InCheck - An Integrated Recovery Methodology for Fine-grained Soft-Error Detection Schemes

by

Sai Ram Dheeraj Lokam

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved November 2016 by the
Graduate Supervisory Committee:

Aviral Shrivastava, Chair
Lawrence Clark
Anuj Mubayi

ARIZONA STATE UNIVERSITY

December 2016

ABSTRACT

Soft errors are considered as a key reliability challenge for sub-nano scale transistors. An ideal solution for such a challenge should ultimately eliminate the effect of soft errors from the microprocessor. While forward recovery techniques achieve fast recovery from errors by simply voting out the wrong values, they incur the overhead of three copies execution. Backward recovery techniques only need two copies of execution, but suffer from check-pointing overhead.

In this work I explored the efficiency of integrating check-pointing into the application and the effectiveness of recovery that can be performed upon it. After evaluating the available fine-grained approaches to perform recovery, I am introducing **InCheck**, an in-application recovery scheme that can be integrated into instruction-duplication based techniques, thus providing a fast error recovery. The proposed technique makes light-weight checkpoints at the basic-block granularity, and uses them for recovery purposes.

To evaluate the effectiveness of the proposed technique, 10,000 fault injection experiments were performed on different hardware components of a modern ARM in-order simulated processor. InCheck was able to recover from all detected errors by replaying about 20 instructions, however, the state of the art recovery scheme failed more than 200 times.

Dedicated to my parents, ***Radha*** *and* ***Ramaiah***

for all their efforts & sacrifices that turned into words in my thesis

ACKNOWLEDGEMENT

" *You can't connect the dots looking forward; you can only connect them looking backwards. So you have to trust that the dots will somehow connect in future. You have to trust in something - your gut, destiny, life, karma, whatever. This approach has never let me down, and it has made all the difference in my life.*" – Steve Jobs

So, I want to connect all the dots that made this thesis possible. Firstly, I would like to thank *Dr. Aviral Shrivastava* for all the help and guidance. A special thanks to *Moslem Didehban* for mentoring me on some amazing ideas here at *Compiler Microarchitecture Lab*. I also am grateful to *Dr. Anuj Mubayi* for recognizing my abilities that I never noticed & for his never ending support and encouragement through the tough times. My regards to *Dr. Lawrence Clark* for showing keen interest in my research and for adding an immense value to my thesis by being a part of my committee.

I want to thank every CMLer by name for being an integral part in shaping my career as a researcher, but I am limited by space. If it was not for their timely care & help, this thesis would have been an impossibility.

Without a bunch of amazing friends this journey would have been very lonely. My hearty thanks to *Eddie, Ujvala, Srinivas, Priya, Ankita, Rajat, Kundana, Akshay, Aneesh, Sourabh, Shail, Aishwarya, Sneha, Eric* and many more for having faith in me and supporting me even on the days I lost faith in myself.

Living away from family is not easy. I am lucky that I found people who treated me like family. I am grateful to *Beatrice, Bill, Sakshi and Kabir* for all the warm memories that I am going to cherish.

Finally, *Anju Babai, Revathi Pinni, Ramani Pinni, Chinna Babai* I will never be able to thank you enough for everything you people have stood for in my life.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Chapter 1

INTRODUCTION

Due to massive technology scaling, we have reached a point where we are able to make high performing processors that are affordable and power-efficient. Computing systems started becoming so crucial that their failure might be jeopardizing human safety. In applications ranging from smart cars to space shuttles, system dependability plays a critical role in processor design decisions. Soft-Error resilience is one of the key challenges to be addressed in making such processors reliable.

The term Linear Energy Transfer (LET) is generally used to describe the action of the radiation upon a processor. It can be quantified as the energy transferred by an ionizing particle while it traverses a unit distance through the processor. Soft-Errors (SE) are said to happen due to LET from particles with enough energy to alter the data stored in processor memory elements like registers, flip-flop, latches. The term "soft" originates from the fact that these errors do not result in a permanent change to processor circuitry. Sources of SEs' include but are not just limited to Cosmic rays, mechanical vibrations in moving parts, heat dissipation, radio-active elemental traces etc. The rate at which they happen highly depends on altitude of location under consideration.

Soft-Errors are a rare phenomenon. High energy cosmic flux hits the upper layers of earth's atmosphere at a rate of 1000 particles/m^2-s. As they progress towards the earth's surface, atmospheric collisions cause a cascade and raise the flux to 1000000/m^2-s at altitudes of around 40,000 ft. However due to the higher density of lower atmosphere, most of this flux gets absorbed making it only 10 times higher than the initially incident flux. (~1/cm^2-s) Ziegler *et al.* (1996). Not every particle

strike from this flux will cause a soft-error induced system failure. Experiments done on DRAM cells by Li *et al.* (2010), report a 0.061 `FIT` per Mbit. (Failure In Time: Number of expected failures in 10^9 hours of operation).

Hardware soft error detection strategies, ranging from low-level circuit design techniques to high-level redundancy-based techniques, have been employed in many dedicated mission-critical systems. However, in a new class of emerging application, the so called mixed-critical systems, critical and noncritical programs share a common underlying microprocessor. In such systems, application-level FT techniques are more effective because of their flexibility. Several independent radiation-based testing and low-level soft error emulation studies have already shown the effectiveness of application-level FT approaches.

Most of software-level fault tolerance techniques Didehban and Shrivastava (2016); Feng *et al.* (2010); Khudia *et al.* (2012); Wang *et al.* (2007); Oh *et al.* (2002a); Zhang *et al.* (2012a,b); Mitropoulou *et al.* (2014); Reis *et al.* (2005); Oh *et al.* (2002b) focus on error detection while assuming some kind of backward recovery is available. Restarting a program from the beginning is the simplest method of backward recovery Rennels and Hwang (2001). However, it suffers from a very high error recovery latency and is not applicable in some cases, i.e, a long running-application which is executing in an environment with the soft error frequency more than the the applications execution time . The most common method of recovery is checkpointing/rollback strategy. In such techniques, the program execution is paused periodically and a snapshot of the program state, called checkpoint, will get preserved in a safe storage. If the error detection unit declares the manifestation of any error, the program execution gets terminated, and resumes (rollbacks) from the last checkpoint. However, the usage of checkpoint-based recovery techniques is limited to High Performance Computing applications. This is due to the significant (about 50% Elnozahy and

Plank (2004); