Software Techniques For Dependable Execution

by

Moslem Didehban

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved November 2018 by the
Graduate Supervisory Committee:

Aviral Shrivastava, Chair
Carole-Jean Wu
Lawrence Clark
Scott Mahlke

ARIZONA STATE UNIVERSITY

December 2018

ABSTRACT

Advances in semiconductor technology have brought computer-based systems into virtually all aspects of human life. This unprecedented integration of semiconductor-based systems in our lives has significantly increased the domain and the number of safety-critical applications – application with unacceptable consequences of failure. Software-level error resilience schemes are attractive because they can provide commercial-off-the-shelf microprocessors with adaptive and scalable reliability. Among all software-level error resilience solutions, in-application instruction replication based approaches have been widely used and are deemed to be the most effective. However, existing instruction-based replication schemes only protect some part of computations i.e. arithmetic and logical instructions and leave the rest as unprotected.

To improve the efficacy of instruction-level redundancy-based approaches, we developed several error detection and error correction schemes. nZDC (near Zero silent Data Corruption) is an instruction duplication scheme which protects the execution of whole application. Rather than detecting errors on register operands of memory and control flow operations, nZDC checks the results of such operations. nZDC ensures the correct execution of memory write instruction by reloading stored value and checking it against redundantly computed value. nZDC also introduces a novel control flow checking mechanism which replicates compare and branch instructions and detects both wrong direction branches as well as unwanted jumps. Fault injection experiments show that nZDC can improve the error coverage of the state-of-the-art schemes by more than 10x, without incurring any more performance penalty. Furthermore, we introduced two error recovery solutions. InCheck is our backward recovery solution which makes light-weighted error-free checkpoints at the basic block granularity. In the case of error, InCheck reverts the program execution to the beginning

of last executed basic block and resumes the execution by the aid of preserved information. NEMESIS is our forward recovery scheme which runs three versions of computation and detects errors by checking the results of all memory write and branch operations. In the case of a mismatch, NEMESIS diagnosis routine decides if the error is recoverable. If yes, NEMESIS recovery routine reverts the effect of error from the program state and resumes program normal execution from the error detection point.

This dissertation is dedicated to my exceptional wife. Zohreh, your devotion and support goes beyond the words I can put on this paper.

ACKNOWLEDGMENTS

This dissertation was only made possible by the support and guidance of my advisor Dr. Aviral Shrivastava who his guidance and dedication inspired me over the course of my PhD. I would like to thank my committee members, Dr. Carole-Jean Wu, Dr. Lawrence Clark and Dr. Scott Mahlke for their insights and feedback on my research. I would like to sincerely thank all the researchers in the field of soft error resilience from whom I received knowledge and inspiration.

I would like to thank all members of the Compiler Microarchitectural Lab, both past and present, for providing support and great working environment. I am grateful to all of my friends, especially Adel Dokhanchi, Yooseong Kim and Hwisoo So, for their support and encouragement. Many thanks to you all.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

Chapter 1

INTRODUCTION

Advances in semiconductor technology has made computer-based systems coupled with virtually all aspects of human life – ranging from inside our body systems (e.g., cardiovascular defibrillators or "emergency room in the chest") to close or body (e.g., wearable technology, cell phones and health monitoring devices) to commutation and transportation systems (e.g., autonomous vehicles, autonomous flight Systems and drones). This unprecedented growth of semiconductor-based systems has significantly increased the domain and the number of safety/mission critical applications – applications with unacceptable consequences of failure.

The execution of safety critical applications should be protected against hardware malfunctions specifically random transient faults (Avižienis *et al.*, 2004). Among the many sources of transient faults in the semiconductor devices (e.g., electrical noise, external interference, cross-talk, etc.) sub-atomic energetic particles that strike on sensitive areas of a chip cause majority of transient fault induced failures in electronic devices (Baumann, 2005a). These transient faults or soft errors temporary alter the logical value stored in one (or more) storage element of microprocessor and their impact will usually disappear when the erroneous data is overwritten. In most cases the impact of such temporarily permutations do not affect the system response due to various micro-architectural and software level derating factors (Sanda *et al.*, 2008; Asadi and Tahoori, 2006). In some cases, however, the erroneous value can survive all masking effects and lead to a system level failure – unexpected output or system crash. System failure can have different level of risk based on application reliability requirements. While occasional failures in video game console are trivial, failures

obstacle detection task in autonomous vehicles can lead to a tragedy. For instance, a recent NVIDIA paper (Li *et al.*, 2017), explain situations that soft errors hitting a DNN (Deep Learning Neural Network) can cause crash in self-driving cars. Therefore, soft error mitigation strategies should be adopted to eliminate or reduce the severity of unaccepted failures in safety critical applications.

Traditionally, soft errors considered as a reliability issue for high altitude applications like spacecrafts, satellites and aircraft mainly because of the amount and frequency of high-energy protons and heavy-ion rays in their working environment. Fortunately, once such particles reaches to the earth's atmosphere they cascades to many secondary harmless low-energy particles. Yet, among these secondary particles, neutrons occasionally cause soft error at ground-level applications (Ziegler *et al.*, 1996; May and Woods, 1978; Normand *et al.*, 2010). Additionally, studies show that due to aggressive sub-nano transistor scaling (10nm-7nm) and near-threshold supply voltage, nowadays even low-energy terrestrial particles like muons can cause soft errors (Sierawski *et al.*, 2011; Silberberg *et al.*, 1984; Hubert *et al.*, 2015). International Technology Roadmap for Semiconductors (ITRS) (IRC, 2015) executive report lists ground-level energetic (i.e., muon-induced) particles as a difficult reliability challenge for future microprocessors.

Many researches have been performed to quantify and predict the impact of soft errors on different microprocessor components including DRAM memory, SRAM memory, sequential and combinational logic circuits. While early studies predicts exponential growth in transistor-level soft error rate by technology scaling (Cohen *et al.*, 1999; Shivakumar *et al.*, 2002; Baumann, 2005b), recent explorations have concluded constant or even decreasing soft error rate in advanced sub 60nm technologies (Cannon *et al.*, 2008; Mahatme *et al.*, 2014). However, due to increased level of integration (more transistors per core, more cores per chip and more chips per system) overall soft

error system-level failures is expected to be increased (Shafique *et al.*, 2014; Henkel *et al.*, 2013).

Soft error reliability issues have influenced industry standards and productions. Contemporary functional safety standards, e.g. (ISO26262, 2011), clearly mentioned to transient fault model as a safety thread which should be addressed to achieve high safety integration level (i.e. ASIL C or ASIL D) for autonomous cars. State-of-the-art modern microprocessors have already adopted a variety range of soft error mitigation schemes in their design. For example, Intel Xeon microprocessor E7 family microprocessors is equipped with error detection and correction codes (ECC) in both on-chip caches and CPU registers to alleviate the issue of transient faults (Xeon, 2011). ARM cortex-R dual/triple core lock-step microprocessors (Iturbe *et al.*, 2016; Lyons, 2010) have adopted redundant execution strategy to detect/mask the manifestation of errors in real time applications. Even for general high performance applications, ARM provides protected configuration which is equipped with ECC-protected memory subsystem i.e. private and shared caches as well as TLBs(ARM®, 2014). Overall, the wide adoption of transient fault mitigation schemes in major microprocessor vendors implies the importance of soft error resilience issue even for safety-critical terrestrial applications.

1.1 Soft Error Mitigation Schemes

Historically, designers of aircraft and satellite applications utilize RHBP (radiation-hardened-by-process) microprocessors to cope with space-level radiation rate. In RHBP microprocessor fabrication process is modified to mitigate soft error concerns. Generally, such microprocessors are orders of magnitudes slower that high performance microprocessors. For instance, the second generation of RAD750 microprocessor, state-of-the-art commercially available RHBP microprocessor, are built on

0.18 micron feature size with 200 clock frequency and computational power of 400 MIPS (Haddad *et al.*, 2011). RHBP is the most extensive error protection strategy, however, the expensive cost of such techniques have restricted their applicability to cost-agnostic systems.

To achieve high-performance and affordable error resiliency, designers adopt RHBA (radiation-harden-by-architecture) microprocessors which in error mitigation is accomplished mainly by μarchitectural-level modifications. Mianly, RHBA techniques explore spatial redundancy and execute two redundant versions of computations on different hardware components – different core or different execution path inside a core, and compare the redundantly computed results. Examples are IBM Z-series servers (Spainhower and Gregg, 1999), HP NonStop systems (Bernick *et al.*, 2005), and HERMES embedded microprocessor (Clark *et al.*, 2014, 2016). These techniques achieve high degree of fault coverage in the cost of more than 2x performance, power and area overhead. Less aggressive hardware redundant multithreading solutions execute redundant threads on simultaneous multithreaded processors and specific results (mainly store instruction's value and address) for error detection purposes (Reinhardt and Mukherjee, 2000; Mukherjee *et al.*, 2002). These approaches suffer from remarkably less performance degradation than fully redundant lock-stepped designs, but they still require hardware modifications (Mukherjee *et al.*, 2002).

Software level schemes can also be employed for soft error protection through applying redundancy at various level of abstractions. Fine-grained instruction replication schemes (Oh *et al.*, 2002a; Reis *et al.*, 2005), run two (or three) versions of assembly instructions and check their results periodically for error detection (masking). Thread-level replication solutions (Wang *et al.*, 2007; Zhang *et al.*, 2012b; Mitropoulou *et al.*, 2016), execute two redundant threads of computations (possibly of different cores of a multicore microprocessor) and detect the manifestation of

errors by comparing their results. Similarly, coarse-grained process-level redundancy schemes (Shye *et al.*, 2009; Zhang *et al.*, 2012a), apply redundancy at process level for error detection.

Software-level schemes are applicable on commercial microprocessors without any hardware modification. Because of this feature they are preferable especially for mixed-critical systems where different tasks have different level of resiliency requirements (Barhorst *et al.*, 2010; Kang *et al.*, 2014; Wells *et al.*, 2009; Burns and Davis, 2017). For instance, consider an application which in safety-critical tasks and non-critical tasks share underlying microprocessor. Software error mitigation techniques (e.g. task replication) can be applied to the safety-critical tasks for error protection, while normal tasks can be executed without any performance degradation.

1.2 Scope of This Research

In this research we focus on instruction level soft error mitigation schemes because of their extreme fine-grained flexibility and fast error detection capability. These solutions can selectively be applied on the most critical part of computations (instructions) even inside a task or function. In fact, recent studies (Feng *et al.*, 2010; Laguna *et al.*, 2016), have shown that considerable error protection can be achieved by selective instruction-level redundancy in the cost of low performance degradation. Despite of these advantages, the existing instruction-level replication techniques can only meet the reliability requirements for medium- and less-critical computations and fail to achieve high error resiliency (detection+recovery) demanded by high-critical tasks.

The main goal of this research is to advice compiler level soft error resilient schemes which can provide complete, effective and timely recovery for general purpose applications. Particularly, we focus on protecting the execution of programs against against

soft errors affecting microprocessor on-core components excluding caches and TLBs. That is because memory subsystem can be protected efficiently and effectively by error detection and correction codes. In fact, in many embedded processors like ARM Cortex-A53 the costumer can select cache protection version of the processor which in caches and TLB are protected by ECC (ARM®, 2015).

1.3 Contributions

This thesis makes the following contributions:

1. **nZDC:** A novel comprehensive in-thread instruction duplication approach that detects the manifestation of soft errors in all microprocessor on-core components. Against the state-of-the-art schemes which can only protect computational instructions, the proposed scheme protect the execution of all program instructions against soft errors.

2. **InCheck:** An in-application error detection and backward recovery scheme which achieves comprehensive, safe and timely soft error resiliency. The proposed schemes make light-weight verified checkpoints at basic block granularity and reverts the program execution to the beginning of last executed basic block after error manifestation detection.

3. **NEMESIS:** A compiler-level error detection, diagnosis and forward recovery scheme. The proposed scheme expands the sphere-of-protection of existing instruction replication techniques from the execution of computational instructions to the absolutely all program instructions. It replaces computationally expensive software majority-voting routines with cheap error detectors. In the case of error, diagnosis and recovery routines allow for quick recovery, and execution will be resumed error-free.

4. **gZDC:** A general soft error detection scheme which detects hard-to-detect control-flow errors as well as data-flow errors. The proposed solution combines the best aspects of the existing state-of-the-art data and flow schemes and introduces coarse-grained main-redundant instruction scheduler and asymmetric control-flow signatures. Fault injection experiments on different hardware components of synthesizable Verilog description of an OpenRISC-based microprocessor shows that the effectiveness of the proposed solution.

The rest of the dissertation is organized as follows. Chapter 2 reveals the vulnerable windows in the existing instruction-level error detection schemes and describes a novel instruction-replication soft error detection schemes. The proposed scheme considerably increases the error detection coverage of the state-of-the-art schemes. Chapter 3 explains the flaws in state-of-the-art in-application error recovery scheme and introduces a safe and fast backward recovery solution. Chapter 4 discusses a novel soft error forward recovery scheme. Chapter 5 explains a generic soft error detection and shows RTL-level fault injection results. Finally, Chapter 6 concludes this dissertation and proposes possible future extensions.

Chapter 2

ERROR DETECTION BY INSTRUCTION DUPLICATION[1]

2.1 Introduction

In order to achieve a low-cost and flexible fault tolerance mechanism, several software-level techniques have been proposed. Among software-only fault tolerance techniques, low-level instruction duplication based schemes (Oh *et al.*, 2002a; Reis *et al.*, 2005, 2007; Feng *et al.*, 2010; Khudia *et al.*, 2012; Mitropoulou *et al.*, 2013b; Yu *et al.*, 2009) are the most popular ones because they provide flexible protection without imposing any inter-thread or inter-core communication overhead. The key idea behind such methods is time redundancy and that error protection is possible by redundant execution and result checking. Instruction-level error detection schemes usually divide programmer available registers into two sets of registers and replicate program instructions with different registers. Error detection is accomplished by simply checking the results of redundant instructions. In this chapter, we review the state-of-the-art instruction duplication error detection schemes. Then, We highlight the limitations of existing schemes and propose our solution to enhance the error detection ability of existing solutions. Finally, we evaluate the effectiveness of our proposed scheme by processor-wide μarchitectural-level fault injection experiments.

[1]This chapter combines two published papers, Didehban, Moslem, and Aviral Shrivastava. "nZDC: A compiler technique for near Zero Silent Data Corruption." Proceedings of the 53rd Annual Design Automation Conference. ACM, 2016 and Didehban, Moslem, and Aviral Shrivastava."A Compiler Technique for Processor-Wide Protection From Soft Errors in Multithreaded Environments."IEEE Transactions on Reliability 67.1 (2018): 249-263The publishers permit authors to include partial or complete papers of their own in a dissertation.

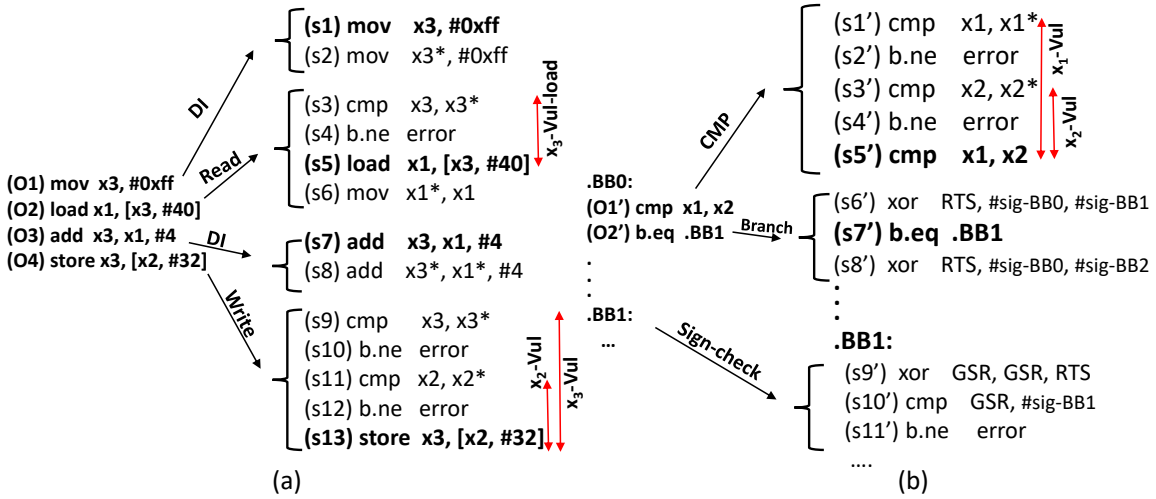

Figure 2.1: SWIFT transformation: part (a) data flow protection and (b) control flow protection.

2.2 Related Work

Error Detection by Duplicated Instructions or EDDI (Oh *et al.*, 2002a) scheme is one of the pointers of instruction-replication based schemes. EDDI partitions programmer visible registers and program memory space into partitions. EDDI transformation replicates all program computational and memory instructions with different set of registers. EDDI checks the results of redundant computations for error detection right before the execution of memory write and conditional branch instructions.

In an attempt to improve the performance and fault coverage of EDDI, SWIFT (Reis *et al.*, 2005) eliminates the need of memory duplication by assuming that the memory subsystem is protected (by other means such as ECC). In SWIFT transformation, all computational/logical instructions are duplicated with different registers and the checking instructions are inserted before three type of instructions: a) memory read operations, b) memory write operations, and c) control flow instructions, i.e., compare, branch, and function calls. Figure 2.1(a) shows the SWIFT data flow transformation and the corresponding original code. In the Figure, shadow registers

are differentiated from the master ones by an star (x1* is the shadow for x1). In the snippet code presented in the Figure, original "mov"(inst. O1) and "add"(inst. O2) instructions are duplicated by SWIFT transformation which are marked as `DI` (**D**uplicatable **I**nstruction) in the Figure. SWIFT transformation does not duplicate the memory read instructions and it checks for errors in memory address register of memory read operations before the execution of such instructions and copies the loaded value into the corresponding shadow register right after the execution of load instruction. SWIFT memory read instruction transformation is marked by `read` in the Figure. For the "load" (inst. S5) instruction, the value of address register `x3` is checked against the redundant-computed value `x3*`, before the execution of "load" instruction and the loaded value `x1` is copied into the corresponding shadow register `x1*` right after the "load" instruction. We named this extra "move" instructions (inst. S6) as "copying-move" instruction. SWIFT transformation only executes one version of memory write instructions and the checks the register operands of such instructions before their execution for error detection. The memory write instruction transformation is marked as `Write` in the Figure. The "store" instruction register operands, `x3` and `x2` are checked against their shadows `x3*` and `x2*` before the execution of `store` instruction.

To protect the execution of programs against control flow errors, SWIFT checks register operands of "compare" instructions and uses statically-assigned signatures to detect errors affecting the execution of "branch" instructions. Figure 2.1(b) shows SWIFT control flow transformation. Before the execution of `cmp` (inst. s5'), its register operands `x1` and `x2` are checked against their shadow registers `x1*` and `x2*`.

SWIFT transformation statistically assigns a specific number (called signature) to each basic block and uses two specific registers (named RTS and GSR) to dynamically recompute basic blocks signatures. It declares a control flow error, if dynamically cal-

culated signature does not match with the statistically assigned signature. Before each "branch" instruction or at the end of each basic block, the RTS (Run-time-Signature) register is computed from the current basic block signature and the branch destination basic block signature (inst. S6' and S8'). The control flow error detection takes place in the beginning of each basic block by extracting the basic block signature from the RTS and GSR (General Signature Register) registers and checking the dynamically calculated signature against the statically-assigned ones (labeled as `Sign-check` in the Figure).

For more than a decade, SWIFT has been considered as the most effective in-thread instruction duplication technique in terms of fault coverage. Acknowledging the near perfect fault detection ability of SWIFT, several works have tried to improve the performance overhead of SWIFT transformation (Feng *et al.*, 2010; Khudia *et al.*, 2012; Mitropoulou *et al.*, 2013b; Yu *et al.*, 2009; Mitropoulou *et al.*, 2013b). For instance, authors of Shoestring (Feng *et al.*, 2010) paper express that "SWIFT has the advantage of being purely software-based, thus requiring no specialized hardware, and can achieve **nearly 100%** coverage". Targeting non-critical applications, Shoestring compromises between fault coverage and performance overhead of instruction duplication by taking advantage of low-level hardware symptom detectors and applying instruction duplication to those instructions which error on them most likely cause no symptom. Similarly, research IPAS (Laguna *et al.*, 2016), also apply instruction duplication judiciously of most critical instructions of programs and trades off coverage for effiency. On the other hand, DRIFT (Mitropoulou *et al.*, 2013b) and ESoftCheck (Yu *et al.*, 2009) try to improve SWIFT performance overhead without jeopardizing its fault coverage. A recent study explores the use of vector instruction for instruction replication-based fault tolerant techniques and illustrates considerable performance improvements (Chen *et al.*, 2016).

2.3 Limitations of state-of-the-art schemes

Ideally, a perfect software fault detection scheme should be able to protect the execution of a program against soft errors on various hardware components. However, the main error coverage limitation of existing instruction duplication techniques is that they can only detect the impact of errors on the execution of some instructions (i.e., the ones that they replicate) and leave the rest unprotected. To provide a detailed analysis of the protection offered by instruction duplication techniques, in particular SWIFT (Reis *et al.*, 2005), we investigate the impact of single bit flip transient faults on different hardware components while executing a SWIFT-protected program. We consider an in-order baseline processor (shown in Figure 2.2) and examine the impact of errors on different type of instructions in a SWIFT-protected code. We limit our analysis to the microprocessor core components excluding branch predictor and memory subsystem. We do not consider soft errors on branch predictor structure because we assume that soft errors affecting branch predictor will affect the performance of the processor not its functionality (Mukherjee *et al.*, 2003). We assume that memory subsystem including TLBs and caches ECC are protected by ECC (Error detection and Correct Code) and the data is protected while it is in memory. However, faults on cache controllers while decoding an address can still result to a wrong access (read/write) to the memory (Borucki *et al.*, 2008).

First, we classify the instructions in a SWIFT-protected code into 7 categories: i) Duplicable Instructions (DIs): these are program computational/logical instructions which are duplicated by SWIFT transformation, ii) Memory Read Instructions (MemRead): These are load instructions that SWIFT checks their address register before their execution, iii) Copying-move (CopMov) instructions: these are inserted after MemRead instructions to provide consistent input replication for shadow and master

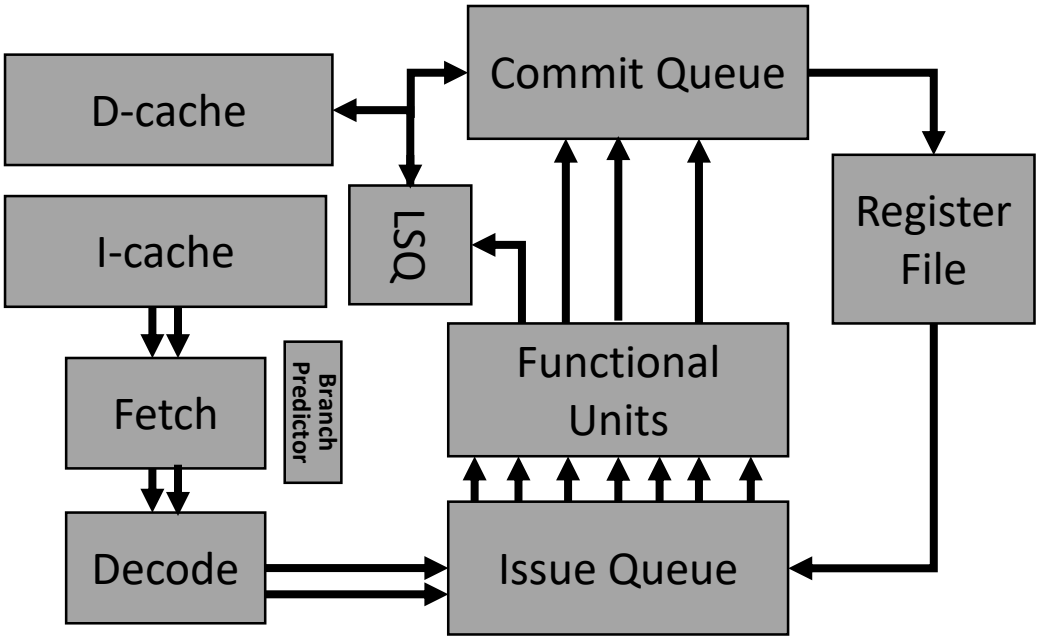

Figure 2.2: Baseline processor. Note that Caches and branch predictor are excluded from our error coverage analysis.

instructions, iV) Memory Write instructions (MemWrite): wherein value and address register operands are checked before their execution, v) Original Compare (OrgCMP) instructions: wherein register operands are checked to prevent control flow errors, vi) Original Branch (OrgBR) instructions, and vii) Checking Instructions (CIs), which includes instruction responsible for updating the RTS and GSR registers, checking for discrepancy in signatures or redundantly computed registers, and terminating the program execution in case of error.

Table 2.1 summarizes the protection offered by SWIFT in the presence of single bit-flip error in different hardware components. The rows in the Table represent various type of instructions in a SWIFT-protected program and the columns show various hardware components. If SWIFT transformation is able to detect the effect of errors in component `Comp.`, while it is utilized with instruction `Inst.`, the letter **P** (**P**rotected) is placed in the corresponding location `(Inst.,comp.)` in the Table.

Table 2.1: Software-hardware view of SWIFT Protection

Inst. Type	Fetch	Decode	IQ	FUs	Commit	LSQ	RF
DI	P[1]	P	P	P	P	na	P
MemRead	NP[2]	NP	NP	NP	NP	NP	NP
CopMov	P	P	P	P	P	na[3]	NP
MemWrite	NP	NP	NP	NP	NP	NP	NP
OrgCMP	NP	NP	NP	NP	NP	na	NP
OrgBr	NP	NP	P	P	P	na	NP
CI	NP	NP	P	P	P	na	P

[1] Protected. [2] NotProtected. [3] not applicable.

Otherwise, it is filled by **NP** (**N**ot**P**rotected), which means SWIFT transformation cannot protect the execution of instruction `Inst.` against faults in the hardware component `Comp`.

Duplicable Instructions (DIs): The first row in the table says that SWIFT transformation can detect errors affecting the execution of duplicated instruction, regardless of the location of fault. If error happens on any hardware component while processing a DI instruction, the impact of error will get masked or detected by SWIFT checking instructions. It is worth mentioning that if an error hits some specific bits in the micro-architectural resources, e.g., the valid-bit of an entry holding two redundant instants of a duplicated instruction, the error alters both instructions in the same way and remains undetected.

Memory Read Instructions (MemRead): As the second row of table 2.1 shows, SWIFT transformation leaves memory read instructions unprotected during their

execution. This is because there is no redundant version or execution check for load instructions. Therefore, all errors which modify the address or the size of loaded value can result to a failure. Examples of such errors are: errors that hit fetch, decode, or issue stage registers while they are occupied by load instruction. Errors on functional unit that is responsible for load effective address calculation. Errors on memory read request address or target data size while the load request is processing in the load-store queue. Even if an error hits the source register of a memory read instruction (e.g., register `x3` of `load` in Figure 2.1(a)) before getting **directly**[2] accessed by the memory read instruction, it will cause a wrong-memory-location access error. Note that in this case, since the state of load address register is different from its shadow, the error will remain undetected if the next access to the load source register is a write.

Note that the execution of memory read instructions are unprotected in many similar instruction-based replication techniques (Chen *et al.*, 2016; Reis and August, 2006; Mitropoulou *et al.*, 2013a; Martinez-Alvarez *et al.*, 2012; Reis *et al.*, 2006, 2007; Wang *et al.*, 2007; Zhang *et al.*, 2012b; Yu *et al.*, 2009; Khudia *et al.*, 2012; Yu *et al.*, 2008, 2007).

Copying Mov Instructions (CopMov): Soft error on all microprocessor hardware components, expect register file, while processing CopMov instructions is covered by SWIFT transformation. If an error hits a CopMov instruction source registers (e.g., `x1` of CopMov instruction (inst. s6) in the Figure 2.1(a)), the error will propagate from master register (`x1`), to its corresponding shadow register (`x1*`) and remain unnoticed.

All instruction replication based techniques that do not duplicate the memory read

[2] An instruction accesses a register directly, if that is the last access to the register so far.

instructions (Chen *et al.*, 2016; Reis and August, 2006; Reis *et al.*, 2006; Mitropoulou *et al.*, 2013a; Martinez-Alvarez *et al.*, 2012; Reis *et al.*, 2007; Wang *et al.*, 2007; Zhang *et al.*, 2012b; Yu *et al.*, 2009; Khudia *et al.*, 2012; Yu *et al.*, 2008, 2007), also suffer from this vulnerable interval.

