/return address can result in accessing arbitrary

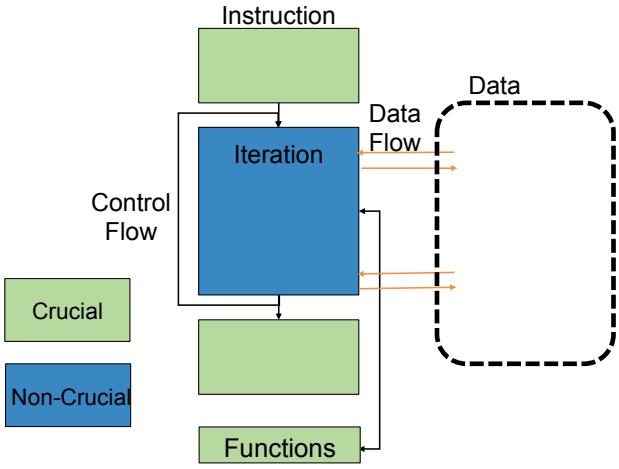

Figure 2.1.1: Original program, which in general is composed of loops.

code and cause unpredictable effects, as shown in Figure 2.1.2: ①. Thus I claim that control flow instructions are also crucial, and must always be protected. Moreover, a conditional control flow instruction may incorrectly calculate its branch condition, thus all such indirect calculations also need to be protected.

The store instructions can access crucial code unexpectedly due to incorrect store address calculation in non-crucial code, as shown in Figure 2.1.2: ②. Thus store addresses must always be protected. For stores with indirect addressing, the programmer (or profiling tool) identifies address calculations that are revealed naturally, such as calculations of the array index. These calculations need to be protected.

The non-crucial code can also affect program outcome when data is committed to memory, as shown in Figure 2.1.2: ③. In order to illuminate the commit process, I divide the data of each non-crucial loop iteration into two categories: *local* and *global*. Local data is only used within the non-crucial code. It is also temporal because it is not used after exiting the non-crucial code, for example, variables defined or initialized within the loop iteration. Data that is consumed outside the non-crucial code is considered global. Local data is accumulated to global data through compu-

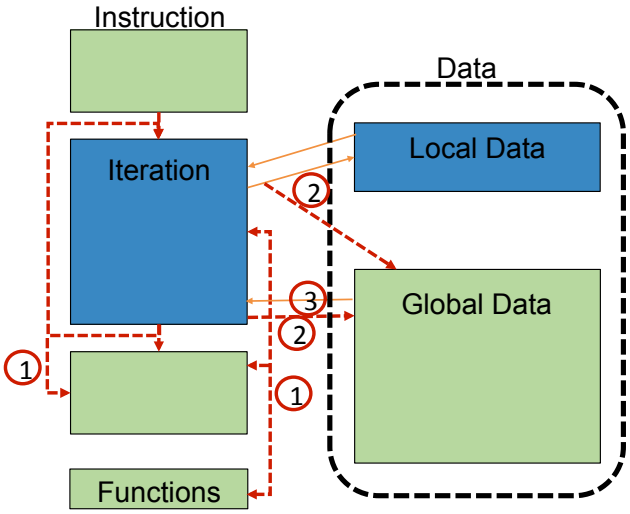

Figure 2.1.2: Instruction within loops can affect the program outcome in three ways: ① Control Flow, ② Store Address, and ③ Data Value.

tations. Thus local data is always non-crucial, and global data can be crucial. Values accumulated to global data are considered critical to the program correctness and must be protected.

2.2 Proof of Concept Prototype

In order to justify the crucial/non-crucial region concept, I consider recent research that has explored approximate programming [30, 31, 32]. Esmaeilzadeh et al. [30] focused on developing a language to generate code that executes on approximate hardware. However, the proposed approach is not limited to approximate hardware and is more general i.e., find code regions of a general program that can be potentially approximated without significantly impacting program outcome. The prototype [35] is inspired from loop perforation [31, 32], which achieves the performance and accuracy tradeoff by reducing the number of iterations in certain program loops. A profiler tool identifies loops that can be perforated with negligible loss in program's accuracy. For

example, **for** (i=0; i<iter; i++) can be changed to **for** (i=0; i<iter; i+=2). The profiler identifies perforable loops. This meet the the program structure shown in Figure 2.1.1. Code outside the perforable loop's iterations is considered crucial, and must be strongly protected. As mentioned earlier, soft-errors introduce new challenges since code in the perforable loops can unexpectedly affect other parts of the program and compromise program correctness. Thus certain resiliency mechanisms are also needed for perforable loops to execute without impacting program correctness.

2.2.1 Hardware Support for Declarative Resilience

In order to deliver high soft-error coverage when necessary, I developed a state-of-the-art hardware resilience mechanism (HaRE) that guarantees high soft-error coverage [19] (details in Section 3). In HaRE, each core re-executes its own atomic instruction sequences, and rollback to safe state when soft-error is detected. For the purpose of deterministic and deadlock-free re-execution, HaRE guarantees atomicity for instruction sequences: modified data is not committed or transferred until control and data flow is checked for soft-errors. It has two main phases: *regular execution*, and *redundant execution*. At the beginning of regular execution, all necessary states such as register file and program counter are duplicated to ensure a safe state checkpoint. During regular execution, memory updates are held in a per core Speculative Store Buffer (SSB), and control and data flow is captured as signatures. During the redundant execution phase, the instruction sequence is re-executed and validated by comparing signatures. If the signatures of two executions mismatch, that core safely rolls back to the beginning of regular execution phase and starts another execution. If signatures are matched correctly, memory updates are bulk-committed from the SSB. A resilient cache coherence protocol [23] is used to enable holistic coverage for computation and the communication hardware.

A key feature of HaRE is that one core's execution and re-execution does not affect other cores. For example, if core-1 sends a coherence message to core-2, whether core-2 is in regular execution, redundant execution, or about to start redundant execution, core-1's execution will be oblivious to core-2's status. The only difference is delay in the coherence reply message. This ensures that HaRE can be independently turned on/off at each core, without introducing further complexity. A software-hardware interface is developed on top of HaRE to make it capable of setting re-execution on/off based on program's demand. The program turns HaRE on/off using a set of special instructions, and a reserved re-execution status bit in each core. When the status bit is cleared, the corresponding core initiates re-execution of the instruction sequence from its last checkpoint and then performs resilience check. At that point, the core bypasses HaRE mechanism, until it is turned back on.

2.2.2 Declarative Resilience Implementation

The following schemes are implemented to protect the program:

1. As shown in Figure 2.2.1: ①, all crucial instructions are protected through HaRE, which includes all instructions outside perforable loops, and the control flow instructions within them. Some branch conditions are conspicuous, such as the "i" variable in **for** (i=0; i<iter; i++). When HaRE is enabled around the "for" instruction at program level, calculations of "i" are automatically protected. For other conditional branches, I rely on the profiling tool to identify whether the code that resolves the branch target is within the non-crucial code. For such instructions, wrong branch conditions can be tolerated since they do not affect the crucial code. The non-crucial code selected by the profiler only has branch conditions that affect local computation. In the worst case, wrong branch conditions only lead to unacceptable local data values, which is discussed later. Since there are no control flow instructions when

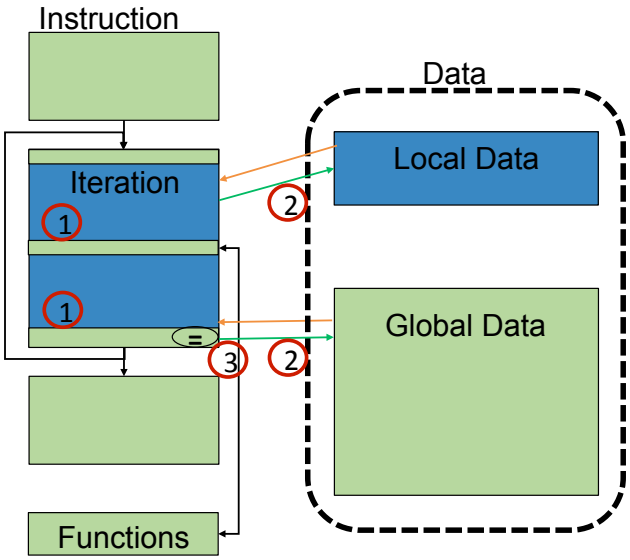

Figure 2.2.1: Overall protections: ① Crucial Code: HaRE, ② Store Address in Non-crucial Code: redundant address calculation (SHR), ③ Data Value in Non-crucial Code: software checker (SHR).

HaRE is off, an exception is triggered for any unexpected change in the program counter.

2. As shown in Figure 2.2.1: ②, store instructions are checked by hardware-level redundant address calculation with additional overhead. This is part of SHR and is only used when HaRE is off. Store operations proceed after their address calculation is verified.

3. Data Value: With the above protection schemes, the selected non-crucial code's control and data flow can only affect the program outcome through commit to global data. As shown in Figure 2.2.1: ③, the data value is checked for its bound at software-level, before committing to global store. This is also part of SHR, and used only for committing global data in non-crucial code. For numerical data types, which is the only case I had in this thesis, the bound is determined offline based on the program's functionality and runtime statistics. Arising from loop perforation, it is acceptable to drop the results of certain iterations. Thus data is dropped if it fails the checker. The checker is implemented in software, and HaRE is turned

on before and turned off after the checker is executed.

2.2.3 Application Illustration

I instrument four PARSEC [36] applications using the perforation profiling tool, and selectively turn HaRE ”off” for the non-crucial code within perforable loops. Moreover, I make the following software modifications to non-crucial code.

`Streamcluster`: Declarative resilience is set within **dist()** function, which computes euclidean distance between two points. From the loop perforation tool, its ”for loop” iteration is heavily used and does not have any indirect stores or control flow instructions. The HaRE is turned ”off” when an iteration is invoked, and turned back ”on” after the distance calculation. Furthermore, the global variable *result* is checked through software checker code (SHR) and dropped if it is out of a statically determined bound. In order to reduce the checker overhead, the compiler unrolls the loop. Instead of checking the result of every iteration, the accumulated result of ten iterations is checked and committed before next loop iteration.

`Swaptions`: Function **HJM_SimPath_Forward_Blocking()** is selected, which computes and stores an HJM Path for a given trial. HaRE is turned on before all the control flow instructions such as for loops and function calls. A data checker (SHR) is applied to “dDiscSwaptionPayoff”, since it is the final result of each iteration.

`Canneal`: Function **calculate_delta_routing_cost()** is selected and calculation is mainly done in **swap_cost()**. HaRE on/off is set around the two for loops in the swap_cost function. Since the result of calculate_delta_routing_cost falls in one of the ”movement” categories, there is no need for data checker. In the worst case, the classification of one ”movement” could be wrong, but it only affects the accuracy.

`Blackscholes`: The calculation of selected loop is mainly done in **BlkSchlsEqEuroNoDiv()**.

HaRE on/off is set around the ”if” branches, function calls, and same for inside the CNDF() function. For the calculated result, ”price” value is bound checked (SHR) for each iteration.

2.2.4 Methods

I use the `Graphite` multicore simulator [37] to model the prototype. The default architecture parameters are summarized in Table 2.2.1. I model all hardware resiliency mechanisms proposed in [19]. The hardware-software interface for declarative resilience is implemented using a special function instrumented in the application, which passes the HaRE on/off pragma to the simulator. With the in-order single-issue core setup, I assume a five-stage pipeline. The HaRE ”on” switch needs 3 cycles to create a safe state and start capturing signatures. For HaRE ”off” switch, the pipeline needs to be flushed with additional 1 cycle delay to re-execute and check the previous instruction sequence. Redundant store address calculation (SHR) incurs one cycle latency since it needs to stall the compute pipeline. It is only enabled when HaRE is ”off”, and uses existing HaRE hardware to check the results. The data checker (SHR) is implemented in the application using regular instructions, so it adds to the instruction footprint but incurs no additional hardware overhead.

2.2.5 Evaluation

Figure 2.2.2 shows the completion time of several resilient schemes normalized to a baseline that does not support resiliency. The ”Re-exe” scheme (HaRE) that re-executes all instructions has an average of 1.75$\times$ performance overhead. The synchronization time increases due to the re-execution time of instructions within locks or barriers. Memory stall time also increases, because memory operations that trigger re-executions (such as invalidating a cache line), need to wait until

Architectural Parameter	Value
Core	64 In-Order, Single-Issue cores
Memory Subsystem	
L1-I/D Private Caches per Core	32 KB, 4-way Set Assoc., 1 cycle
L2 Shared Cache per Core	256 KB, 8-way Set Assoc., 8 cycles, Inclusive
Coherence Protocol	Directory Invalidation-based, MESI
DRAM Memory Interface	8 controllers, 5 GBps/controller, 100 ns latency
Electrical 2-D Mesh with XY Routing	
Hop Latency	2 cycles (1-router, 1-link) + link contention
Flit Width	64 bits

Table 2.2.1: Architectural parameters.

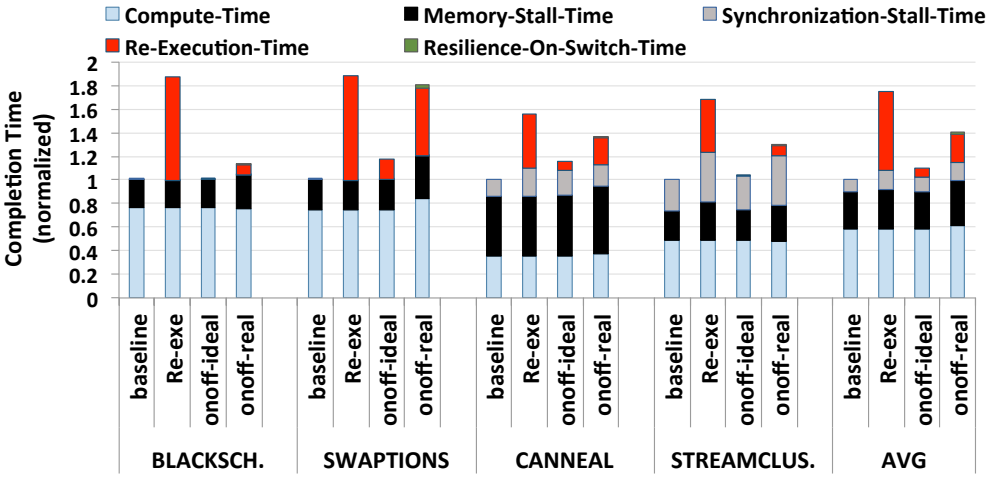

Figure 2.2.2: The breakdown of completion time. The "baseline" is without any soft-error protection scheme. "Re-exe" uses HaRE for all program code. "onoff-ideal" only applies HaRE for non-perforable crucial code, leaving perforable loops unprotected. "onoff-real" is the proposed declarative resilience scheme.

the re-execution completes.

To demonstrate the efficacy of the profiling tool, ”onoff-ideal” only applies HaRE to the classified non-perforable crucial code, and the control and data flow within the perforable loops is not protected by HaRE or SHR. The performance overhead is reduced significantly to on average 1.1×. Thus it is clear that perforable code is executed heavily in the benchmarks, and have great contribution to the resilience overhead. This certifies the importance of the profiling process, and also serves as the upper limit for declarative resilience.

The proposed declarative resilience scheme also needs to protect all the crucial code, and against the side effects of non-crucial code that can potentially propagate to crucial code. The ”onoff-real” scheme introduces the following overheads on top of ”onoff-ideal”: control flow protection (HaRE) in perforable loops, redundant store address checking (SHR), and data checkers (SHR) for non-crucial code. On average the performance overhead of ”onoff-real” is 1.38× of the baseline. This translates to a 21% performance improvement compared to state-of-the-art scheme that uses redundant execution for the entire execution of an application. As shown in Figure 2.2.2's ”Resilience-On-Switch-Time”, the resilience on switching only contributes slightly to the overall delay, which is on average 0.89% of the total completion time. Since resilience off switch would trigger re-execution, it is accounted within ”Re-Execution-Time”.

The performance advantage of declarative resilience varies for each benchmark. For example, I observe 1.81× overhead for `swaptions`, whereas, the upper limit as shown by the “onoff-ideal” scheme is 1.17×. The overhead is mainly affected by the following factors: the number of control flow instructions, store instructions, and total instructions in the non-crucial code. Because control flow instructions need to be protected, benchmarks with large number of control flow instructions in their non-crucial code would have less benefit from the scheme. I observe that all benchmarks except `swaptions` have >80 instructions per resilience ”off” sections of code. This allows more non-crucial code to execute in between turning resilience on and off. On the other

hand, `swaptions` has <20 instructions per resilience "off" sections. The reason is that many instructions in the loop identified by the profiler are control flow instructions, thus HaRE is turned on frequently. Moreover, at the end of each non-crucial code iteration, the data value checker incurs overhead as well. In general, the proposed scheme achieves better performance when the non-crucial code has less control flow and store instructions, and large number of instructions in each section of resilience "off" code.

Finally, I note that declarative resilience does not explicitly perform loop perforation. In the worst-case scenario when data checker (SHR) fails, the non-crucial loop iteration is dropped. Thus, the probability of dropping iterations is less than the soft-error rate. Considering the quality of service study in loop perforation [31, 32], the scheme maintains a high level of program accuracy.

2.3 Summary

With the prototype, I have demonstrated the idea of declarative resilience, which explores resilience overhead tradeoffs with program accuracy, while not compromising soft-error coverage. The preliminary analysis shows 1.38× performance overhead over a system without resilience. This is an average 21% performance improvement over state-of-the-art hardware resilience scheme that always protects the executed code.

The four main research aspects will be further explored in this thesis:

1. *Formulation*: Currently, profiling non-crucial code, applying resilience on/off pragmas, and adding data checkers are all done manually by the programmer. This requires extra work for the programmer when developing a resilient application and is prone to errors. However, the formulation of declarative resilience can be generalized in an unambiguous way.

2. *Applicability*: I have evaluated only four benchmarks. Due to similarity of coding style,

many applications are expected to have heavily executed non-crucial loops, which should benefit from declarative resilience. Thus I plan to extend the work to a wider range of parallel applications.

3. *Generality*: I plan to use an established reliability analysis technique such as fault injection to validate whether the assumptions of crucial and non-crucial code hold when applied in practice.

Chapter 3

Efficient Resilient Multicore with High Soft-error Coverage

In this section, a novel resilience architecture (referred as HaRE) is introduced in detail, which is a fundamental component of declarative resilience.