Memory Write Instructions (MemWrite): Similar to MemRead instructions, memory write instructions are also vulnerable all through their execution. Errors affecting opcode, data and address register pointers, immediate, shift, size and rotate field while a memory write instruction is processing by pipeline registers are the examples of such undetected errors. Likewise, errors altering memory effective address while the instruction is utilizing functional unit or load-store queues also remain undetected. Furthermore, the operands of store instructions are also susceptible to soft errors that occur before the register gets accessed directly by such instructions. However, there is a chance that such errors in register file are detected by the upcoming checking instructions, if the next access to faulty registers is a read. Memory write instructions are also single-point-of-failure in many fine-grained instruction replication techniques (Xu *et al.*, 2013; Chen *et al.*, 2016; Reis and August, 2006; Mitropoulou *et al.*, 2013a; Martinez-Alvarez *et al.*, 2012; Feng *et al.*, 2010; Reis *et al.*, 2007, 2006; Wang *et al.*, 2007; Zhang *et al.*, 2012b; Yu *et al.*, 2009, 2007, 2008; Mitropoulou *et al.*, 2013b; Liu *et al.*, 2015; Xiong and Tan, 2013; Chen *et al.*, 2016).

Original Compare Instructions (OrgCMP): As the Figure 2.1(b) shows, SWIFT transformation adds checks to the operands of `compare` (inst. s5') to avoid wrong-direction control flow errors. However, since the `compare` instruction itself is not duplicated, errors during the execution of `compare` instruction itself, can change the control-flow of the program and remain undetected. Note that in this case, the

signature-based part of SWIFT control flow checking mechanism is also unable to detect the error because the RTS register is set for both (taken or not-taken) directions (inst. s6' and s8'). There are many similar techniques (Xu *et al.*, 2013; Chen *et al.*, 2016; Reis and August, 2006; Mitropoulou *et al.*, 2013a; Martinez-Alvarez *et al.*, 2012; Feng *et al.*, 2010; Reis *et al.*, 2007, 2006; Yu *et al.*, 2009, 2007, 2008; Mitropoulou *et al.*, 2013b; Liu *et al.*, 2015; Xiong and Tan, 2013; Chen *et al.*, 2016; Oh *et al.*, 2002a) that suffer from the same control-flow vulnerability.

Original Branch Instructions (OrgBr): The original branch instructions are vulnerable to soft errors when they are in the fetch and decode units. If an error changes the branch opcode or condition field of a branch instruction, it can change the direct of the branch from taken to not-taken or vice versa, and the error remains unnoticed. Most of the errors that change the target address of a branch will get detected by the signature-part of the SWIFT control flow checking mechanism. However, if an error causes a jump back to the body of the source basic-block (after the signature-checking instructions) it still remains undetected because the signature-checking part is already passed. Furthermore, in table 2.1 it is noted that the branch instructions are not protected in the register file. This is because the source register for (direct or indirect) branch instructions is the program status flag register, which holds Zero, Overflow, Negative, and Carry bits, and if error happens on the flag register, the direction of branch will be changed and the error is undetectable. For function calls, depending on the implementation, if just one copy of registers is sent to the callee function, the registers are unprotected between checking (in caller function) and duplicating time (in callee function). Many of similar works (Xu *et al.*, 2013; Chen *et al.*, 2016; Reis and August, 2006; Mitropoulou *et al.*, 2013a; Martinez-Alvarez *et al.*, 2012; Feng *et al.*, 2010; Reis *et al.*, 2007, 2006; Yu *et al.*, 2009, 2007, 2008; Mitropoulou *et al.*,

2013b; Liu *et al.*, 2015; Xiong and Tan, 2013; Chen *et al.*, 2016) also suffer suffer the same problem.

Checking Instructions (CI): SWIFT transformation inserts a plenty of checking instructions into the code to ensure the correct execution of the program. These checking instructions are either redundant-register mismatch-checking instructions (i.g. instructions s3, s4, s9, s10, s11, s12, s1', s2', s3' and s4' in the Figure 2.1) or signature setting/checking instructions (i.g. instructions s'6, s8', s9', s10' and s11'). Generally, errors affecting these instructions just cause false alarms and not lead to a failure. However, in some special cases (e.g., opcode of a CI instruction changes to a store instruction), it is possible that the program experiences failure because of soft error on a CI instruction.

Overall, the memory and control-flow instructions are the main single-point-of-failures in many in-thread instruction replication-based techniques. In (Blem *et al.*, 2013; Patterson and Hennessy, 2013) the amount of MemRead, MemWrite, OrgCMP and OrgBr instructions are reported as 50%, 55%, and 40% on average for X86, ARM, and MIPS processors, respectively.

2.4 nZDC: Our Proposed Instruction Duplication Error Detection Solution

In this section, we present nZDC (a compiler technique for near Zero silent Data Corruption), a compiler approach to almost eliminate SDCs. In line with previous researches, we assume that soft errors can modify the data within the CPU but memory and caches are protected by other orthogonal techniques such as ECC. The salient features of nZDC are:

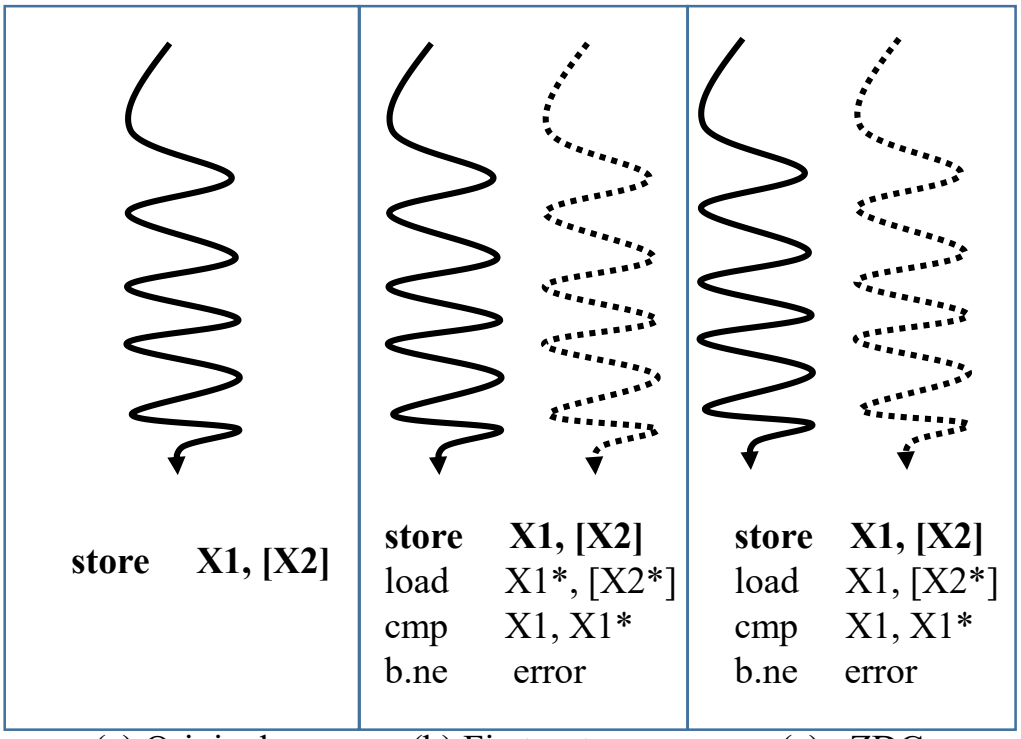

Figure 2.3: nZDC store checking mechanism

nZDC Protects Stores by Checking-load Strategy

nZDC introduces the concept of "checking load instruction" to make sure that the store instruction has executed fault free. The main idea here is to load back the stored value from the memory and check that against the stored value, if they match there is no error otherwise error handler routine get involved. Figure 2.3(b) shows the first-cut of our approach to insert checking load instruction. Although it can detect errors which affect the address part of the store instruction, yet errors on value part can simply propagate from the store's value register to checking load value registers, and remain undetected. For instance, if soft error alters the value of `X1`, store instruction writes the erroneous value into memory. Later, the checking load instruction loads the corrupted value to `X1*` register. Now, both `X1` and `X1*` have same wrong values and comparing them cannot catch the error. This happens because the value register

of checking load instruction is the shadow of the value register of store instruction. nZDC protects stores by using the same value register for both store and checking load instructions and later check that register against its shadow for soft error (Figure 2.3(c)). By this method, in addition to the producer chains of store value and address registers, the execution of store instruction itself is also protected.

The performance overhead of nZDC solution for store instruction may seem high at first, but thanks to store-to-load forwarding mechanism in LSQ of modern microprocessor, the checking load instructions normally take their values from forwarding path in load-store unit and executes very fast. The only problem here is, if error happens on store buffer after that the store forwarded its data to the checking load instruction, the error may remain undetected. This unprotected interval can be completely removed in two ways; i) flush store buffer after each store, or ii) use ECC in store buffer. The former comes with significant performance degradation; it is similar to the case that there is no store buffer at all. The second approach may not have performance overhead, but the ECC code should be generated before the stores arrive at the store buffer.

nZDC Protects Loads by Relax Duplication Strategy

Memory read instructions are the most frequent unprotected instruction in SWIFT and many other software redundancy based techniques (Khudia *et al.*, 2012; Yu *et al.*, 2009). These instructions behave such as the input of duplicated instructions chain, and if a memory read instruction gets faulty, the execution can go wrong and checking instructions are unable to detect such an error.

Our solution for memory read instructions is simple, we use "load" duplication, which in memory read instructions get duplicated as well as logical and computational instructions. The load instruction duplication has two advantages; 1) it protects load

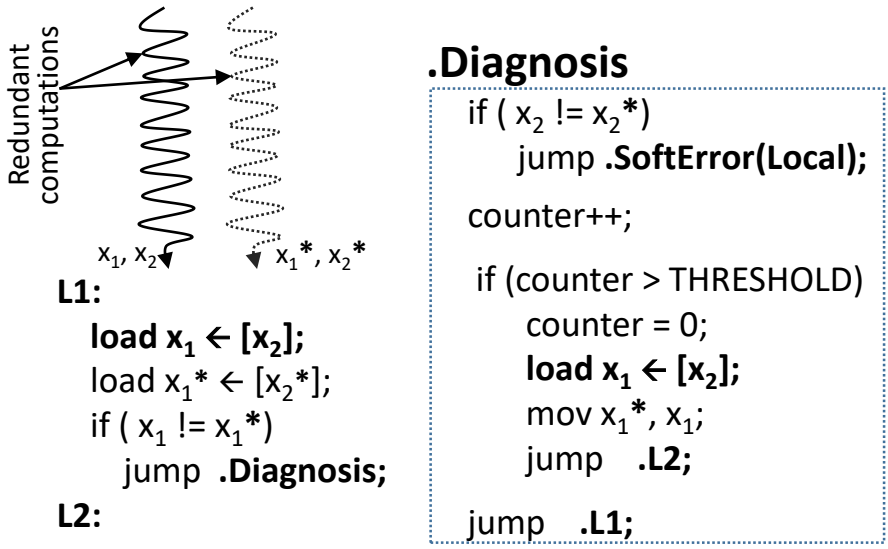

Figure 2.4: nZDC load instruction transformation for shared memory access in multithreaded applications.

instructions completely during their execution in pipeline, LSQ and removes load-related register file vulnerable intervals (Marked as $_{\text{X3-vul-load}}$ in Figure 2.1), and 2) it saves performance overhead by reducing the number of checking instructions.

The load instruction duplication may introduce false alarms for memory access to the shared memory in multithreaded applications if an intervening store changes the state of the memory between redundant load accesses. Figure 2.4 illustrates nZDC memory read instruction transformation for shared memory access in multithreaded applications. As it shows, nZDC transformation redirects the program control to a diagnosis block if the results of redundant loads are different. In the diagnosis block, it first checks for errors in loads address registers (`x2` and `x2*` in the Figure 2.4). In case of mismatch, the soft error detection flag will be raised. If there is no error in replicated load address registers, there can be two possibilities for the inconsistency: (1) soft error happens during the execution of one of the load instructions and for instance, alters the effective address, or (2) intervening store from the other thread has modified the state of the memory. Either way, by jumping back to right before

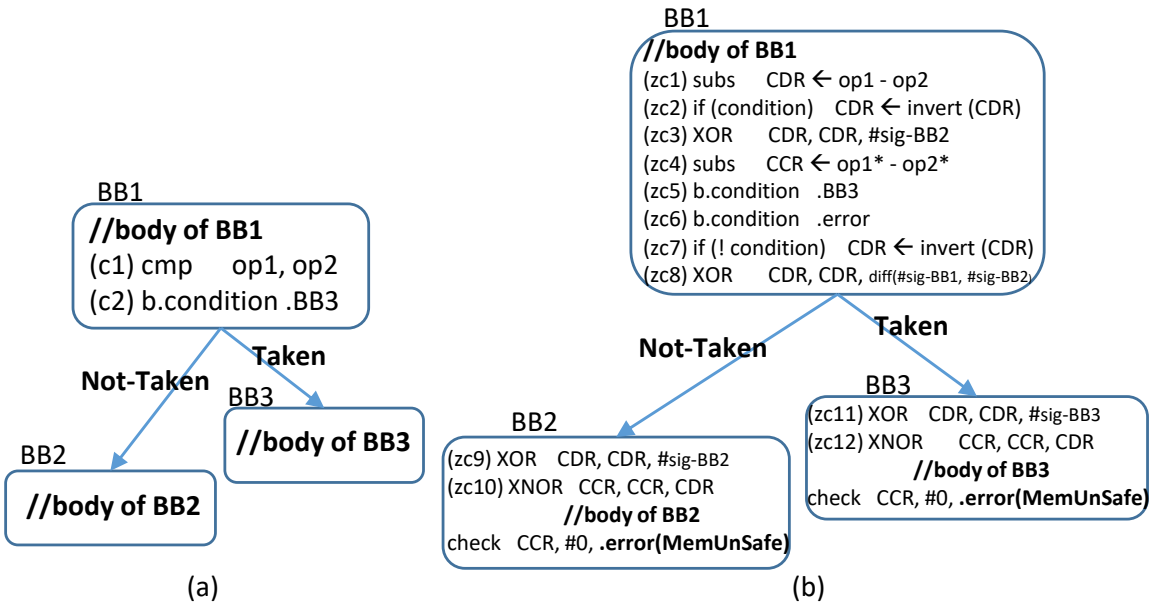

Figure 2.5: nZDC Control Flow Checking Mechanism. All errors detected by control-flow checking mechanism are considered as global errors (error is propagated to the memory).

redundant-load instructions, the problem will be solved. If soft error was the reason for the discrepancy (mismatch between `x1` and `x1*` in Figure 2.4), simply re-execution provides recovery. If the discrepancy comes from an intervening store instruction (like race condition), the problem will be solved by repeating the execution of redundant loads. However, to prevent a program from going to an infinite loop (due to frequent memory update by other threads between the redundant loads instructions) nZDC transformation uses a threshold-based mechanism. Basically, it counts the number of iterations that a particular diagnosis routine has been involved (`counter++` in Figure 2.4). If the number of iterations exceeds a predefined value, the nZDC first resets the counter. Then performs just one load and copies the loaded value to the corresponding shadow register, and transfers the program execution to right after the checking instructions in the original control flow of the program. Note that since the execution of diagnosis block is rare, the performance overhead of nZDC transformation is acceptable.

nZDC Control-Flow Checking Mechanism

An effective control flow checking mechanism should be able to protect all of the control flow determining parts of the execution; which are a) Operands of compare instructions, b) pipeline registers while executing compare instruction, c) conditional registers (NVCZ flags) and d) branch instructions. Against the existing control flow checking mechanism which can partially protect some of these parts, nZDC control flow checking mechanism can effectively protect all control flow checking determining components. The nZDC control flow mechanism (shown in Figure 2.5) demands two general purpose registers, called CDR (Compare Destination Register) and CCR (Compare Check Register). The nZDC control flow checking mechanism works based on three main insides: a) compare and branch instruction replication, b) protect NVZC flag register by conditionally inverting the value of CDR register based on the direction of the following conditional branch, and c) use static signatures for source-encoding/destination-decoding to make sure that the control flow of the program is traversed correctly. The nZDC control flow mechanism consists of five main steps:

Duplicating CMP instruction: Generally, in ARM and X86 ISAs, a compare (CMP) instruction is implemented by a subtraction (SUB) instruction which disregards the results of the subtraction operation and updates the program status flags (NVZC). Leveraging this fact, nZDC control flow transformation converts all program compare instructions to their equivalent subtraction operations and duplicates them. However, rather than disregarding the results, nZDC control flow transformation saves the results of the subtraction operation into CDR and CCR registers. In Figure 2.5, instructions (zc1) and (zc4) are for duplicated versions of the original CMP instruction (c1).

Conditionally inverting the CDR: Since the NVZC register is not duplicable,

nZDC uses time redundancy to protect that register against soft errors. For instance, as Figure 2.5 shows, at time *t*, the first CMP instruction(zc1) sets the NVZC flag, which is going to be read by the following conditional invert instruction(zc2) at the next cycle (assuming 1 cycle per instruction). The second CMP instruction(zc4) will be set to the NVZC flag at time *t+3*, and the flag register will be read by conditional branch instruction(zc5) at time *t+4*. If the first CMP instruction(zc1) sets the NVZC flag in such way that the condition of the following conditional branch(zc5) is true, the CDR register gets inverted right after the first SUBS instruction (zc1). On the other hand, if the condition is not true and branch is suppose to be not taken, the CDR gets inverted after the branch (zc7). In a fault free run of the program for each conditional branch, the CDR inverts just one time. Although this conditional instructions is ISA dependent, it can be replaced with a micro if the ISA does not support these instructions.

Duplicating branch Operations: nZDC duplicates all programs conditional branch instructions. However, the branch target addresses for the copy branch (zc6) is error handler basic block. The purpose of branch duplication is to protect the soft errors on the branch opcode. The main idea is if the condition is true, the main branch (instruction I5) will change the control flow of the program and the redundant one does not execute. If the condition is not true neither of the branches changes the control flow. But if errors happen on the original branch opcode, e.g., branch changes from blt (branch less than) to bgt (branch greater than) and if it changes from taken to not-taken, the redundant branch detects the error. On the other hand, if soft error alters the direction of a conditional branch from not-taken to taken, nZDC control-flow checking instructions detects that in the wrong target basic block.

Adding destination signature: Based on the possible destination, the CDR register gets XORed with a unique and basic block signature assigned to all program

Table 2.2: Simulator parameters

Parameter	Value
CPU Model	ARM64 bit in-order processor
Pipeline	Two issue/4-stage
# of FUs	2Int, 1Mul, 1Div, 1Float, 1Mem
L1 D/I-Cache	64KB (2-way) / 32KB (2-way)
Integer register file	32 registers (64-bit width)
Store buffer size	5 entries

basic block statically. Instruction(zc3) shows the destination related signature coding if branch is taken, otherwise instruction(zc8) performs the signature coding for the next basic block.

Inserting control-flow checking instructions: In order to check if the direction of a branch has been taken correctly, nZDC inserts two instructions at the beginning of the all program's basic blocks. The first instruction XORs the CDR register with the current basic block signature (zc10 in BB2 and zc9 in BB3) and saves the result back to the CDR register. The next instruction XNORs the CDR and CCR registers (zc12 in BB2 and zc10 in BB3) and stores the result in CCR. In a fault free run of the program execution, before XNOR instruction, the value of CDR should be equal to the inverted value of CCR. Therefore, after the XNOR instruction the CCR register value should be always Zero because the inputs for the xnor instruction are each other inverse (one's complement). Finally the nZDC control flow error detecting instructions will be inserted into two points of execution: 1) before each write to the CCR (between inst zc3 and zc4), and 2) before all function calls and direct branches.

2.5 Experimental methodology

We have performed extensive fault injection testing to evaluate the effectiveness of nZDC and SWIFT in reducing SDCs.

Compilation framework: We have implemented nZDC and SWIFT transformations as late backend passes in LLVM3.7 compiler infrastructure (Lattner and Adve, 2004) after register allocation and instruction scheduler. This implementation enables us to take advantage of all of the advanced compiler optimizations including Common Sub-expression Elimination (CSE) and Dead Code Elimination (DCE). We test the effectiveness of SWIFT and nZDC on applications from the Mibench benchmark suite.

Simulation environment: We have used the gem5 (Binkert *et al.*, 2011) - a popular cycle-accurate microarchitectural simulator. The simulator was run in ARM syscall emulation mode and modeled the ARMv8-a profile of the ARM64 bit architecture. We have used a two-way in-order ARM architecture for fault injection experiments with the details of the processor configuration in Table 2.2. The simulated CPU model is very close to "cache protection configuration" of ARM Cortex-A53, a popular modern high performance low power embedded microprocessor. In this configuration of ARM cortex-A53 the memory subsystem including TLB, instruction and data L1 caches, and L2 data cache are protected with error detection and correction codes.

Fault sites: Instead of injecting faults just on processor's register file, we inject faults on all the major sequential component of the processor. For the in-order ARM, this includes the pipeline registers, load-store queue and functional units. All the other components are either not vulnerable (e.g., the branch predictor), or are already assumed protected (e.g., the caches and the TLBs). Additionally, to show the

effectiveness of the nZDC control-flow checking, we specifically perform fault injection on the branch and compare instructions while they are in the processor's pipeline.

Fault injection experiments: For each fault site, a random bit in a random time is selected and inverted. For example in the case of register file, at the start of each experiment a physical register, a bit and a cycle is selected randomly for fault injection. Simulation runs the program normally till the selected cycle. Then the value inside the selected bit gets inverted, and the program runs until completion or allowable simulation time (which is 10 times the nominal execution time) gets over. For each fault site, 400 faults are injected which gives us a 5% margin of error and 95% confidence interval (Leveugle *et al.*, 2009). This means, that for each version of the program (original, SWIFT and nZDC), 2400 fault injection experiments are performed. Among them 2000 faults are injected into register file, pipeline registers, load store queue, functional units, and the rest, 400 faults, are specifically injected into the main branch and compare instructions. Overall, we inject 72,000 faults in various components of the processors.

Output classification: Since the main goal of this work is to prevent a program from producing the wrong output because of soft errors, the result of each fault injection trial is classified into two categories: *1. SDC:* The simulation runs which are terminated normally, but produce wrong output, and *2. Others:* All other scenarios i.e, masked faults, detected fault, segmentation fault and crash fall into this category.

Comparison Metric

The common practice to evaluate the efficacy of software fault tolerant techniques is by comparing percentage of failures/SDCs extracted from statistical fault injection with performing the same number of fault injection experiments for original and protected versions of the programs (Xu *et al.*, 2013; Chen *et al.*, 2016; Reis and

August, 2006; Mitropoulou *et al.*, 2013a; Martinez-Alvarez *et al.*, 2012; Feng *et al.*, 2010; Reis *et al.*, 2007, 2006; Wang *et al.*, 2007; Zhang *et al.*, 2012b; Yu *et al.*, 2009, 2007, 2008; Mitropoulou *et al.*, 2013b; Liu *et al.*, 2015; Xiong and Tan, 2013; Chen *et al.*, 2016; Oh *et al.*, 2002a). However, since usually software fault tolerant methods prolong the execution time of the program, they can decrease the percentage of SDCs just by increasing the amount of masked or detected errors caused by the faults that influenced the program-irrelevant parts of execution (Schirmeier *et al.*, 2015). Program-irrelevant parts of execution is the segment of the execution time that is not part of the original program, but is needed to protect the original program, such as the duration of time that the processor spends to execute the redundant and control flow checking related instructions in SWIFT/nZDC protected programs. To clearly express the main idea of this section, we use a simplified example of running original and FT versions of a program on a simple in-order CPU. The execution time trace is shown in Figure 2.6.

As the figure shows, the original program, marked as ORG, starts its execution at time 0 and finishes at time 10. During the execution of this program, we assume some intervals as Vulnerable **(V)** and some as Non-Vulnerable **(NV)**, which means if soft error happens on **V** interval, it leads to program failure, and, if fault occurs on **NV** interval, it will get masked. This program spends 4 units of time in **V** intervals and 6 in **NV** intervals. Now, assume that 10 random fault injection have been performed on this program. In an ideal random fault injection, one fault would happen on each unit of time, and since 6 units of execution time is **NV** and 4 is **V**, the amount of failures should be 4, or 40%. Now, consider a hypothetically FT version of the program. FT version of the program has two parts; first part is exactly similar to the original ones, and the second part is just No-operations (NOP). The execution time of the FT version is as twice as the original one, which makes the execution time of the FT

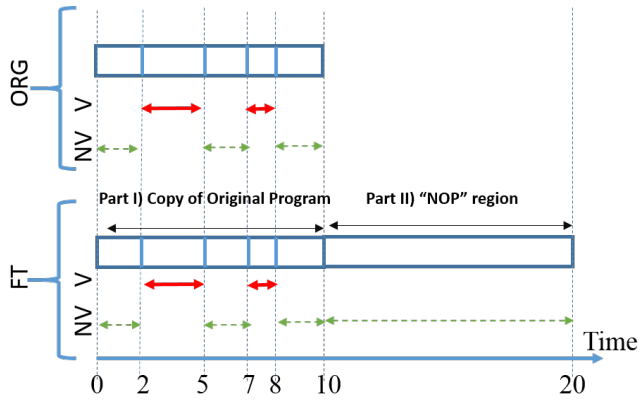

Figure 2.6: Vulnerable and Non-Vulnerable intervals for original and FT version of hypothetical program

version 20 units of time. Now, assume we perform the same random fault injection experiments that we did on the original version of the program (10 fault injection). If we randomly select 10 cycles to perform fault injection, statistically speaking, most likely 5 of them would occur on the second part of the program (the NOP execution part) which is **NV** interval, and therefore, will not result in failure. From the 5 remaining faults, happening on the first part of the FT program, 2 will happen on **V** intervals and 3 should happen on **NV** intervals. Therefore, the number of failures in this case is 2, or 20% of total injected faults. Although the ideal statistical fault injection results show significant (from 40% to 20%) reduction in failure rate, the question, however, is whether this reduction is real or just a false conclusion.

Intuitively, it is clear that the correct interpretation of SFI results should demonstrate exactly the same amount of failures for both original and FT version of the program. In this work, we use **P**probability **o**f **F**ilure (PoF) as a comparison metric that can be calculated by involving the execution time overhead of the FT version in two ways; a) adjusting the number of fault injection experiments according to the execution time overhead and just comparing the absolute number of failures not percentage. For example, injecting 20 random faults instead of 10 random faults of the

FT version of the program in the above example should get us 4 failures, which is exactly equal to the number of failures of the original program. b) Multiplying the number/percentage of failures into the execution time overhead. For example, for FT version of the program in the example above, we can use 20% * 2 = 40%, which is equal to the original program failure rate.

In addition, as pinpointed by (Schirmeier *et al.*, 2015), showing the number or percentage of detected or masked fault is also misleading. For instance, in our simple example, injecting 10 faults in FT version of the program would lead to 80% masked faults, which in comparison with the original program fault injection is 20% more. However, this fault masking improvement is the result of faults which are injected into the program-irrelevant parts of the execution of the program. Similar to NOPs, in the imaginary FT version of the above example, redundant instructions in the in-thread duplication FT schemes are also irrelevant to the main program. Faults that affect these original program irrelevant parts will result in either masked or detected/SegFault and should not be considered as the fault detection ability of the FT method.

In conclusion, since comparing the absolute or percentage of SDCs of original and protected versions of the program can result in an overestimation of the effectiveness of the software fault tolerance techniques, in this paper we use the PoF metric which was calculated as follow:

$$PoF = Percentage\,of\,SDCs \times execution\,time\,overhead \tag{2.1}$$

For the original version of the programs, the execution time overhead is considered as 1; therefore, the PoF is equal to the percentage of SDCs. For protected versions of the program, the execution time overhead is calculated as the protected program execution time divided by the original program execution time.

2.5.1 Experimental Results

Error Coverage

Graphs in Figure 2.7 present the PoF for each hardware component. In each graph, the PoF is plotted on the Y-axis for each benchmark on the X-axis. We study the PoF for original, SWIFT and nZDC versions of the benchmarks for the following components:

Figure 2.7(a) presents the FP extracted from fault injection experiments on LSQ. For the LSQ, although the FP is 3.4% for the original program, but it actually increases to 3.9% for SWIFT. As mentioned in section 2.4, SWIFT does not protect the loads and stores - they are executed only once. Therefore the LSQ is vulnerable. However, we observe that it is actually more vulnerable than the original program. This is because to implement SWIFT, we need to reserve half of the registers in the processor, and that increases the register pressure and increases the spill code - causing a spike in the number of load and store instructions. This leads to an increase in the number of entires in the LSQ, and therefore the probability that a fault will cause an error/failure. nZDC has 0% failure. nZDC is effective, since it protects loads by duplicating them, and protects the stores by reading the stored value again and checking it against the duplicate.

The result of fault injection on FUs is shown in Figure 2.7 (b). For FUs, the average FP for original and SWIFT versions of the programs are 9.8% and 1.7%, respectively. We explored the failure experiments in SWIFT, and discovered almost all of them are the result of faults affecting the FU while computing effective address of memory instructions. As expected, Zero SDC for programs protected with nZDC mechanism is observed.

Figure 2.7(c) presents the FP extracted from fault injection trails on pipeline reg-

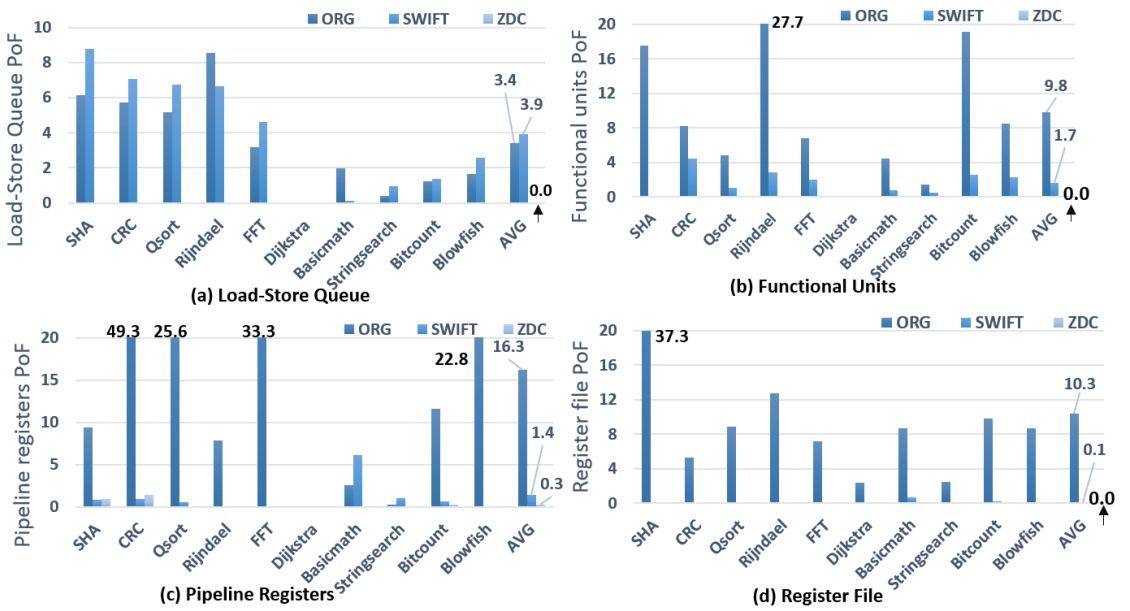

Figure 2.7: Component wise probability of failure for Original, SWIFT-protected and nZDC-protected programs

isters. As it shows, nZDC reduces the FP of the original program by about 55X to fairly close to zero, and SWIFT reduces pipeline registers FP by about 11X. In contrary with what we expect, the FP for nZDC in not absolutely zero. This discrepancy is not a deficiency of our technique, but has its roots in our evaluation methodology. We analyzed the failures, and found that the reason is because of working with unmodified library calls. The soft error happens on the destination register pointer part of a checking instruction before a library call, and the fault changed the destination register from register Zero to the register X1, which was already checked. As a result the argument value of the library function call was wrong and it produces a SDCs. These errors would not happen when all code, including libraries is treated by nZDC.

Figure 2.7(d) displays the FP for register file, which on average for Original, SWIFT and nZDC is about 10.3%, 0.1% and 0.0%, respectively. Our fault injection results on register file is in accordance with previous works (Reis *et al.*, 2005; Feng *et al.*, 2010; Mitropoulou *et al.*, 2013b) which show almost near zero SDC for SWIFT.

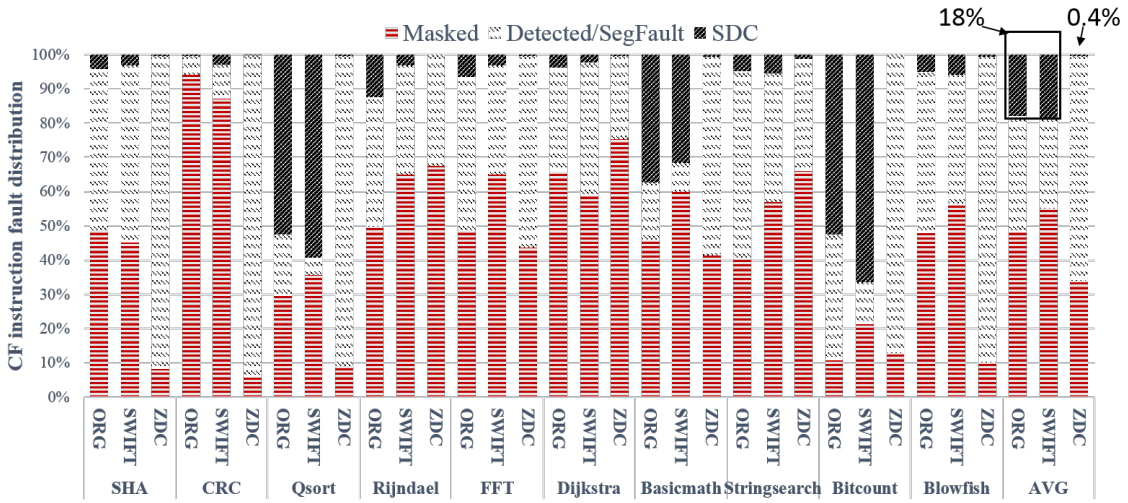

Figure 2.8: Fault injection results on CF instruction

However, because of the register file vulnerable periods (marked by vertical lines in figure 2.1), there is always a chance of SDC caused by soft error in the register file. On the other hand, since nZDC can completely close the register file vulnerable intervals by performing checking instructions after memory write instructions instead of before them, the PoF is Zero.

Figure 2.8 shows the results of fault injection on the branch address and compare instructions of the programs to examine the efficacy of the nZDC control flow mechanism. In these experiments we randomly inject faults on the original compare and branch instructions (excluding checking instructions) while in pipeline register, and conditional registers. Since the target of fault injection has been selected among the program original instructions, in Figure 2.8 we show the amount of Masked, Detected/SegFault and SDCs. The Detected/SegFault portion of the stacked bars demonstrate the percentage of injected faults which are either detected by SWIFT/nZDC or by OS as segmentation fault.

As figure 2.8 demonstrates original and SWIFT are almost identical in failure rate, about 18% SDCs! This is because, SWIFT CFC can just only detect wrong direct

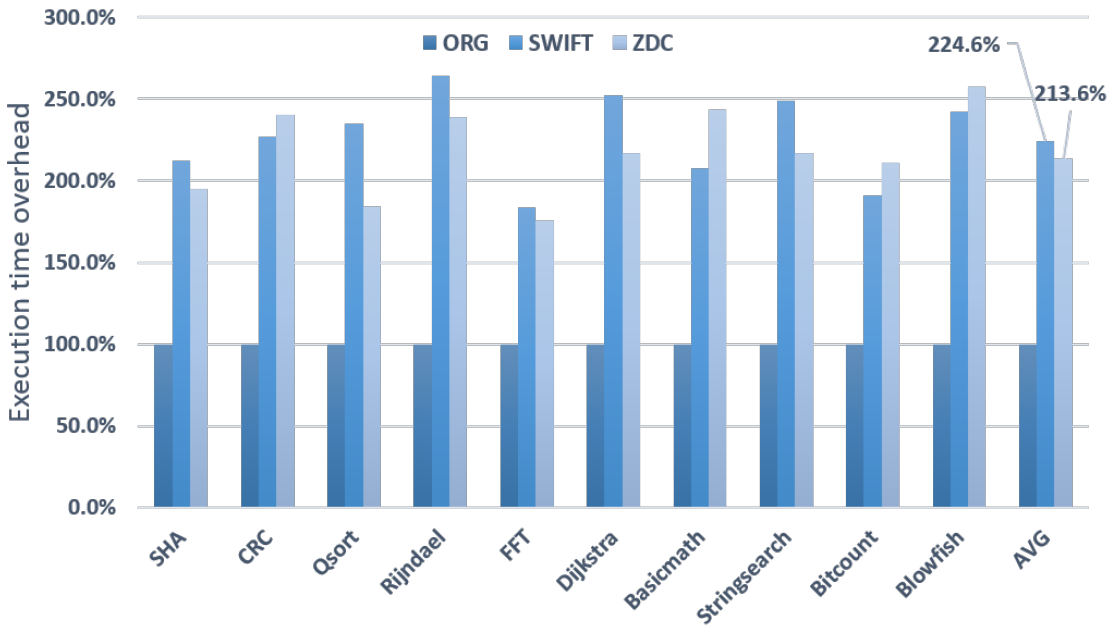

Figure 2.9: Execution time overhead for SWIFT and nZDC

branches to the beginning of a basic block not to the middle. Overall, the SWIFT CF mechanism is not effective, however, nZDC CF mechanism detects about 55% of faults and just 0.4% of faults lead to SDCs.

Performance evaluation

Figure 2.9 presents the results of performance overhead of nZDC and SWIFT for an in-order ARM processor with the configuration shown in table 2.2. On average, the execution time overhead for nZDC and SWIFT is about 224% and 213%. Performance overhead is higher than similar works, and it happened because of an inaccuracy that some of previous works have in their performance measurement. For instance, research (Feng *et al.*, 2010) assumed that library functions are protected by other means, but they do not consider the performance overhead of the library function protection, and, since a program can spend a considerably large amount of its execution time in the library calls (CRC benchmark spends more than 90% of its execution time

inside library calls), this leads to performance overhead underestimation. However, in this work, for performance overhead evaluation, we just consider the cycles that a program spends in user functions.

2.6 Summary

The significant amount of non-replicated operations in the existing software-level soft error mitigation schemes significantly restricts their error coverage. This chapter proposed nZDC error detection scheme which either duplicates program operations or verifies their executions with advanced control-flow checking and store checking mechanism. Statistical μarchitectural error injection experiments show significant error detection improvements.

Chapter 3

BACKWARD RECOVERY[1]

"Organization of redundancy and fault-tolerance for ultra-high reliability is a challenging problem: redundancy management can account for half the software in a flight control system and, if less than perfect can itself become the primary source of system failure (Owre *et al.*, 1995)."

3.1 Overview

An ideal solution for soft error resilience should mask the effect of soft errors from user and provide correct results at expected time. Many of the software-level fault tolerance techniques are *incomplete*, because they provide error detection and assume some sort of checkpoint/roll-back for recovery. Restarting a program from beginning is the simplest rollback recovery strategy. However, re-starting is not applicable in many cases, i.e, long running, real-time and interactive applications (Zhang and Chakrabarty, 2003), and even if possible it accompanies a high error recovery latency – expected recovery latency is half of the program execution time. These problems can be alleviated by building full-system checkpoints (preserving the whole memory and register stats) during the execution of a program (Duell, 2005; Lu *et al.*, 2013). However, to solve the problem of latent errors (errors which may happen before checkpointing and will be detected after checkpointing) frequent checkpoints are required, which impose unacceptable performance overhead to the system (Elnozahy and Plank, 2004; Schroeder and Gibson, 2007; Aupy *et al.*, 2013).

[1]This chapter is an enhancement of a published papers, Didehban, Moslem, Sai Ram Dheeraj Lokam, and Aviral Shrivastava. "InCheck: An in-application recovery scheme for soft errors." Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE, 2017. The publisher permits authors to include partial or complete papers of their own in a dissertation.

In-application fault tolerant techniques can potentially eliminate the need for full-system checkpointing and memory replication, while providing efficient and timely error handling by combining both error detection and recovery within the application itself. Unfortunately, the existing in-application error tolerant schemes are significantly weaker (in terms of error coverage) than corresponding error detection only schemes due to the vulnerabilities added by their complex (and unprotected) error recovery routines. For instance, SWIFTR (Reis *et al.*, 2007) was proposed to provide error recovery to SWIFT (an error-detection only technique described in section 2.2) (Reis *et al.*, 2005) by adopting forward recovery strategy. SWIFT-R divides programmer available registers into 3 redundant sets, executes 3 versions of each computational instruction and performs majority-voting between register operands of memory and control-flow instructions before their execution. Surprisingly, our analysis of SWIFTR-protected programs reveals that they suffer from ∼16x less error coverage than SWIFT-protected programs! This is because: **i)** SWIFTR-protected programs include considerably more unprotected memory instructions than SWIFT-protected programs due to high register pressure imposed by register reserving required for redundant computations, **ii)** SWIFT-R has software vulnerability windows larger than SWIFT, as it replaces light (in terms of machine instructions) error detection checks of SWIFT with heavier voting operations, and **(iii)** SWIFTR offers unsafe recovery by blindly masking the effect of certain errors on registers that may have already propagated to memory or may even have altered the program control-flow.

Realizing the above-mentioned limitations of SWIFTR, we build upon our proposed nZDC error detection scheme (explained in Chapter 2) and introduce an effective error detection and recovery scheme. Our solution, named InCheck (In-application Checkpointing and Recovery), is a software-only scheme for complete, safe & timely recovery from soft errors. InCheck makes light-weight error-free check-

points at basic block granularity, and safely reverts the program execution to the beginning of last executed basic block using preserved checkpoints. The main features of InCheck are:

i) Verified Register File Preservation. InCheck transformation not only preserves registers value into memory (no latent error), but also makes sure that the preserving process is performed correctly.

ii) Single Memory-location Checkpointing. Rather than checkpointing the whole memory state, InCheck temporarily preserves the state of each memory location before the corresponding writes to those locations.

iii) Safe & Timely recovery. Instead of performing recovery regardless of the error propagation scope, InCheck invokes a diagnosis routine which allows recovery only when its safe. The recovery latency of InCheck is negligible as it involves re-execution of just one basic-block's instructions apart from diagnosis and recovery routines.

3.2 Limitations of Related Work

Traditionally Checkpointing/rollback has been used a recovery strategy in High Performance Computing (HPC) systems. The program execution in such systems is periodically paused to save checkpoints (snapshots of the entire program state including memory footprint and register values) into a safe storage. In case of an error, program execution is resumed from the the latest checkpoint (Duell, 2005).

Applying full system checkpointing/rollback as error recovery in embedded critical applications in not efficient because of: i) Significant recovery latency (millions or billion of instructions depends on the checkpointing interval), ii) Unacceptable performance overhead (frequent checkpointing is required for latent error recovery (Aupy *et al.*, 2013)) and iii) Need of extra safe storage ($\sim$ 0.5-1 Giga bytes per checkpoint).

In fact, this huge overhead of frequent and multiple checkpoints (required for successful and fast recovery from silent errors) has restricted the usage of such recovery techniques to HPC systems alone (Elnozahy and Plank, 2004; Schroeder and Gibson, 2007).

To overcome the limitations of full checkpointing, techniques like Encore (Feng *et al.*, 2011), Clover (Liu *et al.*, 2015) and FASER (Xu *et al.*, 2013) propose fast and low-overhead recovery schemes by taking advantage of the idempotent regions of the program codes. Idempotent segments of a code do not have any Write after Read (WAR) dependencies. Therefore, multiple re-executions of such region always produces same result after execution. Nevertheless, idempotent-based recovery techniques have been designed for non-critical applications and have narrow fault coverage. For instance, as mentioned in Encore paper (Feng *et al.*, 2011), faults which cause a write into the wrong memory address or control flow errors (a fairly large amount of errors) cannot be recovered even if they occur in idempotent region of code. Hence, the ineffectiveness of idempotent-based recovery schemes makes such solutions unsuitable for critical applications.

3.2.1 Coarse-Grain Forward Recovery

Forward recovery schemes which are based on the triplication and voting strategy, eliminate the need for checkpointing and can provide timely recovery. Forward-recovery can take place at coarse-grained (like process/task/thread replication) or fine-grained (assembly-level instruction replication) modular redundancy. State-of-the-art coarse-grained techniques like PLR (Process-level Redundancy (Shye *et al.*, 2009)) apply redundancy at process-level, and perform voting between the redundantly-computed system call arguments at system-call boundaries. However, since errors may affect a program without manifesting themselves in system call arguments, PLR

approach is not effective for critical applications. For instance, if a system call (like `fwrite`) takes a memory pointer and size as arguments, there can be errors in the actual data referenced by the pointer even if the arguments (redundantly computed pointers and size) are equal. In addition, software voting operations and the execution of system call themselves are the single-points-of-failures in such techniques.

3.2.2 Fine-Grained Recovery

Fine-grained assembly-level techniques (Reis *et al.*, 2007, 2005; Oh *et al.*, 2002a; Didehban and Shrivastava, 2016; Mitropoulou *et al.*, 2013b; Xu *et al.*, 2013; Yu *et al.*, 2009; Wang *et al.*, 2007) are the most related techniques to the work presented in this chapter. They are very popular as they can potentially provide high degree of reliability. This is because such techniques are implemented as machine-level instructions, and have the ability to effectively check for the errors which may cause a failure, i.e, in memory operations and control flow direction. Unfortunately, existing complete fine-grained techniques, SWIFTR (Reis *et al.*, 2007) and FASER (Xu *et al.*, 2013), suffer from significantly higher failure rate that the fined-grained error detection schemes.

SWIFTR: Unsafe Recovery

The idea of providing the error detection and recovery at assembly-level was introduced in SWIFTR (Reis *et al.*, 2007) research paper. SWIFTR (SWIFT+Recovery) is developed based on the well-known SWIFT (Reis *et al.*, 2005) error detection only technique. It executes three versions of program's computational instructions with different sets of registers and performs 2-of-3 majority-voting between redundant registers values before memory and control-flow instructions to mask the effect of error from computations. However, SWIFTR transformation not only imposes significant performance overhead, it also increases SWIFT failure rate due to following reasons:

1) SWIFTR transformation results in considerably more unprotected instructions than SWIFT. SWIFTR requires about two thirds (∼66%) of programmer available registers for error recovery, while SWIFT reserves about half of registers for error detection. This extra register preservation forces compiler to generate more memory instructions which are single-point-of-failure in SWIFT-based transformation (Didehban and Shrivastava, 2016). To quantify the effect of register reservation on programs, we compiled Mibench (Guthaus *et al.*, 2001) programs with LLVM 3.7 (Lattner and Adve, 2004) infrastructure for ARMv8-A architecture which has 32 general purpose integer registers. [2] We found out that SWIFTR imposes more than 4x memory operations compared to SWIFT. Note that, FASER (Xu *et al.*, 2013), a SWIFT-based backward-recovery, also shares this problem with SWIFTR.

2) SWIFTR suffers from considerably larger software vulnerability window than SWIFT. SWIFTR increases software vulnerability window (interval between value checking and actual usage of that value) of SWIFT, because it replaces light error-detectors (1-2 machine instructions), with expensive majority-voting operations (8-10 machine instructions). Figure 3.1 illustrates the software vulnerable window of SWIFT and SWIFTR transformations for a simple memory write operation. SWIFT just checks the stores value (`r1`) and address (`r2`) registers, against their shadows (`r1*` and `r2*`) before the actual memory write instructions. Therefore, there is a small interval between the checking and using of register values. If an error happens in the register during this interval, error may remain undetected. SWIFTR transformation, on the other hand, requires performing majority-voting operations between redundant registers that are used for stores value (`r1`, `r1*` and `r1**`) and address (`r2`, `r2*` and `r2**`) operands. Since the software implementation of the 2-of-3

[2]Implementation of SWIFTR and FASER on an ARMv7 microprocessor is problematic (if not possible) because only 16 user visible registers are available.

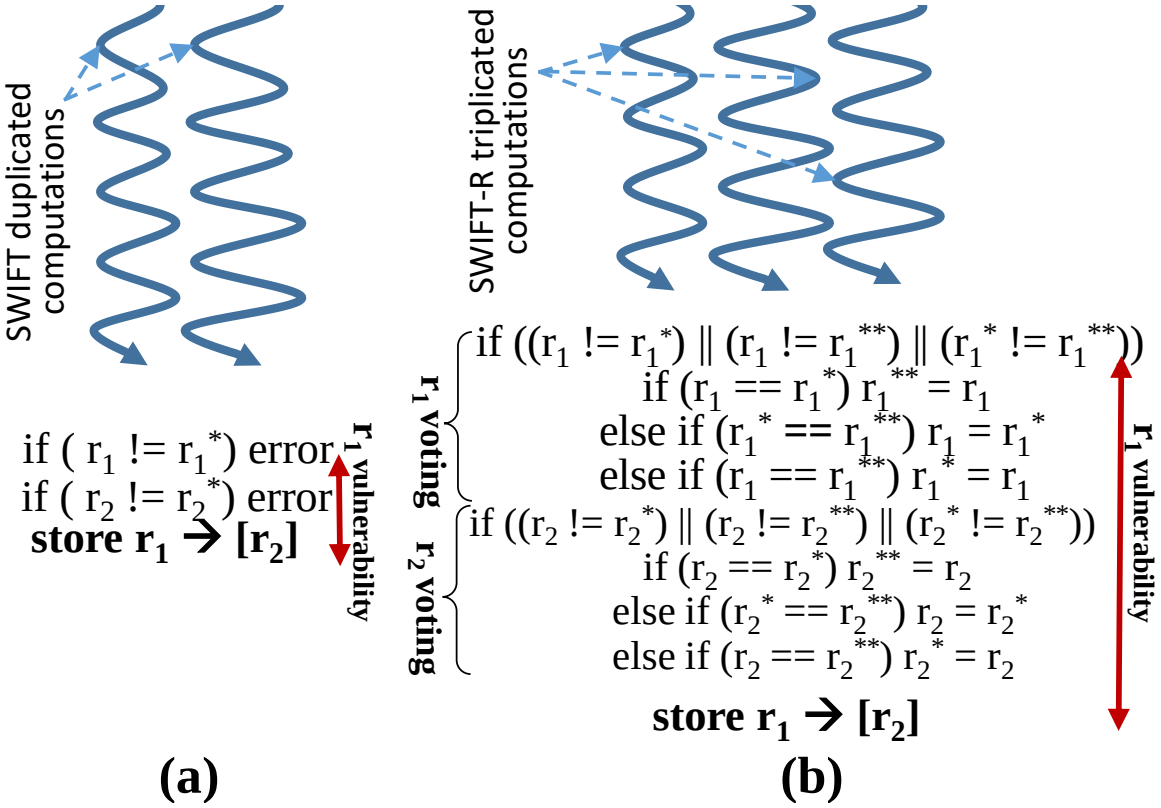

Figure 3.1: SWIFTR (part b) software vulnerability interval is considerably more than SWIFT (part a)

majority-voting needs more machine instructions than just error checking, software vulnerable intervals of SWIFTR is longer than SWIFT. As SWIFTR demands more memory (therefore voting) operations, the number of these software vulnerability intervals are considerably more than those in SWIFT. Note that these software vulnerability intervals are considerable because they exist before all memory, control-flow and function call instructions.

3) SWIFTR unsafe recovery eliminates the effect of error on registers even though memory or control flow is faulty. If an error happens during SWIFT vulnerability window, it probably corrupts the memory state and/or alters the program control-flow. SWIFT can still detect such errors, if the mismatch between redundant registers reaches to the successive error detectors. However, since SWIFTR

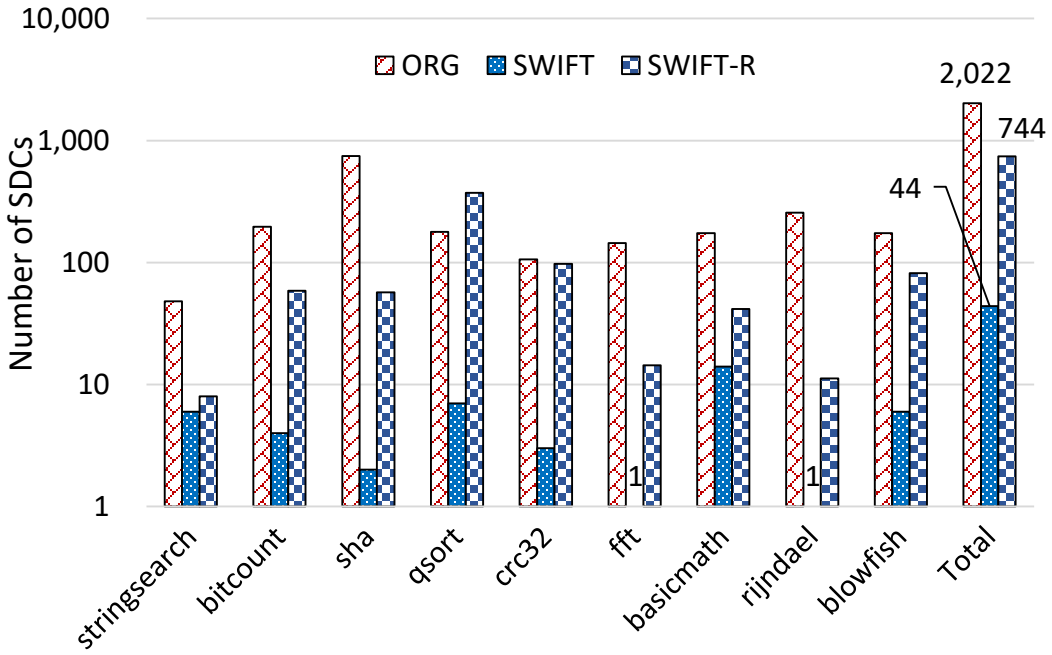

Figure 3.2: SWIFTR protected programs experience more than 16x failure than SWIFT-protected ones!

and FASER recovery schemes, try to blindly recover from any discrepancy (without determining the scope of error propagation), they can mask the effect of error in registers while the memory state or the program control flow is erroneous.

In order to quantify SWIFTR negative impacts on failure rate, we performed 54k fault injection experiments on register file of a simulated microprocessor while running SWIFT and SWIFTR protected programs. The details of the simulator configuration and fault injection set up are presented in section 3.4. Figure 3.2 demonstrates the results of fault injection experiments. As it can be seen, SWIFTR transformation caused ∼16x more failure (SDC) than previous error detection version!

3.3 InCheck: Our Proposed Fine-grained Backward Recovery Solution

In this section, we propose a safe and timely recovery scheme which in combination with nZDC (explained in Section 2.4) as the underlying error detection scheme,

will serve as a complete, safe and timely error handling strategy. InCheck error handling process can be divided into two main parts: DF (Data-Flow) and CF (Control-Flow) error recovery. DF error recovery (shown in Figure 3.3) consists of four main steps: 1) Verified Register Preservation. It takes place at the beginning of each basic block and stores the value of live registers into the checkpointing area of memory. InCheck makes sure that no error can cross the preservation phase by checking the process of preservation itself in-addition to checking the reserved register values. 2) Single Memory-location checkpointing. Right before each store instruction, InCheck preserves the data presented in about-to-be-updated memory location to a specific register. This register will further be used for memory restoration in the case of errors. 3) Checks for Safe Recovery. InCheck diagnosis routine checks if the error is recoverable. This is necessary because safe recovery is not possible in some cases. For instance, errors which cause a write into a memory location different from the one that the backup load reads from cannot be recovered because the backup itself is not valid. 4) Timely Recovery. The program execution is resumed from the beginning of basic block after the Memory and Register file state is restored to fault-free state that was present at the beginning of that basic block. Timely recovery is possible since the overall operations needed for diagnosis and recovery are implemented within 100 instructions.

Control-flow error recovery is similar to DF error recovery, however, the challenge is to determine from where the program re-execution should be restarted. InCheck CF diagnosis routine separates wrong-direction (errors which alter the direction of branch) CF errors from wrong-target ones(errors which cause an illegal jump), and provides recovery for the former and safe-stop for latter.