As mentioned, today's microprocessors implement multicores, featuring diverse set of compute cores and on-board memory sub-systems connected via increasingly complex communication interconnection networks and protocols. Most of the prior research on hardware-level redundancy [38, 39, 40, 41, 42, 43] Shared memory protocols are notoriously complex, and any error in the protocol operation within the compute cores or the communication fabrics can lead to a catastrophic system failure, such as deadlock or program state corruption. Thus, there is a need to develop mechanisms for failure-resilient execution of parallelized applications running on future shared memory multicores. Recent hardware-level resiliency mechanisms for multicores utilize coarse-grain thread-level redundancy (TLR) to detect errors and implement global checkpoint and rollback to recover to a safe state [44, 20]. Although TLR provides adequate soft error coverage (at coarse granularity), it suffers from two key drawbacks: (1) The *loss of parallelism* due to redundant

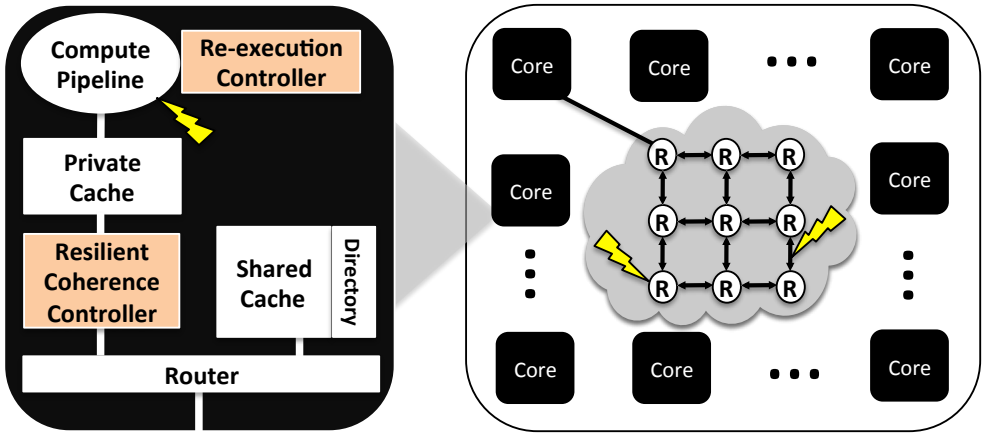

Figure 3.0.1: A *resilient* shared memory multicore: proposed novel *time redundancy* mechanisms (identified by re-execution and resilient coherence controllers) to detect and correct soft errors in the compute cores and the communication fabrics.

execution translates to performance loss in a multithreaded application. This trend will worsen as network latency increases when more cores are added, since the diameter of most on-chip networks increases with the number of cores. (2) For error recovery, TLR implements architectural support to checkpoint and rollback program state so as to provide a deterministic and consistent set of inputs to the rolled program state. Rolling back a core to a previously known good state requires partial undoing of shared memory state in a multicore environment, which is complicated and error prone due to the inherent non-determinism in the shared memory execution model. Additionally, *global checkpointing and rollback* adds to the performance overheads of TLR.

The vision is to develop a holistic resilient multicore architecture that minimizes the implementation overheads of redundant execution and enables a flexible method to tradeoff performance with soft error coverage at runtime. To that effect the novel mechanisms are developed that (1) enable *opportunistic* redundant execution while fully exploiting the multicore parallelism, and (2) do not require global checkpoint and rollback support.

To illustrate the working of the proposed architecture, a multicore is setup (shown in Fig-

ure 3.0.1) where each *core* has a compute pipeline, private L1 and shared L2 caches, and a router for communication. Each core in the chip communicates with others via an on-chip mesh interconnection network. All private caches across the cores are kept coherent using a traditional invalidation-based MESI directory protocol in which the physically distributed directory tracks the sharing information. The directory locations are co-located with the shared L2 cache that is physically distributed across the entire multicore. A resilient multicore is envisioned where cores running multithreaded applications perform autonomous re-execution of instruction(s) safely without deadlocking the shared memory system.

- Similar to TLR, it is assumed that the on-chip caches are protected from soft errors using information redundancy mechanisms such as parity or error correcting codes.

- A novel deadlock-free time redundancy mechanism is developed: each core can autonomously rollback the program state on a distributed per-thread granularity and restart execution based on the *locally available states*. For deadlock-free operation of shared memory, the mechanism allows *deterministic re-execution* of fixed-length instruction sequences in a core as long as no shared data in that sequence is modified by an external agent, such as an invalidation request from another core to acquire ownership of a shared data block.

- *Opportunistic re-execution* of selective instruction sequences is enabled in the compute cores without deadlocking the shared memory system. Selective re-execution (when the hardware is most vulnerable) enables a dynamic architectural knob to control the tradeoffs in performance with soft error coverage.

- The resilient coherence protocol [23] is integrated with the proposed redundancy mechanism to enable a *holistic* soft error tolerant shared memory multicore. For each coherence transaction (initiated via a private cache miss), the resilient coherence protocol guarantees either a

reply is received by the requesting core, or in case of an error ridden coherence transaction, a *timer* expiration at the requesting core initiates a deterministic re-execution of the private cache miss.

It is quantitatively shown that the resilient multicore architecture delivers significant performance advantage over an *idealized* TLR implementation, because it fully exploits multicore parallelism, and it minimizes and hides the latency of redundant execution. The simulation analysis of a 64-core multicore running multithreaded applications shows that an idealized TLR system performs $\sim$1.9$\times$ worse compared to a multicore with no redundant thread execution. In comparison, the proposed method with high soft error coverage is $\sim$1.6$\times$ worse, because it fully exploits multicore parallelism, and it minimizes and hides the latency of redundant execution. When opportunistic redundancy (e.g., using private cache miss as the *only* trigger to initiate re-execution) is enabled, the method delivers 45% soft error coverage at a small 12% performance loss.

3.1 Soft-Error Resilient Multicore

The basic idea is to enable a fine-grain redundant execution mechanism for shared memory multicores without global checkpoint/rollback support and minimal performance loss.

The shared memory programming model requires some form of inter-core communication protocol to keep a consistent view of data among cores. Rolling back a core to a previously known good state requires partial undoing of shared memory state, which is complicated by the inherent non-determinism in the shared memory execution model. Recently, Rashid and Huang [20] proposed a coarse-grain and highly-decoupled thread-level redundant execution framework based on checkpoints. In case of asynchronous progress of redundant cores that could lead to data races, [20] proposed to mask this issue as a transient fault. In the worst-case scenario, not even rollback guar-

antees deadlock freedom in their proposal. So an order tracking mechanism is proposed which enforces the same access pattern in redundant threads. Such mechanism implies determinism by recurrent creation of bulk-synchronous sub-intervals using expensive global synchronizations. In addition to complex and error prone support for global synchronizations, checkpointing and rollback recovery, [20] incurs significant performance degradation due to loss of parallelism of thread-level redundancy.

A unique deadlock-free time redundancy mechanism is proposed; each *core* can autonomously rollback the program state on a distributed per-thread granularity and *restart execution based on the locally available states*. For deadlock-free operation of shared memory, the mechanism allows deterministic re-execution of a fixed-length instruction sequence in a core as long as no shared data in that sequence is modified by an external agent, such as an invalidation request from another core to acquire ownership of a shared data block.

Figure 3.1.1 shows the key mechanisms to enable the proposed soft error resilient multicore architecture. The key mechanisms inside a core are introduced and related with the time varying steps involved in the redundant execution process. There are two key phases to the approach: (1) *START – INITIATE RE-EXEC* captures the necessary states (register file, PC, memory) to ensure a deterministic and deadlock-free replay mechanism when a core is required to re-execute an instruction sequence. (2) *INITIATE RE-EXEC – DONE*, if and when initiated, re-executes the instruction sequence and validates the execution. In case the execution signatures mismatch, the core can be safely rolled back to *START* and deploy a recovery procedure (e.g., another re-execution). Using this two-phase approach, each core in the multicore autonomously enables redundant execution for instruction sequences.

Identifying Instruction Sequence for Re-execution: During the first phase, each core in the multicore autonomously initiates an instruction sequence capture process by saving the register file and program counter contents in a protected (error tolerant) storage structure (shown as ① in

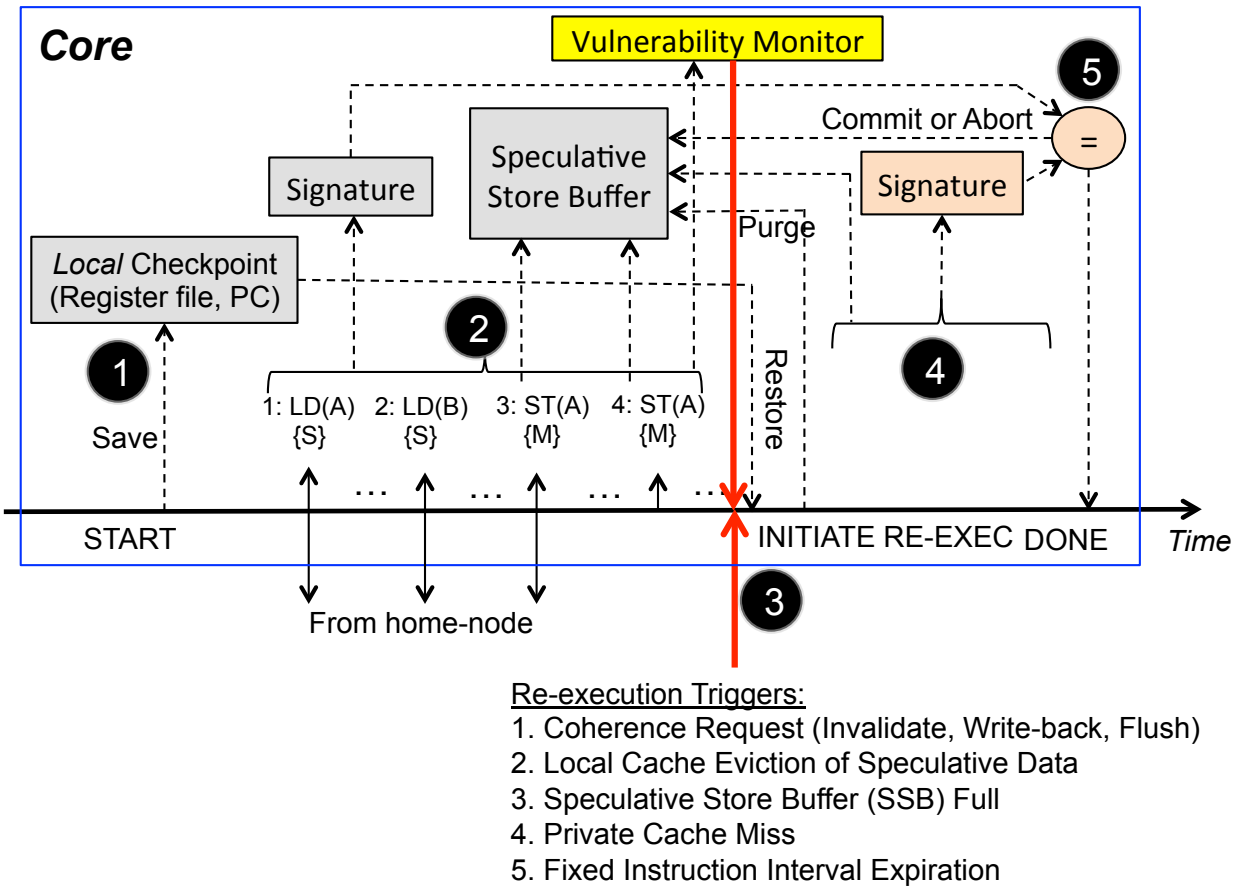

Figure 3.1.1: Proposed mechanism for fine-grain per-core time redundancy in shared memory multicores. Steps ① and ② capture the necessary local states to enable a deterministic and deadlock-free replay mechanism to re-execute the captured instruction sequence. Step ③ determines when to stop capturing an instruction sequence and consults a *vulnerability monitor* to initiate re-execution. Steps ④ and ⑤ are invoked to validate the instruction sequence.

Figure 3.1.1). As instructions execute and commit, a special mechanism is introduced for memory load/store instructions, so that when a data block is modified (and later consumed) by the local instructions, all stores are performed speculatively. For this, I introduce a fully associative Speculative Store Buffer (SSB) that allows the local data cache to only capture the pre-modified loads and the associated coherence states. For illustration, I will use the directory-based MSI invalidation protocol for cache coherence activity between the cores initiating/receiving coherence messages to/from the core with the directory (home-node).

② shows an example sequence of load/store operations on data blocks A and B. When A is loaded for the first time, the directory state S (for *Shared*) is captured in the local cache. A subsequent store on A requires the coherence protocol to transition the directory state to M (for *Modified*) in the local cache, but the actual data modifications on this block are only captured in the SSB. All further memory accesses on A are read and written to the SSB. This mechanism allows the core's private cache to hold data blocks in their pre-modified clean state. To ensure determinism and deadlock freedom of shared memory, I continue execution of instructions and capture their associated states until any of the three *required* conditions arise (shown in ③):

1. A coherence request to *Invalidate*, *Write-back* or *Flush* indicates that another core intends to share or acquire ownership of a locally held data block. To avoid deadlocks (cf. Section 3.2.1), the core stops and initiates re-execution of the instruction sequence.

2. An eviction of a local cache's data block that has been speculatively consumed (via local loads or stores) indicates the loss of determinism in re-executing the instruction sequence. Therefore, I introduce a special bit in the local cache (*speculative bit*) to indicate whether a data block, since *START*, is consumed speculatively, and if so, the core stops and initiates re-execution.

3. When SSB is full since *START*, the core stops and initiates re-execution.

Additionally, two *optional* conditions are included to trigger re-executions: when a fixed instruction interval has lapsed and upon a private cache miss. It is important to note that opportunistic re-execution may choose to re-execute on any of these conditions or simply discard and initiate a new *START*. These two are introduced as microarchitectural knobs to create opportunities for selectively re-executing instruction sequences and/or hiding the re-execution latency. For example, a private cache miss that results in an off-chip memory access may stall the requesting core for 100s of clock cycles; perhaps it will be beneficial to initiate re-execution and hide its latency. Similarly, in applications with extensive communication, a core that results in a coherence miss will potentially need to wait until all the shearers of the requested cache line re-execute their instruction sequences. A trigger that limits the number of instructions to accrue for re-execution may help with minimizing the performance impact in such scenarios.

Deterministic Redundant Execution: During ② (in Figure 3.1.1), the core captures a golden signature (cf. Section 3.2.4) of the executed instructions. When a core initiates the intention to re-execute (③), the *vulnerability monitor* (cf. Section 3.2.5) is consulted to determine whether to re-execute or continue without error detection (i.e., initiate a new *START*). To initiate re-execution, the initial register file and PC state is restored and SSB is purged. The already pre-loaded data in the local caches allow efficient re-execution by exploiting instruction-level parallelism of the compute pipeline (④). When re-execution completes, the new signature is compared to the golden signature. In case of a match, the SSB is bulk-committed and *DONE* indicates success (⑤). This process allows the core to detect and recover from soft errors. It is important to note that only after *DONE* is signaled, an external request (③) is allowed to be serviced by the core.

Redundant Execution of Private Cache Misses: The deadlock-free time redundancy mechanism described above only covers the cores and leaves the communication hardware (the un-core) vulnerable. In shared memory multicores, an unreliable interconnection network can lose or corrupt coherence messages, causing the entire chip to deadlock. Recently, [23] proposed modifications

to the traditional directory coherence protocol to tolerate the loss of coherence messages and argued for a system-level resiliency solution to tolerate an unreliable underlying on-chip network. [23] introduces a systematic methodology to transform a coherence protocol to a resilient one by extending the coherence controller state machines with "safe states" and incorporating additional handshaking messages into the coherence transactions (cf. Section 3.3). This enables a requesting core to safely re-execute its private cache misses.

As shown in Figure 3.0.1, a recovery mechanism is proposed where each private cache miss is protected using a dedicated *timeout counter* at the core that initiates a coherence transaction. In addition, I incorporate the resilient coherence protocol into the shared memory architecture. Upon a fault in the un-core and the eventual timer expiration (because of lost or corrupt messages), the requesting core replays the suspended private cache miss (from its current state to completion) identically as the fault-free scenario, without introducing any protocol-specific synchronization messages. It is important to note that re-execution of the private cache misses can only be initiated during ② when an un-checked instruction sequence is initially executed and data is brought into the core for computation.

3.2 Architectural Details

The proposed redundant execution mechanism for shared memory multicores is fully compatible with the *Total Store Order* (TSO) memory model [45], commonly deployed across many commercial architectures, such as SPARC and x86. TSO defines a total memory order that is consistent with each core's program order, except that stores may be delayed provided a core's loads see its own stores immediately. Figure 3.2.1 shows an example deterministic redundant execution of a single valid TSO interleaving (*bottom left*) from among the several possible alternatives. Two

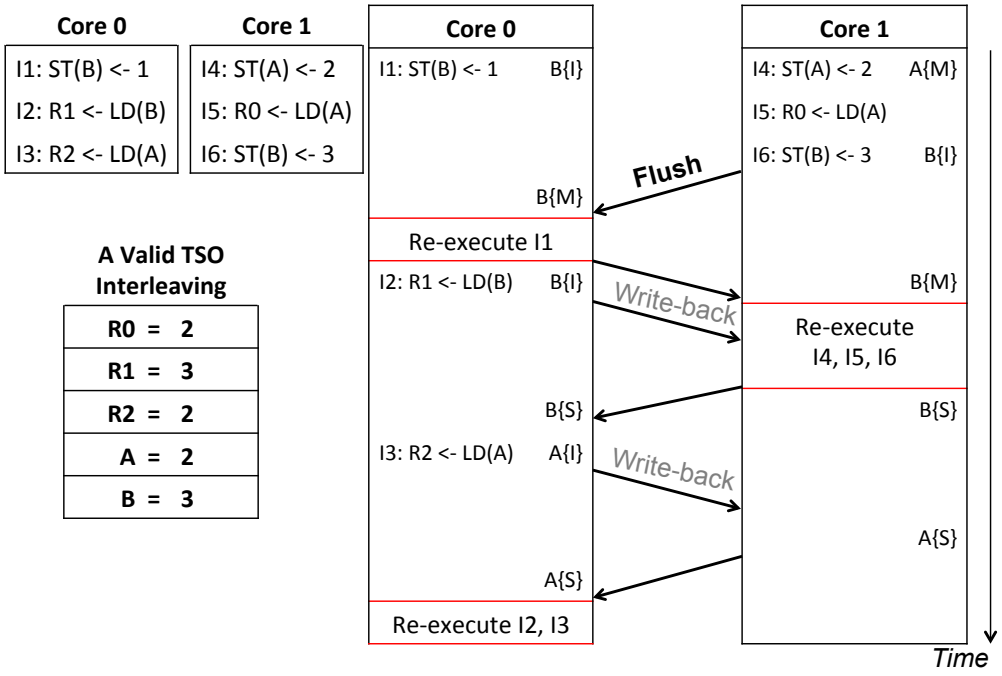

Figure 3.2.1: An illustrative example showing how the proposed redundant execution mechanism enforces a valid TSO interleaving (*bottom left*) for the execution of shared memory operations in two cores (*top left*). The instruction execution (*top right*) shows the timing aspects of the memory operations and their cache/coherence state management on a per-core basis, along with the events and conditions to trigger autonomous and deadlock-free re-executions.

cores executing three memory instructions each (*top left*) are shown to illustrate the time varying behavior of redundant execution using Core0 and Core1 (*top right*).