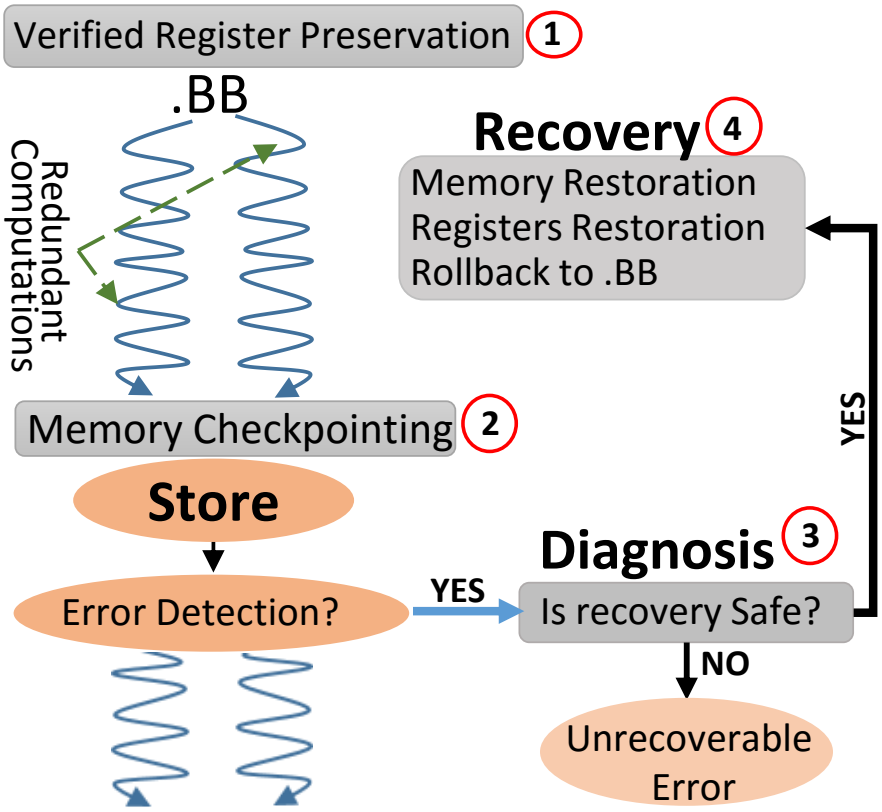

Figure 3.3: InCheck data-flow error recovery Overview

3.3.1 Verified Register File Preservation

InCheck saves the values of live registers into a designated memory location called register preservation area at the entrance of each basic block. Error-free registers should only be preserved and the preservation process itself should be error-free to prevent failures. InCheck validates the correctness of preservation process by applying checking-load strategy introduced in Chapter 2 Section 2.4 – It loads back the saved register value from the register preservation area and checks that against its shadow register. Note that the program counter register (PC) is always considered as live and gets preserved. The PC preservation is crucial for recovery from control-flow errors (described in section 3.3.5) detected in fan-in basic blocks that potentially contain multiple re-execution points.

3.3.2 Single Memory-Location Checkpointing

InCheck introduces a novel and efficient method for memory checkpointing. Rather than saving the entire memory state or being rely on idempotent regions of code (Feng *et al.*, 2011; Liu *et al.*, 2015; Xu *et al.*, 2013), it just backs-up the memory location which is about to be updated to facilitate safe recovery. Since memory subsystem can be protected by ECC, it is intuitive to avoid saving the whole memory state. However, ECC is ineffective if memory write operation data or address is faulty. Therefore, the previous value of about-to-be-updated memory location is needed for memory restoration.

InCheck provides memory checkpointing by inserting a load instruction (back-up load) from the exact address as the following memory write instruction into an specific register, named MBR (Memory Back-up Register). InCheck transformation forces compiler to break down basic blocks with potentially conflicting memory operations (multiple write and read operations from the same memory location) into sub basic blocks without such memory dependencies. This basic block purification is required for recovery from basic blocks with conflicting memory operations, because InCheck on-the-fly single memory location checkpointing strategy just provide backup for one memory location.

3.3.3 Checks for Safe Recovery

One of the features of InCheck that distinguishes it from its related techniques is its diagnosis routine that's essential for safe recovery. Basically, if error affects the execution of redundant computations or error detection instructions (shown is Figure 3.3) the error is always recoverable. Since the data will be written into an unknown (unbacked-up) memory location, if error impacts the execution of store instruction

in such a way that the effective memory address gets modified, the error should be considered as unrecoverable.

Figure 4.4 shows an example of InCheck data-flow error detection and diagnosis. The first `load` (left side of the Figure, before `store`) is the "back-up load" which performs on-demand memory checkpointing. The error detection takes place after the `store` instruction, and program control goes to diagnosis routine in the case of a mismatch. Firstly, diagnosis routine checks for errors in the computation of `store` value register (`r1`). If a mismatch is observed, the error is flagged to be recoverable. The program flow then jumps to the recovery routine (not shown in Figure). Secondly, diagnosis routine checks for mismatch from the `store` address register. Depending on the time of error occurrence, it may or may not be recoverable. If error occurs on address register `r2` before the back-up load, the error is flagged to be recoverable. In this case, the back-up for wrongly updated memory is available and thus memory restoration is possible. However, if the error happens after the execution of back-up load, the recovery is not possible since the value of MBR is not the same as previously updated wrong memory value. To determine the time of occurrence of error, diagnosis routine loads the data back from the memory location with the same address as `store` into a temporary register (`temp`) and compares that against `store` value register (`r1`). If they match, it assumes that error has modified the address of both back-up load and the `store` in the same way (back-up is valid) deems it recoverable. In the third step, diagnosis routine compares the value of MBR to `temp`. If different, it implies that the `store` has written incorrect data into right memory location. This type of error is also recoverable because memory back-up is valid. In the fourth step, diagnosis checks for errors on detection instructions which are just false alarms and easily recoverable. False alarms can be checked by repeating the error detection instructions. Ultimately, if none of the above situation were true, the diagnosis routine declares the error as

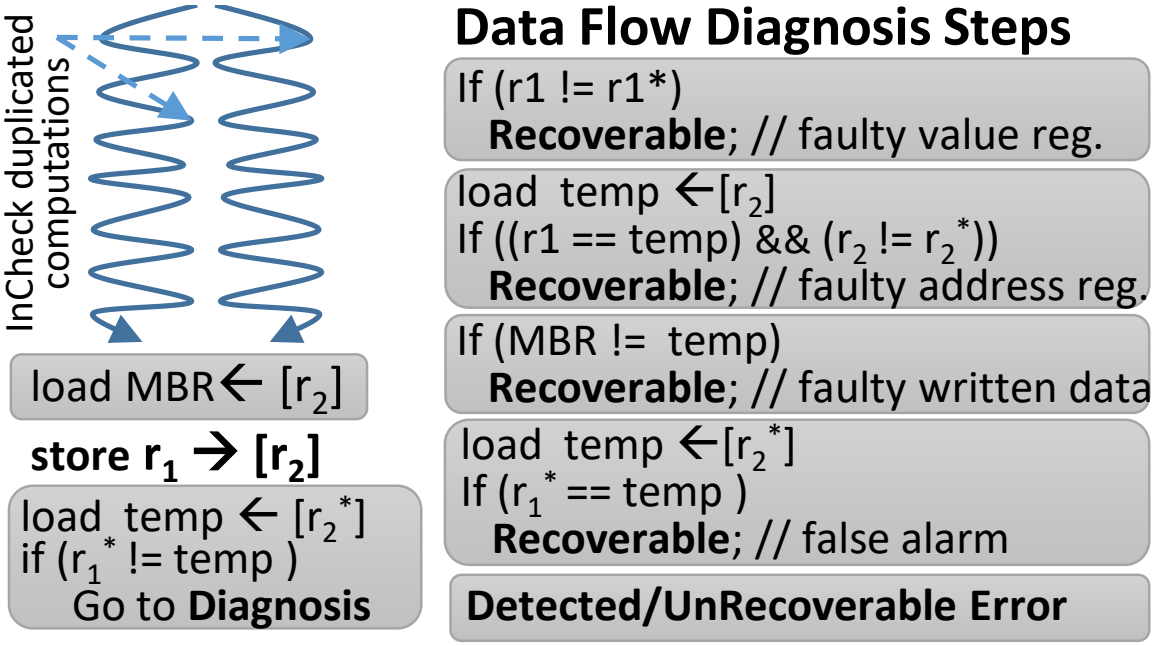

Figure 3.4: An example of InCheck data-flow diagnosis

detected/Unrecoverable and terminates program execution.

3.3.4 Timely Recovery

In the last phase of InCheck error handling, the actual recovery takes place by performing memory and register file restoration and re-executing the program from the beginning of the basic block. First, the memory state will be restored to the same state as before the write instruction. This is done by writing the MBR register into the memory write target location. Error-free live registers will then be loaded back from the register preservation area to the corresponding registers. Finally, the main program execution is resumed. Since the first two steps of InCheck (safe register preservation and memory checkpointing) should be executed in all (erroneous or error-free) cases, the recovery latency of InCheck is equal to the execution time of diagnosis and recovery routines and replicated instructions (instructions from the beginning of basic-block till the error detection point). Since the diagnosis, recovery routines

and the average basic block size are small, the overall recovery latency is practically negligible.

3.3.5 Control-Flow Error Recovery

InCheck employs nZDC control-flow (CF) checking mechanism (explained in Section 2.4), but in addition to nZDC CF error detection checks (positioned close to the end of each basic block) it also performs error detection checks at the beginning of each basic block (before safe register preservation). If a CF error gets detected by the first checks, InCheck invokes the corresponding CF-specific diagnosis routine. These routines are different from DF diagnosis routine (described in section 3.3.3), and their responsibility is to determine if the detected CF error is a wrong-direction or a wrong-target error. A control-flow error will be considered as wrong-direction error, if the last preserved PC is in the list of predecessors of the current basic block. If that is the case, error will be treated as recoverable and recovery takes place by restoring memory and register values to the initial state of the previously executed basic blocks. Otherwise, the error will be considered as wrong-target CF error and errors which are detected by nZDC CF error detectors (positioned at the end of basic block) will be considered as unrecoverable – diagnosis routine terminates the execution of program. Fortunately, as (Shrivastava *et al.*, 2014) demonstrated, most of control-flow errors are wrong-direction errors and are therefore recoverable by InCheck error handling scheme.

3.4 Experimental Methodology

To quantify the effectiveness of InCheck, we implemented InCheck and SWIFTR (as the the-state-of-the-art related work) techniques as late back-end passes in LLVM 3.7 infrastructure (Lattner and Adve, 2004) for an ARMv8-a ISA (64-bit architec-

ture). We compiled 9 programs from Mibench benchmark suite (Guthaus *et al.*, 2001) with *-O3* compiler optimization flag. For each program three versions (Original, InCheck and SWIFTR) were produced. It should be noted that all experiments and results were performed on user functions (library functions and system calls were excluded).

We performed extensive fault injection experiments on major sequential hardware components of a modern ARM cortex A-53 like simulated microprocessor. Experiments were performed on gem5 (Binkert *et al.*, 2011), a cycle accurate μarchitectural-level simulator with the configuration shown in Table 2.2. We performed single bit-flip fault injection experiments on major core components including, integer register file, issue and decode pipeline registers, functional units and load-store unit buffer registers. For each component 2000 faults were injected per version of program, which means 72,000 (4 * 2000 * 9) faults per each program version – overall 216,000 (72k * 3) faults. For each fault injection experiment, a target component and a (*bit*, *cycle*) were randomly selected before the simulation run. Once the simulator reaches the target fault injection *cycle*, simulation is paused and the selected *bit* is inverted. The simulation then resumes with the faulty value until it gets terminated or reaches the allowed simulation time (3x of normal execution time). The result of each simulation run is classified as one of the following:

1) Masked: Program terminates normally and the output is correct.

2) Failed/SDC: Program terminates, but the output is incorrect.

3) Detected/Unrecoverable: This outcome occur just in InCheck protected programs, and happens when an error is detected, but cannot be recovered from.

4) Others: Program encounters a fatal error, such as segmentation fault or simulation time reaches its limit.

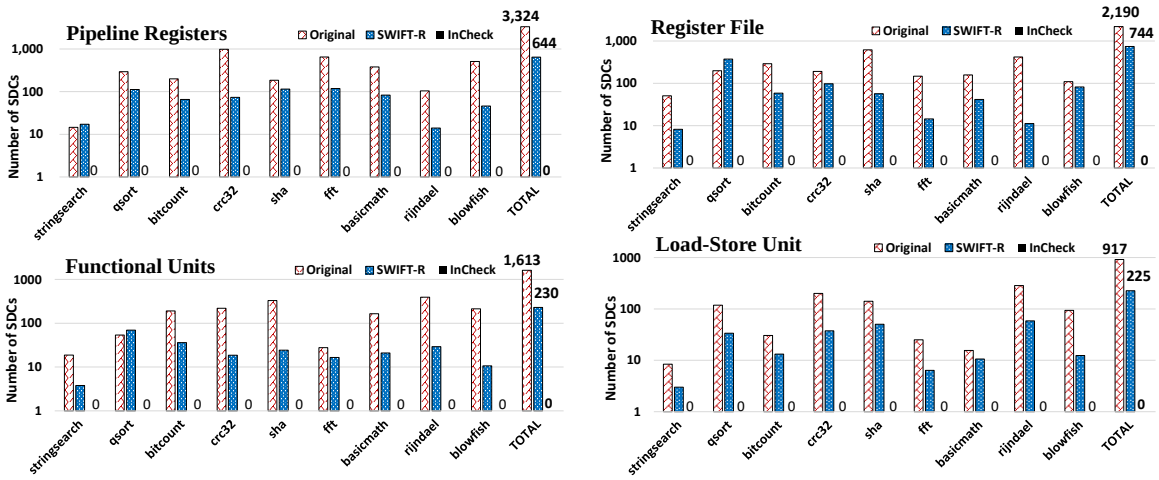

Figure 3.5: SDCs Distribution in Component-wise Fault Injection Experiments

3.5 Experimental Results

3.5.1 Error Coverage

Figure 4.7 depicts the absolute number (in logarithmic scale) of failures (SDCs) per hardware component. We did not use fault coverage metric, because it can be misleading (Schirmeier *et al.*, 2015)). Regardless of the target fault injection component, InCheck-protected programs never resulted in a failure! This implies that 1) No error could skip InCheck+nZDC error detectors, 2) The diagnosis routine always distinguishes recoverable errors from unrecoverable ones accurately, and 3) If the detected error was recognized as recoverable, the recovery routine is always successful. InCheck is extremely effective as it protects functionally-related instructions of the program as well as error handling (preservation and checkpointing) operations. However, in comparison to original programs, SWIFTR transformation reduces the overall failure count by 4.3x (5.2x, 2.9x, 8x and 4x for pipeline registers, register file, functional units and load-store unit, respectively). Our investigation from failed experiments reveals that SWIFTR provides correct recovery only from the faults which affect the computational instructions, and the rest of the faults either get masked by

the programs or lead to failures or segmentation faults.

InCheck-protected programs can potentially recover from Soft Errors which lead to segmentation faults if their diagnosis routines initialize at the beginning of signal handler functions of applications. Since in gem5 system call emulation mode, the simulator terminates the program execution without forwarding segmentation fault signals to the application, the results shown here do not fully demonstrate the InCheck recoverability.

InCheck Diagnosis routine declared around 96% of detected faults as recoverable. In less than 4% of the cases, diagnosis routine provided safe-stop and prevented failure by terminating the program. If left unterminated, these unrecoverable faults could have either directly impacted the execution of a memory write operation or caused an unexpected jump in the program. Restarting can anyways be employed as recovery strategy in these scenarios.

3.5.2 Performance Overhead

Figure 3.6 shows the execution overheads of InCheck+nZDC and SWIFT-R protected programs normalized to Original Program. It can be seen that on an average, an InCheck version of a program can run 36% faster than its SWIFTR equivalent. InCheck is faster because it pushes the uncommon diagnosis and recovery routines off the critical-path of execution. The performance overhead of frequent live register preservation is acceptable, because the corresponding memory preservation locations are usually presented in the cache and will therefore execute fast. Furthermore, the performance overhead of back-up loads (inserted right before program store instructions) are also not significant, because they do not cause any more memory misses – if the data is not in the cache, miss is inevitable. If not by back-up load, it will eventually happen by store instruction itself.

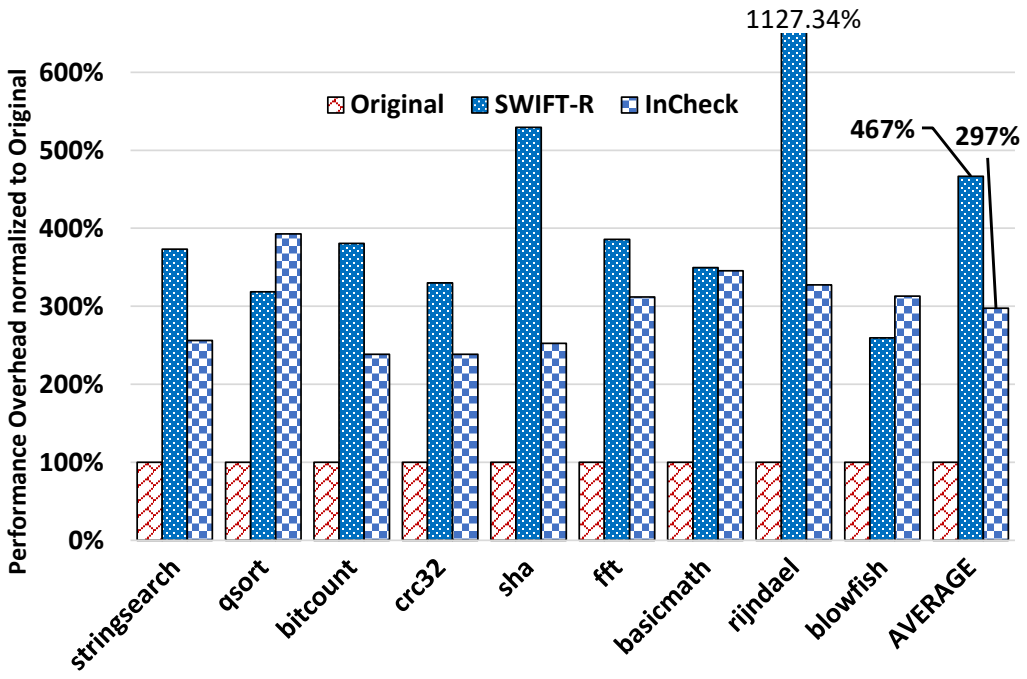

Figure 3.6: Execution Overhead of SWIFT-R & InCheck

To quantify the recovery-latency of InCheck, we counted the average number of extra instructions which were executed when an injected fault was detected and recovered. On an average, the InCheck recovery spans for 180 dynamic instructions. This latency is unnoticeable in most cases.

3.6 Summary

In this chapter we present InCheck, as an in-application soft Error detection, diagnosis and recovery scheme. InCheck protects the execution of resilince-related routines like checkpointing operations as well as main program instructions. InCheck uses verified register file preservation and single-memory-location checkpointing. We also performed diagnosis after error detection to provide safe recovery. Fault injection experiments demonstrate that InCheck offers quicker and better error recovery compared to the state-of-the-art approaches.

Chapter 4

FORWARD RECOVERY[1]

This chapter presents NEMESIS a compiler-level fine-grained soft error resilience technique that enables computation even in the presence of soft errors by adopting a forward error recovery scheme. It replaces computationally expensive software majority-voting routines with cheap error detectors. On detecting an error, diagnosis and recovery routines allow for quick recovery, and continued error-free execution with a reasonable performance overhead.

4.1 NEMESIS: Overview

NEMESIS is a set of compiler transformations which provide a soft error hardened code by adding redundancy and reforming control flow of the original code. NEMESIS partitions programmer-available machine registers into three sets, called M-regs (Master Registers), D-regs (Detection Registers) and R-regs (Recovery Registers), and runs three independent sequences of instructions, named M-stream, D-stream, and R-stream. The M-stream has all instructions needed for functionally correct execution of a program. D-stream is a redundant copy of M-stream which does not include any memory write and functional call instructions, however, it does include all arithmetic, memory read, compare and branch instructions. R-stream just contains arithmetic and memory read instructions – register-modifying instructions. This is because that R-stream results only will be used in majority-voting to mask the effect of the errors from the general purpose register file. In addition to these three redun-

[1]This chapter is an enhancement of a published papers, Didehban, Moslem, Aviral Shrivastava, and Sai Ram Dheeraj Lokam "NEMESIS: A software approach for computing in presence of soft errors." Computer-Aided Design (ICCAD), 2017 IEEE/ACM. Publisher permits authors to include partial or complete papers of their own in a dissertation.

dant streams, a NEMESIS-protected program includes error detection, diagnosis and recovery instructions. NEMESIS assumes ECC-protected caches and memory, and its sphere-of-protection includes the entire microprocessor core (excluding memory subsystem). The objective of NEMESIS is to detect and correct the effect of all transient faults which may lead to SDC or timing failures. These errors are the hardest to detect and correct because they usually do not generate any visible symptoms like exceptions, segmentation faults (Wang and Patel, 2006) and not easily detectable by low-cost symptom-based error detection techniques. The salient points of NEMESIS, and why it is effective are mentioned below:

i) NEMESIS protects the execution of all (critical and noncritical) instructions: This is the main philosophical difference between SWIFTR (most related work explained in Section 3.2.2) and NEMESIS. SWIFTR tries to mask the impact of error from register operands of critical instructions, but it does not check the execution of critical instructions themselves. NEMESIS, on the other hand, verifies the execution of all programs instructions by making sure than the program executes the right store instructions, correctly. NEMESIS achieves that by checking the outcome of program branch and memory write instructions. In the case of any discrepancy, NEMESIS calls a diagnosis routine and then attempts to recover from the error. For store instructions, error detector verifies if the stored value is correctly written into the correct memory location by loading back the stored data and checking that against the D-stream computed store value. For branch instructions, direction check takes place by executing the corresponding D-stream compare and branch instruction, and verifying the destination basic block. Nevertheless, if the presence of any error is detected, a diagnosis routine will get invoked and determines the scope of error in the program. If the error is diagnosed as recoverable (marked as 1 in Figure 4.1), the effect of error from the register file and memory is eliminated, then the recovery

routine re-executes the corresponding critical instruction and the program execution resumes. Otherwise, if the diagnosis routine declares the presence of an unrecoverable error(marked as 2 in Figure 4.1), the program execution will be terminated.

ii) NEMESIS transformations leaves no software vulnerability window: Software vulnerability window, defined as the duration between checking a value in software and the time to use the value, exist in almost all existing software-level techniques. This interval can be a major source of failure especially for Voting-based techniques. Software voting, not only has vulnerable periods, but also imposes considerable performance overhead. Instead of voting, NEMESIS just checks for errors in the results of critical instructions. Since this checking will take place after the execution of store instructions, NEMESIS needs to preserve the value inside each memory location before write, for recovery purposes. This strategy eliminates the software vulnerability window – no unprotected interval between value checking and usage. If there is no error, then the execution proceeds. Otherwise, diagnosis routine decides about the faith of program. If error considered as recoverable, the recovery scheme uses a voting mechanism and memory preserved data to eliminate the impact of errors from register file and memory. Note that since the execution of post-error handling routines is rare (just in the case of error), they usually do not have much impact on the performance.

4.2 NEMESIS: Details

4.2.1 Memory write operation error detectors

NEMESIS utilizes the load-backing data flow error detection strategy of nZDC-based protection schemes (Didehban and Shrivastava, 2016; Didehban *et al.*, 2017a). It detects errors on memory write instructions by loading back the written value

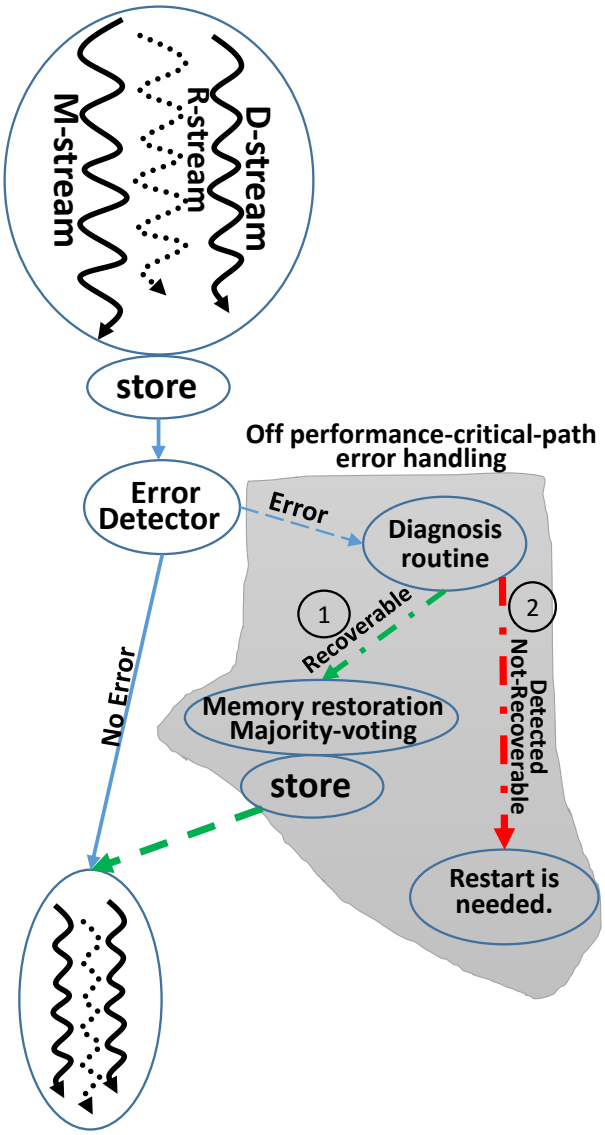

Figure 4.1: NEMESIS data-flow error handling strategy. After each store instruction, the Error detector unit checks for errors, and if any observed, the diagnosis routine will get involved and classifies the error as either Recoverable or Detected/not-recoverable. If an error is recoverable, memory and register restoration will take place and program continues with executing the store instruction. Otherwise, the program stops the execution by raising an error flag.

from the memory and comparing it against corresponding D-stream computed value. However, such post-store error detection strategy cannot detect errors affecting the address of silent stores and in this section we present our solution for such undetected errors.

Silent store vulnerable window. By definition, a store is said to be silent if it writes a value into a memory element which is already holding the same value (Bell *et al.*, 2000; Lepak *et al.*, 2001). If an error affects the execution of a silent store, i.e, alters stores effective address, the write operation can make a random modification to the state of memory and the error cannot be detected by load back strategy because the loaded value from the memory is as same as the stored vale.

Figure 4.2 exemplifies an undetected error case in store-loadback strategy. As it shows, the store is silent because the value in memory location addr, *val*, before executing `store` instruction (upper part of fig.4.2) is equal to the values which are computed by main and detection streams, val^M and val^D, respectively. Therefore, the state of memory should not get changed by the execution of `store` instruction. Now, assume that the soft error hits the base address register of the store, and alters the store's effective address from `addr` to `f-addr`. Consequently, the store writes its data into the faulty memory address `f-addr` rather than `addr`, and changes the state of memory while it is not supposed to do so(lower part of fig.4.2). This error remains undetectable, since the following checking-load instruction will load the value, *val*, from the correct address (computed by the detection stream), $addr^D$, which is equal to val^M and val^D. Note that simply inserting a check for the base address register `store` wouldn't solve the problem since the error can alter the store address without affecting the address register, i.e, errors affecting functional unit or pipeline register while processing the store instruction. Since silent stores can consist around 18% to 64% of total program's store instructions (Bell *et al.*, 2000), fixing the silent store

vulnerability is important in critical applications.

First-cut solution for silent store vulnerable window. Since silent stores do not alter the state of microprocessor, not executing such useless instructions will eliminate their vulnerability without harming the correctness of program. Thus, an obvious solution could be to jump over silent stores in the program. Figure 4.3(a) illustrates the first cut attempt for eliminating the silent store problem from the store-loadback error detection strategy. In the Figure redundantly computed values of M-/D- and R-streams are differentiated by a superscript M, D or R letter. For instance, val^M, val^D and val^R denote redundantly-computed store's value by M, D and R streams, respectively. Initially in this scheme, a silent-check load (inst. 1) reads back from exactly the same address that `store` is going to write into, and saves the loaded value in an specific register, called Silent Check Register (SCR). Then, SCR is compared against the M-stream computed store's value, val^M (inst. 2), to determine if the `store` is silent. If the condition is true, the program jumps over the `store` instruction, otherwise, `store` (inst. 3) will get executed and the following checking-load (inst. 4) instruction reloads the store's written value from the memory into SCR register. In the end, regardless of the store being silent or not, the SCR register should get checked against the value produced by D-stream, val^D (inst. 5), to make sure if store's value,val^M, was computed correctly.