I assume, initially Core1 has data block *A* in *M (for Modified)* coherence state and Core0's caches are cold. Core0 makes the store request for instruction *I1* and waits for the data from DRAM. In the meantime, Core1 executes instructions *I4 – I6*, and *I6* results in a coherence message *Flush* to Core0 to acquire ownership of data block *B* in Core1. As Core0 receives the *Flush* request, according to the mechanism (cf. Figure 3.1.1), Core0 stops and re-executes *I1*. When *I1* re-execution is complete and following error detection, either Core0 recovers from an error by rolling back to *I1*, or continues to process *I2* and a reply (with data block *B*) is sent back to Core1. Because Core1 is done with executing its three instructions, it initiates re-execution of instructions *I4*, *I5*, and *I6* to validate them before finishing. In the meantime, Core0 continues to execute *I2* and *I3*, and sends two coherence messages (*Write back* requests) to Core1 to bring the data blocks *B* and *A* in Core0 with *S (for Shared)* coherence states respectively. When *I3* finishes, re-execution of instructions *I2* and *I3* is performed before completing the work assigned to Core0. Using this mechanism, the protocol ensures redundant execution of shared memory multicores (1) *deterministically* and *autonomously* at per core granularity, and (2) without incurring any *deadlocks*.

3.2.1 Coherence Protocol Deadlock Avoidance

It is well known that shared memory cache coherence protocol implementations are notoriously complex, and if the protocol operations are not architected properly within the compute cores and the communication fabrics, the multicore execution can end up in a *deadlock* [46]. For example, Figure 3.2.2 shows the execution of two threads running on Core0 and Core1, and accessing shared data blocks *A* and *B*. Initially, Core0 has *A* in its local private cache with *M* coherence state, and

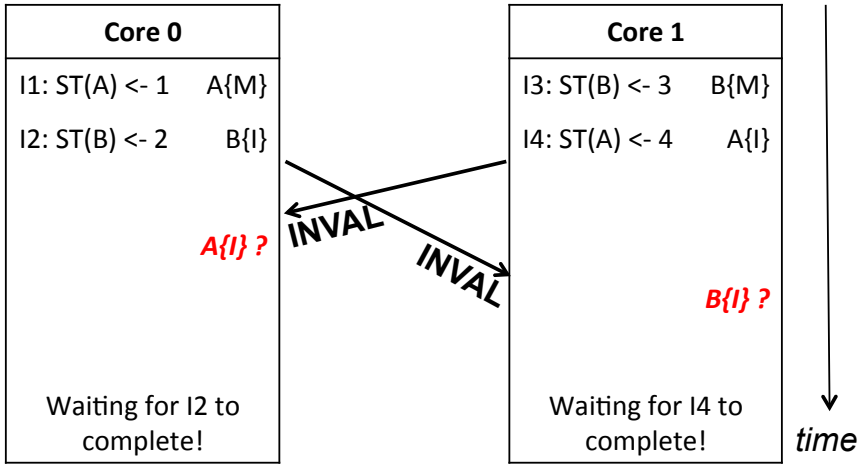

Figure 3.2.2: An illustrative execution of shared memory operations highlighting the possibility of *deadlock* when care is not taken during re-execution of instructions within the cores.

Core1 has *B* in its local private cache with *M* coherence state. When Core0 executes instruction *I2*, a coherence request is sent to Core1 to *invalidate* the data block *B* so that Core0 can get exclusive access to it. Concurrently, Core1 executes instruction *I4*, and a coherence request is sent to Core0 to *invalidate* the data block *A* so that Core1 can get exclusive access to it.

During the normal coherence operation, without any support for deterministic redundant execution, each core will service the invalidation requests by invalidating the requested data blocks (actions identified using *?* in the figure). But with the protocol, *determinism* of redundant execution requires that I re-execute and validate instructions that have their data in speculative state, before propagating any of the local effects globally. Therefore, in this scenario, Core0 cannot allow data block *A* to be invalidated before re-execution of instruction *I1*, and Core1 cannot allow data block *B* to be invalidated before re-execution of instruction *I3*. So, to ensure *deadlock-free* and *deterministic* redundant execution, the protocol initiates re-execution before servicing an *Invalidation*, *Flush* or *Write-back* coherence message. Note that for illustration, I am assuming a directory MSI invalidation-based coherence protocol that will initiate these messages from the directory. A

variation of this protocol may require additional messages that when seen by a processor tile, will initiate re-execution.

3.2.2 Handling of Synchronization in Shared Memory

A typical shared memory model requires instruction set architecture support for synchronization primitives. Test-and-set is a typical way to achieve synchronization where only one processor tile is allowed to access a critical section. In general, this technique involves using a shared variable called the semaphore and assigning values to this variable for the *Lock-OFF* or *Lock-ON* state (Figure 3.2.3 shows an example code snippet using the MIPS ISA-style instructions). Semaphore can be interpreted as a lock to some critical section. Each processor tile checks if the lock is off; if so, it tries to lock it by modifying the variable appropriately. The shared memory protocol must guarantee that only one processor tile, at a time, can get the ownership to modify the variable and lock or unlock it. Once a processor tile gets the lock, it is then allowed to modify some restricted data or access the critical section. After it is done, it gets out of the critical section and modifies the semaphore to the *Lock-OFF* state so another processor tile can get a chance to access it. Figure 3.2.3 to 3.2.6 present an illustrative example showing synchronization primitive operations and redundant execution.

Figure 3.2.4 to 3.2.6 show three possible scenarios where a processor tile X is spinning when another tile has access to the *critical section* (Figure 3.2.4), the lock is OFF but tile X's attempt to acquire the lock fails (Figure 3.2.5), and the lock is OFF and the tile X succeeds to acquire the lock to access the *critical section* (Figure 3.2.6). Figure 3.2.4 shows an *invalidation* from another tile that may be attempting to acquire ownership to *S*, therefore, tile X stops and initiates re-execution before servicing this coherence request. Similarly, if an *invalidation* is observed between instructions *I1* and *I4* while the lock is OFF (Figure 3.2.5), re-execution will ensure that

```
Loop  I1: R2 <- LL(S)          // R2 <- semaphore (1=Lock is ON, 0=Lock is OFF).
                               // Tile sets LLBit.
      I2: ORI R3, R2, 1
      I3: BEQ R3, R2, Loop     // Loop if locked (R2=1).
      I4: SC(S) <- R3          // Try to store semaphore. After completion of SC
                               // R3 returns 0 if SC failed, else SC succeeded.
                               // Success will invalidate request to other tiles to break
                               // link by resetting LLBit.
      I5: BEQ R3, 0, Loop      // Loop if R3=0.
                               // Another tile succeeded in acquiring the lock.
          .
          .
          CRITICAL SECTION
          .
          .
      I6: ST(S) <- R2          // semaphore <- R2 where R2=0.
                               // Invalidate requests to other tiles.
```

Figure 3.2.3: Code snippet for locking and subsequently unlocking access to the *critical section* in a processor tile.

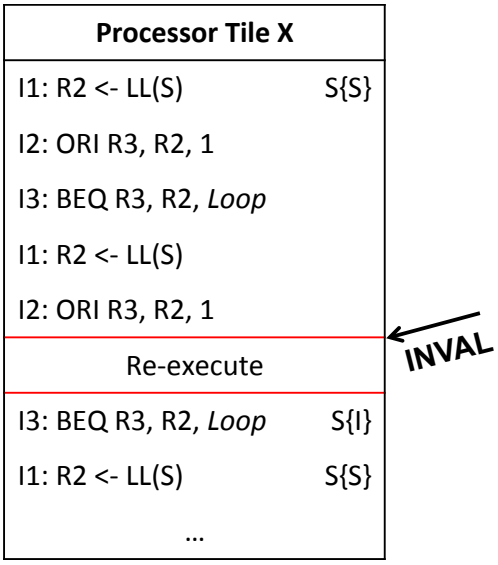

Figure 3.2.4: *Lock-ON*, so tile X spins until an *invalidation* on *S* initiates re-execution followed by an attempt to lock by tile X.

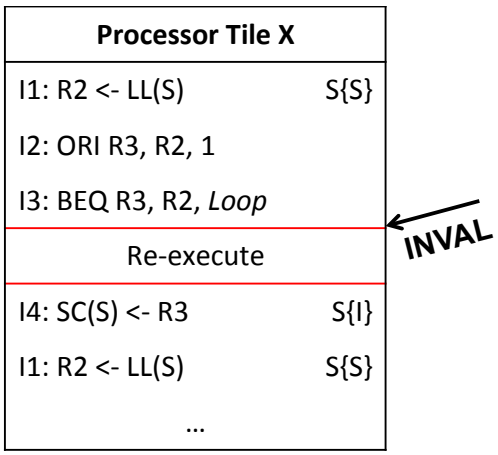

Figure 3.2.5: *Lock-OFF*, but an *invalidation* on *S* initiates re-execution and clears *LLBit* in tile X. The Store-Conditional (SC) subsequently fails to acquire the lock.

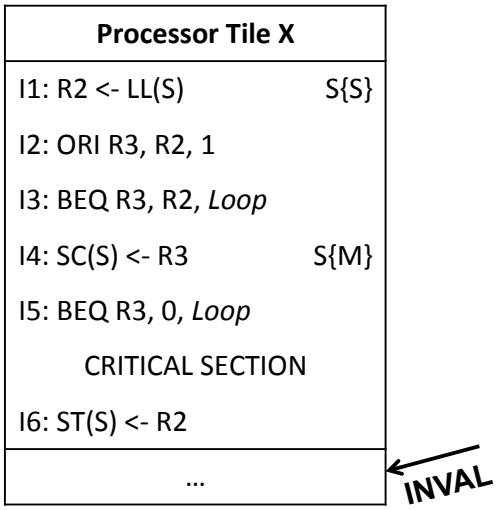

Figure 3.2.6: *Lock-OFF* and tile X successfully executes Store-Conditional (SC) to acquire the lock (*I4*). Other tiles will spin and tile X will have exclusive access to the *critical section* until instruction *I6* releases the lock.

data block shared memory behavior and synchronization semantics are preserved. In case of a successful acquisition of lock (Figure 3.2.6), shared memory synchronization semantics ensure all other processor tiles will spin wait and not interfere with tile X's access to the *critical section*. In this case, the instructions executed between *I1 – I6* can re-execute at any desirable sub-intervals.

3.2.3 Extensions to Support Hard Errors

While redundant execution within a processor tile is sufficient to detect soft and transient errors, detection of hard (aka permanent) errors on an individual tile poses special challenges as re-execution may end up with the same erroneous result. Hard error detection requires re-execution of the code in different hardware. Relying on an additional processor tile for redundant execution requires either fetched instructions and operands to be transmitted to the other core, creating large communication and power overhead [47, 48], or extend recent execution migration mechanisms [49, 50, 51] that ensure deterministic and deadlock-free aspects of the proposed redundant execution mechanisms. Further, to reduce performance overhead, the other tile must be free and available whenever an instruction sequence needs to be verified, which may require the second tile to be a slave dedicated core within the processor tile. In prior solutions such as DIVA [52], a slave core was used for instruction verification. A slave core adds significant power overhead due to data movement between cores, not to mention the hardware overhead.

Data movements between tiles are notoriously expensive in power, and therefore, an ideal solution from power perspective involves re-executing the code in the same tile, which is at odds with the first objective of hardware diversity. Note that the *execution errors* can be easily detected, if the instructions are simultaneously dispatched to two separate execution units within the same tile. This requires a full-time spare execution unit(s). Alternately, additional tag bits may be added in the reorder buffer (ROB) of an out-of-order core to indicate a unit ID where an instruction was

executed and to re-dispatch the instruction for re-execution to a different unit. This takes care of all execution errors. *Control errors* may be handled analogously. Instructions that modify the program counter value may be inserted into a queue for re-verification with the previous result. Together, these solutions ensure robustness of program execution without adding power for instruction and operand re-fetch, without a slave tile, and without power overhead for data movement between tiles. The key research questions are (i) optimum resources and their sizing, and (ii) optimal instruction scheduling for minimizing/eliminating any performance impact. While program recovery is easy for soft/transient errors within the proposed mechanism, micro-architectural solutions are also needed for error isolation in case of hard errors. The proposed research will address these questions.

3.2.4 Verification of Instruction Sequences using Signatures

Ensuring correct execution of instructions can take many forms. Today's microprocessors typically employ fault-avoidance techniques that aim to attain correctness by design margining. Voltage and frequency margins are inserted into acceptable operating ranges. An extensive set of signal integrity checks are then performed in simulation to ensure that sufficient margins have been added to account for all execution scenarios. These techniques, however, are becoming less attractive due to increasingly uncertain nature of sources of error and the rising cost of design and resource overheads.

Dynamic verification of instruction sequence execution based on signatures is proposed to detect soft errors. In order to perform lightweight dynamic verification of a block of instructions, I simplify the checking operation by relying on signatures (similar to [17]). Whenever an error is detected by comparing signatures, the program state of each core is rolled back by invalidating stores in the speculative store buffer (SSB). Usage of SSB can potentially allow a larger window

of instructions to be verified. While redundant execution within a single core is sufficient to detect transient errors, detection of permanent errors on a single core poses special challenges as re-execution may result in the same erroneous result. Relying on additional core for redundant execution requires fetched instructions and operands to be transmitted to the other core, creating large communication and power overhead [DATE'09, ATS'11]. Further, to reduce performance overhead, the other core must be free and available whenever an instruction block needs to be verified, which may require the second core to be a slave core. In prior solution such as DIVA [XX], a slave core was used for instruction verification. A slave core adds significant power overhead due to data movement between cores, not to mention the hardware overhead. Avoiding instruction by instruction comparison of results improves performance and reduces power consumption. In the following I outline proposed solutions to avoid instruction by instruction comparison of results and data transmittal for instruction verification.

The key idea in avoiding instruction-by-instruction comparison of results is to capture signatures of the program execution that reflect both the control flow [53] and the data execution [54]. Specifically, I collect two signatures for each executing thread. The first one compresses the program counter values associated with each committed branch instruction into a multiple-input signature-register (MISR). The MISR is re-initialized for each instruction sequence, collecting the signature of a fixed number of executed branches and comparing against the reference obtained dynamically. Such a comparison reveals errors in the control flow of the program. The second signature is associated with store instructions. Whenever a store instruction is executed, a write occurs into a memory address. In practice, such writes will modify the cache but may never get written back into the main memory before the program crashes. To avoid the loss of such information, I collect a signature of the data value and the data address in a MISR on every committed write instruction. Since the signatures are collected on committed instructions, speculative execution has no effect on these signatures. The virtual address of the branch/store instructions is used

to compute the signature. Hence, the signatures are easy to match. Lastly, the traps and interrupts are modified for verification of the instructions to be completed before transfer of control. When a fault occurs, the signatures obtained from the two executions differ and the fault is detected. The key research problems involve (i) quantitative assessment of performance overhead of this solution and (ii) micro-architectural solutions to keep the overhead at minimum.

3.2.5 Identifying Vulnerable Instructions for Re-Execution

The concept of vulnerable instruction sequences applies primarily to soft errors. By their very nature, the exact timing of such errors cannot be predicted. For example, soft errors that result from cosmic radiations depend on the particle flux, energy distribution of particles, and angle of incidence. Therefore, it is most beneficial to enable resiliency when the hardware is vulnerable to such errors. Opportunistic transient-fault coverage has been proposed for superscalar processors since it allows the system to tradeoff performance and coverage [39].

The probability that a fault in a compute pipeline will result in a visible error in the final output of a program has been previously defined as the architectural vulnerability factor (AVF) of that structure [55]. Thus, the key to optimizing an instruction sequence re-execution is to understand the AVF of instructions in flight in terms of hardware vulnerability. It is also well known that the AVF of a program instruction depends on how long the instruction was in flight during its execution. Previously it has been assumed [55] that instructions are vulnerable after they are fetched and before the results are retired. If an instruction(s) or its operand(s) reside for a long time in the compute pipeline, its vulnerability for single event upset increases. Thus, the probability of incorrect execution of an instruction depends on the time elapsed between insertion and retirement. Longer the duration, the greater the chance for system error.

I propose to maintain a *vulnerability monitor* in each core (cf. Figure 3.1.1) that tracks the

hardware vulnerability of the compute pipeline and consequently initiates instruction re-execution. Typically, instructions stay longer in the pipeline due to stalls that result from private cache misses. Therefore, in this thesis I utilize a private cache miss based trigger for re-execution. Since such stalls imply increased hardware vulnerability, the heuristic is expected to lower AVF, while incurring a small performance penalty (since most of the re-execution can be hidden behind the memory stall). With such opportunistic resiliency heuristic, all other required re-execution triggers (cf. Figure 3.1.1) are serviced by simply purging the SSB and initiating a new *START*.

In a modern superscalar processor, after an instruction has been fetched, it occupies a slot in the reorder buffer (ROB), which is used to enable out-of-order execution and in-order commit. The record for the instruction is maintained in ROB until it retires. If an instruction(s) or its operand(s) stay too long in an ROB, its vulnerability for single event upset increases. Thus, the probability of incorrect execution of an instruction depends on the time elapsed between insertion and retirement of entries from the ROB. Longer the duration, the greater the chance for system error. I propose to maintain a timer register in the ROB to measure time elapsed between instruction issue and retire. Similarly, an elapsed time accumulator (ETA) is used to collect elapsed time for a block of instructions.

In Figure 3.1.1, I show a processor tile that initiates the intention to re-execute (③). At this time, if the value of the ETA exceeds certain threshold to be determined experimentally, the instruction sequence is re-executed. This is the central idea behind *Vulnerability Monitor*. Even without an explicit request, the vulnerability monitor may potentially re-execute individual instructions based on large elapsed time.

The early results indicate that this is an efficient solution. Typically, instructions stay longer in the pipeline due to stalls that result from branch mis-predictions or private cache misses. Whenever there are stalls, execution resources are idle and using them for instruction re-execution can be effective.

3.3 Evaluation Methodology

I evaluate a 64-core shared memory multicore. The important architectural parameters used for evaluation are shown in Table 3.3.1. All experiments are performed using the modified GRAPHITE multicore simulator [37]. All the mechanisms and protocol overheads discussed in Section 3.1 are modeled. GRAPHITE simulator requires the memory system (including the cache hierarchy) to be functionally correct to complete simulation. This is a good test that all proposed re-execution mechanisms are working correctly given that I have run 25 parallel benchmarks to completion.

Architectural Parameter	**Value**
Number of Cores	64 @ 1 GHz
Compute Pipeline per Core	In-Order, Single-Issue
Memory Subsystem	
L1-I Private Cache per Core	32 KB, 4-way Associative, 1 cycle
L1-D Private Cache per Core	32 KB, 4-way Associative, 1 cycle
L2 Shared Cache Slice per Core	256 KB, 8-way Associative, 8 cycles, Inclusive
Directory Coherence Protocol	Invalidation-based MESI
Num. of Memory Controllers	8
DRAM Bandwidth	5 GBps per Controller
DRAM Latency	100 ns
Electrical 2-D Mesh with XY Routing	
Hop Latency	2 cycles (1-router, 1-link) plus Contention delay
Flit Width	64 bits
Default Re-execution Trigger Parameters	
Speculative Store Buffer Size	4 entries, Fully Associative, 1 cycle
Fixed Instruction Interval	100 instructions
Private (L1) Cache Miss	Enabled
Per-trigger Overhead (Local Checkpoint)	1 cycle

Table 3.3.1: Architectural parameters.

The electrical mesh interconnection network uses XY routing. Since modern network-on-chip routers are pipelined, and 2- or even 1-cycle per hop router latencies have been demonstrated (e.g., Tilera multicores), I model a 2-cycle per hop delay; I also account for the appropriate pipeline

latencies associated with loading and unloading a packet onto the network. In addition to the fixed per-hop latency, network contention delays are also modeled.

Application Benchmarks: Thirteen SPLASH-2 [56] benchmarks, four PARSEC [36] benchmarks, three Parallel MI-Bench [57], a Travelling-Salesman-Problem (TSP) benchmark, a Depth-First-Search (DFS) benchmark, a Matrix-Multiply (MATMUL) benchmark, and two graph benchmarks from the DARPA UHPC program (CONNECTED-COMPONENTS & COMMUNITY-DETECTION) [58] are evaluated using GRAPHITE. The graph benchmarks model social networking based applications. Each application is run to completion using the recommended medium or large input sets.

Performance Models: For each simulation run, I measure the *Completion Time*, i.e., the time in *parallel* region of the benchmark; this includes the instruction processing, memory access, and the synchronization latencies. I implement the *re-execution triggers* due to private cache misses or evictions (The re-execution trigger on a private cache miss supersedes the required trigger for the local cache eviction of speculative data.), SSB full, the coherence requests observed by each core, and fixed instruction intervals. I also model all the performance overheads of re-execution, including local checkpoints, and instruction, memory and communication latencies. I measure the *Re-execution Interval Length* by tracking the average number of committed instructions for each re-execution trigger.

Compute Pipeline Coverage: I measure the soft error coverage of the compute pipeline by capturing the ratio of the original execution time of the instructions that are eventually re-executed, and the original execution time of all instructions (excluding their re-execution time).

Resilient Coherence Models: I model all protocol and performance overheads of the resilient cache coherence [23]. Resilient coherence protocol enables each coherence transaction from a core to produce a unique outcome when receiving the message for the first time and anytime thereafter. The protocol ensures deterministic replay of coherence transactions via three properties: "(1) All initiators of transactions stay in a transient state till all other cores involved in the transaction have

completed their part and transitioned back to a stable state, (2) Previously transmitted messages can be re-transmitted: cores retain sufficient information to regenerate any previous message for each outstanding transaction, and (3) All cores involved in a transaction can tolerate duplicate messages and still produce the same outcome, i.e. transition to the same state and generate the same message."

I model the performance overheads of resilient coherence by incorporating additional coherence messages over the on-chip network. The requester and directory interface is extended with *unblock* and *done* messages to ensure the directory only transitions its coherence states after the requester has safely completed the coherence transaction. Additionally, the directory and sharer(s) interface is extended with *permission* and *ack* messages to ensure data transmission has safely completed at the directory core before transitioning the coherence states at the sharer(s).

3.4 Results

3.4.1 Resilient Cache Coherence Implications

I first evaluate the performance overhead of the additional coherence messages introduced by the resilient coherence protocol [23]. Figure 3.4.1 shows this overhead when compared to a traditional directory-based coherence protocol. On average resilient coherence results in a 5.8% performance loss. It is observe that SPLASH-2 and PARSEC benchmarks incur a smaller 2.5% performance penalty (similar to [23]). The remaining benchmarks result in a higher penalty because they have relatively more communication between parallel threads. Since resilient coherence overheads materialize on a private cache misses, I can potentially hide much of this overhead behind the re-execution latency.

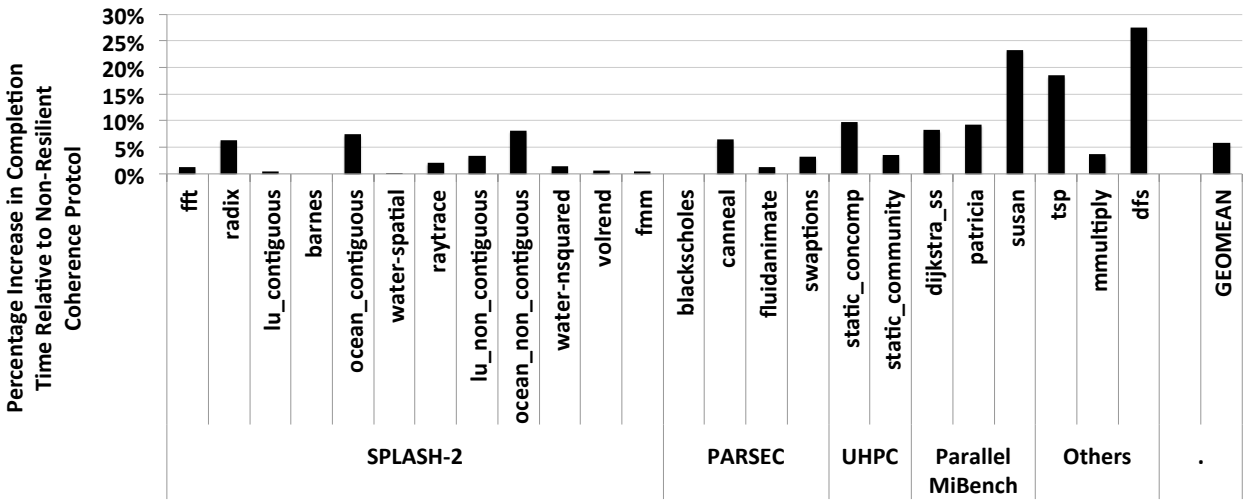

Figure 3.4.1: Performance overhead for the resilient directory-based coherence protocol.

3.4.2 Performance Advantage over Idealized Thread-Level Redundancy

I compare the completion time of the proposal with an idealized thread-level redundant (ideal-TLR) architecture where each thread of a parallel program is redundantly executed to detect and correct soft errors at runtime. In the implementation, an ideal-TLR is approximated by running each parallel application on half of the cores (using the same inputs that are used to run the application on all available cores). This is similar to a recent proposal by Rashid and Huang [20] that suggested an *implementable* thread-level redundancy mechanism that performs worse than the ideal-TLR. A comparison to the ideal-TLR reveals the true merit of the proposed architecture. It is important to note that in addition to performance advantages the proposal is *lightweight* as it inherently guarantees deadlock-freedom and is therefore exempt from expensive global rollback and recovery mechanisms.

Figure 3.4.2 shows the performance advantage of the proposal compared to the ideal-TLR. In the first implementation, I do not consider re-execution triggers due to private cache misses. This configuration does not enable all the scenarios to hide the latency of re-execution as the compute

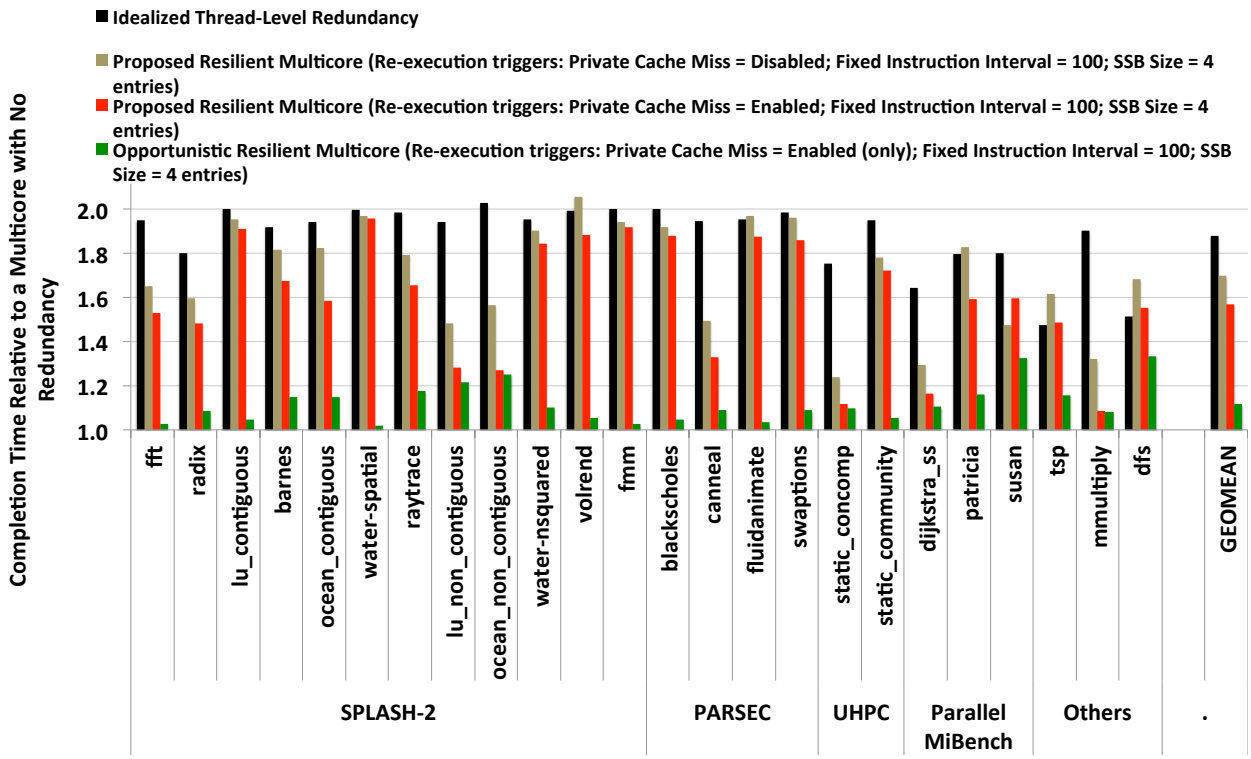

Figure 3.4.2: Performance comparison of ideal-TLR ($\sim$1.9$\times$ worse), the proposed resilient multicore architecture ($\sim$1.7$\times$ and 1.55$\times$ worse without/with latency hiding) @ high coverage, and opportunistic ($\sim$1.11$\times$ worse) @ 41% coverage, relative to a baseline multicore with *no* redundancy.

pipeline may need to stall frequently and wait for each re-execution to complete. I observe that the baseline re-execution mechanism (without all latency hiding optimizations) outperforms ideal-TLR by an average of 10%, because:

- I exploit the inherent parallelism of a multicore and the application, and
- Re-execution is *isolated* within a core where the latency of each instruction execution is optimal (e.g., all memory accesses result in L1 cache hits during re-execution).

Next an optimized resilient multicore is evaluated, which in addition to the required re-execution triggers also initiates re-execution for each private cache miss. Hiding latency further improves the results since much of the re-execution latency is hidden behind the private cache misses (that can take 10s to 100s of clock cycles). Figure 3.4.2 shows that the optimized resilient multicore outperforms the ideal-TLR by an average of 17%. Since each *eviction* (a necessary conditions for initiating re-execution) results in a private cache miss as well, the opportunity to hide the re-execution latency of the requesting core is highly likely. Similarly, a *coherence miss* (another necessary condition that initiates re-execution of the sharer tiles) that is now possibly more expensive under the proposed architecture, the requesting core can potentially initiate its own re-execution and hide some the associated latency while the communication protocol services the miss.

The simulation results in Figure 3.4.2 validate the intuition that the proposal is viable under bottlenecks such as high communication applications and the overhead of per-core local checkpoints (using SSB and shadow register file hardware).

Re-execution Interval Length

To expect good performance from the proposed architecture, the number of instructions in a re-execution sequence must be neither too small nor too large. A small re-execution sequence may

not be able to amortize the cost of the overhead associated with initiating re-execution, whereas a large re-execution sequence limits the potential advantage from the latency hiding optimizations. Keeping these tradeoffs in mind, I varied the size of the per-core SSB from 64 entries down to 4 entries per SSB, and quantified the number of committed instructions per re-execution trigger. Note that hiding the latency of re-execution due to an "SSB full" trigger is highly unlikely since such hardware-dependent trigger may not align with an opportunity where the core may need to stall for an extended period to service an instruction.

The results (not shown as a figure) show that a small 4-entry SSB results in a re-execution trigger every $\sim$50 committed instructions (on average). When the SSB size is increased to 8, 32 and 64 entries, the re-execution trigger improves to only $\sim$65, $\sim$78 and $\sim$82 committed instructions respectively. I conclude that this improvement in re-execution interval length does not justify the increase in SSB complexity and overhead (specifically latency of SSB access compared to a single-cycle L1 cache). The performance results (not shown here) also validated this conclusion since I observed minimal performance improvements by increasing the SSB size when compared to the results presented in Figure 3.4.2.

Breakdown of Re-execution Triggers

To validate the intuition about the potential latency hiding optimizations, I must justify that a low-overhead 4-entry SSB does not overwhelm the frequency of re-execution triggers. Therefore, I conducted an independent experiment where only re-execution triggers due to L1 misses, coherence requests, and "SSB full" are allowed. I plotted the re-execution triggers relative to the total re-executions for each benchmark (Figure 3.4.3). The average values are also plotted for the SSB sizes of 4 and 8 entries.

On average, the re-execution triggers due to "SSB full" are $\sim$25% and $<$10% of the total

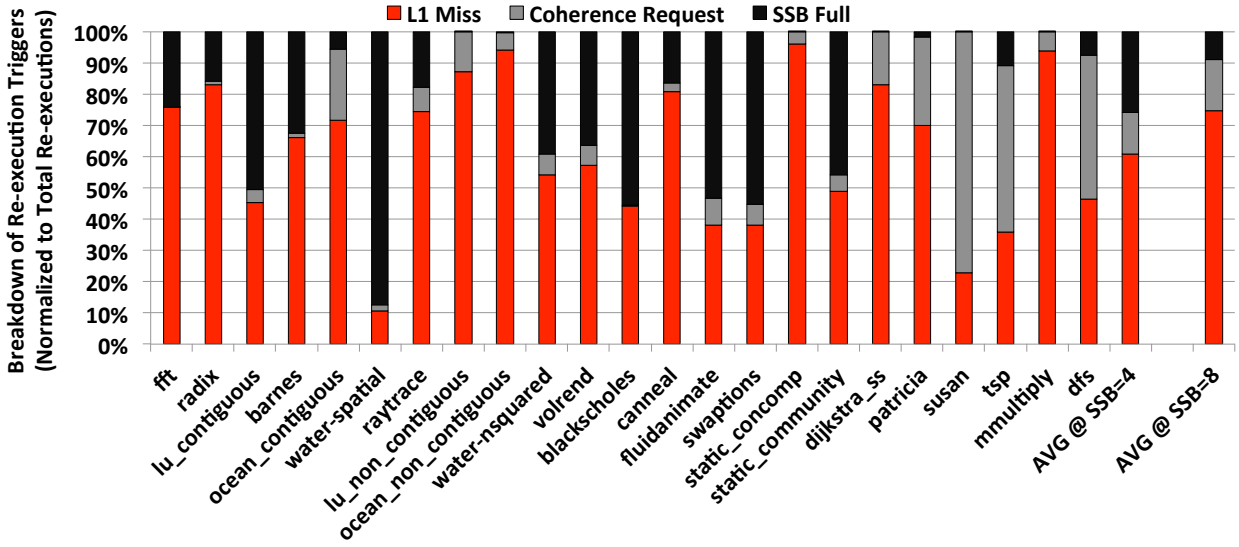

Figure 3.4.3: Breakdown of re-execution triggers for all benchmarks with a 4-entry SSB per core.

number of re-executions for SSB sizes of 4 and 8 entries respectively. I also observed (not shown here) that a 32-entry SSB reduces the "SSB full" triggers to $<$0.1%. This result highlights the lightweight nature of the proposal, whereas the re-execution trigger due to coherence requests ($\sim$15%) nicely justifies that the proposal is not a bottleneck due to communication in applications. Even in applications such as TSP and DFS, where more than 40% of the re-execution triggers are due to coherence requests, the performance of these benchmarks still matches the ideal-TLR (cf. Figure 3.4.2).

Finally, I make an important observation that $\sim$60% of the re-executions are due to L1 misses in a core. As confirmed in Figure 3.4.2, I can safely conclude that incorporating latency hiding optimizations is indeed a viable strategy to hide much of the latency of re-executions.