NEMESIS solution for silent store vulnerable window. Although the first-cut method solves the problem of silent stores, it introduces yet another possibly undetected error scenario as the silent-check instructions (inst. 1, 2 in Figure 4.3(a)) themselves are unprotected. Therefore, if any error alters the effective address of a silent-check load instruction in such a way that the wrongly loaded value is equal to the store value, a non-silent store will be treated like a silent one, and memory state would not get updated when it must. We named this type of errors as missing-memory

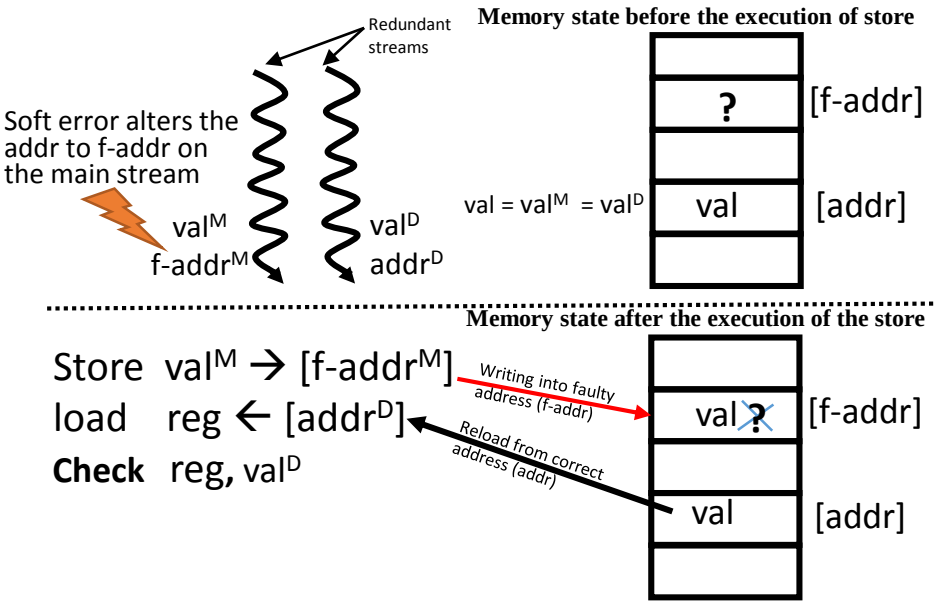

Figure 4.2: Silent store undetected error scenario in checking load mechanism. Since the store instruction is silent, writing into the wrong memory location error could not get detected by checking load instruction.

update errors. These errors differ from Silent Store scenarios by a fact that the former does not change the state of memory while it should, and the later updates the state of memory while it should not. Note that similar to the silent-store problem, adding one more checking instruction for the address register would not solve the problem, since errors can alter the load address in data-path or load-store unite while the base address register is error free.

NEMESIS error detection scheme counter-intuitively eliminates the problem of silent-store vulnerability and missing-memory update by redundant and intertwined execution of silent-check operations. This is best explained with pseudo code example shown in Figure 4.3(b). Firstly, the value within the `store` destination memory location is loaded back into two specific registers, VCR (Value Check Register) and SCR (Silent Check Register) by two redundant `load` instructions (marked as inst 1 and 2 in part (b) of Figure 4.3) which use M- and D- stream's redundantly-computed registers as their address operands. Then in order to find out whether the `store` is

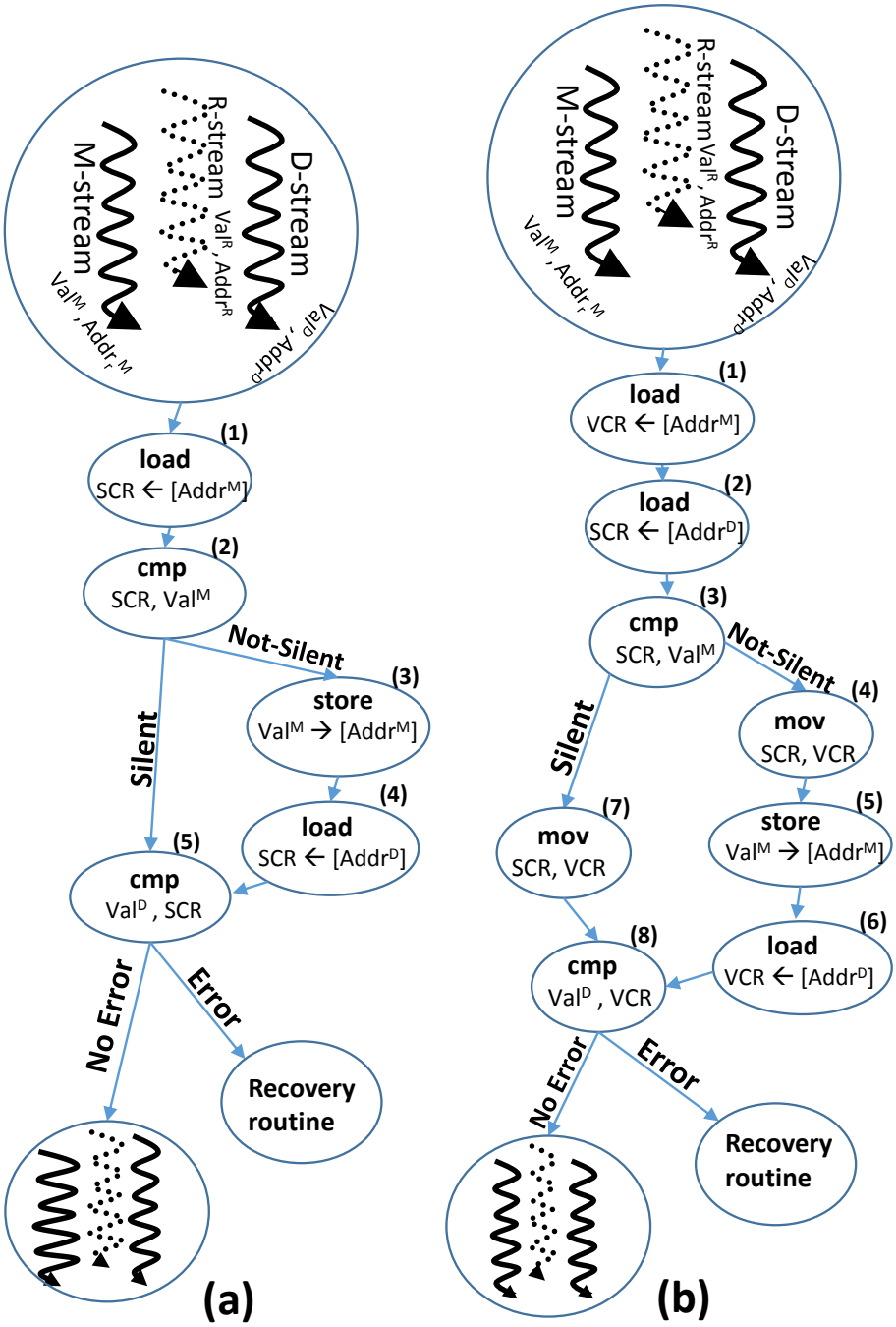

Figure 4.3: Memory write instruction checking mechanisms. (a) The first-cut solution which suffers from missing-memory update error and (b) NEMESIS memory write checking mechanism which solves the problem of silent-store and missing-memory update.

silent, the SCR register gets compared against the `store` value register computed by M-stream, val^M. If the `store` is identified as silent, the VCR register gets compared against the `store` value register computed by D-stream (inst. 8) for error detection. Since, the results of redundant silent-check loads are compared with the two M- and D-stream computed store's values, the missing-memory update error will get detected. Now assume that the store is not silent, and `store` and `checking-load` (inst. (5) and (6)) will get executed. Instruction (5) performs the actual memory write operation and instruction (6) reloads the written value back to the VCR register. The VCR register then gets compared against val^D, which is the redundant copy of `store` value computed by D-stream, for error detection purpose (inst. 8).

Since in NEMESIS transformation the effect of the error is detected after memory write instructions, it is necessary that a backup of about-to-be-written memory location preserved before each store instruction. Fortunately, the previous state of memory is already loaded to VCR register (inst. (1)). However, since this value will be overwritten by the checking load instruction (inst. (6)) in the case of not-silent stores, a copy of VCR is preserved in SCR register (inst. (4) and (7)).

4.2.2 Diagnosis/Recovery for Memory Write Errors

The main responsibility of NEMESIS diagnosis routine is to decide whether a detected error is recoverable. A memory write detected error is considered as recoverable if the state of the register file and memory can revert to an error-free state before the execution of store instruction. Generally, the effect of error from registers can be masked by performing 2-of-3 majority-voting between corresponding M-, D- and R- registers. And, the memory state can be rolled-back to an error-free sate by writing back the backed-up data into the memory. However, there are two rare cases in which NEMESIS diagnosis routine declares a detected error as not recoverable:

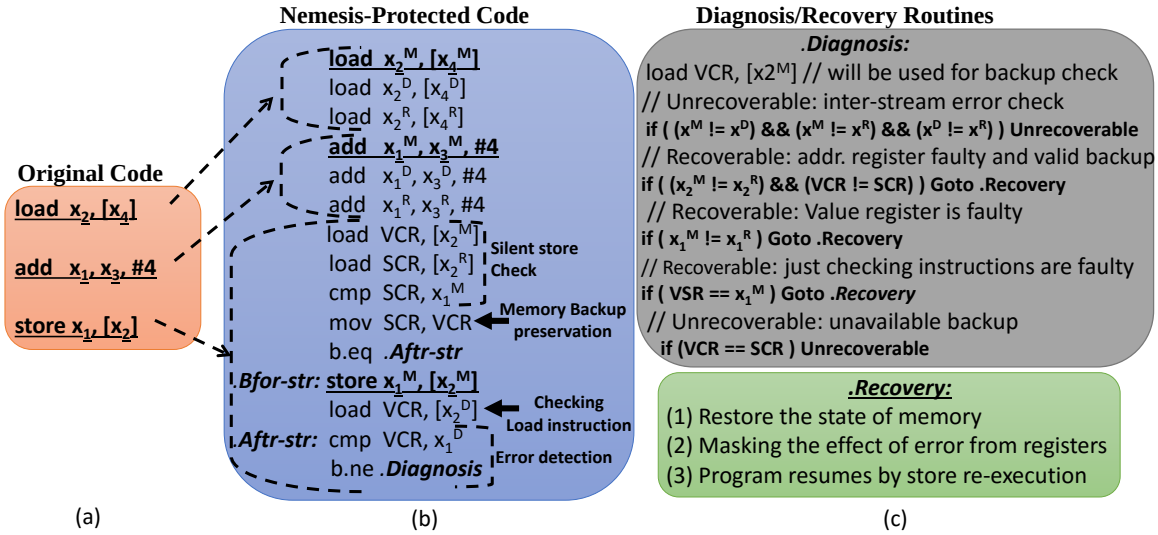

Figure 4.4: An example of NEMESIS memory write error detection, diagnosis, and recovery. Part (a) shows the original code. Part (b) shows the Nemesis transformation for error detection and recovery, the original instructions are distinguished from the error management operations by being underlined. Part (c) shows NEMESIS off performance-critical-path post error diagnosis and recovery routines.

Case 1: Inter-stream error propagation. If the effect of error has crossed the boundary of redundant streams, it is possible that all three redundant-computed registers contain different values, and, therefore, performing majority-voting cannot mask the effect of the error. For example, consider an error on the decode stage of the processor pipeline that alters the destination register pointer of an M-stream instruction to a D-stream register which is going to be used as the operand for the corresponding redundant D-stream instruction. In this case, the three corresponding redundant registers will be holding different values, and, such an error can be detected but is unrecoverable.

Case 2: Unavailable memory backup. If the memory write instruction modifies a different memory location from the preserved one, the error is unrecoverable. Errors affecting store address register after the first silent-check load and before the

store instruction (between inst. (1) and inst. (5) of part (b) of Figure 4.3), or error altering the effective address of store instruction while it is processing in the processor are deemed to be detected but unrecoverable errors.

If none of the above-mentioned cases are encountered, the diagnosis routine provokes the recovery block in which memory restoration takes place by writing the backup data into the faulty written memory address, and then, followed by majority-voting between registers. Once the effect of error gets eliminated from the memory and the registers, a program can continue its error-free execution by retrying store instruction.

4.2.3 Fault Coverage Analysis for Store Instructions

In order to clarify how NEMESIS memory error detection, diagnosis and recovery work, in this section we explore the effect of some different errors on a sample NEMESIS-protected code, presented in Figure 4.4. Note that the order of checks and the structure of diagnosis and recovery blocks remain the same for different memory write instructions, but the register numbers and labels need to be customized per store instruction.

1) Error affects computation of store value register during the execution of redundant streams. Assume that error happens during the execution of the M-stream `add` instruction and the value saved in x_1^M register becomes erroneous. Depending on the effect of the error on the data within the x_1^M register, two different cases can happen: 1) Wrong value written to the correct memory location, or 2) skipping a memory update by considering non-silent store as silent.

In the first case, in order to error propagates to the memory the erroneous value in register x_1^M has to be different from the data within the about-to-update memory

location (saved in SCR), otherwise, the execution jumps over the `store` instruction. In this case, the error will be discovered by NEMESIS error detection checks because the loaded back value (saved in VCR register) will be different from the x_1^D (the D-stream computed version of x_1^M). In the diagnosis routine, we first load from the store memory address into VCR register, which will be used for checking 1) the (un)availability of memory backup and 2) the correctness of checking-load instruction. Then, the diagnosis routine checks for inter-stream error propagation in store value and address registers. Nevertheless, for this example, we assumed that the inter-stream error propagation is not the case. In the next step of diagnosis, if the `store` address register is faulty and the memory backup is valid, the error is considered as recoverable and the program will go to the recovery block. However, in this case, the memory write address register, x_2^M, is not faulty and the diagnosis routine will proceed to the next step. Now, it is time to check for errors in the `store` value register which is positive, and the recovery block will be engaged. In the recovery block, first, the effect of error from the memory will get eliminated by writing the memory backup data (preserved in SCR register) into the `store` target address memory location. Then, the effect of the error from the registers will be corrected by performing majority-voting. And, finally, the error-free program execution continues from before memory write instruction.

In the second case, error alters the data of x_1^M register in a way that the erroneous value became equal to the value loaded from an about-to-update memory location, which is saved in SCR. In this case, the non-silent store will be treated as a silent one. However, based on the facts that (1) the VCR value is equal to SCR (computed redundantly with correct load instructions), (2) the SCR value is equal to x_1^M-faulty value, and, (3) x_1^M value is different from x_1^D because of the fault, the value of VCR register will not be equal to the x_1^D value, and the program goes to the diagnosis/recovery

routine. Diagnosis/recovery routine will behave exactly similar to the first case, and recovery block gets engaged by the third check (mismatch in `store` value register).

2) Error alters the effective address of a store instruction during its execution. If an error occurs on functional unit flip-flops during the calculation of effective address or errors on load-store unit queues while processing a memory write request, a correct value will be written into a wrong memory location.

Considering the fact that in an NEMESIS-protected program, only not-silent store instructions will actually execute, these errors are easily detectable by the load-back strategy, because the reloaded value is different from the D-stream computed version of `store` value (x_1^D). However, recovery from these types of errors seems extremely hard, because the wrongly-written memory location is unknown. In the diagnosis block, a copy from the `store` memory target location will be saved in VCR register initially. The next four checks, inter-stream error propagation, `store` address register, `store` value register and checking-load instruction execution checks will take place in order, and no discrepancy will be found. Ultimately, the diagnosis routine will perform its last check which reveals that the `store` has not written its data into the memory location that it is supposed to write into. It is concluded from the fact that the value within the `store` target memory location has not changed by the execution of the non-silent store. In this case, the presence of an unrecoverable error will be announced by the diagnosis routine.

4.2.4 Control Flow Error Detection

Soft errors can alter the control-flow of a program by producing unexpected jumps or wrong-direction branches in a program. An unexpected jumps arises when the program control-flow alters in a way which is not permitted in its control flow graph (CFG). Errors which directly modify to PC register or alter the target address of a

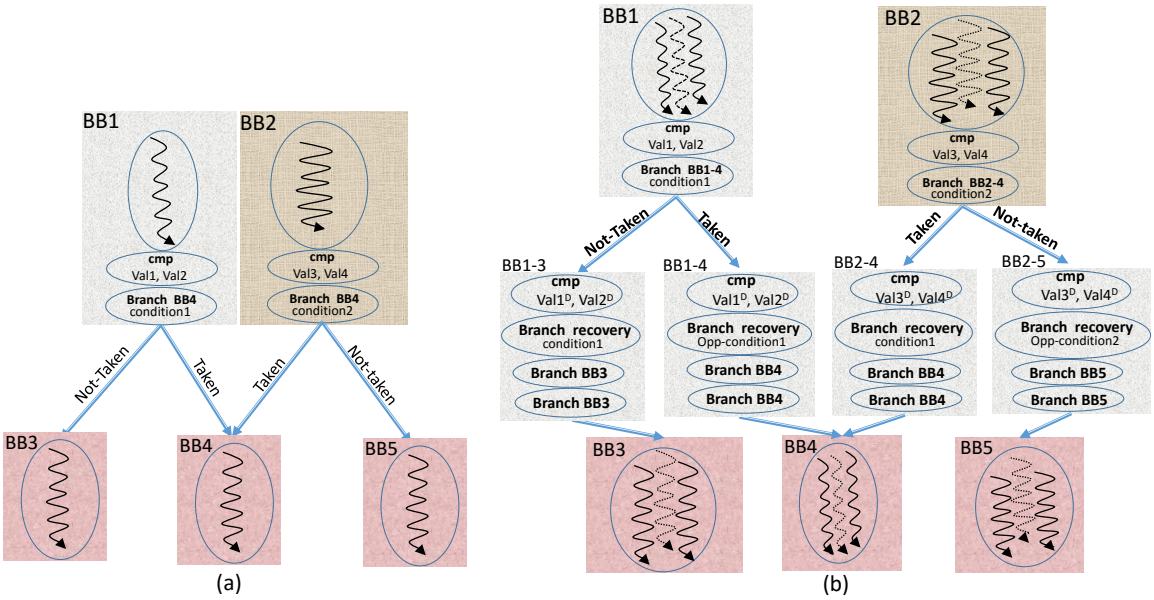

Figure 4.5: Control flow protection in NEMESIS. (a) shows unprotected program control flow. (b) shows NEMESIS branch direction double checking mechanism.

taken branch instruction (if the branch is not taken the error will be masked) are examples of unexpected jumps. Such control flow errors can be detected or even recovered with signature-based control flow checking techniques (Oh *et al.*, 2002b; Vemu and Abraham, 2011; Vemu *et al.*, 2007). A wrong-direction control flow occurs when a branch direction changes from taken to not-taken or vice-versa, which can be caused by errors affecting the compare instruction register operands, execution and the opcode of compare and branch, or even program status flag registers. If such error occurs, all three streams will steer to the wrong direction and will execute the wrong sequence of memory write instructions. The wrong-direction control flow errors are more frequent than wrong-target errors, and, cannot be detected with most of the existing signature-based control flow checking techniques (Shrivastava *et al.*, 2014; Didehban and Shrivastava, 2016). Hence, the focus of NEMESIS transformation is to detect and recover from wrong direction control flow errors while a signature-based control flow checking technique can be employed for handling unexpected jumps

control flow errors.

In order to detect wrong-direction control flow errors, NEMESIS double checks the direction of a conditional branch by placing an intermediate block, called direction-check block, between each source and destination BBs (Basic Blocks) in the original program control flow. These intermediate, direction-check blocks are necessary because they make direction double checking possible even for multiple-entry destination BBs. In each direction-check block, NEMESIS inserts a redundant (D-stream) compare instruction and announces the presence of error only if the redundant compare changes the direction of the conditional branch in a different way from the original (M-stream) compare instruction. However, if no error is detected, the program control goes to the destination BB by a direct branch which is positioned at the end of direction-check block. These unique features of NEMESIS control flow checking mechanism provide complete wrong-direction error detection and also maximize the masking effect of compare instruction.

Figure 4.5 demonstrates NEMESIS control flow transformation for a simple program which has both, single-entry (`BB3` and `BB5`) and multiple-entry destination (`BB4`) basic blocks. In the Figure, NEMESIS direction-check blocks are marked as `BB1-3`, `BB1-4`, `BB2-4`, and `BB2-5`. A direction-check block contains four instructions, a D-stream copy of the source BB compare instruction, a conditional branch instruction to the diagnosis/recovery block, and two redundant direct branches to the destination BB. The condition of the conditional branch instruction in a direction-check block is specified in such way that it will not change the control flow of the program in a fault-free run, and, the control flow will reach to the destination BB. For that purpose, if the control flow changes when the branch is not-taken (from `BB1` to `BB3` or from `BB2` to `BB5`), the condition of the conditional branch instruction in the direction-check block is as same as the condition of branch in the source BB. On the other hand, if

the control flow changes from source to the destination BB when a branch is taken (from `BB1` or `BB2` to the `BB4`), the condition of the conditional branch instruction in the direction-check block will be opposite to the condition of the branch in the source BB. For instance, if the branch in source BB is "b.eq (branch equal)", the conditional branch in the direction-check block will be "b.ne (branch not equal)".

In a fault-free run of a program, the conditional of the conditional branch in direction-check block is always false, and the control flow goes to the destination BB with the first direct branch instruction. However, if soft error alters the direction of the source BB branch, the conditional branch in the direction-check block will change the control flow of the program to the corresponding recovery block. Note that, the second direct branch in the direction-check block (the last instruction) just gets execute if an error affects the execution of first directed branch in a way that it cannot change the control flow of the program.

4.2.5 Control Flow Error Diagnosis/Recovery

The control flow error diagnosis routine is simple because it just needs to check for inter-stream error propagation, and, if that is the case, the error is consider as detectable/not-recoverable. Otherwise, diagnosis routine transfers the control of the execution to the recovery block, where, majority-voting takes place between compare register operands, and the program resumes its error-free execution from the M-stream compare instruction.

4.3 Experimental Methodology

4.3.1 Compilation and Simulation Framework

In order to evaluate the effectiveness of NEMESIS fault tolerant technique, we implemented NEMESIS and SWIFTR techniques as late back-end passes in LLVM 3.7 infrastructure (Lattner and Adve, 2004) for an ARMv8-a ISA. This implementation enabled us to take advantage of the all advanced compiler optimization. We compiled twelve benchmarks from MiBench benchmark suite (Guthaus *et al.*, 2001) with *-O3* compiler optimization flag. For each program we produced three versions, Original, SWIFTR and NEMESIS. Please note that we did not modify the standard library functions and therefore we exclude them from all of fault injection and performance overhead evaluation results shown in this work. We performed extensive fault injection experiments on different hardware components of a modern, high-performance low-power, ARM cortex-A53 like microprocessor simulated in gem5 (Binkert *et al.*, 2011) a cycle accurate simulator. Table 2.2 shows the details of the processor configuration.

4.3.2 Fault Model and Fault Injection Set-up

Fault model and fault sites: We inject single bit-flip per execution as our fault model in this work. We injected faults on different bits of various hardware components including general purpose integer register file, pipeline decoder and instruction queue registers, integer functional units and load-store unit buffers. Injecting faults only in register file does not accurately capture the effect of soft errors that happen in the entire system (Shrivastava *et al.*, 2014). Very importantly, injecting faults in register file will not cause inter-stream error propagation. However, they can happen for example, when errors happen on an instruction register and alters a register

number from one set of registers (M-/D- and R-regs) to another. Our fault injection scheme – where we insert faults everywhere in the processor – allows for such errors.

Number of fault injections experiments and outcome classification: In an attempt to cover all cases, we randomly inject 100,000 faults for each version of a program per fault site. For each version of a program, we injected 400,000 (4 * 100,000) faults in four hardware components. Overall, we inject 14,400,000 (400,000 * 3 * 12) fault injection experiments. According to (Leveugle *et al.*, 2009), these extensive fault injection experiments provide us more the 95% confidence interval with less than 0.1% error rate, which is 10x less error rate than previous works (Didehban and Shrivastava, 2016; Mitropoulou *et al.*, 2013b; Feng *et al.*, 2010; Reis *et al.*, 2007).

For each fault injection experiment, a target component and a (bit, cycle) are randomly selected statically. Once the simulator reaches the target fault injection cycle, simulation is paused and the selected bit is flipped, then, the simulation run resumes its execution till simulation terminates or the allowable simulation time gets over. The result of each simulation run is classified into one of the following category:

Masked: Program terminates and the output is correct. Note that faults, which are recovered by SWIFTR and NEMESIS techniques, also count as masked faults.

Failed: Program terminates normally, but, the output is incorrect. This is the case of Silent Data Corruptions or SDCs.

SegFault: Program terminates by generating some symptoms such as segmentation fault or simulation time is over. We consider segmentation faults as detected, and focus our work on Silent Data Corruptions that go undetected, and are hardest to catch, and therefore recover from.

Detected/Not-Recoverable: In this case, program terminates by announcing the presence of a unrecoverable error. This type of outcome can only happen while the

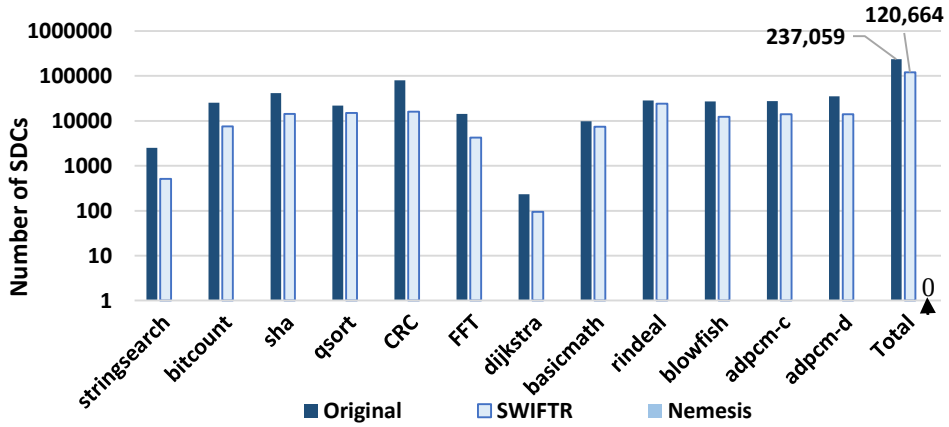

Figure 4.6: Out of 15 million fault injection experiments (evenly distributed between original, SWIFTR and NEMESIS versions of programs), 237K result in SDCs in the ORG program, 120K in the SWIFTR program, and 0 in the NEMESIS program.

fault is injected during the execution of an NEMESIS-protected program.

4.4 Evaluation and Analysis

4.4.1 Fault Injection Results

Figure 4.6 shows the number of fault injection runs that resulted in SDCs for unprotected (ORG), SWIFTR and NEMESIS versions of the programs. The Figure shows that of the 5 million runs, the original program results in SDC about 237K times. The SWIFTR version ends up with 120K times occurrence of SDCs. As compared to these, NEMESIS protected programs do not cause any SDCs.

Figure 4.7 depicts the number of fault injection experiments that lead to SDCs per hardware component. We show the actual number of failures because, commonly used metrics such as fault coverage or even the percentage of masked faults may lead to fault detection/correction overestimation in techniques like NEMESIS or SWIFTR which expand the bit-cycle fault space of the original program (Schirmeier *et al.*, 2015). Note that the Y-axis of the graph, or `Number of SDCs` is in exponential scale and that the last bar in each graph shows the total number of experiments that

Figure 4.7: Fault injections in different hardware component of simulated microprocessor never lead to failure, while running NEMESIS-protected programs.

resulted in SDC. We can see from all the four plots that fault injection in NEMESIS-protected programs never resulted in SDCs. This implies that NEMESIS error detection is able to detect all injected faults. The NEMESIS diagnosis routine always correctly distinguished recoverable errors from unrecoverable ones, and the recovery routine always eliminated the effect of error completely. NEMESIS error detection is able to detect all errors because it checks for the results of critical instructions rather than their operands, and it covers infrequent cases like silent-stores and missing-memory updates. In comparison with original versions of programs, SWIFTR-protected programs, on average, produced 35%, 73%. 86% and 89% less failures for faults injected in load-store unit, register file, pipeline registers and functional units, respectively. Surprisingly, in some cases such as fault injection in the load-store unit for programs like basicmath and rijndeal, SWIFTR transformation actually increases the number

of failures! The reason is that in comparison with original codes, SWIFTR transformation dramatically increases the number of memory operations for register-hungry programs and leaves them unprotected. NEMESIS transformation also increases the number of memory operations, but NEMESIS protects them either by triplication (read operations) or loadback techniques (write operations). Voting time-to-check to time-to-use is the cause of failures due to faults in the register file for SWIFTR-protected programs. Faults that happen on pipeline registers or functional units, while they are processing the execution of critical instructions, e.g., memory, compare and branch instructions, may lead to a failure in SWIFTR-protected program. For instance, faults which convert the opcode of a compare instruction to some other instruction may change the control flow of a program, but will remain undetectable in programs protected by SWIFTR.