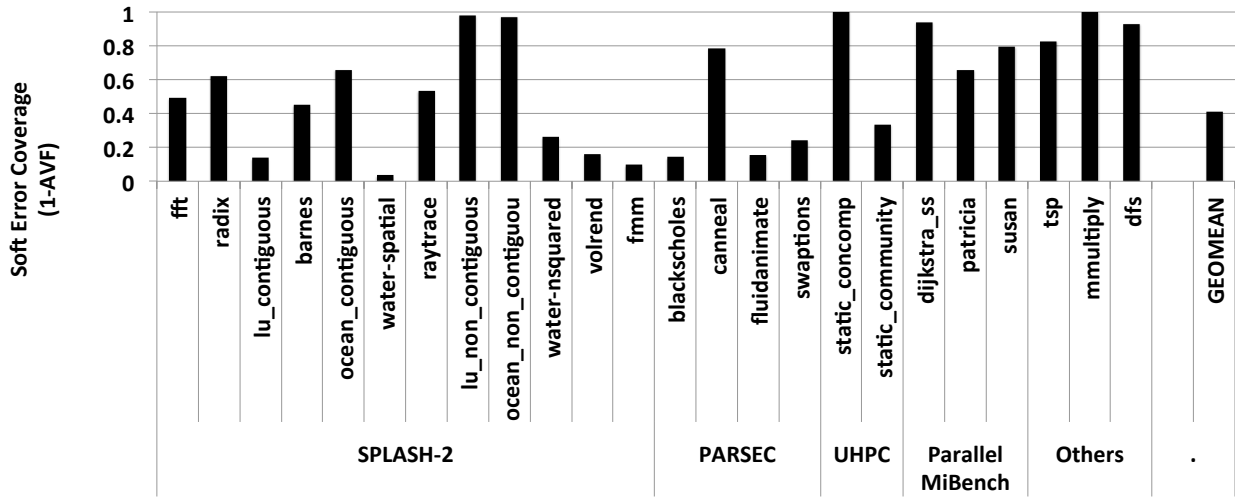

Figure 3.4.4: Soft Error Coverage for the compute pipelines when re-execution is opportunistically triggered on private (L1) cache misses. All other re-execution triggers result in purging the SSB and initiating a new *START* (cf. Figure 3.1.1).

3.4.3 Performance and Coverage Tradeoffs with Opportunistic Re-execution

So far, the evaluation has focused on high soft error coverage while the performance and hardware overhead of the proposed resiliency mechanism is optimized. As discussed in Section 3.2.5, opportunistic re-execution within the proposed resiliency method allows the system to tradeoff performance with coverage. An intuitive heuristic only allows re-execution on private cache misses and disregards all other re-execution triggers (by simply purging the SSB and initiating a new *START*). Since the hardware is most vulnerable to soft errors for instructions that are idling in the compute pipeline due to long latency memory stalls, I hope to hide most of the latency of re-execution behind such stalls while delivering a reasonable level of error coverage.

Figures 3.4.2 and 3.4.4 show the performance and compute pipeline coverage results for the opportunistic re-execution heuristic. The results show only 11% performance degradation when compared to a baseline with no redundancy, while the hardware coverage is 41% of the compute pipeline and the un-core logic. I note that the compute pipeline coverage is highly correlated to the

private cache miss rate and latency. For example, applications such as *lu_contiguous, water_spatial, water-nsquared, volrend, fmm, blackscholes, static_community* with private cache miss rates of $\sim$1% have the smallest coverage of 3 to 15%. On the other extreme, applications with $>$5% miss rates (*ocean_contiguous, lu_non_contiguous, ocean_non_contiguous, static_concomp, dijkstra_ss, mmultiply, dfs*) show nearly $\sim$100% instructions are covered.

In summary, the results suggest a dynamic knob that can be used to tune the soft error coverage and performance overhead of resiliency at runtime.

Chapter 4

Declarative Resilience Framework

The proposed declarative resilience architecture for shared memory multicores applies different resilience schemes to different code regions based on their criticality. A Hardware Redundant Execution (HaRE) [19] scheme was introduced earlier, which is based on temporal redundancy. It relies on a local per-core re-execution mechanism to recover from detected errors for the compute core, and implements resilient coherence protocol for the on-chip communication [23]. This avoids expensive global checkpoint and roll-backs, while enabling holistic protection for the multicore system. It also ensures the feasibility of switching the re-execution on/off at each core since one core's re-execution does not affect instruction execution of another core. Declarative resilience reduces the performance overhead by eliminating unnecessary redundant execution for certain code regions. Strong scheme (HaRE) is used for the code regions (crucial) that effect program correctness. However, lightweight schemes are used for the regions (non-crucial), where compromising program accuracy is acceptable in case of soft-errors.

The key idea of the proposed declarative resilience framework is protecting different code regions with different resilience schemes. The novelty comes from the way code regions are iden-

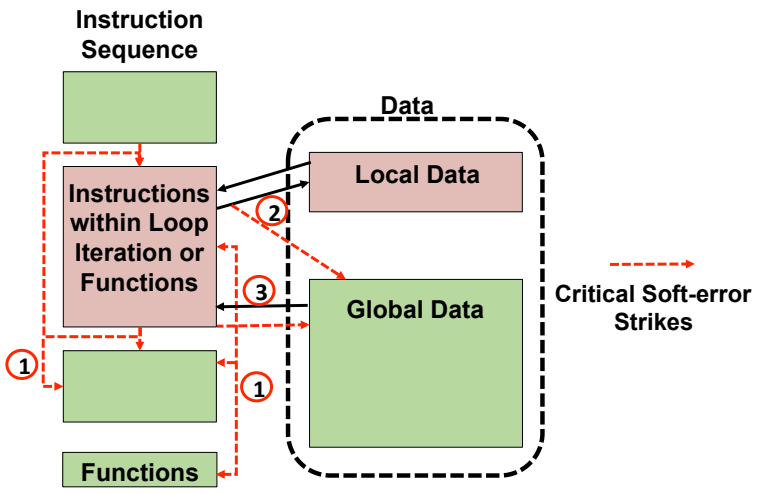

Figure 4.1.1: Soft-errors' effects to program control and data flows

tified: based on their impact to program outcome. The framework can be applied with various hardware/software protection schemes on different systems, such as Xeon Phi, or GPU, to trade off performance, accuracy, and coverage. As shown in Table 1.2.1, the proposed architecture also targets high coverage for SDC in addition to program crashing, deadlock, and livelock type of errors.

4.1 Guidelines of Non-Crucial Instructions

Based on the insights about soft-error effects on program control and data flow (explained in Section 2.1), the ideal candidates for non-crucial instructions are *compute instructions* in-between control flow that only execute on local data and have minimum impact on program outcome. One example would be random number generation instructions in Monte Carlo method. However, in real applications instructions that meet all these conditions are rare. In addition, turning HaRE on and off introduces overheads that must be amortized by lowering the frequency of switching

between crucial and non-crucial instructions. Next, a set of lightweight software and hardware resilience mechanisms are introduced that enable the proposed architecture to compose non-crucial code regions.

In a nutshell, following scenarios elaborate soft-error effects of compute, load and store instructions;

(1) *The opcode is perturbed.* Due to the fact that there should be no control-flow instructions in non-crucial code regions, when the operands are perturbed, such that the compute, load, and store instructions are turned into control flow instructions, exceptions would be triggered. For the same reason, the PC of next instruction should always be ”current PC + 4”, otherwise an exception should be triggered.

(2) *The operand is perturbed (When no exception triggered).* Since load values are only used for computation in non-crucial code regions, perturbed load operand (address) would only result in wrong value for the following computation in the regions (in case of no segmentation fault). Thus, bit-flips in the operands (such as in the decode or execution stage) of compute and load instructions would only result in perturbed value of temporal variables, which would be checked and dropped when committed, if they are out of bound. Meanwhile soft-errors in the operands of store instructions would result in accessing arbitrary memory location, thus the store address calculation is protected by SHR.

(3) *When exceptions are triggered.* All exceptions should be handled as usual. Such as in the case of a perturbed load access out of bound address, the OS would handle the triggered segmentation fault. Additional OS support is needed to handle the exception when control-flow instructions are present in non-crucial regions.

4.2 Non-Crucial Code Regions with SHR

In order to make declarative resilience beneficial for programs, more instructions need to be made resilient for certain soft-errors, and it is favorable to have them in continuous sequences. Such continuous regions can be obtained by the use of *loop-unrolling*. It is a loop transformation technique that attempts to optimize the execution speed of the program by reducing instructions that control the loop such as *end of loop* tests on each iteration. Loop-unrolling involves loop to be re-written (can be done in an autonomous fashion) as a repeated sequence of similar independent commands and statements. This allows the soft-error effects on the program outcome to be restricted for certain code regions. For this purpose, a set of lightweight software and hardware resilience (SHR) mechanisms are introduced on top of HaRE.

First, at the hardware level, SHR applies protection for store instructions. Based on the insight that store addresses are always critical to the program, SHR performs hardware-level redundant address calculations. These calculations incur additional overheads. Store operations proceed after their address calculation is verified. Otherwise they are re-executed. This is to ensure that store instructions do not access the crucial region unexpectedly. Second, at the software level, the value committed to global data is checked, when necessary (in software), to ensure program correctness and accuracy. This protection is critical as the non-crucial code can affect the program response if the values committed to the memory are not checked. In machine learning and graph analytic applications, majority of critical variables are passed through a bound-checking process. Bound checkers provide an upper bound of the values to be committed to the memory. For example, consider a simple program P that increments the value of x by 1 ($x++$) each time for 100 iterations. For P, the bound-checker would keep track of x that it never goes beyond 100 or is never less than 0– this could happen if the soft-error strike perturbs x to some arbitrary value. If the program fails to pass the bound-checking process, it is re-executed. Otherwise the program proceeds to the next

step. It is practical to use a programmer defined software-level bound checker to provide certain resilience protection.

With SHR, store instructions and local data computations with predictable outcome are able to be included in non-crucial code regions. These make it more feasible to form individual non-crucial instructions into code regions. When taking SHR into consideration, code regions can be classified as non-crucial if they do not contain control flow instructions, and meet one of the following: (1). They do not access global data; (2). The value committed to global data does not affect the program outcome. (3). The value committed to global data can be bound checked. Based on the relaxed criteria, a reasonable amount of code in machine learning and graph analytic programs can be considered non-crucial.

4.3 Systematic Assist

In the design flow (as shown in Figure 4.3.1), the programmer first identifies the non-crucial instructions based on the guideline. As mentioned earlier, it is beneficial to have contiguous compute instruction sequences within each non-crucial region. With SHR in mind, schemes such as software level bound checking and loop-unrolling are used to compose non-crucial instructions into code regions. This provides better performance while ensuring coverage. Accuracy analysis is later performed to verify the accuracy loss of the non-crucial code regions, and help the programmer select the proper combination of regions under certain constraints, such as soft-error rate and accuracy loss threshold. This combination is a subset of all the non-crucial regions, which is referred as configuration in this thesis. With the proper configuration, the transformed program can be deployed on the proposed cross-layer architecture. In the current setup, programmer's effort is mainly spent in instruction classification. The other steps can be assisted, and automated to certain

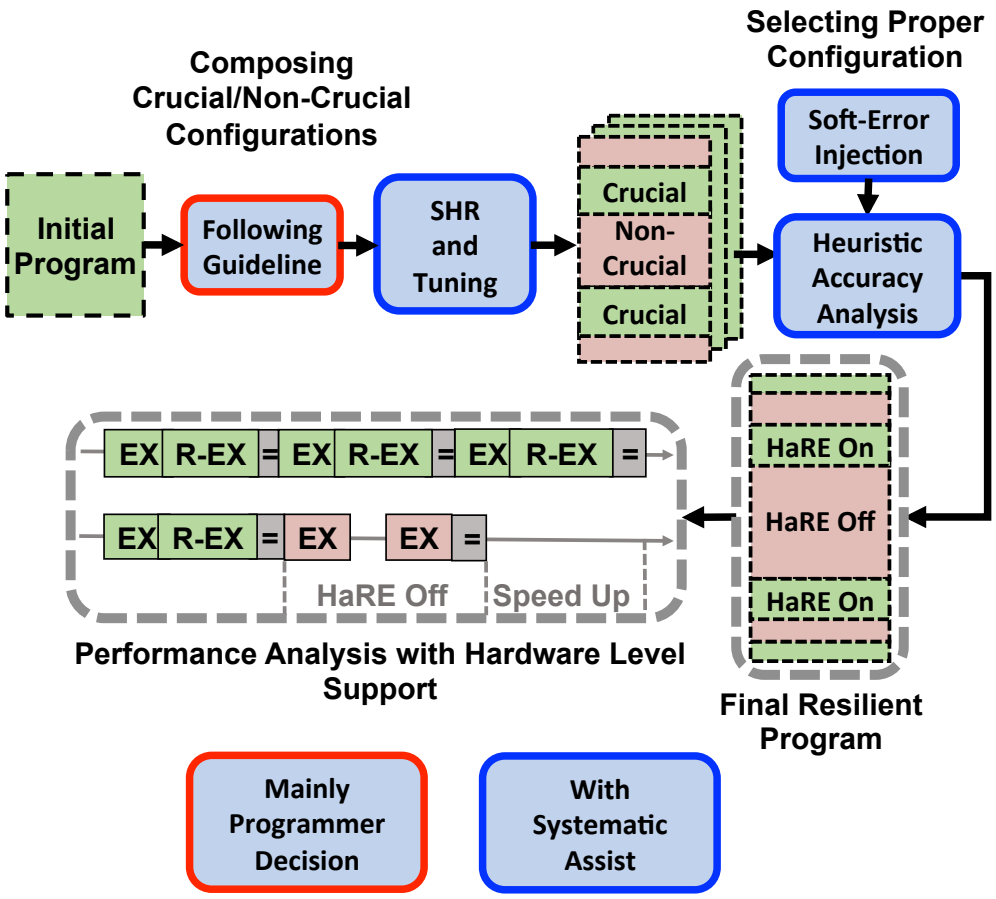

Figure 4.3.1: Declarative Resilience Framework.

extend.

The accuracy analysis can be done using an exhaustive search. However, in order to cover all the possible crucial/non-crucial region combinations, it needs to run 2^I setups (I is the number of potential non-crucial regions), with each one simulated multiple times. This involves great effort when I is large (such as CNN-ALEXNET $I = 9$ shown in Table 5.1.2). I propose heuristic based accuracy analysis to speedup this process with fewer simulations, and provide near optimal configuration, C_n. As shown in Algorithm 1, the goal is to find a C_n that meets the accuracy requirement, while the programmer needs to provide the following factors used in the analysis.

1. Potential non-crucial code regions R based on the guideline and SHR.

2. Acceptable program accuracy loss threshold: T.

3. Soft-error rate: $e\%$ based on the system setup.

In order to obtain other factors, such as program accuracy of regions (ar_n) and configurations (A_{Cn}), program level fault injection is performed. It imitates random errors in non-crucial code regions. It is done in such a way that a program variable prone to soft-errors is exposed to random values in the range determined by its data type. The probability of the injection is determined based on the error rate ($e\%$) and the execution time of that code region (exe_n), which is acquired from the time spent in that code region during simulations. The accuracy is determined based on application dependent metrics. Both realistic and aggressive error rates are used to build up the confidence. In the fault injection analysis, a single error is injected in the non-crucial region of the program code. The accuracy is defined based on application dependent metrics (Section 4.4). The notion of the fault injection is that after applying the protection schemes, the resilience framework ensures store address and control flow instructions are always protected (using SHR or HaRE). The remaining vulnerable code is other data flow instructions, meanwhile the out of bound value is

never committed. The accuracy loss of each region is obtained by only applying fault injection to it. Multiple simulations (10000 times) are performed for each configuration of non-crucial regions to obtain the average program accuracy.

Algorithm 1 Configuration Selection (Section 4.3)

1: Potential non-crucial regions: $R = \{r_1, r_2, ..., r_I\}$, $|R| = I$
2: Acceptable program accuracy loss threshold: T
3: Soft-error rate: $e\%$
4: Accuracy loss of regions: $AR = \{ar_1, ar_2, ..., ar_I\}$
5: Execution time of regions: $EXE = \{exe_1, exe_2, ..., exe_I\}$
6: Number of non-crucial regions in practice: n, $0 \leq n \leq I$
7: Configuration with n non-crucial regions: $C_n \in R$
8: Accuracy loss of configuration C_n: A_{Cn}
9:
10: $n = I$; ▷ *All regions non-crucial.*
11: **while** $n \geq 0$ **do**
12: Run simulations using C_n with $e\%$; ▷ (Get A_{Cn}).
13: **if** $A_{Cn} > T$ **then**
14: $n = n - 1$; ▷ One more region crucial.
15: C_n =
16: newConfig(n, AR, EXE, T);
17: **else** Return C_n;

As shown in Algorithm 1, at the beginning of the analysis all regions are assumed non-crucial. If the program accuracy loss exceeds the threshold, a new configuration with one less non-crucial region is selected (shown as C_n at Algorithm 1 line 14 - 16). The region with maximum accuracy loss (ar_n) in the previous configuration is considered as crucial. In case multiple regions have similar accuracy loss, it chooses the one with minimum execution time (exe_n). In the worst case, only I number of simulations are needed (compared to 2^I when using an exhaustive search).

As shown in Figure 4.3.1, after selecting the proper configuration the switching between crucial and non-crucial is placed in the program. These are passed as HaRE on/off pragmas to the hardware. The hardware-software interface for declarative resilience is implemented using a special function instrumented in the programs. HaRE is turned on before the bound checker to protect

it. The final resilient program is executed on the proposed cross-layer architecture, with hardware level support implemented in the simulator. Detailed code demonstrations are shown next.

4.4 Application Illustration

Machine learning and graph analytic applications are ubiquitously used in many domains, where the systems could be exposed to soft-errors. Due to their unique structure and computational behaviors, research has been done on relaxing their accuracy for performance benefits. Likewise they have potentials in the proposed framework for resilience-accuracy tradeoff. I evaluate six machine learning, and nine graph analytic applications. In this section, I illustrate how applications are transformed using the declarative resilience framework onto the cross-layer architecture.

4.4.1 Machine Learning

In general, machine learning algorithms work on massive data and perform perception computations. They can be used in many applications, including image recognition, video analysis and natural language processing. Such applications have the potential to be deployed in safety-critical systems, where they face resilience challenges. On the other hand, due to their inherent heuristic nature, individual floating point calculations hardly impact program outcome. The benchmarks evaluated in this thesis use supervised learning algorithms. The rate of ”correct classification / the number of tests” is defined as the accuracy of the application. For example, when applying 100 hand-written digits through CNN-MNIST, if 95 are classified correctly, its accuracy is defined as 95%. The accuracy loss is normalized to the program accuracy in soft-error free condition. In this section, the transformation of CNN is discussed in detail [59].

Algorithm 2 CNN Convolutional Layer Pseudo Code

```
1:  ConvolutionLayer(input, conv_out, tid, threads) {
2:  for each neuron in the thread do
3:      \* The following 3 level loop is Unrolled *\
4:      for (number of kernels k, kernel height, h, kernel width, w) do
5:          \* Assign temp_k/h/w *\
6:          HaRE Off
7:          conv_out += do_conv(input,temp_k/h/w)
8:          \* Update temp variables *\
9:          conv_out += do_conv(input,temp_k/h/w)
10:         \* Update temp variables *\
11:         :
12:         HaRE On
13:         \* Update k, h, w *\
14:     Bound_Checker(conv_out)
15: }
```

Convolutional Neural Network (CNN)

CNN is a highly prevalent neural network type. In this thesis, I evaluate four of the most commonly used convolutional neural networks: AlexNet (ALEXNET) [34], hand-written digits recognition (MNIST) [60], recognition of German traffic signs (GTSRB) [9], and VGG [61]. All of them consist of 4 types of layers: input, convolutional, fully connected, and output layers. The computation within each layer can be identified as non-crucial. In this section convolutional and fully connected layers are illustrated, as they contribute most of the execution time. The parallelization strategy for CNN is to divide the neurons in each layer among the available threads. Since the subsequent layers consume the outputs of prior ones, barriers are used to synchronize the threads after each layer, which are protected through HaRE.

In CNN, the convolutional layer takes the input feature map, then convolves it with the kernels to give an output feature map. This results in an output feature matrix cell value. These computations (shown in Algorithm 2 lines 7-9) can be considered as non-crucial, since the effect of

individual cell value to the program outcome is limited. Furthermore, each kernel produces an output feature matrix of its own. Cell values are compared with each other to find the maximum one, which is later used to construct a single output feature map. When exposed to soft-errors, the output map can get affected only if the maximum cell values are perturbed into larger ones, since smaller values are masked out. Note that, out of bound values are dropped. In that case, the second largest value would be used for the corresponding cell in the output feature map. The loops (shown in Algorithm 2 lines 4) are unrolled, and the loop counters (k,h,w) are updated with HaRE protection. Only temporary variables (temp_k/h/j in Algorithm 2) are written in non-crucial region.

Algorithm 3 CNN Fully Connected Layer Pseudo Code

```
1:  FullyConnectedLayer(input, fully_out, tid, threads) {
2:  for each layer do
3:      for each neuron do
4:          \* The following loop is Unrolled *\
5:          for each input i do
6:              HaRE Off
7:              O += (input(i) * weights(i))
8:              i = 1
9:              O += (input(i) * weights(i))
10:             :
11:             i = 2, 3, ....
12:             O += (input(i) * weights(i))
13:             HaRE On
14:             Temp = I
15:             Bound_Checker(O)
16:         fully_out = Sigmoid(O)
17:     Barrier
18: }
```

The fully connected layer (as shown in Algorithm 3) in CNN is a feed-forward network, in which all the neurons in one layer are connected to the neurons in the next layer. The neuron count reduces towards the end of fully connected layer. The first layer of this feed-forward network provides the output data set of the previous layer to the neurons as input. The later layers

perform accumulations and multiplications of the inputs with their respective weights to compute the sigmoid. The result is further propagated to the next layer. The computations done in the fully connected layer can be considered non-crucial, since the remaining unperturbed accumulations could overcome it. To ensure correctness of the program, bound checkers (shown in Algorithm 2 line 14 and 3 lines 15) are introduced in the code so that the effects of the perturbations can be reduced. Statically determined bounds are used. For fully connected layers, the accumulated results are used to compute the sigmoid. Based on the definition, this complex sigmoid computation always results a value with in the range of 0 to 1. For the sigmoid to output a 0 or a 1, the respective values (O) are -90 and 10. Thus, to limit the result with in this range of 0 to 1 (excluding 0 and 1), I limit the accumulations to -90 and 10.

Multi-Layer Perceptron (MLP)

Multi layer perceptron (MLP) is a feed-forward neural network in which all the neurons in one layer are connected to every neuron in the next layer. It's structure resembles the fully connected layer of CNN. Thus, these layers and the output layer can be considered non-crucial.

K-Nearest Neighbors (KNN)

In KNN, objects are classified using a number of known examples called training data. The distances between the new object and the each known object are calculated and sorted to determine k-nearest neighbors. Class of the new object is decided by majority vote over k-nearest neighbors. The Calculate_Distance function (**Distance**) in KNN involves accumulations of multiple examples. Each example has minimum impact to the program, hence its calculation can be considered as non-crucial. If a distance value gets perturbed, its affect on the overall outcome would be insignificant. Moreover, as the distance calculation of one neighbor is independent of the other

neighbor, perturbations do not propagate.

4.4.2 Graph Analytics

In general, graph analytic benchmarks traverse the vertices in the input graph, and compute based on the connectivity and weights on the connected edges. They may consist of different phases and iteration counts. Such phases consist of either data flow or control flow, alongside synchronization. In terms of graph analytics such phases traverse vertices and edges, and may propagate data to subsequent phases. These phases can also be within abstract level iteration counts within algorithms. Such computations may iterate over the input graph multiple times. Most of these computations within each phase, corresponding to for example floating point operations and arithmetics, can be identified as non-crucial. The accuracy metrics are defined based on individual benchmark's outcome. In this section, the non-crucial region(s) of each benchmark is described, with Triangle Counting being discussed in detail.

Triangle Counting (TRI_CNT)

Triangle counting is an important graph workload to measure graph statistics regarding vertex connections in applications such as web connectivity. This benchmark consists of three phases, as shown in Algorithm 4. The first (**AddEdges**) phase adds side counts from each edge to a global data structure for each vertex. The sides need to be incremented atomically for each edge in a parallel setting, thus locks are needed. The second (**Reduc**) phase reduces these added counts. It involves more computations, specifically divisions to get triangles per vertex. Moreover, these triangles are accumulated per vertex into a total triangle count in each thread. Bound checkers are needed to ensure total triangle counts are within acceptable bounds. The maximum number of triangles in each thread should not exceed the total edge count for a vertex in that specific thread,

which is used as the upper bound of the checker. The final (**GetTri**) phase gets the total number of triangles. Only master thread is involved in this phase, to sum up the total number of triangles. Barriers are needed between these phases to ensure proper functionality. Accuracy is quantified by comparing the total triangle counts.

Algorithm 4 Resilient TRI_CNT Pseudo Code.

```
1:  \* First phase *\
2:  HaRE On
3:  for (each vertex, v) do
4:      for (each neighbor, u, of v) do
5:          Lock (u)
6:          Add_Edges(u)
7:          UnLock (u)
8:  Barrier
9:  \* Second phase, the following loop is unrolled *\
10: for (each vertex, v) do
11:     HaRE Off
12:     Temp_T_Count = Reduce(v)
13:     HaRE On
14:     \* Bound check *\
15:     if (Temp_T_Count < V_tid * DEG) then
16:         Thread_Count
17:         = Temp_T_Count
18: Barrier
19: \* Final phase, the following loop is unrolled *\
20: Master Thread:
21: for (each thread, tid) do
22:     HaRE Off
23:     Temp_S += Get_Total_Triangles(tid)
24:     HaRE On
25:     \* Bound check *\
26:     if (Temp_S < Max_T) then
27:         S = Temp_S
```

Page Rank (PageRank)

PageRank initializes the probability of each vertex to the inverse of the total number of vertices, and then computes the page ranks of each vertex. The rankings are probabilities, which specify the likelihood of that corresponding page to be visited over the Internet. Each thread works on a chunk of vertices, thus no locks are needed. Main calculations are done by looping all the neighboring edges of a vertex and accumulated to give the pagerank of that vertex. These **Ranking** computations can be considered as non-crucial because in case one calculation gets perturbed the remaining unperturbed accumulations still produce a viable pagerank. As the pageranks determined are normalized to an upper bound of "1", I include a bound check at the end of each pagerank calculation. Moving further, the calculations within the loop are unrolled to reduce resilience switching overhead. Accuracy is determined by comparing the pagerank value of each vertex.

Single Source Shortest Path (SSSP)

SSSP computes shortest paths for graphs with non-negative edge weights. The algorithm starts from a user-defined vertex, and hops over all the vertices in the graph, updating neighboring vertices with lowest path costs from the starting vertex. The work is divided using the range based parallelization strategy [62] [63] amongst threads. Each thread works on a chunk of allocated vertices. The distance update must be done with atomics, since two threads can update the same vertex from different paths. Although the distance calculation can be classified as non-crucial, this would introduce frequent resilience switching overheads, because locks need to be protected by HaRE.

Based on the parallelization strategy, the graph is traversed multiple times (256), due to the fact that later updated vertices can affect the distance of earlier ones. When having certain traversals (towards the end of program execution) protected by HaRE, the accuracy loss is minimized because

the later traversals can possibly recover the wrong distance due to soft-error in previous ones. Thus initially **80% Iterations** are defined as non-crucial. Accuracy is defined as the percentage differences between perturbed (error injected) and optimal (no error) shortest path distances for each vertex.

Breadth First Search (BFS)

BFS searches for a target vertex in a given graph, while doing "neighbors first" type search. Similar to SSSP, work is dynamically divided into frontiers amongst threads, where each thread works on a chunk of allocated vertices within the current frontier. It traverses all the neighboring edges of the current vertex. Lock-based updates are needed to avoid race conditions. Vertex updates (**Vertex Check**) can be considered as non-crucial. Soft-errors can cause vertices to become unchecked even though they were checked earlier, however it is highly possible that these vertices will be checked again by another thread due to edge sharing. The accuracy is calculate as " number of correctly checked vertices / total number of vertices".

Depth First Search (DFS)

DFS uses a branching parallelization strategy to search vertices in a first-come first-served manner using disjoint stacks. Each thread maintains a stack to keep record of the vertices checked. Similar to BFS, the computations within of each vertex (**Vertex Check**) can be considered as non-crucial. Moreover, if a wrong vertex (outside the vertex set) is pushed into the stack due to soft-error, a bound checker will drop the iteration, which leaves that vertex as unchecked. Non-crucial computations are also unrolled to increase the non-crucial region for overcoming resilience switching overheads to allow greater performance benefits. Program accuracy is determined similarly as done in BFS.

Community Detection (COMM)

Community Detection uses the Louvain method [64] to detect communities, and uses graph division to parallelize vertices amongst threads. It optimizes modularity, which is a measure of connectivity in a graph, which is later used to detect communities. This algorithm runs in three primary phases. The first (**Mod**) phase finds the maximum modularity gain for each vertex. The modularity calculation is computationally intensive and can be considered as non-crucial. A bound checker is implemented for each modularity calculation, to ensure it is not perturbed to very large value due to soft-errors. The bound value can be found in literature [65], in this case $Max\_Edges * Max\_Deg$ is used. The second (**Recons**) phase reconstructs the graph based on the computed modularities. This step requires a subtraction between calculated community values, which are then written to the source graph. Locks are required for updating shared edges. Perturbations due to soft-errors can also write detrimental values to the graph, which can further propagate. The final (**Reduc**) phase allocates communities for all vertices using a community reduction heuristic. It does not require locks, and can be protected using a bound checker (value: number of vertices). Accuracy checking for COMM is done by comparing the modularities.

Connected Components (CONN_COMP)

Connected Components is used primarily to measure connected regions in image graphs, and in clustering applications. The classic Shiloach-Vishkin (SV) algorithm [66] is used, while the parallelization applied is graph division, which divides a graph's vertices amongst threads. This requires atomically updating shared edges. The **Denoting** process is defined as non-crucial. Due to program structure, this process has many in-direct accesses to the cluster array. These are resolved by added bound checkers, which ensure the indexes of the array are always within range. The program accuracy is defined as the percentage of vertices with correct connected component values.

D* (D-STAR)

The D-STAR algorithm is a popular incremental search algorithm that applies heuristics to speedup Dijkstra's Algorithm, while keeping a proof of convergence [67]. Approximate heuristic distance, from the current vertex to the destination, steers the path finding. It only checks and selects the next best vertex from neighbors. The shortest path (**Distance**) calculation of each vertex can be defined as non-crucial, and there are no locks in the benchmark. Accuracy is quantified in the same fashion as for SSSP: the percentage differences between perturbed and optimal shortest path distances for each vertex.

Betweenness Centrality (BTW_CENT)

The Betweenness Centrality benchmark identifies important vertices in a graph. The first (**Distance**) phase computes all the shortest paths in a graph between all the pairs of vertices. Then the second (**Centr**) phase identifies the number of shortest paths passing through each given vertex. It is statically divided amongst threads, with each thread reading shortest path values and updating the centralities via atomic locks. The accuracy analysis is done by comparing the centrality of each vertex.

4.4.3 Accuracy Threshold Selection

The heuristic accuracy analysis requires the programmer/user to provide an accuracy threshold to mark certain regions as crucial or non-crucial. In the context of machine learning applications, the accuracy threshold can be decided based on frames per second (fps) parameter. Consider an example of a self-driving car (a real-time system) processing images via CNNs with a certain fps rate to predict the next action. The fps rates range from 30 to 60 frames per second [68]. Higher

the fps rate, better and safer would be the decision made by the CNN. Depending on the conditions around the vehicle, the fps value can be allowed to change. Suppose, when the traffic is normal, even lower fps rates should allow the neural network to predict a close-to optimal action. This is because in a normal traffic the changes happening on the road are rare and CNN would be processing redundant images most of the times. Hence, losing some frames (for example 2% of the total – 1 image) from the image stream should not impact the output of the system to much extent. Similarly, in terms of graph analytic, graph applications are tolerant to random errors, as long as the nodes with higher connectivity are not affected [69]. Injecting single error in the graph applications should not have a significant impact on the final response. This is because the probability of the single injected error effecting the highly connected nodes is very less. Hence, 1% accuracy threshold can be a reasonable verge. However, the selection of accuracy threshold depends on the time, performance, and energy constraints provided by real-time system and the threshold would vary from one condition to the other.

Chapter 5

Evaluation

5.1 Simulation Methods

5.1.1 Performance Analysis Setup

Graphite multicore simulator [37] is used to model the cross-layer architecture. The default architecture parameters are summarized in Table 5.1.1. All hardware resiliency mechanisms proposed in [19] are modeled. The hardware-software interface is implemented using a special function instrumented in the application, which passes the HaRE on/off pragma to the simulator. With the in-order single-issue core setup, I assume a five-stage pipeline. The HaRE "on" switch needs 3 cycles to create a safe state and start capturing signatures. For HaRE "off" switch, the pipeline needs to be flushed with an additional 1 cycle delay to re-execute and check the previous instruction sequence. Redundant store address calculation (SHR) incurs one cycle delay since, it needs to stall the compute pipeline. It is only enabled when HaRE is "off", and uses existing HaRE hardware to check the results. The data checker (SHR) is implemented in the application using regular

Architectural Parameter	Value
Core(s)	64 (In-Order)
Memory Subsystem	
L1-I/D Private Caches (per Core)	32 KB, 4-way Set Assoc., 1 cycle latency
L2 Shared Cache (per Core)	256 KB, 8-way Set Assoc., 8 cycle latency, Inclusive
Coherence Protocol	Directory, Invalidation-based, MESI
DRAM Memory Interface	8 controllers, 5 GBps/controller, 100 ns latency
Electrical 2-D Mesh with XY Routing	
Hop Latency	2 cycles (1-router, 1-link) + link contention
Flit Width	64 bits

Table 5.1.1: Architectural Parameters.

instructions, so it adds to the instruction footprint but incurs no additional hardware overhead. The completion time is broken into following categories:

1. *Instructions:* Time spent retiring instructions.

2. *L1-I Fetch Stalls:* Stall time due to instruction cache misses.

3. *Compute Stalls:* Stall time due to waiting for functional unit (ALU, FPU, Multiplier, etc.) results.

4. *Memory Stalls:* Stall time due to load/store queue capacity limits, fences and waiting for load completion.

5. *Branch Speculation:* Stall time due to mispredicted branch instructions.

6. *Synchronization:* Stall time due to waiting on locks, barriers, and condition variables.

7. *Resilience:* Stall time due to hardware level resilience schemes.

The analysis is based on the fact that non-crucial code regions should only contain compute instructions, load instructions, and store instructions that access temporal variables. For example the non-crucial code region in the convolutional layer of CNN-MNIST only contains following X86 instrutions: movl, addl, cltq, salq, addq, movq, subl, movslq, leaq, movsd, mulsd, addsd.

5.1.2 Accuracy Analysis Setup

The heuristic accuracy analysis described in Section 4.3 is based on the fact that non-crucial code regions should only contain compute instructions, load instructions, and store instructions that access temporal variables. According to the soft-error effects in non-crucial regions, we believe injecting error(s) to the variable(s) before committing to global is sufficient. Considering soft-errors as bit-flips, which can happen anywhere in the non-crucial region and have unpredictable effects to the variables about to commit, we mimic them using random values. As in reality occurrence of soft-errors is rare [70], the primary focus is on the case that single soft-error happens during the program's execution. However, in order to evaluate the possible soft-error effects in more extreme cases and explore the accuracy tradeoff, aggressive error rates are used. The error rate is applied as the probability of single iteration being perturbed. For example, if there are total 1000 iterations in the non-crucial region, when the error rate is 1%, on average there would be 10 iterations with random values. In case there are both crucial and non-crucial instructions in the iterations, errors are injected with a probabilistic distribution based on the execution time of the non-crucial instructions. The random value ranges are determined based on the data types of the variables in order to cater for the possible bit-flip scenarios. For example, if the data type is *double*, random values ranging from 1.7E +/- 308 are injected. The results are shown for the single error injection analysis.

Application	Setups
Machine Learning	
CNN-VGG (IMAGENET [71])	16 convolutional, 3 fully connected layers
CNN-ALEXNET (IMAGENET [71])	5 convolutional, 3 fully connected layers
CNN-GTSRB (GTSRB [72])	2 convolutional, 2 fully connected layers
CNN-MNIST (MNIST [73])	1 convolutional, 1 fully connected layer
MLP-MNIST([73])	2 intermediate layers
KNN-MNIST([73])	5 nearest neighbors
Graph Analytic (CRONO [1])	
SSSP, D-STAR, BFS PAGERANK, TRI-CNT, DFS, CON-COMP, COMM, BTW-CENT	California Road Network (Sparse Graph Input), Mouse Brain Retina 3 (Dense Graph Input)

Table 5.1.2: Benchmark Setup.

5.1.3 Benchmark and System Setups

The benchmark setups are shown in Table 5.1.2. We evaluate nine graph analytic benchmarks from CRONO suite [1], with different graph inputs to evaluate input dependency. California road network [74] is used as a sparse input, and mouse brain graph [75] as a dense input. For MNIST machine learning benchmarks (CNN, MLP, KNN) handwritten digit dataset [73] is used. CNN-GTSRB uses German traffic sign [72] as an input. Imagenet [71] is provided as an input to CNN-ALEXNET and CNN-VGG.

The following section discusses the results. There are following system setups in the evaluation:

1. **BASELINE** is the original program without any resilience schemes.

2. **HaRE** redundantly executes all instructions.