4.4.2 Analysis of Injected Faults

Figure 4.8 shows the fault injection experimental result distribution for original and NEMESIS-protected versions of the programs. It can be seen that very significant percentage of faults are masked, even in original programs (on average about 77%), and adding protection increases the number of masked faults, on average, by about 13%. NEMESIS-transformation also slightly decreases the number of SegFault, on average, from 11.5% to 9.5%. We believe most of these segmentation faults can be recovered by NEMESIS, if programs execute NEMESIS diagnosis and recovery routines in the their signal handling functions. However, since that in our simulation environment the simulator does not forward the segmentation faults signal (SIGSEGV signal) to the program and directely terminates the simulation, we have segmentation faults in our results. The graph also shows that just a negligible amount of faults (about 0.42%, on average) lead to unrecoverable errors.

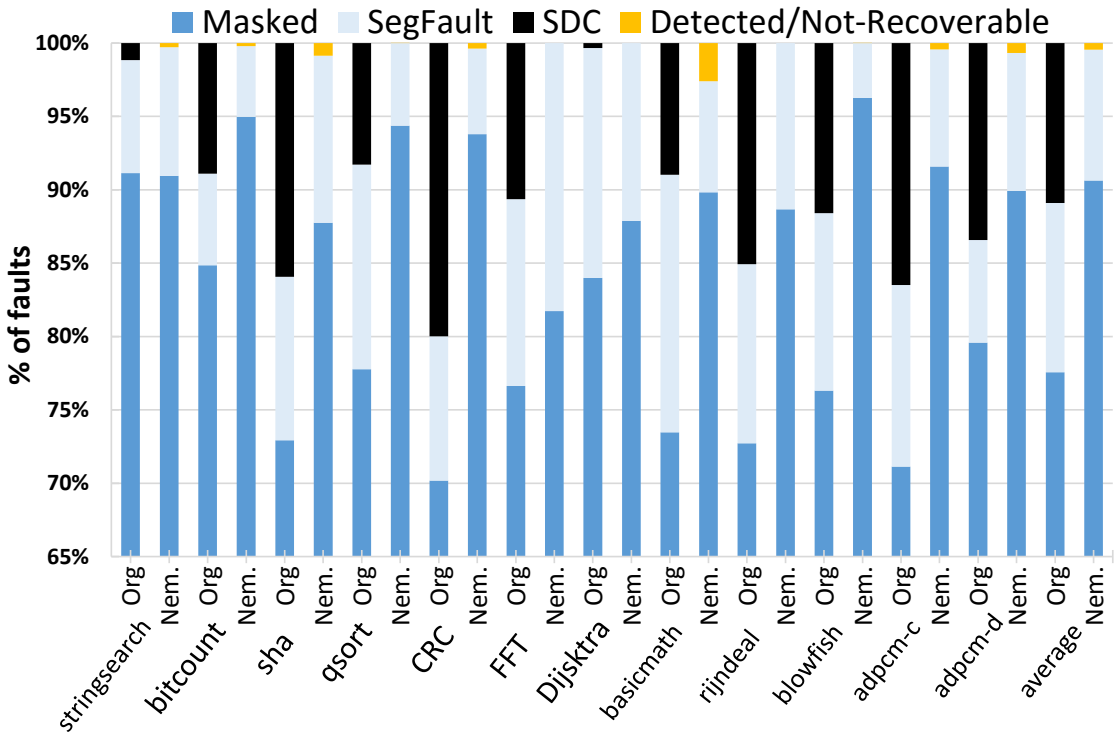

Figure 4.8: NEMESIS protected code increases the percentage of masked faults by 13%, and completely eliminates SDCs.

4.4.3 Analysis of Diagnosis Routine Outcomes

Figure 4.9 shows the outcome distribution of diagnosis routine. The diagnosis routine determines the scope of error propagation and decides if the detected error is recoverable. The graph shows that more than 97% of the detected errors are recoverable. 3% of the cases, where NEMESIS cannot recover are when the faults cause an inter-steam error propagation, or when the fault causes an error in the effective address of the store instruction during the execution of the store instruction. In these cases, NEMESIS cannot find out the correct register value (because all are different) or wrongly-written-into memory location, and therefore can detect but not recover.

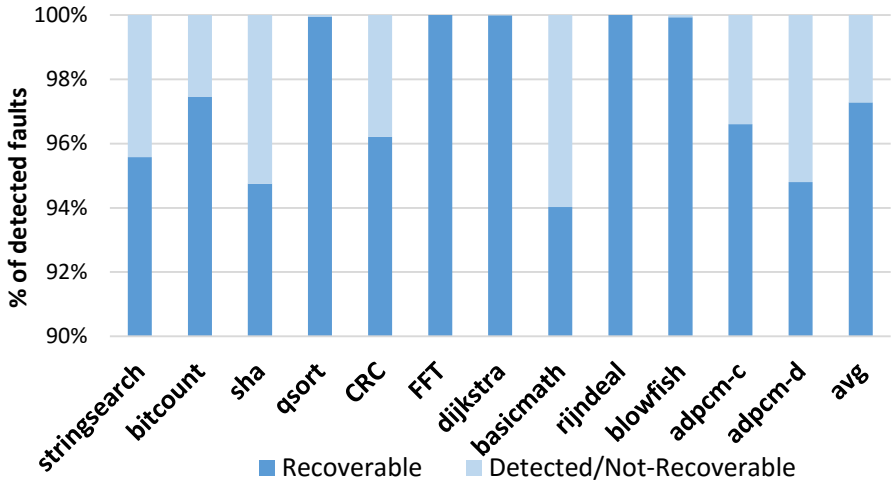

Figure 4.9: Nemesis transformation is able to successfully recover from more than 97% of detected faults, while 3% of detected fault remains unrecoverable

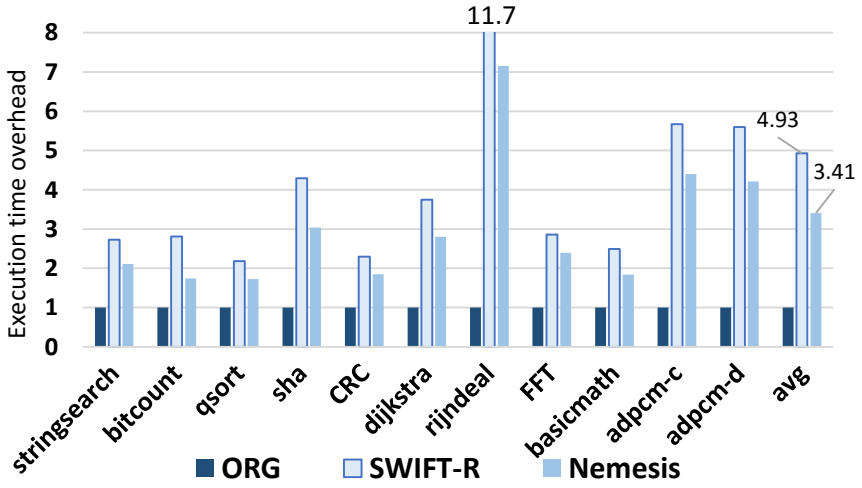

Figure 4.10: Nemesis-protected programs, on an average run 30% faster than SWIFTR protected ones.

4.4.4 Execution Time Overhead

Figure 4.10 shows the execution time overhead for SWIFTR and NEMESIS versions of the programs. Compared to the original code, SWIFTR and NEMESIS transformations, on average, increase the execution time of the program by about 5X and 3.4X, respectively. The NEMESIS-protected programs run, on average, 30% faster than SWIFTR-protected ones, because they execute less number of instructions. For instance, for each memory read operation, NEMESIS transformation just adds two extra redundant instructions while SWIFTR transformation requires a majority-voting, which needs 4 machine instructions in our implementation, and two extra move instructions. The number of extra instructions added for memory write operation protection, for both NEMESIS and SWIFTR transformations is 8 machine instruction. However, since NEMESIS does not execute silent stores, it executes on average 18% less memory write operations. For each compare instruction, SWIFTR transformation requires two voting operations (so a total of 8 instructions), however, NEMESIS transformation adds just 3 extra instructions. Note that the SWIFTR and NEMESIS register reservation for redundant instructions, impose considerably high overhead in some register-hungry applications like rijndeal.

The performance overhead reported in this work may seem higher than similar/previous works because for two main reasons: First, we exclude the number of cycles that a program spends in standard library calls from our execution time estimation, because we didn't modify such functions, and second similar works usually select a much wider and aggressively out-of-order processor, that can hide the performance overhead due to the execution of the extra instructions. Nevertheless, considering the fault coverage provided by NEMESIS, its performance overhead is reasonable and even competitive with many coarse-grain software and hardware error detec-

tion/recovery technique. This is because such techniques usually demand two extra cores for running redundant codes and some extra operations for majority-voting, which overall can be considered as more than 3x overhead.

4.5 Conclusions

We present NEMESIS as a complete solution for protecting computing against soft errors. NEMESIS is a software level redundancy-based technique which checks the results of memory write operations and the direction of branch instructions. If any violation is detected, NEMESIS diagnosis and recovery schemes undo the effect of error from processor and memory state, and the program can continue its execution. NEMESIS strategy provides near-immediate error recovery which is crucial for workloads like real-time and interactive applications. However, those applications which do not require such rapid error recovery can still benefit from NEMESIS error detection technique as it can detect all faults that cause SDCs – a claim that most previous techniques cannot make. We evaluated the effectiveness of NEMESIS against state-of-the-art error recovery techniques by performing about 15 million processor-wide single bit-flip fault injection experiments on original and protected versions of different programs. Our experimental results show that NEMESIS protected programs do not incur any SDC. On an average, NEMESIS protected programs run 30% faster than SWIFTR protected programs.

Chapter 5

LOW LEVEL CRASH TESTING AND ADJUSTMENTS

The goal of this chapter is to quantify the error coverage capability of the proposed software-level solutions through RTL-level fault injection experiments. To propose a general (ISA-independent) solution, we first propose gZDC, a combination of nZDC data-flow error detection and NEMESIS control-flow error protection. gZDC also utilizes coarse-grained scheduling and asymmetric control-flow signatures to reduce the chance of hard-to-detect control-flow errors. Statistical single bit-flip fault injection experiments on different hardware components of a synthesizable Verilog description of an OpenRISC-based microprocessor reveals that gZDC transformations can achieve more than 99.9% error coverage.

5.1 gZDC Overview

An error may lead to failure by changing application data-flow or its control-flow. To detect data-flow errors, gZDC utilizes nZDC load-backing strategy introduced in Section 2.4. Control-flow errors can furthermore be divided into (1) Wrong direction control flow errors i.e. errors alter the direction of a branch, and (2) Unexpected jumps i.e. errors which cause a random jump to an arbitrary location. In following subsections, we describe gZDC solution for each type of these control-flow errors.

5.1.1 Wrong Direction Control Flow Errors

To detect wrong-direction control-flow errors, gZDC adopts NEMESIS control-flow detection scheme (Didehban *et al.* (2017b)) and performs redundant check on the direction of each conditional branch. It raises error detection flag only if an error

has altered the direction of the branch. Note that simple branch duplication is ineffective because usually the result of a branch is a change in program control flow (i.e. update on PC), not a value that can be simply duplicated and check later. Therefore, since simple branch duplication is useless and we use branch checking instruction to verify the direction of branches. Branch checking instructions use the redundant registers and their destinations are always error detection routine. Contingent upon the direction of the original branch instruction, the opcode of branch checking instructions (branch condition) can be either same or opposite to the original branch condition. Basically, there are two possible paths after each conditional branch instructions: (1) Taken path – when branch condition is true and control of execution should be changed, and (2) Fall through path – when branch condition is false and there will be no change in program control flow. Based on the possible outcomes of a conditional branch (taken or untaken) the opcode and the position of branch checking instructions are determined as follow:

Direction checking for taken branches: Placing checking branch instructions after taken conditional branches is useless simply because branch checking instructions would not even get a chance to be executed. A naive solution can be placing branch checking instructions right in the beginning of the branch target basic block. Unfortunately, such solution will lead to a false alarm in the cases that the branch target basic block is a merge basic block – it has more than one predecessors. For instance, consider program shown in Figure 5.1(a). In the code, control can reach to `.BB3` (target address of the conditional branch `BNE R1, R2, .BB3`) from either `.BB0` or `.BB1`. When control reaches to the `.BB3` from `.BB0` the value of `R1` register is definitely "not equal(NE)" to the value of `R2`. However, that condition may or may not be true if the control lands to `.BB3` from `.BB1` block.

To solve this problem, gZDC transformation first creates an intermediate blocks (`.BB-CH` in Figure 5.1(b)), then verifies the direction of branch by executing branch checking instruction (marked as ② in the Figure), and finally transfers the control flow to the desirable block by inserting a direct jump instructions (marked as ③ in the Figure). The condition of branch checking instruction in the taken branch cases in oppose to the original conditional branch instructions. For example, in this case the opcode of branch checking instruction in intermediate block `.BB-CH` is "BEQ (Branch Equal)" which is opposite to the "BNE (Branch Not Equal)" operation. The reason is that if the original branch is taken, the condition is true (`R1` is not equal to `R2` in our example) and the opposite condition should be false. Therefore, the branch checking instruction is always untaken and control will transfer to the destination block by the next direct jump. However, if an error influences the direction of branch and alters its direction from untaken to taken, the control flow will reach to the intermediate block wrongly. In those cases, the program control flow will be directed to the error handling block by checking branch instruction because their condition is always opposite to the original error-free branch – the checking branch instruction is taken because the original error-free branch was untaken.

Furthermore, to make sure that actual jump takes place (control redirects to the destination BB), we insert a direct jump to error detection block, after each direct jump instruction in the code (marked as ④ in the Figure). This is required for the cases which because of soft error (i.e., errors affecting the opcode of jump) a jump instruction alters to another instruction (e.g., arithmetic or even memory operation).

Direction checking for untaken branches: For cases that original branch in untaken and control flow goes to the basic block right after the branch, gZDC inserts a branch checking instruction with the exact same opcode (condition) but shadow

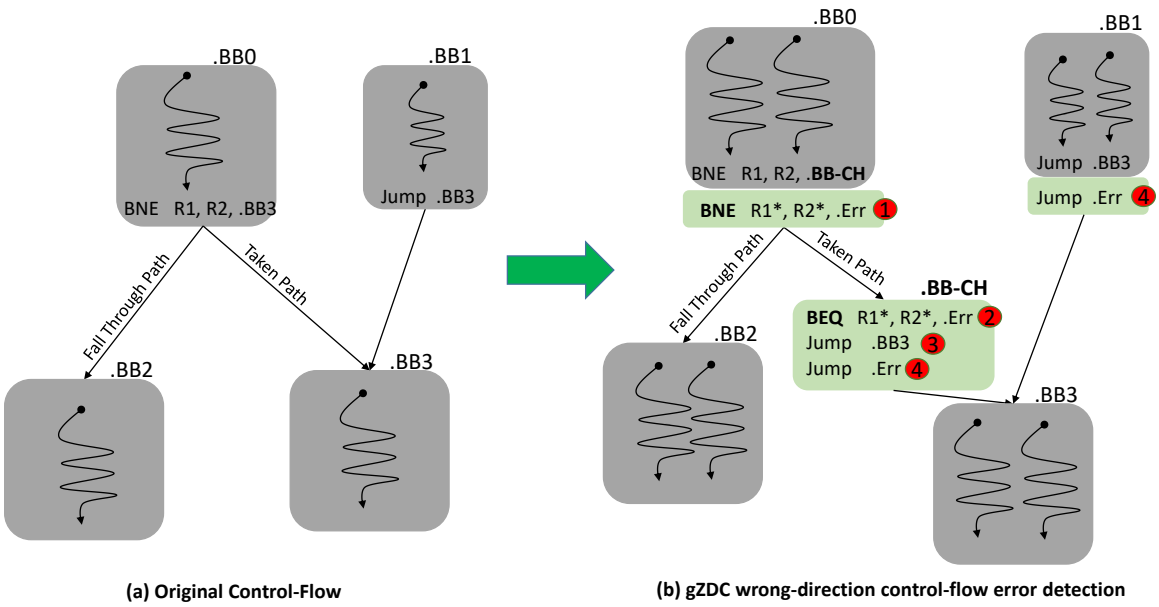

Figure 5.1: gZDC inserts a branch direction check basic block between all control flow edges from a taken conditional branch to a merge basic block. The inserted BB always composed of a branch direction-check instruction followed by two direct jump instructions.

register operands from the original branch immediately after the original branch. [1] Instruction marked as ① in the Figure is an example of branch checking instruction for untaken branches. The key point here is that if the original branch in untaken, the checking branch instruction is untaken as well and the program execution will continue as expected. However, if an error alters the direction of the original branch from taken to untaken, the checking branch instruction will direct the control flow operation to the error handling block. Since all architectures provide conditional branch operation, this solution is applicable on all targets. Note that all conditional branches in 5.1 can be changed to separate compare and branch instructions (e.g. for ARM ISA) without any affect on the protection.

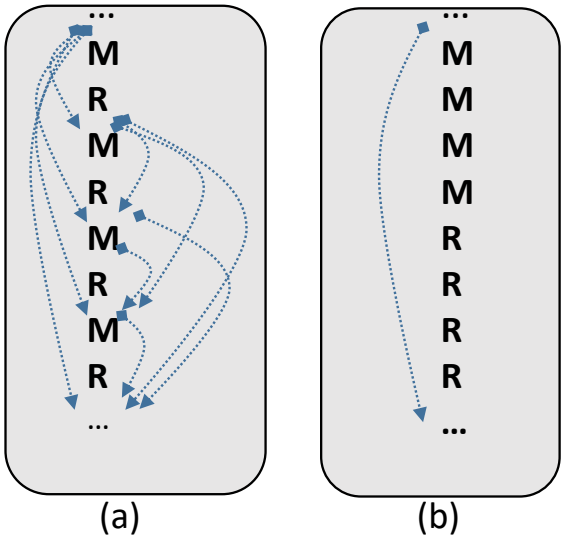

Figure 5.2: The impact of fine-grained vs coarse-gained instruction scheduling on intra-BB undetected unwanted jumps. Main and Redundant instructions are shown by M and R letters respectively and arrows represent undetected intra-BB forward unwanted jumps. Part (a) shows fine-grained instruction scheduling which leaves many unwanted jumps (dashed arrows) as undetected because such jumps cause no mismatch between the state of redundant registers. Part (b) shows coarse-grained scheduling policy which has a lesser chance of undetected unwanted jumps errors.

5.1.2 Unexpected Jumps

We define unexpected (or unwanted) jump error as causes that an error changes program control flow in a way that is not allowed by program static control flow graph. Examples are errors on PC, errors on target field of direct/indirect branch instructions, errors altering the opcode of a non-branch instruction to a branch/jump and errors on address field of branch target address buffer structure. We divide unwanted jump errors into intra-BB (unwanted jumps from one basic block to itself) and inter-BB (unwanted jumps from one basic blocks to another) jumps. gZDC adopts different solutions to address each of these cases:

[1]Technically, in this case a new basic block will be inserted between original basic block and the fall-through basic block.

Intra-BB unwanted jump detection: To detect the manifestation of intra-BB unwanted jumps, we introduce a different static instruction scheduling than the one that usually is used in the state-of-the-art error detection scheme. Figure 5.2(a) illustrates widely-used scheduling policy for interleaving main and redundant instructions. Such instruction scheduling (i.e. interleaving main and redundant operations one by one) is extremely vulnerable to unexpected short jumps.

Basically, any unexpected jump from an *equal-point-of-execution* which skips (in forward or backward direction) exactly same number of main and redundant instructions will remain undetected. We define *equal-point-of-execution* as program execution points[2] which at that point the value of all main and redundant registers pairs are equal. For instance, all program execution points before the execution of main instructions (represented by M in Figure 5.2(a)) are *equal-point-of-execution*. [3] Dashed arrows in Figure represent forward unwanted jumps that basically cannot be detected by instruction-duplication based schemes because they cause no mismatch in the state of registers.

Note that in the Figure we just show forward undetected jumps, undetected backward jumps can be easily pictured by reversing the direction of arrows. Figure 5.2(b) shows an alternative scheduling policy (called coarse-grained scheduling of main and redundant instructions) which significantly reduces the chance of undetected jump errors. The main idea behind coarse-grained scheduling is that if an unexpected jump leads to a discrepancy between the state of main and redundant registers, most probably will be detected later on by further error checking operations.

Furthermore, the presence of gZDC data-flow and control-flow error checking op-

[2]Program execution points are points between two consecutive instructions in a program.

[3]Note that if redundant instructions are inserted into the code before instruction scheduler phase in compilation pipeline, the scheduling of main and redundant instructions may be slightly different. Nevertheless, our definition of *equal-point-of-execution* is independent on scheduling policy.

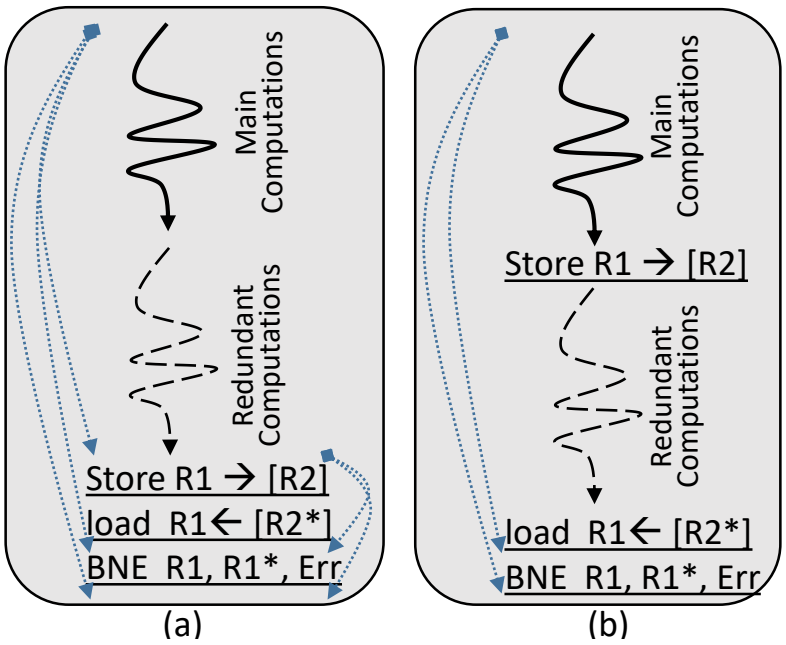

Figure 5.3: Coarse-grained scheduling in the presence of store and checking operations. As part (a) shows scheduling store and corresponding checking operations at the end of basic block introduces new possibilities for undetected unwanted jumps. Dashed arrows represent these undetected jumps. Part (b) shows gZDC coarse-grained main-redundant instruction scheduling policy in presence of store and checking instructions.

erations restricts the window of coarse-grained scheduling policy. For instance, Figure 5.3(a) shows a naive implementation of proposed scheduling strategy for a basic block with a store instruction. As it shows compared to Figure 5.2(b), the number of possible undetected unwanted jumps actually increase from '1' to '5'. Now, jumps from the beginning of the basic block to the right before store or right before error checking instruction (`BNE R1, R1*, Err`) is also undetectable. In addition, jumps from right before store to the end of basic block or right before error checking instruction are also remain undetected. gZDC solution to mitigate this problem is to consider store instruction as a main and checking load operation as well as error checking instruction (`BNE R1, R1*, Err`) as redundant instructions. Figure 5.3(b) shows the gZDC instruction scheduling policy. As the Figure shows, such scheduling can reduce the number of undetected jumps from '5' to '2'.

We did not count a short jump skipping over the last two instructions in Figure 5.3(b), because such unwanted jumps are benign and do not change the functionality of the program. Note that if there is a dependency between store and prior load operations inside a basic block (they access same memory location), the redundant load should be inserted before the conflicting store. Similar to store case, placing wrong direction control-flow checking operations at the end of basic blocks (Figure 5.4(a)) also introduces new vulnerable cases for unwanted-jump errors.

Figure 5.4b demonstrates gZDC wrong direction control-flow errors with coarse-grained scheduling policy. As shown, gZDC transformation first schedules main stream of instructions followed by the conditional branch instruction. Then it inserts corresponding redundant stream of instructions and direction checking operations in both taken and fall through paths.

Inter-BB unwanted jump detection: One of the main advantages of coarse-grained main/redundant instruction scheduling is that not only such scheduling reduce the chance of intra-BB unwanted jumps, but also is effective in inter-BB unwanted jump error detection. The reason lies in the drastic reduction in the number of *equal-point-of-execution* (i.e. the program points that *live* main and redundant registers have the same values). Generally, all unwanted (intra and inter basic block) jumps from program point S to E remain undetected if both S and E are *equal-point-of-executions* in instruction-replication schemes. That is because such unwanted jumps cause no discrepancy in the state of main and redundant registers and therefore there is no chance to be detected by error checking instructions. Coarse-grained scheduling policy reduces the chance of undetected unwanted errors by simply reducing the number of *equal-point-of-execution* in a protected by an instruction-duplication scheme.

To further reduce the possibility of undetected jumps, similar to existing signature-based control-flow checking schemes (Goloubeva *et al.*, 2003; Abadi *et al.*, 2005) gZDC

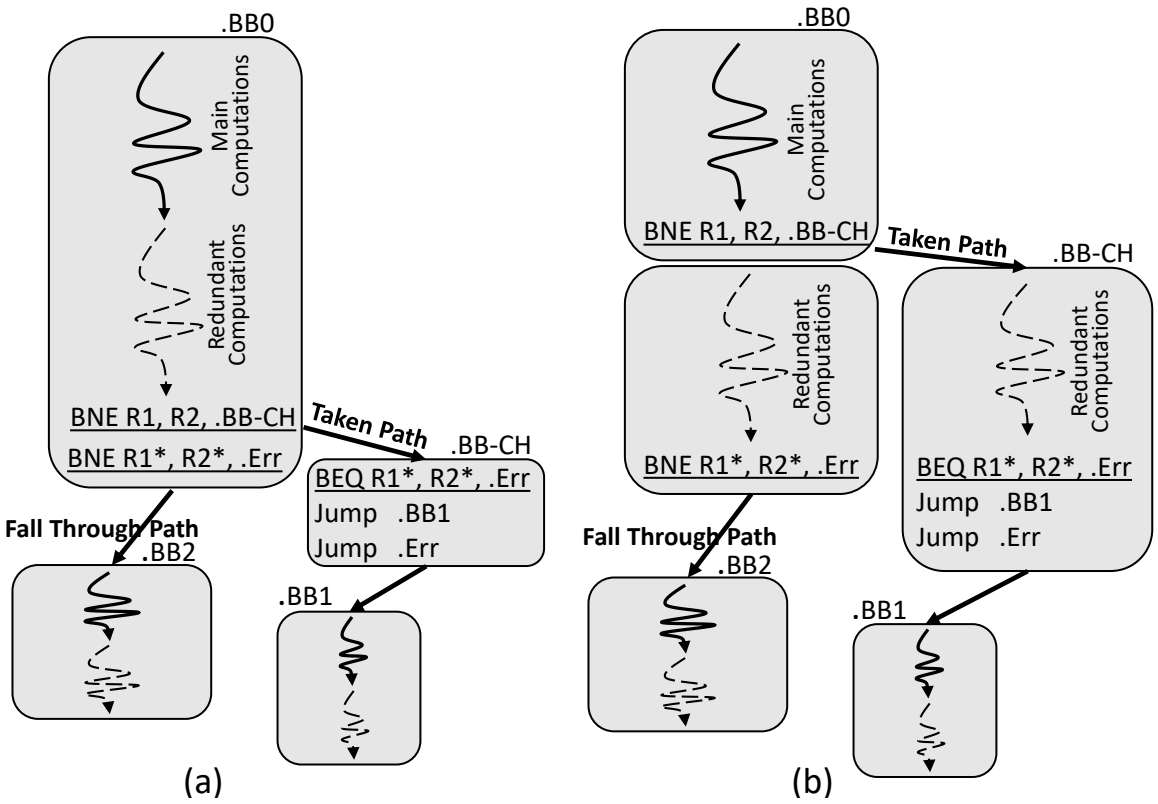

Figure 5.4: Coarse-grained scheduling in the presence of conditional branch operations. (a) Naive scheduling by inserting conditional branch at the end of basic block. (b) gZDC wrong direction control flow and coarse-gained instruction scheduling.

encodes static control-flow footprints (signatures) into the program and dynamically checks their results at the critical point of execution i.e. before and after system calls. Particularly, gZDC transformation designates two specific general purpose registers, called MICR (Main Instruction Check Register) and RICR (Redundant Instruction Check Register) to control-flow signature updating and checking. These two registers get updated during the execution of program and their values are checked against each other for error detection purposes only before and after system calls. Defining these two redundant registers forces another condition to program *equal-point-of-executions*. Now, for a point of execution to satisfy the condition of *equal-point-of-executions* apart of equal value for all live main and redundant registers, MICR and RICR registers should also have same exact values. The key point in gZDC unwanted-jump detection

strategy is **asymmetrical updates** for MICR and RICR registers which tries to keep the values of MICR and RICR registers different as far as it is possible and therefore removes the *equal-point-of-executions*.