3. **DR** is the proposed cross-layer architecture based on declarative resilience architecture,

Benchmark	Layers/Regions	Accuracy Loss (%)
CNN-ALEXNET	C1	0.3
	C2	0.4
	C3	1.9
	C4	0.4
	C5	0.3
	F1	0.8
	F2	1.0
	F3	1.3
	Output	13.1

Table 5.2.1: Program accuracy of CNN-ALEXNET when single error is injected in the each region (over 10000 runs). (C - Convolutional Layer, F - Fully Connected Layer, Output - Output Layer)

which selectively applies HaRE and SHR.

5.2 Results

5.2.1 Non-crucial Region Selection

The selection of code regions as non-crucial is considered based on the heuristic described in Section 4.3. Let us take the example of CNN-ALEXNET to illustrate this selection process. As shown in Table 5.2.1, all convolutional (C1 – C5) and fully connected (F1 – F3) layers in CNN-ALEXNET have relative low accuracy loss. When single error is injected in these layers, the maximum accuracy loss of 1.9% is observed. In contrast, the output layer observes a loss of more than 10%. The programmer sets an accuracy threshold for the selection heuristic to mark certain code regions as crucial. The accuracy loss threshold is set to 2% (for both machine learning and graph analytic workloads), which classifies the output layer as crucial and all convolution and fully connected layers as non-crucial.

In order to explore the accuracy tradeoff of selected non-crucial regions, aggressive error rates were also considered. However, the results for single error injection in the performance analysis are shown. Overall, DR is able to select configurations with reasonable amount of non-crucial code, resulting in low accuracy loss even with high error rates. The accuracy loss in different benchmarks highly depends on the code structure and the functionality. For code regions with similar functionality in a benchmark, the more execution time the code region has, the more accuracy loss it may contribute. This behavior is observed in the convolutional layers of CNN. However, this pretext would not hold if the code regions have different functionalities, such as the output layer in all machine learning benchmarks. It does not have much execution time, but contributes a relatively large amount of accuracy loss. This is because errors in the output layer could directly affect the final program outcome.

Table 5.2.2 and 5.2.3 show the percentage time spent in non-crucial code and the percentage accuracy loss of the selected configurations at a single injected error per program execution. All machine learning benchmarks consider their output layers as crucial. The second column in Table 5.2.2 represents which layers of the benchmark were considered as non-crucial. For example "C:1-5 + F: 1-3" for AlexNet depicts that all convolutional layers from 1 to 5 were selected along with all the fully connected layers from 1 to 3. Moreover, the selected non-crucial code regions account for, overall, $>75\%$ of the execution time, while the accuracy loss is observed at less than 1% (satisfying the threshold).

For the graph benchmarks, when different input graphs are applied, the observed accuracy loss is relatively stable, while the time spent in the selected non-crucial code regions vary significantly. For example, for the path finding heuristic D-STAR, edges are involved in the distance calculation. A sparse graph has fewer edges per vertex, while a dense graph has much higher edge connectivity. Therefore, the work done in non-crucial region for a dense graph is much larger for a dense graph as compared to a sparse graph. However, the accuracy loss of the selected non-crucial code region

Applications	Selected Non-Crucial Regions	Non-Crucial Time (%)	Accuracy (%)
CNN-ALEXNET	C: 1-5 + F: 1-3	91	99
CNN-VGG	C: 1-16 + F: 1-3	92.5	98.8
CNN-GTSRB	C: 1,2 + F: 1,2	88	99.1
CNN-MNIST	C + F	87	99.3
MLP-MNIST	I: 1,2	75	99.91
KNN-MNIST	Distance	94	99.9

Table 5.2.2: Selected configurations of machine learning benchmarks when single error is injected in the program (over 10000 runs). (C - Convolutional Layer, F - Fully Connected Layer, I - Intermediate Layer.)

remains under 1% for both input graphs. Similar behavior is observed for other graph benchmarks, where the worst-case accuracy loss is observed at 1.8% among all graph benchmarks. The time spent in non-crucial code regions is directly proportional to the expected performance gains from executing the graph benchmarks under the proposed DR architecture.

5.2.2 Performance

Under normal conditions, the probability of soft-error strikes is very low (less than 1 per day for the current technology node [76]). In order to reveal the soft-error effects on program execution, an aggressive error rate of 0.1% is used. The accuracy threshold is defined as 90% to select the proper configurations. The completion time of selected configurations of each application is plotted in Figure 5.2.1.

The BASELINE completion time for most benchmarks is dominated by memory stall and/or synchronization. This is due to benchmarks' inherent characteristics: numerous memory accesses to shared data, and heavy contentions on locks or barriers. For graph analytic workloads, such as SSSP and BFS, results show higher synchronization delay when using the dense graph input. This

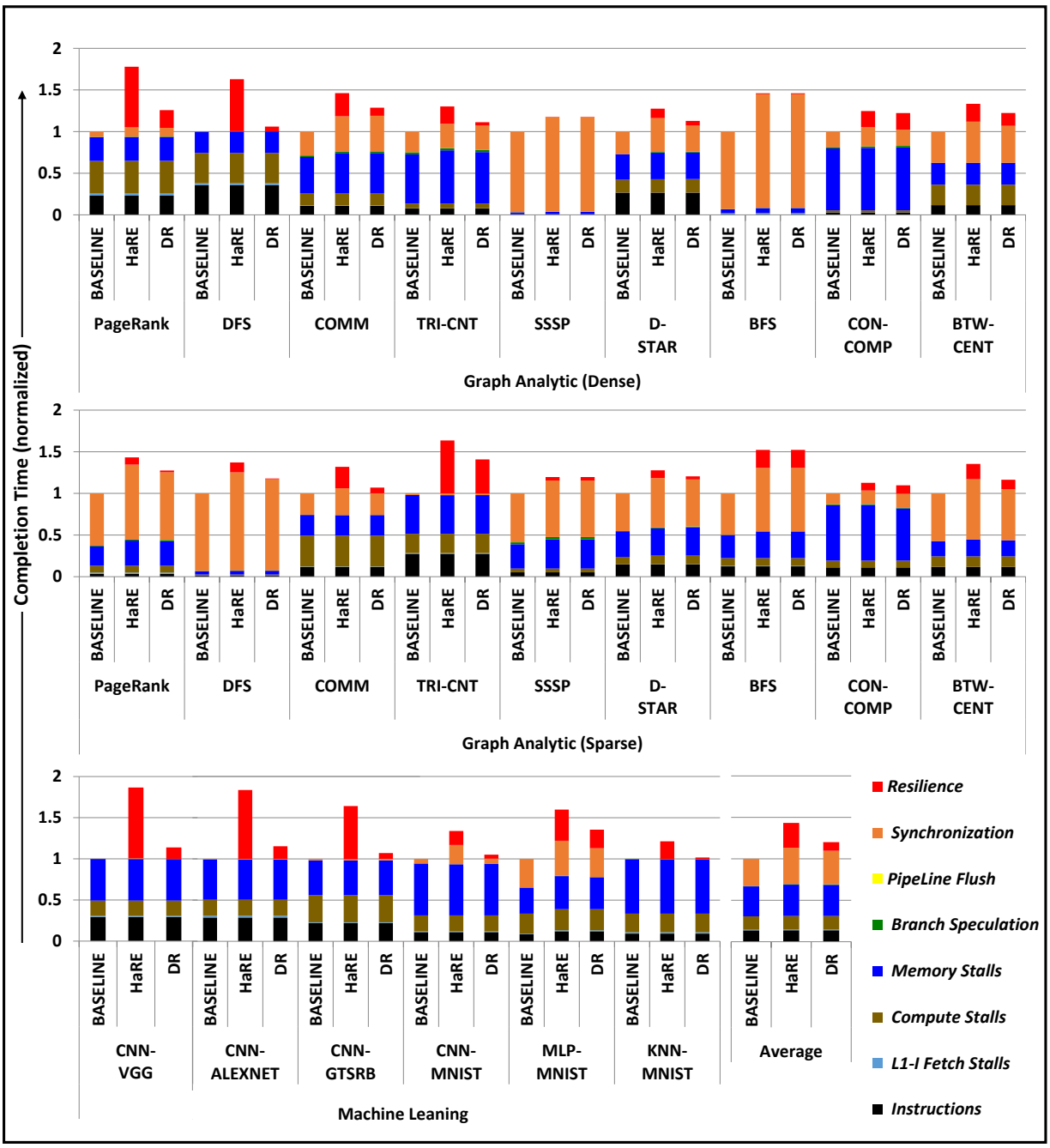

Figure 5.2.1: Completion time of selected configurations using proposed cross-layer architecture.

Applications	Selected Non-Crucial Regions (Heuristic Analysis)	Non-Crucial Time Time (%)		Accuracy (%)	
		Sparse	Dense	Sparse	Dense
SSSP	80% Iterations	58	64	98.2	98.91
D-STAR	Distance	72	91	99.7	99.45
PageRank	Ranking	74.3	78	99.9	99.3
TRI_CNT	AddEdges, Redu, GetTri	89	86	99.4	99.05
BFS	Vertex Check	21.4	27.8	99.97	99.7
DFS	Vertex Check	77	81	99.9	99.6
COMM	Mod, Recons, Reduce	88	89	99.76	99.29
CON-COMP	Denoting	58	74	98.87	98.38
BTW-CENT	Distance, Centr	84	86	99.86	99.34

Table 5.2.3: Selected configurations of graph benchmarks when single error is injected in the program (over 10000 runs). The non-crucial region names represent their functionalities as in CRONO [1]

is because of higher number of shared edges, which causes more lock contention. For benchmarks like PageRank and DFS, more time is spent in synchronization for the sparse graph input. This is because they use coarse-grain locks over the vertices (not the edges).

HaRE performs reasonably well for most benchmarks, because it performs local redundant execution and exploit core-level locality. Moreover, it hides re-execution latency behind cache miss stalls. Memory stalls of HaRE are increased over BASELINE, because memory operations trigger re-executions (such as invalidating a cache line). Messages need to wait until the re-execution completes. The synchronization of HaRE also increases over BASELINE due to the re-execution time of instructions within locks or barriers.

When applying DR, all machine learning applications show remarkable performance improvement over HaRE. DR reduces the completion time of CNN significantly compared to HaRE (from 1.83× to 1.15× for ALEXNET). This is because HaRE is not able to hide resilience overheads, meanwhile a major amount of computations is identified as non-crucial in DR. For benchmarks

with high resilience overhead in HaRE, DR is always able to reduce it. Note that the time spent in non-crucial regions (shown in Tables 5.2.3 and 5.2.2) is not directly proportional to the performance gain in the proposed architecture across benchmarks. This is mainly due to the following reasons. (1) The non-crucial time reported in Tables 5.2.3 and 5.2.2 refers to the execution time of the whole layer/phase, which includes locks and checkers. Thus the execution time of actual non-crucial instructions could be much smaller. (2) For benchmarks having highly contended fine-grain locks, such as SSSP and BFS, their resilience overhead is hidden behind synchronization delay. Although DR reduces the re-execution within locks, they may not overcome the delay added due to software level checker and on/off switching of HaRE. (3) For benchmarks with coarse-grain locks, such as DFS, DR is able to reduce the synchronization delay (sparse input) because of the preferable longer instruction sequence within locks. In the case of COMM, DR causes workload imbalance between threads and slightly increases the synchronization delay of barriers. I observe that SSSP and BFS do not provide much performance benefit. This is mainly due to the fact that most of the computations are done within the locks, which is why there is high synchronization in the reported results. The computations done within the locks can be considered non-crucial. However, synchronization instructions such as barriers and locks need to be protected. This limits us from unrolling the non-crucial instructions and thus, not much benefit is observed from the proposed architecture. In case of CON-COMP, the performance gain over HaRE is not much because of the structure of the algorithm. This algorithm contains many indirect accesses of form $A[A[i]]$, which need to be considered as crucial. Protection of such instructions leaves very little number of non-crucial instructions that cannot be unrolled for performance benefits. Overall, the proposed DR architecture shows significant performance improvement over HaRE. It reduces the performance overhead of resiliency from $\sim$1.43$\times$ to $\sim$1.2$\times$ on average.

5.2.3 Configuration Selection

Soft-error rates change due to different conditions in real world environmental conditions. In addition, the acceptable accuracy loss is determined from case to case. I introduce heuristic accuracy analysis (Section 4.3) in this context to help the programmer select the proper configurations. Figure 5.2.2 shows the performance improvements over HaRE of selected configurations, at different accuracy thresholds with different error rates. Some benchmarks, such as D-STAR, have relatively low accuracy loss ($<= 1\%$ for D-STAR) in the original configurations, which can meet the lowest accuracy threshold (3%). Their configurations are not changed when applied with different error rates and accuracy thresholds. Thus their performance numbers remain constant in this analysis, and are not shown in the figure.

As shown in Figure 5.2.2, three different accuracy thresholds are applied, from top to bottom, 10%, 5%, and 3% accordingly. In general, a higher accuracy threshold and lower error rate results in configurations with more non-crucial code regions. For example, when using an accuracy threshold of 10%, AlexNet is able get the best performance at 0.1% or lower error rates. According to Table 5.2.2, all of its convolutional and fully connected layers are considered as non-crucial at this point, which contribute 91% of the program's execution time. Thus the proposed cross-layer architecture has great performance improvement over HaRE (37%). As the error rate increases, the same configuration cannot satisfy the 10% accuracy threshold. It gets to 31% at an error rate of 0.5%, which was 37% earlier (error rate of 0.1%). Using the proposed heuristic accuracy analysis (Section 4.3), I am able to get the accuracy loss back by making the 5th convolutional layer and the 3rd fully connected layer crucial. However, this results in less non-crucial code (76% of execution time), thus has less performance improvement (26%). In extreme cases, the performance improvement can reduce to 0%. This means all code regions have to be considered as crucial, which is effectively HaRE.

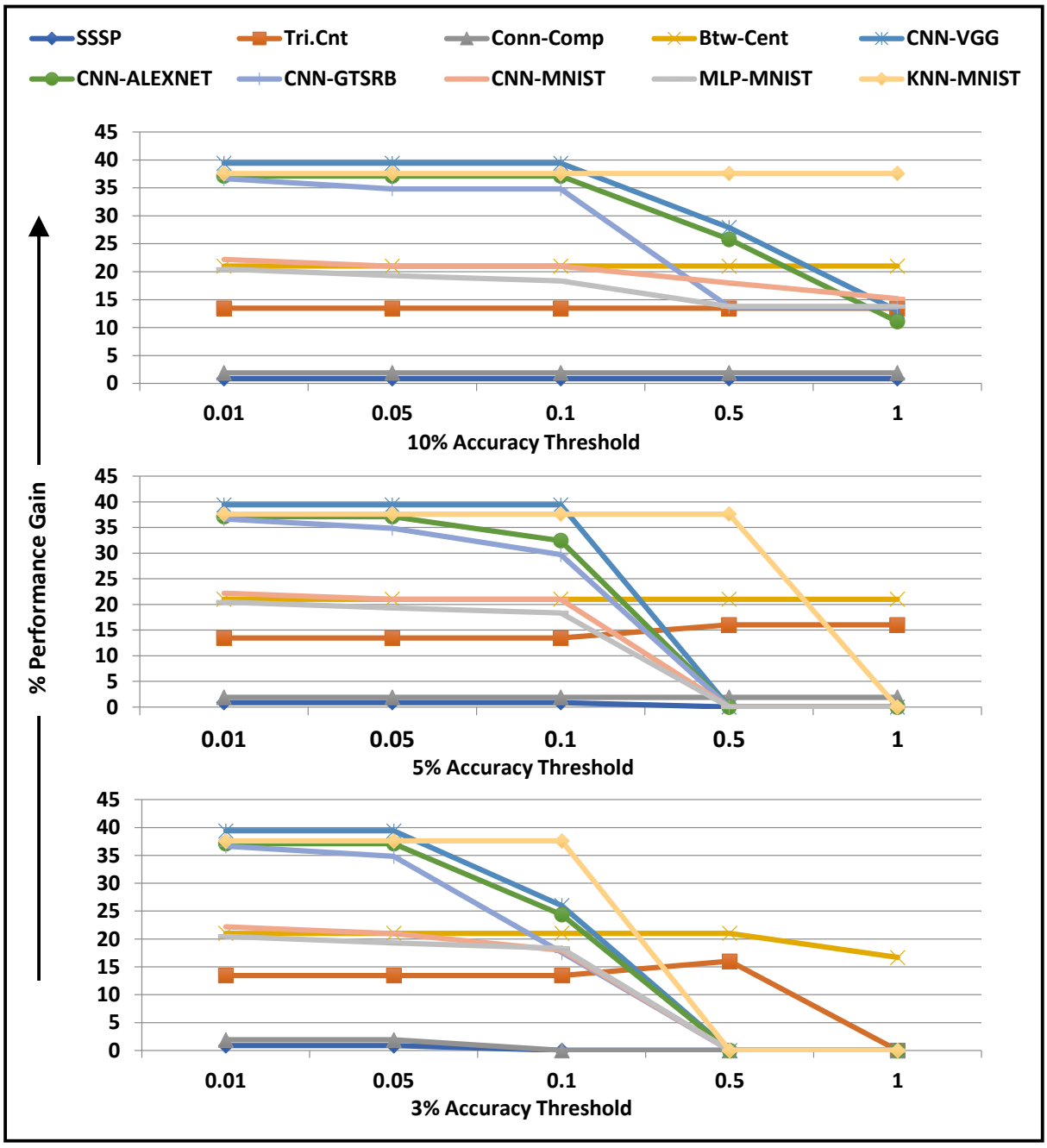

Figure 5.2.2: Performance improvement over HaRE at 10%, 5% and 3% accuracy thresholds with different error rates.

Chapter 6

Related Work

6.1 Resilience Scheme

Symptom-based mechanisms [77, 78] have low coverage since they rely on coarse-grain detectors such as fatal-traps, hangs, panics etc. However, they incur low area, power and performance overheads. *Software* solutions such as instruction duplication [13, 14, 15], and invariant checking [17, 52, 18] improve coverage, but incur higher overheads. To deliver 100% coverage, several proposals utilize temporal and spatial redundant execution. Redundant multithreading [79, 80] uses the processor's symmetric multithreading contexts to run two copies of the same thread, where the trailing thread verifies the results of the leading thread. This approach is generalized as *n-modular redundancy* [81, 82, 20, 16] where *n* copies of the same thread are executed and verified in parallel. All *n-modular* techniques incur significant performance and energy overheads because multithreaded applications are unable to exploit the hardware's thread-level parallelism. To improve performance, researchers have explored selective resilience within applications [21, 22]. These schemes obtain efficiency by providing high resiliency for high vulnerability code; however, they

tradeoff performance with soft-error coverage.

6.2 Approximate Computing

Research has explored approximate programming [30, 31, 32, 33], which relaxes program accuracy when possible. Esmaeilzadeh et al. [30] focused on developing a language to generate code that executes on approximate hardware. The proposed architecture is different from these works mainly in the following two aspects. First, the proposed approach is not limited to approximate hardware and is more general i.e., find code regions of a program that can be potentially ”approximated” without significantly impacting program outcomes. Unlike approximate computing schemes, the proposed architecture does not actively relax accuracy. Program accuracy may be compromised only when soft-errors perturb non-crucial code. Second, soft-errors introduce new challenges since code in the non-crucial regions can unexpectedly affect other parts of the program and compromise program correctness, thus identifications for approximate computing cannot be directly applied to declarative resilience. However, the proposed architecture can benefit from profiling schemes of approximate computing, which can assist programmers to identify potential crucial/non-crucial code regions.

6.3 Crucial/Non-Crucial Code Identification

Works have been done on selecting crucial/non-crucial code of a program using different criteria, such as Rely [83], which is a programming language that enables developers to specify the reliability requirements for program functions. They verify the program reliability with respect to the specification of the underlying hardware system. In terms of code region selection, the proposed

declarative resilience framework is mainly different from previous schemes in the following ways. (1). In declarative resilience, programmers are involved in identifying the code regions, however they cannot actively reduce reliability. If it is possible for some code to have critical impact on program outcome, it cannot be marked as non-crucial. (2). Research (such as SoftBeam [84]) points out that different microarchitectures as well as different instructions have different inherent soft error vulnerability. However, in declarative resilience I do not classify code according to its hardware level vulnerability. This is based on the insight that vulnerability cannot be used directly to guide the code's impact on the program. High vulnerability code may not have severe effects. On the other hand, soft errors in low vulnerability code may still crash the program, although the possibility of the soft error happening is low.

6.4 Algorithm Level Accuracy Tradeoff

Because of the algorithm complexity, and system timing constraints, works have been done on fast algorithms with lower accuracy [24, 25, 26]. These algorithm level schemes have better performance, however they do not provide protections to soft-errors. The proposed framework can be applied on top of such schemes seamlessly for resilience purpose. Due to their inherent accuracy relaxation, the proposed framework may bring less impact to the program outcomes. Algorithms, such as D* [85] approximates path planning and provides better performance than exact algorithms with relaxed accuracy. Such algorithms open avenues to apply declarative resilience to new domains, if program accuracy can be relaxed actively.

Chapter 7

Summary

Multicore technology is increasingly being deployed in numerous areas, from traditional data center systems to emerging ones, such as unmanned aerial vehicles (UAVs) and self-driving cars. Conventional hardware/software resiliency schemes can provide high soft-error protection, however with high overhead. Hardware Redundant Execution (HaRE) is developed to deliver high coverage. It relies on a local per-core checkpoint and rollback mechanism to recover from detected errors. This scheme is advantageous, because it (1) fully exploits multicore parallelism; (2) minimizes the latency and energy consumption of redundant execution by exploiting locality on the local core; (3) hides the latency of redundant execution, for example, by initiating redundant execution on long latency private cache misses.

Based on the analysis of soft-error effects to program control flow and data flow, I observe that not all transient errors affect program correctness, some errors only affect program accuracy, i.e., the program completes with certain acceptable deviations from error free outcome. Moreover, certain applications running on such systems have inherent resilience due to their unique structure and computational behaviors. Popular machine learning and graph analytic applications work on

massive data and perform perception computations. They can be used in many applications, including image recognition, video analysis and natural language processing. Such applications have the potential to be deployed in safety-critical systems, where they face harsh conditions that lead to vulnerability against transient perturbations in the hardware system. It is practical to improve efficiency by trading off resilience overheads with program accuracy.

This thesis introduces a novel declarative resilience framework for graph analytics and machine learning applications. The key idea is to explore resilience overhead tradeoff with program accuracy, while not compromising soft-error coverage and safe execution of the program. On top of the framework, I developed a cross-layer resilient architecture. It guarantees program correctness, while only incurs $\sim$1.2$\times$ performance overhead over a system without resilience. This is an average 16% performance improvement over state-of-the-art hardware resilience scheme that protects the whole program.

The potential future directions for this work are discussed below.

- *Automation*: Currently, profiling non-crucial code, applying resilience on/off pragmas and data checkers are mainly done by the programmer with systematic assist. This requires extra work for the programmer when developing a resilient application and is prone to errors. However, the formulation of declarative resilience can be generalized in an unambiguous way. A compiler or a runtime tool can do the work for the programmer.

- *Generality*: Due to similarity of coding style, many applications are expected to have heavily executed non-crucial code, which should benefit from declarative resilience. Thus, this work can be applied to a wider range of parallel applications, also with different systems, such as GPUs. Other than fault injection at program level, there are established reliability analysis techniques such as architectural/program vulnerability factor and failure analysis. It would be beneficial to validate the framework using such techniques as well.

Bibliography

[1] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan. Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores. In *Workload Characterization (IISWC), 2015 IEEE International Symposium on*, pages 44–55, Oct 2015.

[2] Subhasish Mitra, Tanay Karnik, Norbert Seifert, and Ming Zhang. Logic soft errors in sub-65nm technologies design and cad challenges. In *Proceedings of the 42nd annual Design Automation Conference*, DAC '05, pages 2–4, 2005.

[3] Alaa R. Alameldeen, Ilya Wagner, Zeshan Chishti, Wei Wu, Chris Wilkerson, and Shih-Lien Lu. Energy-efficient cache design using variable-strength error-correcting codes. In *Proceedings of the 38th annual international symposium on Computer architecture*, pages 461–472, 2011.

[4] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. Shoestring: Probabilistic soft error reliability on the cheap. *SIGPLAN Not.*, 45(3):385–396, March 2010.

[5] V. Roberge, M. Tarbouchi, and G. Labonte. Comparison of parallel genetic algorithm and particle swarm optimization for real-time uav path planning. *IEEE Transactions on Industrial Informatics*, 9(1):132–141, Feb 2013.

[6] J. Kim, H. Kim, K. Lakshmanan, and R. Rajkumar. Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car. In *Cyber-Physical Systems (ICCPS), 2013 ACM/IEEE International Conference on*, pages 31–40, April 2013.

[7] A. Vega, C. C. Lin, K. Swaminathan, A. Buyuktosunoglu, S. Pankanti, and P. Bose. Resilient, uav-embedded real-time computing. In *2015 33rd IEEE International Conference on Computer Design (ICCD)*, pages 736–739, Oct 2015.

[8] R. Viguier, C. C. Lin, K. Swaminathan, A. Vega, A. Buyuktosunoglu, S. Pankanti, P. Bose, H. Akbarpour, F. Bunyak, K. Palaniappan, and G. Seetharaman. Resilient mobile cognition: Algorithms, innovations, and architectures. In *2015 33rd IEEE International Conference on Computer Design (ICCD)*, pages 728–731, Oct 2015.

[9] P. Sermanet and Y. LeCun. Traffic sign recognition with multi-scale convolutional networks. In *Neural Networks (IJCNN), The 2011 International Joint Conference on*, pages 2809–2813, July 2011.

[10] I. Sato and H. Niihara. Beyond pedestrian detection: Deep neural networks level-up automotive safety. In *GPU Technology Conference*, 2014.

[11] H. Li, D. Song, Y. Lu, and J. Liu. A two-view based multilayer feature graph for robot navigation. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 3580–3587, May 2012.

[12] Qingchuan Shi, Kartik Lakshminarashimhan, Christopher Noll, Eelco Scholte, and Omer Khan. A lightweight spatio-temporally partitioned multicore architecture for concurrent execution of safety critical workloads. In *SAE Technical Paper*. SAE International, 09 2016.

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

[14] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. Swift: software implemented fault tolerance. In *International Symposium on Code Generation and Optimization*, pages 243–254, March 2005.

[15] N. Oh, S. Mitra, and E. J. McCluskey. Ed4i: error detection by diverse data and duplicated instructions. *IEEE Transactions on Computers*, 51(2):180–199, Feb 2002.

[16] J. Wadden, A. Lyashevsky, S. Gurumurthi, V. Sridharan, and K. Skadron. Real-world design and evaluation of compiler-managed gpu redundant multithreading. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 73–84, June 2014.

[17] A. Meixner, M. E. Bauer, and D. J. Sorin. Argus: Low-cost, comprehensive error detection in simple cores. *IEEE Micro*, 28(1):52–59, Jan 2008.

[18] V. Reddy and E. Rotenberg. Coverage of a microarchitecture-level fault check regimen in a superscalar processor. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 1–10, June 2008.

[19] Q. Shi and O. Khan. Toward holistic soft-error-resilient shared-memory multicores. *Computer*, 46(10):56–64, October 2013.

[20] M. W. Rashid and M. C. Huang. Supporting highly-decoupled thread-level redundancy for parallel programs. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 393–404, Feb 2008.

[21] S. Rehman, F. Kriebel, Duo Sun, M. Shafique, and J. Henkel. dtune: Leveraging reliable code generation for adaptive dependability tuning under process variation and aging-induced effects. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2014.

[22] T. Li, M. Shafique, J. A. Ambrose, S. Rehman, J. Henkel, and S. Parameswaran. Raster: Runtime adaptive spatial/temporal error resiliency for embedded processors. In *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pages 1–7, May 2013.

[23] Konstantinos Aisopos and Li-Shiuan Peh. A systematic methodology to develop resilient cache coherence protocols. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 47–58, New York, NY, USA, 2011. ACM.

[24] Yangguang Fu, Mingyue Ding, and Chengping Zhou. Phase angle-encoded and quantum-behaved particle swarm optimization applied to three-dimensional route planning for uav. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, 42:511 – 526, 2012.

[25] G. R. Jagadeesh, T. Srikanthan, and K. H. Quek. Heuristic techniques for accelerating hierarchical routing on road networks. *IEEE Transactions on Intelligent Transportation Systems*, 3(4):301–309, Dec 2002.

[26] L. Murphy and P. Newman. Risky planning: Path planning over costmaps with a probabilistically bounded speed-accuracy tradeoff. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 3727–3732, May 2011.

[27] S. Rehman, F. Kriebel, Duo Sun, M. Shafique, and J. Henkel. dtune: Leveraging reliable code generation for adaptive dependability tuning under process variation and aging-induced effects. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2014.

[28] T. Li, M. Shafique, J. A. Ambrose, S. Rehman, J. Henkel, and S. Parameswaran. Raster: Runtime adaptive spatial/temporal error resiliency for embedded processors. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–7, May 2013.

[29] D. S. Khudia and S. Mahlke. Harnessing soft computations for low-budget fault tolerance. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 319–330, Dec 2014.

[30] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Architecture support for disciplined approximate programming. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 301–312, New York, NY, USA, 2012. ACM.

[31] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. Quality of service profiling. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 25–34, May 2010.

[32] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 124–134, New York, NY, USA, 2011. ACM.

[33] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 449–460, Washington, DC, USA, 2012. IEEE Computer Society.

[34] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[35] Q. Shi, H. Hoffmann, and O. Khan. A cross-layer multicore architecture to tradeoff program accuracy and resilience overheads. *IEEE Computer Architecture Letters*, 14(2):85–89, July 2015.

[36] C. Bienia, S. Kumar, J.P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*, 2008.

[37] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12, Jan 2010.

[38] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th annual International symposium on Computer architecture*, ISCA '00, pages 25–36, 2000.

[39] Mohamed A. Gomaa and T. N. Vijaykumar. Opportunistic transient-fault detection. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, pages 172–183, 2005.

[40] N. Madan and R. Balasubramonian. Power efficient approaches to redundant multithreading. *IEEE Trans. Parallel Distrib. Syst.*, 18(8):1066–1079, August 2007.

[41] M. Wasiur Rashid, Edwin J. Tan, Michael C. Huang, and David H. Albonesi. Exploiting coarse-grain verification parallelism for power-efficient fault tolerance. In *Proceedings of the*

14th International Conference on Parallel Architectures and Compilation Techniques, PACT '05, pages 315–328, 2005.

[42] Jared C. Smolens, Jangwoo Kim, James C. Hoe, and Babak Falsafi. Efficient resource sharing in concurrent error detecting superscalar microarchitectures. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 257–268, Washington, DC, USA, 2004. IEEE Computer Society.

[43] Albert Meixner, Michael E. Bauer, and Daniel Sorin. Argus: Low-cost, comprehensive error detection in simple cores. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 210–222, Washington, DC, USA, 2007. IEEE Computer Society.

[44] Jared C. Smolens, Brian T. Gold, Babak Falsafi, and James C. Hoe. Reunion: Complexity-effective multicore redundancy. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 223–234, 2006.

[45] Sudheendra Hangal, Durgam Vahia, Chaiyasit Manovit, and Juin-Yeu Joseph Lu. TSOtool: A program for verifying memory systems using the memory consistency model. In *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, pages 114–, 2004.

[46] X. Shen and Arvind. Specification of Memory Models and Design of Provably Correct Cache Coherence Protocols. *Group Memo 398, Massachusetts Institute of Technology*, 1997.

[47] Abhisek Pan, Omer Khan, and Sandip Kundu. Improving yield and reliability of chip multiprocessors. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 490–495, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[48] Rance Rodrigues and Sandip Kundu. An online mechanism to verify datapath execution using existing resources in chip multiprocessors. *Asian Test Symposium*, 0:161–166, 2011.

[49] Omer Khan and Sandip Kundu. Thread relocation: A runtime architecture for tolerating hard errors in chip multiprocessors. *IEEE Trans. Comput.*, 59(5):651–665, May 2010.

[50] Michael D. Powell, Arijit Biswas, Shantanu Gupta, and Shubhendu S. Mukherjee. Architectural core salvaging in a multi-core processor for hard-error tolerance. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 93–104, New York, NY, USA, 2009. ACM.

[51] Myong Hyon Cho, Keun Sup Shim, Mieszko Lis, Omer Khan, and Srinivas Devadas. Deadlock-free fine-grained thread migration. In *Proceedings of the Fifth ACM/IEEE International Symposium on Networks-on-Chip*, NOCS '11, pages 33–40, New York, NY, USA, 2011. ACM.

[52] T. M. Austin. Diva: a reliable substrate for deep submicron microarchitecture design. In *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*, pages 196–207, 1999.

[53] Nancy J. Warter and Wen mei W. Hwu. A software based approach to achieving optimal performance for signature control flow checking. In *FTCS*, pages 442–449, 1990.

[54] A. Meixner and D. J. Sorin. Error detection using dynamic dataflow verification. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 104–118, 2007.

[55] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 29–, 2003.

[56] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA*, 1995.

[57] S.M.Z. Iqbal, Yuchen Liang, and H. Grahn. Parmibench - an open-source benchmark for embedded multiprocessor systems. *Computer Architecture Letters*, 9(2):45 –48, feb. 2010.

[58] DARPA UHPC Program BAA. https://www.fbo.gov/spg/ODA/DARPA/CMO/DARPA-BAA-10-37/listing.html, March, 2010.

[59] Q. Shi, H. Omar, and O. Khan. Exploiting the Tradeoff between Program Accuracy and Soft-error Resiliency Overhead for Machine Learning Workloads. *ArXiv e-prints*, July 2017.

[60] Michael A. Nielsen. Neural networks and deep learning. *Determination Press*, 2015.

[61] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[62] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron. Pannotia: Understanding irregular gpgpu graph applications. In *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, pages 185–195, Sept 2013.

[63] Jin Y. Yen. An algorithm for finding shortest routes from all source nodes to a given destination in general networks. *Quarterly of Applied Mathematics*, 27(4):526–530, 1970.

[64] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, 2008.

[65] T. N. Dinh and M. T. Thai. Community detection in scale-free networks: Approximation algorithms for maximizing modularity. *IEEE Journal on Selected Areas in Communications*, 31(6):997–1006, June 2013.

[66] M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on gpus. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pages 141–151, Nov 2012.

[67] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[68] Mariusz Bojarski. End-to-end deep learning for self-driving cars.

[69] Paolo Crucitti, Vito Latora, Massimo Marchiori, and Andrea Rapisarda. Efficiency of scale-free networks: error and attack tolerance. *Physica A: Statistical Mechanics and its Applications*, 320:622–642, 2003.

[70] T. Karnik, B. Bloechel, K. Soumyanath, V. De, and S. Borkar. Scaling trends of cosmic ray induced soft errors in static latches beyond 0.18 /spl mu/. In *2001 Symposium on VLSI Circuits. Digest of Technical Papers (IEEE Cat. No.01CH37185)*, pages 61–62, June 2001.

[71] J. Deng, W. Dong, R. Socher, L. J. Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255, June 2009.

[72] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel. The german traffic sign recognition benchmark: A multi-class classification competition. In *Neural Networks (IJCNN), The 2011 International Joint Conference on*, pages 1453–1460, July 2011.

[73] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.

[74] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *CoRR*, abs/0810.1355, 2008.

[75] J. W. Lichtman, H. Pfister, and N. Shavit. The big data challenges of connectomics. In *Nature Neuroscience 17(11)*, pages 1448–1454, Sept 2014.

[76] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, Lorenzo Alvisi, Ibm Technical, Contacts John Keaty, Rob Bell, and Ram Rajamony. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of International Conference on Dependable Systems and Networks*, pages 389–398, 2002.

[77] N. J. Wang and S. J. Patel. Restore: symptom based soft error detection in microprocessors. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 30–39, June 2005.

[78] Siva Kumar Sastry Hari, Man-Lap Li, Pradeep Ramachandran, Byn Choi, and Sarita 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*, MICRO 42, pages 122–132, New York, NY, USA, 2009. ACM.

[79] 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*, pages 84–91, June 1999.

[80] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multi-threading alternatives. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pages 99–110, 2002.

[81] T. J. Siegel, E. Pfeffer, and J. A. Magee. The ibm eserver z990 microprocessor. *IBM J. Res. Dev.*, 48(3-4):295–309, May 2004.

[82] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. Nonstop reg; advanced architecture. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 12–21, June 2005.

[83] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 33–52, New York, NY, USA, 2013. ACM.

[84] M. Gschwind, V. Salapura, C. Trammell, and S. A. McKee. Softbeam: Precise tracking of transient faults and vulnerability analysis at processor design time. In *Computer Design (ICCD), 2011 IEEE 29th International Conference on*, pages 404–410, Oct 2011.

[85] Anthony Stentz. The focussed d* algorithm for real-time replanning. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, pages 1652–1659, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.