Listing 1 shows gZDC asymmetric control flow registers updating algorithm. [4] Algorithm 1 consists of three phases: 1) updating MICR, 2) updating RICR register, and 3) eliminating symmetric updates. First, the compiler assigns each main-instruction-included-BB (BB with at least one main instruction) a unique number, called basic block signature. Note that main instructions are arithmetic, logical or memory operations that use only main registers as their operands. Then it inserts instructions to update the value of MICR register by xoring the value of MICR register with basic block signature right after the last main instruction in the basic blocks (line 4-6). By the end of this step, each main-instruction-included BB should include one MICR-updating instruction. In the second phase (line 9-16), RICR-updating operations will be inserted only into BBs which are predecessors of all control-flow merge or join BBs. [5] This lazy updates on RICR register lead to asymmetric (updates with different immediate value) on MICR and RICR registers which minimizes the number of *equal-point-of-executions*. To insert a RICR-updating instruction into a basic block, we should first calculate the correct immediate value of xor operation which is an aggregated xor between all basic block signatures in a backward path from current BB to the first join BB (line 11).

Figure 5.5 shows gZDC asymmetric control-flow signature updating algorithm for a simple case. As Figure 5.5(a) illustrates both `BB0` and `BB1` are join BBs. Applying gZDC wrong direction control flow error detection transformation (Figure 5.5(b)) in-

[4]The redundant signature registers only carry the same value before system calls or end of the function and error detection operations also will be inserted both before/after system calls and the end of functions.

[5]Basic blocks with more than one predecessors are join basic blocks.

Algorithm 1 gZDC asymmetric CF signature updating

Input: gZDC coarse-grained scheduled program

Output: gZDC coarse-grained scheduled program with unwanted jump error detection

1: *Initialization* : $MICR = RICR = 0$;
2: Assign unique number Sig_i to each main-instruction-included BB_i in Input
3: */*First Phase: Updating MICR*/*
4: */*Top-down control flow traversal*/*
5: **for** each BB_i in Input **do**
6: **if** (BB_i is a main-instruction-included BB) **then**
7: $lmInst$ = last main instruction in BB_i
8: Create Instruction $Inst$: $MICR = MICR \oplus Sig_i$
9: Insert $Inst$ after $lmInst$
10: **end if**
11: **end for**
12: */*Second Phase: Updating RICR*/*
13: */*Top-down control flow traversal*/*
14: **for** each BB_i in Input **do**
15: **if** (BB_i is a predecessor of a join BB) **then**
16: $aggrSig$ = aggregated xor of BB_i signature with its all consecutive predecessors to the first join BB
17: $frInst$ = first redundant instruction in BB_i
18: Create Instruction $Inst$: $RICR = RICR \oplus aggrSig$
19: insert operation $Inst$ before $frInst$
20: **end if**
21: **end for**

22: */*Third Phase: eliminating symmetric updates*/*

23: */* bottom-up control flow traversal*/*

24: **for** each BB_i in Input **do**

25: **if** (BB_i includes $MICR$ and $RICR$ instructions) **then**

26: $immMICR$ = immediate value of $MICR$ updating instruction in BB_i

27: $immRICR$ = immediate value of $RICR$ updating instruction in BB_i

28: **if** $immMICR == immRICR$ **then**

29: $randomNum$ = a random number

30: $immRICR = immRICR \oplus randomNum$

31: **for** each $preBB_i$ of BB_i **do**

32: **if** $preBB_i$ includes a $RICR$ updating instruction **then**

33: $immRICR$ = immediate value of $RICR$ updating instruction in $preBB_i$

34: $immRICR = immRICR \oplus randomNum$

35: **else**

36: $frInst$ = first redundant instruction in $preBB_i$

37: Create Instruction $Inst$: $RICR = RICR \oplus randomNum$

38: insert $Inst$ before $frInst$

39: **end if**

40: **end for**

41: **end if**

42: **end if**

43: **end for**

creases the number of program BBs by three (BB0_1, BB0_2 and BB0_CH are added by gZDC transformation). This transformed control flow graph includes two main-instruction-included-BB (BB0 and BB0_1) and two BBs with join successor ((BB0_2 and BB0_CH)). According to phase one of the Algorithm 1, MICR-updating instructions are inserted into BB0 and BB0_1 right after the last main instruction. These instructions are shown as MICR $\oplus$ sig_BB0 and MICR $\oplus$ sig_BB0_1 in the Figure. Based on the second phase of the algorithm, RICR-updating operations are inserted into BB0_2 and BB0_CH. To compute the offset of xor operation for BB0_2, we should traverse control flow graph backward to the last join BB (Dashed arrows in the Figure shows this path). Such path consists of BB0_2, BB0_1 and BB0. However, since BB0_2 is not a main-instruction-including-BB it does not have a signature and the aggregated signature is calculated by xoring the signature of BB0_1 (Sig_BB0_1) and signature of BB0 (Sig_BB0). The last instruction in BB0_2 shows inserted RICR-updating instruction. Similar to the BB0_2, a RICR-updating instruction is also inserted in the BB0_CH.

The last phase of Algorithm 1 (line 17 to 36) deals with the problem of symmetric updates on both MICR and RICR registers. The problem in symmetric updates is that they introduce *equal-point-of-executions* which alleviates the chance of undetected jumps. For instance, unwanted jumps from the beginning to the end of a BB which consists of main instructions follow by symmetric updates on MICR and RICR registers and redundant instructions remain undetected because such jumps do not cause any mismatch in the state of pair main/redundant registers. To detect symmetric update cases, gZDC transformation goes over all program basic blocks starting from the end of program (exit basic blocks) and first checks if a basic block contains both MICR-updating and RICR-updating instructions (line 18). If yes, it extracts the immediate field of signature updating instructions and checks if those values are

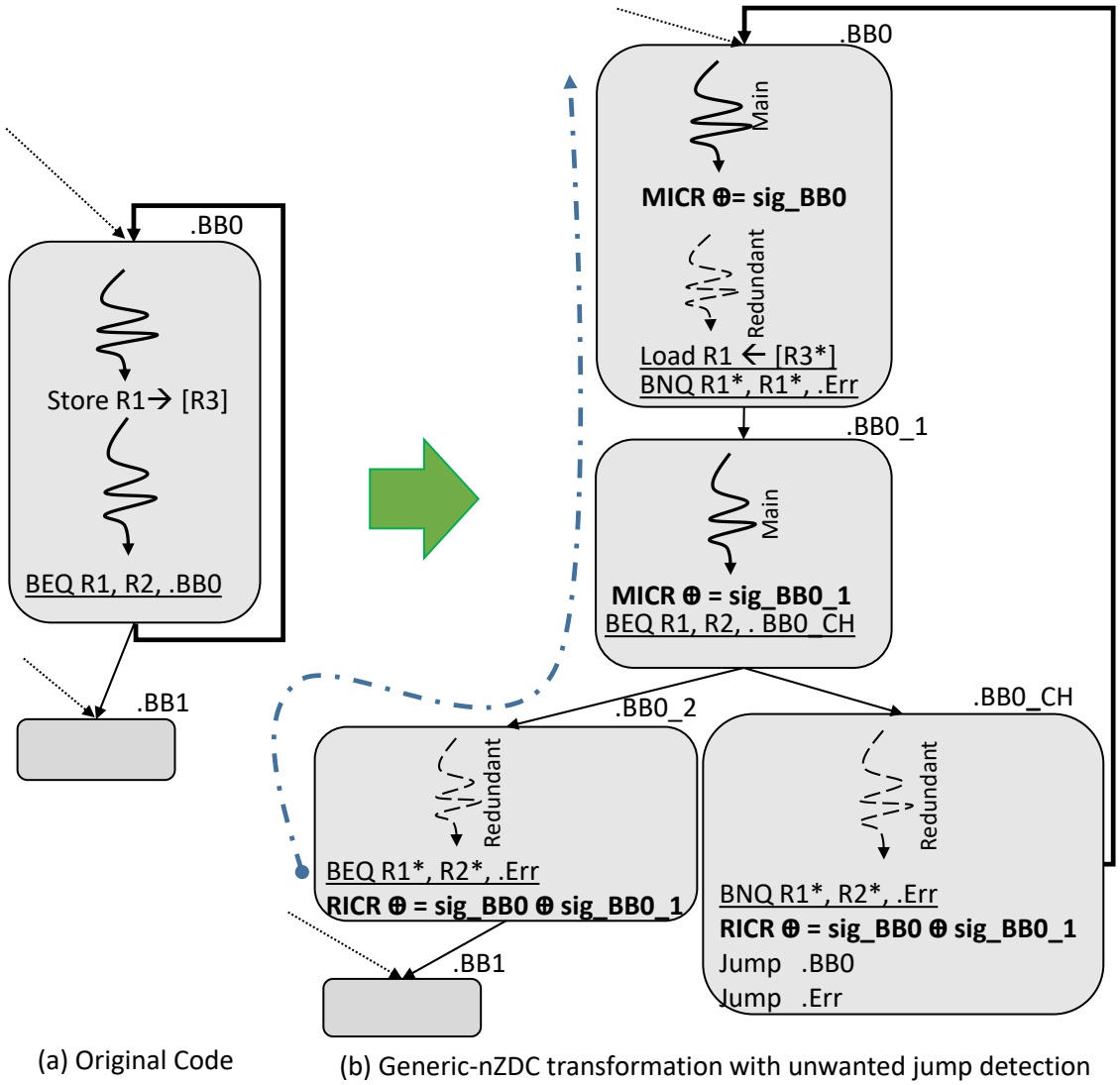

Figure 5.5: Complete gZDC data-flow and control-flow transformations for a simple loop. (a) shows control-flow for a simple loop, and (b) shows corresponding gZDC code with static signature updating operations. MICR-updating instructions are inserted into the main-instruction-included-BB and RICR-updating instruction are inserted into successor BBs of a join BBs. The dashed arrow shows backward trace path to compute the aggregated signature required for RICR-updating instruction in $BB0\_2$.

same (line 18-21). If a BB with symmetric updates is detected, then the algorithm generates random number and updates the immediate field of RICR-updating instruction (line 2-23). This makes signature updating instructions asymmetric. However, to maintain the consistency, signature modification for predecessor BBs of current BB is also required (lines 24 to 33). Basically, same exact random number should be xored with the RICR-updating instruction in the predecessor blocks as well. If the predecessor block does not have such instruction, the algorithm creates one and inserts it into the basic block before its first redundant instruction (line 29 and 31).

5.2 Experimental Methodology

In this section, we describe our experimental environment and evaluation strategy to measure the re-evaluate the error coverage and performance overhead of SWIFT and gZDC transformations. Note that we did not implement nZDC (explained in Chapeter 2) because as we mentioned before nZDC is ISA-dependent (nZDC control-flow is compatible with ARM V8-a ISA) and cannot be implemented on OpenRISC ISA.

5.2.1 Microprocessor and Fault injection Environment

We used single-issue 5-stage pipeline Mor1kx cappuccino microprocessor (the latest version of OpenRISC1000 processor family) which is capable of running Linux operating system (Mor, 2013). The microprocessor configuration is shown in table 5.1. We simulated the synthesizable HDL Verilog codes of Mor1kx microprocessor by Icarus Verilog simulator (Williams, 2006).

Figure 5.6 depicts high-level block diagram of mor1kx microprocessor. The microprocessor core components includes instruction and memory caches, a simple branch predictor, register file, processor data path and load-store unit. Note that for sim-

Table 5.1: Mor1kx microprocessor configuration

Parameter	Value
ISA	OpenRISC
CPU model	In-order Single-issue Mor1kx
Pipeline	5 stage (cappuccino)
# of General Purpose Registers	32
Branch prediction	Backward taken, forward not taken
L1 D/I cache	8KB(2 way)/8KB(2 way)
Cache line size	32 bytes
RAM size	32 MB

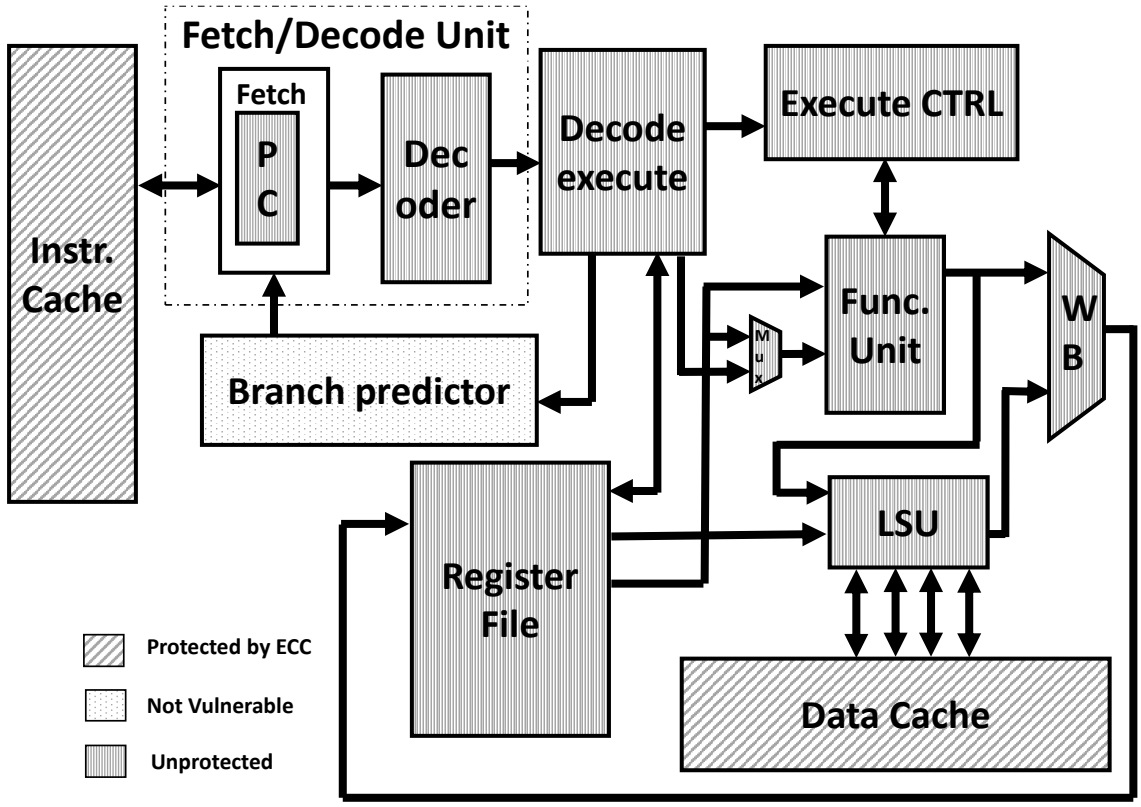

Figure 5.6: Mor1kx architecture. Caches and Branch predictor are excluded from fault injection analysis.

Table 5.2: Fault Injection Features

Component	Fault Model	# of fault sites
Register File	Single Event Upset[a]	1024
Fetch/Decode Unit	Single Event Upset	200
Decode/Execute Unit	Single Event Upset	216
Execute/Control Unit	Single Event Upset	183
Write/Back Unit	Single Event Upset	32
ALU	Single-event transient[b]	36
LSU	Single-event transient	101

[a] Errors are injected on flip-flops.

[b] Errors are injected on combinational circuits.

plicity, pipeline forwarding paths from ALU to Decode/Execute and from WriteBack to Decode/Execute stage are not shown in the Figure. Inline with previous work (Reis *et al.*, 2005, 2007; Feng *et al.*, 2010; Laguna *et al.*, 2016; Chen *et al.*, 2016), we assume that instruction/Data caches are ECC-protected and we exclude them from our fault injection sites (these components are highlighted as protected by ECC in the Figure). We also exclude branch predictor because as pinpointed by mukherjee2003systematic, transient errors on such performance-enhancing structures does not cause lead to a failure. We conducted fault injection experiments on the rest of microprocessor hardware components including Fetch/Decode, Decode/Execute, Execute Control, Register File, ALU, LSU, WriteBack. Table 5.2 summarizes the number of fault sites and the injected fault model in fault injection target components.

5.2.2 Compilation and Binary Generation

We utilize clang and LLVM-or1k (LLV, 2017) compiler infrastructures and compile 8 programs from Mibench (Guthaus *et al.*, 2001) test suite and 10x10 matrix multiplication for mor1kx microprocessor with -O3 optimization level. Note that we compile the whole application as appeared in Mibench test suite and did not reduce programs to micro-benchmarks. We choose to implement error detection transformations as late back-end passes in LLVM-or1k to take advantage of all standard compiler optimizations like dead code elimination and common-subexpression elimination. Note that since SWIFT control-flow transformation is highly dependent on IA-64 X86 predicated instructions, we adopted an alternative implementation explained in (Reis *et al.*, 2007) which is written by same authors as SWIFT original paper. We only apply protection schemes to user-level functions and exclude standard library functions from all evaluations including performance overhead and error coverage estimation. However, to expand the domain of evaluation for the benchmarks which spend most of their execution time in the library calls (e.g., qsort program heavily uses library provided qsort function), we manually copy the source code of the dominant library function call to the program source code from open source GNU C (glibc) library and apply protection on them. To demonstrate the effectiveness of the proposed technique, we generate four executable versions for each benchmark:

Original (ORG) version is the unprotected version of program without applying any software-level protection scheme.

SWIFT version is protected by SWIFT data and control flow protection transformations. Note that we include SWIFT signature-based control-flow error detection in this version.

gZDC-WithoutJumpDet version is protected by nZDC data-flow transformation

(Chapter 2) and generic wrong direction control-flow error detection (explained in Subsection 5.1.1).

gZDC version is the proposed gZDC including coarse-grained main-redundant instruction scheduling policy and asymmetric execution control-flow footprints (Algorithm 1) for detecting the manifestation of unexpected jumps. We develop and evaluate gZDC and gZDC-WithoutJumpDet separately to show the effectiveness of the coarse-gain scheduling and asymmetric control-flow signature techniques on error detection and their implications on performance overhead.

5.2.3 Fault Injection Process and Output Classification

Since the fault models used in this work are random single event transient or upset faults, for each fault injection experiment fault injection site, `b`, and fault injection time, `t` are randomly selected respectively among all 1792 fault injection sites and from all cycles that a program executes user-level functions. After selecting tuple `(b,t)`, we start simulation normally and once the execution hits cycle `t`, we invert the logical value of `b` (by XORing it with '1') for one cycle and let the execution continues till execution terminates permanently or the execution time exceeds from allowable simulation time which is 2 times more than fault free run of program. We classify the results of each fault injection trial into one of the following categories:

Masked: Fault injection simulation terminates normally and generate exactly same output as fault free run.

SW Detected: Cases that protection scheme detects the manifestation of error and raise error detection flag.

OS/HW Detected: Fault injection simulation runs that terminate permanently by generating an exception (i.e. segmentation faults or unkown instruction exception) or cause a time-out error.

Silent Data Corruption (SDCs): Cases that execution terminates normally, but the results is different from the fault-free execution.

For each version of a program we conducted 10,600 random fault injection experiments on hardware components listed in Table 5.2, which statistically provides around 2.5% error margin for 99% confidence level (Leveugle *et al.*, 2009). The hardware (space) distribution of errors is corresponding to the area of hardware components. For instance, since register file includes around 57% of targeted fault sites (1024 of 1792), around 57% (around 6,000) of all injected errors also were injected in register file. Since we have 9 benchmarks, we injected 95400 (10,600 * 9) faults for each version of programs. Overall, for all versions of programs we injected 381,600 (4 * 95400) fault injection experiments.

Since all error detection schemes including SWIFT and gZDC transformations presume some type of backward recovery (i.e. restarting or checkpoint/rollback) as post error detection handling strategy, errors in `Exceptions and Hangs` category can be considered as harmless. The main difference between `Exceptions and Hangs` and `Detected` errors is that the former is identified by operating system or hardware protection schemes and later is recognized by software-level error protection schemes. Nevertheless, in both cases, the recovery routine should be invoked to remove the manifestation of error from the system. Therefore, we consider SDC-induced faults as failed cases because such errors remain unnoticed.

5.2.4 Number of scaled SDCs as Comparison Metric

To fairly quantify and compare the error detection capability of SWIFT, gZDC-WithoutJumpDet and gZDC techniques, we use number of scaled SDCs (or scaled-SDC for short) which is calculated based on an overhead-dependent correction factor which originally proposed by (Schirmeier *et al.*, 2015) and has been used in many

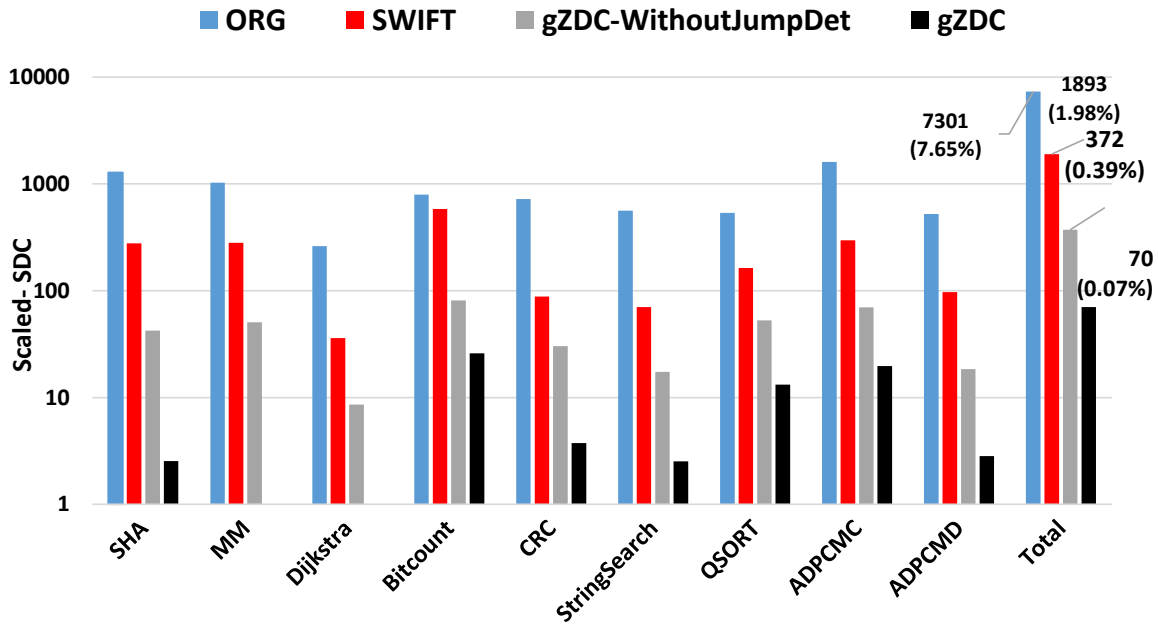

Figure 5.7: Compared to original code, gZDC transformation reduces the number of scaled SDCs by more than 100x.

recent techniques (So *et al.*, 2018; Didehban and Shrivastava, 2018). Scaled-SDC captures the negative impact of runtime overhead (α) of protection schemes on execution reliability and is estimated as below:

$$scaled - SDC = \#ofSDCs \times \alpha \tag{5.1}$$

Note that since there is no runtime overhead for original version of programs the number of scaled SDCs is in fact equal to number of SDCs for original versions of programs. As pinpointed by (Schirmeier *et al.*, 2015) considering number of SDCs (or percentage of SDCs) significantly overestimates the error coverage capability of the protection schemes and will lead to wrong conclusion.

5.2.5 Error Coverage Results and Analysis

Processor-wide error coverage: Figure 5.7 shows number of scaled SDCs (calculated based on equation 5.1) for different benchmarks extracted from our microprocessor-wide fault injection experiments. In the Figure, X-axis represents different benchmarks and Y-axis represents number of scaled-SDCs in logarithmic scale. The rightmost set of bars (denoted as total) represents the aggregated number of scaled-SDCs for each version of programs across all benchmarks. This aggregated sum can be seen as a large application which consists of all the benchmarks. As shown in the Figure from 95,400 fault injection experiments performed on different microprocessor hardware components during the execution of original versions of programs around 7.65% of them result to SDCs. SWIFT transformation can reduce the number of scaled-SDC to 1.98%. gZDC-WithoutJumpDet scheme reduces the number of scaled-SDC down to 0.39%. Finally, gZDC improves the scaled-SDC rate of gZDC-WithoutJumpDet by around 5x (0.07% scaled-SDCs rate). To understand the reasons behind this trend, we analysis the vulnerability of each hardware component before and after applying error detection transformations.

Component-wise error coverage: Figure 5.8 shows the aggregated number of scaled-SDCs per component for different versions of programs across all benchmarks. We start by looking at register file since around 60% (4411 of 7301) of all SDCs in original programs (Figure 5.8(a)) caused by fault injection experiments conducted on register file. We can see that SWIFT transformation is very effective in protecting register file and it reduces the scaled-SDCs to only 174 cases – more than 25x reliability improvement for register file. Further examination (recreating the SDC cases and tracing the trajectory of inserted faults to the final output) shows that almost in all of SDCs cases of SWIFT-protected programs injected fault alters the value of a

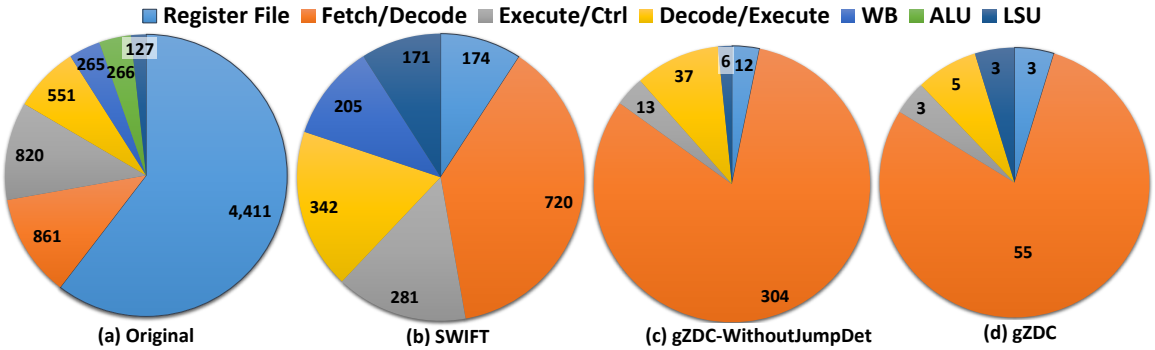

Figure 5.8: Component-wise scaled SDC analysis. While instruction duplication error checking schemes can effectively improve the register file vulnerability but errors affecting fetch/decode stage of pipeline remain the main source of SDCs after applying such schemes.

register operand of a memory or control flow instructions immediately before actual use (read) of register by the critical instruction. Note that in these cases the fault injection targeted register operand is either dead[6] or impact of error gets masked before the next error detection point.

As Figure 5.8(b) shows register file is no more the biggest vulnerable hardware components after protecting programs by SWIFT transformation. Both nZDC-based transformations (Figure 5.8(c) and 5.8(d)) considerably reduce the register file scaled-SDCs rate. That is because they detect errors by checking registers after the execution of critical instruction rather than prior to their execution. However, there is still some SDCs cases for nZDC-based transformations and our tracking reveals that in all SDCs scenarios error hit the address register of a silent stores. As we explained in Section 4.2.1, errors affecting the address part of silent stores are one of the single-point-of-failures in nZDC-based protection schemes.

As Figure 5.8 reveals bits in fetch/decode stage of the pipeline are the most challenging parts of hardware to protect against errors by software-level redundancy schemes as faults injected on them cause the majority of failed cases in all pro-

[6]A register operand is dead if the next access to that register is a write operation.

tected versions of programs. While the number of scaled-SDC due to error injection on fetch/decode stage flip-flops for the original version of programs is around 861, SWIFT transformation can just slightly improve it to 720 cases (around 16%). The main reason for such ineffective protection is because that SWIFT transformation cannot protect program execution against errors which alter unduplicated instructions i.e. load, store and compare operations. gZDC-WithoutJumpDet can reduce the number of scaled-SDC to 304 (64% improvement), however, it is still vulnerable to unwanted jump control flow errors. Examples are errors affecting immediate address of branch/jump instructions or the ones converting a none branch instruction to branch. gZDC, on the other hand, improves the reliability of execution against fetch/decode errors by more than 15x (from 861 to 55). This improvement shows the effectiveness of the coarse-grain scheduling and asymmetric execution control flow footprints strategies.

Bits in Decode/Execute stage of pipeline show similar behavior to fetch/decode stage in terms of effectiveness of protection scheme. We can see, while around 551 SDCs raised due to fault injection on Decode/Execute bits in the original version of programs, SWIFT-transformation can only reduce the number of scaled-SDCs to 342 (only 40% improvement). gZDC-WithoutJumpDet is more effective than SWIFT and can improve decode/execute scaled-SDC rate by 90% (from 551 to 37). However, there is still some SDCs due to errors affecting branch immediate address (branches are resolved at this stage). We can see that strategies proposed in this section for detecting unexpected jumps are effective and none of the errors injected into decode/execute remains undetected in gZDC protected version of programs.

Write back stage of the pipeline is responsible to select the source of register file updates (functional unit in cases of arithmetic/logical instructions or memory in case of load operations). Many of faults affecting this unit (around 14%) lead to SDC in

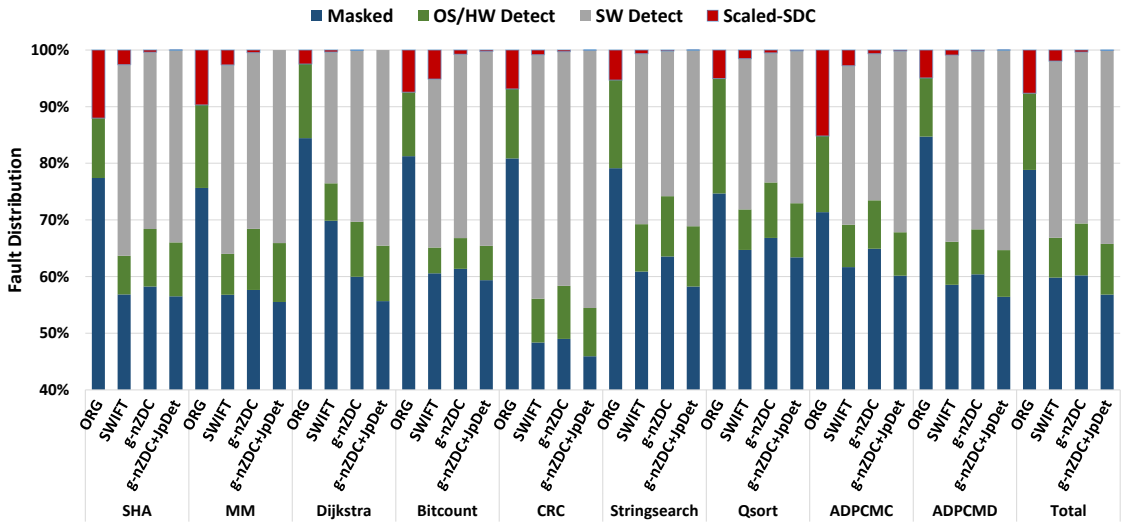

Figure 5.9: Error Distribution.

the original version of programs. Although sensitive to errors, yet since the area of writeback stage is small the chance of faults hitting its circuitry is low and therefore its contribution in processor-wide vulnerability is small. While nZDC-based transformations can completely remove the writeback stage vulnerability, SWIFT transformation only improves its reliability by 22% (from 265 SDCs to 205). It can be seen that all three versions of instruction duplications schemes can completely eliminate the vulnerability of ALU against single event transient faults injected on ALU results. However, it is interesting to see the vulnerability of LSU actually **increases** by a factor of 1.34x after applying SWIFT transformation. That is because SWIFT transformation increases the number of memory instructions due to higher register pressure and leave them unprotected. However, both gZDC-WithoutJumpDet and gZDC transformations significantly improve the sensitivity of LSU (from 127 to 6 and 3 , respectively) because such schemes completely protect the execution of memory operations unless address part of silent store operations.

Impact of protection schemes on fault distribution: Figure 5.9 shows the im-

pact of instruction duplication schemes on error propagation and distribution. In the Figure, X-axis shows different versions of benchmarks and the Y-axis represents the percentage of masked, OS/HW Detected, SW Detected and scaled-SDCs. Across all benchmarks, both SWIFT and nZDC-based transformations (gZDC-WithoutJumpDet and gZDC) considerably reduce the number of masked errors (the lowest segment of each bar in Figure 5.9) by on an average around 20%. Similarly, the number of OS/HW detected faults also reduce on an average by half (from 14% to 7% for SWIFT and to around 9% for gZDC-WithoutJumpDet and gZDC). Since the main goal of instruction duplication error protection schemes is to prevent a program from producing wrong results, an ideal scheme should only detect faults which are going to alter the final program output. However, frequent error checks of SWIFT and nZDC-based schemes introduce some false alarms by detecting benign faults (faults which are going to be masked) and overprotection by detecting errors which are covered by OS/HW error protection schemes. To estimate false alarm and over protection of applied error protection schemes, we can analyze the difference between SW detected errors of protected versions of code and the percentage of SDCs for the original version of programs. While only on an average 8% of injected faults lead to SDCs in original versions of programs, error checks in SWIFT, gZDC-WithoutJumpDet and gZDC raise the error detection flag on average around 31.1%, 30.2% and 34.17% of times, respectively.

5.2.6 Performance Overhead

Figure 5.10 depicts the execution time overhead of SWIFT, gZDC-WithoutJumpDet and gZDC transformations which is on an average around 2.71x, 2.77x and 2.91x, respectively. This amount of performance degradation is expected because around 2x overhead comes from duplication and executing frequent error detection/checking in-

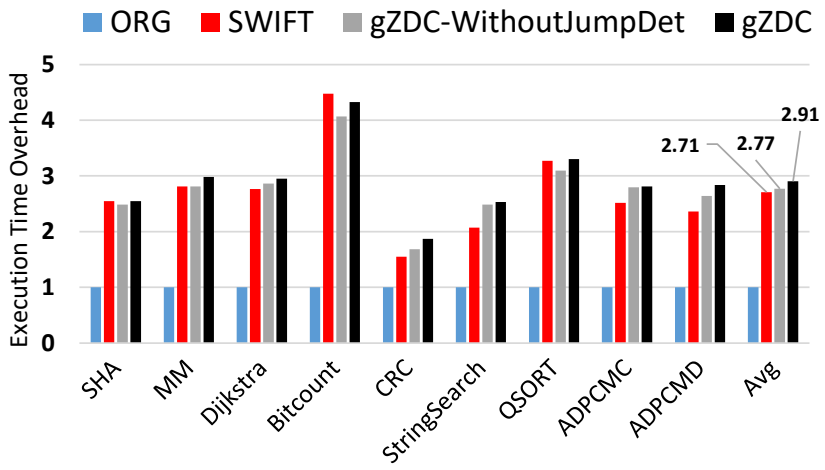

Figure 5.10: Execution time overhead

structions also impose some performance degradation. Almost across all programs (except bitcount and Qsort) gZDC-WithoutJumpDet transformation overhead is slightly more than SWIFT, however, it achieves more than 5x protection. As the rightmost bar for each benchmark shows, the execution overhead of applying coarse-grained master-shadow instruction scheduling and applying asymmetric control-flow footprints is only 4%, but they boost the error coverage by more than 5.2x.

5.3 Conclusions

In this chapter we implement, tune and evaluate the effectiveness of a generalized version of our base error detection solution, named gZDC, on a Verilog description of a simple in-order microprocessor. gZDC transformation consists of three main parts: i) nZDC post-store data-flow error detection (explained in Chapter 2), ii) NEMESIS wrong-direction control-flow error detection (explained in Section 4.2.4) and iii) coarse-grained main-redundant instruction scheduling and asymmetric control-flow signatures for unexpected jump error detection. Verilog-level fault injection experiments show that gZDC can achieve more than 99.9% error coverage.

Chapter 6

SUMMARY AND FUTURE WORK

Unprecedented proliferation of microprocessor-based systems in our lives, on one hand, and microprocessors reliability decreasing trend, on the other hand, have been made resilience of computer-based systems against hardware transient faults such a crucial concern. Software-level resilience solutions are promising because against their hardware-based counterparts they can accomplish flexible and salable protection.

While software-level solutions can be applied in different granularity, in this dissertation we focused on low-level instruction replication error detection and correction schemes. Our detailed analysis of the best known schemes revealed that many of the existing schemes suffer from multitude vulnerability windows. For instance, almost all related schemes replicate program arithmetic and logical instructions and check for errors before the execution of critical instructions i.e. memory write and control flow operations. This pre-store (and pre-branch) error protection strategy leaves the execution of memory write operations (and control flow instructions) unprotected.

To overcome to the limitations of existing instruction-level redundancy-based error protection scheme we propose post-store (and post-branch) error protection. Chapter 2 of this dissertation introduces nZDC as an effective error detection scheme. Against prior solutions, nZDC detects the manifestation of errors on the results of critical instructions rather than their register operands. Furthermore, we built upon nZDC and proposed two error detection and correction solutions. The first recovery solution is called InCheck (Chapter 3) and works based on basic-block level in-application checkpointing and rollback strategy. The second recovery solution (Chapter 4) is a forward recovery technique named NEMESIS which is based on error detection and

on-demand correction rather than simple majority voting based error masking.

We evaluated the effectiveness of our proposed solutions by performing millions microprocessor-wide random single bit-flip error injection at μ-architectural and Verilog level of abstractions (Chapter 5). Our results show that our proposed solutions can accomplish several times higher error coverage than state-of-the-art schemes.

In this work, we have verified the effectiveness of our proposed solutions at μ-architectural and Verilog level of abstractions. The last step in testing process is radiation testing. One future work can be crash testing software solutions against radiation bombardment.

Techniques presented in this dissertation are mainly concerned with single bit-flip transient errors in microprocessor components excluding memory subsystem. However, in reality multiple transient and permanent errors can also occur and threat the correctness of computations. More general (fault model independent) schemes should be developed for systems which are susceptible to wide variety of hardware malfunctions.

The main goal of this dissertation was to introduce highly effective error protection scheme and performance degradation was secondary concern. Adaptive and selective usage of the proposed schemes can significantly improve the application-level execution time. One of the challenge is to find the most critical computations and only apply protection on them. Advanced compiler optimization can be used for automatic critical region of code detection and protection. Similarly, the performance overhead of proposed scheme can be improve significantly by shifting some part of error resilience implementation to hardware. For instance, nZDC data flow protection changes a program in a way that each store instruction has always loading-back load instruction from the same memory location. Load-Store unit component of microprocessor can be modified slightly to check for this property.

"mor1kx - an OpenRISC Processor IP Core.", `https://github.com/openrisc/mor1kx`, [accessed 03-2018] (2013).

"llvm-or1k - Low Level Virtual Machine for Or1k.", `https://github.com/openrisc/llvm-or1k`, [accessed September-2018] (2017).

Abadi, M., M. Budiu, U. Erlingsson and J. Ligatti, "Control-flow integrity", in "Proceedings of the 12th ACM conference on Computer and communications security", pp. 340–353 (ACM, 2005).

ARM®, "Arm cortex-a53 mpcore processor technical reference manual", ARM Limited, Jun (2014).

ARM®, "ARM®: Cortex A-53 Processor", `http://www.arm.com/products/processors/cortex-a/cortex-a53-processor.php` (2015).

Asadi, H. and M. B. Tahoori, "Soft error derating computation in sequential circuits", in "Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design", pp. 497–501 (ACM, 2006).

Aupy, G., A. Benoit, T. Hérault, Y. Robert, F. Vivien and D. Zaidouni, "On the combination of silent error detection and checkpointing", in "Dependable Computing (PRDC), 19th Pacific Rim International Symposium on", pp. 11–20 (IEEE, 2013).

Avižienis, A., J.-C. Laprie and B. Randell, "Dependability and its threats: a taxonomy", Building the Information Society pp. 91–120 (2004).

Barhorst, J., T. Belote, P. Binns, J. Hoffman, J. Paunicka, P. Sarathy, J. Scoredos, P. Stanfill, D. Stuart and R. Urzi, "A research agenda for mixed criticality systems, 2009", White paper (2010).

Baumann, R., "Soft errors in advanced computer systems", Design & Test of Computers, IEEE **22**, 3 (2005a).

Baumann, R. C., "Radiation-induced soft errors in advanced semiconductor technologies", IEEE Transactions on Device and materials reliability **5**, 3, 305–316 (2005b).

Bell, G. B., K. M. Lepak and M. H. Lipasti, "Characterization of silent stores", in "Parallel Architectures and Compilation Techniques. Proceedings. International Conference on", pp. 133–144 (IEEE, 2000).

Bernick, D., B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka and J. Smullen, "Nonstop® advanced architecture", in "2005 International Conference on Dependable Systems and Networks (DSN'05)", pp. 12–21 (IEEE, 2005).

Binkert, N., B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator", ACM SIGARCH Computer Architecture News **39**, 2, 1–7 (2011).

Blem, E., J. Menon and K. Sankaralingam, "Power struggles: Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures", in "High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on", pp. 1–12 (IEEE, 2013).

Borucki, L., G. Schindlbeck and C. Slayman, "Comparison of accelerated dram soft error rates measured at component and system level", in "Reliability Physics Symposium, 2008. IRPS 2008. IEEE International", pp. 482–487 (IEEE, 2008).

Burns, A. and R. Davis, "Mixed criticality systems-a review", Department of Computer Science, University of York, Tech. Rep (Ninth Edition) (2017).

Cannon, E. H., M. S. Gordon, D. F. Heidel, A. KleinOsowski, P. Oldiges, K. P. Rodbell and H. H. Tang, "Multi-bit upsets in 65nm soi srams", in "Reliability Physics Symposium, 2008. IRPS 2008. IEEE International", pp. 195–201 (IEEE, 2008).

Chen, Z., A. Nicolau and A. V. Veidenbaum, "Simd-based soft error detection", in "Proceedings of the ACM International Conference on Computing Frontiers", pp. 45–54 (ACM, 2016).

Clark, L., D. Patterson, C. Ramamurthy and K. Holbert, "An embedded microprocessor radiation hardened by microarchitecture and circuits", (2014).

Clark, L. T., D. W. Patterson, C. Ramamurthy and K. E. Holbert, "An embedded microprocessor radiation hardened by microarchitecture and circuits", IEEE Transactions on Computers **65**, 2, 382–395 (2016).

Cohen, N., T. Sriram, N. Leland, D. Moyer, S. Butler and R. Flatley, "Soft error considerations for deep-submicron cmos circuit applications", in "Electron Devices Meeting, 1999. IEDM'99. Technical Digest. International", pp. 315–318 (IEEE, 1999).

Didehban, M., S. R. D. Lokam and A. Shrivastava, "Incheck: An in-application recovery scheme for soft errors", in "Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE", pp. 1–6 (IEEE, 2017a).

Didehban, M. and A. Shrivastava, "nZDC: a compiler technique for near Zero Silent data Corruption", in "Proceedings of the 53rd Annual Design Automation Conference", p. 48 (ACM, 2016).

Didehban, M. and A. Shrivastava, "A compiler technique for processor-wide protection from soft errors in multithreaded environments", IEEE Transactions on Reliability **67**, 1, 249–263 (2018).

Didehban, M., A. Shrivastava and S. R. D. Lokam, "Nemesis: A software approach for computing in presence of soft errors", in "Computer-Aided Design (ICCAD), 2017 IEEE/ACM International Conference on", pp. 297–304 (IEEE, 2017b).

Duell, J., "The design and implementation of berkeley lab's linux checkpoint/restart", Lawrence Berkeley National Laboratory (2005).

Elnozahy, E. N. and J. S. Plank, "Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery", IEEE Transactions on Dependable and Secure Computing **1**, 2 (2004).

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

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

Goloubeva, O., M. Rebaudengo, M. S. Reorda and M. Violante, "Soft-error detection using control flow assertions", in "Defect and Fault Tolerance in VLSI Systems, 2003. Proceedings. 18th IEEE International Symposium on", pp. 581–588 (IEEE, 2003).

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

Haddad, N. F., R. D. Brown, R. Ferguson, A. T. Kelly, R. K. Lawrence, D. M. Pirkl and J. C. Rodgers, "Second generation (200mhz) rad750 microprocessor radiation evaluation", in "Radiation and Its Effects on Components and Systems (RADECS), 2011 12th European Conference on", pp. 877–880 (IEEE, 2011).

Henkel, J., L. Bauer, N. Dutt, P. Gupta, S. Nassif, M. Shafique, M. Tahoori and N. Wehn, "Reliable on-chip systems in the nano-era: Lessons learnt and future trends", in "Proceedings of the 50th Annual Design Automation Conference", p. 99 (ACM, 2013).

Hubert, G., L. Artola and D. Regis, "Impact of scaling on the soft error sensitivity of bulk, fdsoi and finfet technologies due to atmospheric radiation", Integration, the VLSI journal **50**, 39–47 (2015).

IRC, "International Technology Roadmap For Semiconductors 2.0-Executive Summary", `http://www.itrs2.net/itrs-reports.html`, [accessed 19-November-2016] (2015).

ISO26262, "ISO26262: Road vehicles-Functional safety", International Standard ISO/FDIS (2011).

Iturbe, X., B. Venu, E. Ozer and S. Das, "A triple core lock-step (tcls) arm® cortex®-r5 processor for safety-critical and ultra-reliable applications", in "Dependable Systems and Networks Workshop, 2016 46th Annual IEEE/IFIP International Conference on", pp. 246–249 (IEEE, 2016).

Kang, S.-h., H. Yang, S. Kim, I. Bacivarov, S. Ha and L. Thiele, "Static mapping of mixed-critical applications for fault-tolerant mpsocs", in "Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE", pp. 1–6 (IEEE, 2014).

Khudia, D. S., G. Wright and S. Mahlke, "Efficient soft error protection for commodity embedded microprocessors using profile information", ACM SIGPLAN Notices **47**, 5, 99–108 (2012).

Laguna, I., M. Schulz, D. F. Richards, J. Calhoun and L. Olson, "Ipas: Intelligent protection against silent output corruption in scientific applications", in "Proceedings of the 2016 International Symposium on Code Generation and Optimization", pp. 227–238 (ACM, 2016).

Lattner, C. and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation", in "Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization", p. 75 (IEEE Computer Society, 2004).

Lepak, K. M., G. B. Bell and A. H. Lipasti, "Silent stores and store value locality", Computers, IEEE Transactions on **50**, 11, 1174–1190 (2001).

Leveugle, R., A. Calvez, P. Maistri and P. Vanhauwaert, "Statistical fault injection: quantified error and confidence", in "2009 Design, Automation & Test in Europe Conference & Exhibition", pp. 502–506 (IEEE, 2009).

Li, G., S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer and S. W. Keckler, "Understanding error propagation in deep learning neural network (dnn) accelerators and applications", in "Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis", p. 8 (ACM, 2017).

Liu, Q., C. Jung, D. Lee and D. Tiwari, "CLOVER: Compiler directed lightweight soft error resilience", in "ACM SIGPLAN Notices", vol. 50, p. 2 (ACM, 2015).

Lu, G., Z. Zheng and A. A. Chien, "When is multi-version checkpointing needed?", in "Proceedings of the 3rd Workshop on Fault-tolerance for HPC at extreme scale", pp. 49–56 (ACM, 2013).

Lyons, W., "Enabling increased safety with fault robustness in microcontroller applications", ARM Corporation (2010).

Mahatme, N., N. Gaspard, T. Assis, S. Jagannathan, I. Chatterjee, T. Loveless, B. Bhuva, L. W. Massengill, S. Wen and R. Wong, "Impact of technology scaling on the combinational logic soft error rate", in "Reliability Physics Symposium, 2014 IEEE International", pp. 5F–2 (IEEE, 2014).

Martinez-Alvarez, A., S. Cuenca-Asensi, F. Restrepo-Calle, F. R. P. Pinto, H. Guzman-Miranda and M. A. Aguirre, "Compiler-directed soft error mitigation for embedded systems", IEEE Transactions on Dependable and Secure Computing **9**, 2, 159–172 (2012).

May, T. C. and M. H. Woods, "A new physical mechanism for soft errors in dynamic memories", in "Reliability Physics Symposium, 1978. 16th Annual", pp. 33–40 (IEEE, 1978).

Mitropoulou, K., V. Porpodas and M. Cintra, "Casted: Core-adaptive software transient error detection for tightly coupled cores", in "Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on", pp. 513–524 (IEEE, 2013a).

Mitropoulou, K., V. Porpodas and M. Cintra, "DRIFT: Decoupled CompileR-based Instruction-level Fault-Tolerance", in "International Workshop on Languages and Compilers for Parallel Computing", pp. 217–233 (2013b).

Mitropoulou, K., V. Porpodas and T. M. Jones, "Comet: communication-optimised multi-threaded error-detection technique", in "Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems", p. 7 (ACM, 2016).

Mukherjee, S. S., 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", pp. 99–110 (IEEE, 2002).

Mukherjee, S. S., C. Weaver, J. Emer, S. K. Reinhardt and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor", in "Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on", pp. 29–40 (IEEE, 2003).

Normand, E., J. L. Wert, H. Quinn, T. D. Fairbanks, S. Michalak, G. Grider, P. Iwanchuk, J. Morrison, S. Wender and S. Johnson, "First record of single-event upset on ground, cray-1 computer at los alamos in 1976", IEEE Transactions on Nuclear Science **57**, 6, 3114–3120 (2010).

Oh, N., S. Mitra and E. J. McCluskey, "ED4I: Error Detection by Diverse Data and Duplicated Instructions", IEEE Transactions on Computers **51**, 2, 180–199 (2002a).

Oh, N., P. P. Shirvani and E. J. McCluskey, "Control-flow checking by software signatures", IEEE transactions on Reliability **51**, 1, 111–122 (2002b).

Owre, S., J. Rushby, N. Shankar and F. Von Henke, "Formal verification for fault-tolerant architectures: Prolegomena to the design of pvs", IEEE Transactions on Software Engineering **21**, 2, 107–125 (1995).

Patterson, D. A. and J. L. Hennessy, *Computer organization and design: the hardware/software interface* (Newnes, 2013).

Reinhardt, S. K. and S. S. Mukherjee, *Transient fault detection via simultaneous multithreading*, vol. 28 (ACM, 2000).

Reis, G. A. and D. I. August, "Software fault detection using dynamic instrumentation", in "In Proceedings of the Fourth Annual Boston Area Architecture Workshop (BARC", (Citeseer, 2006).

Reis, G. A., J. Chang and D. I. August, "Automatic instruction-level software-only recovery", IEEE micro **27**, 1 (2007).

Reis, G. A., J. Chang, D. I. August, R. Cohn and S. S. Mukherjee, "Configurable transient fault detection via dynamic binary translation", in "IN: PROCEEDINGS OF THE 2ND WORKSHOP ON ARCHITECTURAL RELIABILITY", (Citeseer, 2006).

Reis, G. A., J. Chang, N. Vachharajani, R. Rangan and D. I. August, "SWIFT: Software Implemented Fault Tolerance", in "Proceedings of the international symposium on Code generation and optimization", pp. 243–254 (IEEE Computer Society, 2005).

Sanda, P. N., J. W. Kellington, P. Kudva, R. Kalla, R. B. McBeth, J. Ackaret, R. Lockwood, J. Schumann and C. R. Jones, "Soft-error resilience of the ibm power6 processor", IBM Journal of Research and Development **52**, 3, 275–284 (2008).

Schirmeier, H., C. Borchert and O. Spinczyk, "Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors", in "Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on", pp. 319–330 (IEEE, 2015).

Schroeder, B. and G. A. Gibson, "Understanding failures in petascale computers", in "Journal of Physics: Conference Series", vol. 78, p. 012022 (IOP Publishing, 2007).

Shafique, M., S. Garg, J. Henkel and D. Marculescu, "The eda challenges in the dark silicon era: Temperature, reliability, and variability perspectives", in "Proceedings of the 51st Annual Design Automation Conference", pp. 1–6 (ACM, 2014).

Shivakumar, P., M. Kistler, S. W. Keckler, D. Burger and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic", in "Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on", pp. 389–398 (IEEE, 2002).

Shrivastava, A., A. Rhisheekesan, R. Jeyapaul and C.-J. Wu, "Quantitative analysis of control flow checking mechanisms for soft errors", in "Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE", pp. 1–6 (IEEE, 2014).

Shye, A. *et al.*, "Plr: A software approach to transient fault tolerance for multicore architectures", Dependable and Secure Computing, IEEE Transactions on **6**, 2 (2009).

Sierawski, B. D., R. A. Reed, M. H. Mendenhall, R. A. Weller, R. D. Schrimpf, S.-J. Wen, R. Wong, N. Tam and R. C. Baumann, "Effects of scaling on muon-induced soft errors", in "Reliability Physics Symposium (IRPS), 2011 IEEE International", pp. 3C–3 (IEEE, 2011).

Silberberg, R., C. H. Tsao and J. R. Letaw, "Neutron generated single-event upsets in the atmosphere", IEEE Transactions on Nuclear Science **31**, 6, 1183–1185 (1984).

So, H., M. Didehban, Y. Ko, A. Shrivastava and K. Lee, "Expert: Effective and flexible error protection by redundant multithreading", in "Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018", pp. 533–538 (IEEE, 2018).

Spainhower, L. and T. A. Gregg, "Ibm s/390 parallel enterprise server g5 fault tolerance: A historical perspective", IBM Journal of Research and Development **43**, 5.6, 863–873 (1999).

Vemu, R. and J. Abraham, "CEDA: Control-Flow Error Detection using Assertions", IEEE Transactions on Computers **60**, 9, 1233–1245 (2011).

Vemu, R., S. Gurumurthy and J. A. Abraham, "ACCE: Automatic Correction of Control-flow Errors", in "2007 IEEE International Test Conference", pp. 1–10 (IEEE, 2007).

Wang, C., H.-s. Kim, Y. Wu and V. Ying, "Compiler-managed software-based redundant multi-threading for transient fault detection", in "Proceedings of the International Symposium on Code Generation and Optimization", (IEEE Computer Society, 2007).

Wang, N. J. and S. J. Patel, "Restore: Symptom-based soft error detection in microprocessors", IEEE Transactions on Dependable and Secure Computing **3**, 3, 188–201 (2006).

Wells, P. M., K. Chakraborty and G. S. Sohi, "Mixed-mode multicore reliability", ACM SIGARCH Computer Architecture News **37**, 1, 169–180 (2009).

Williams, S., "Icarus verilog", On-line: http://iverilog.icarus.com (2006).

Xeon, I., "E7 family: Reliability", Availability and Serviceability: Advanced data integrity and resiliency support for mission-critical deployment (2011).

Xiong, L. and Q. Tan, "A dynamic approach to tolerate soft errors", Cluster Computing **16**, 3, 359–366 (2013).

Xu, J., Q. Tan, L. Tan and H. Zhou, "An instruction-level fine-grained recovery approach for soft errors", in "Proceedings of the 28th Annual ACM Symposium on Applied Computing", pp. 1511–1516 (ACM, 2013).

Yu, J., M. J. Garzarán and M. Snir, "Techniques for efficient software checking", in "International Workshop on Languages and Compilers for Parallel Computing", pp. 16–31 (Springer, 2007).

Yu, J., M. J. Garzaran and M. Snir, "Efficient software checking for fault tolerance", in "Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on", pp. 1–5 (IEEE, 2008).

Yu, J., M. J. Garzaran and M. Snir, "Esoftcheck: Removal of non-vital checks for fault tolerance", in "Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization", pp. 35–46 (IEEE Computer Society, 2009).

Zhang, Y. and K. Chakrabarty, "Fault recovery based on checkpointing for hard real-time embedded systems", in "Defect and Fault Tolerance in VLSI Systems, 2003. Proceedings. 18th IEEE International Symposium on", pp. 320–327 (IEEE, 2003).

Zhang, Y., S. Ghosh, J. Huang, J. W. Lee, S. A. Mahlke and D. I. August, "Runtime asynchronous fault tolerance via speculation", in "Proceedings of the Tenth International Symposium on Code Generation and Optimization", pp. 145–154 (ACM, 2012a).

Zhang, Y., J. W. Lee, N. P. Johnson and D. I. August, "Daft: decoupled acyclic fault tolerance", International Journal of Parallel Programming **40**, 1, 118–140 (2012b).

Ziegler, J. F., H. W. Curtis, H. P. Muhlfeld, C. J. Montrose, B. Chin, M. Nicewicz, C. Russell, W. Y. Wang, L. B. Freeman, P. Hosier *et al.*, "Ibm experiments in soft fails in computer electronics (1978–1994)", IBM journal of research and development **40**, 1, 3–18 (1996).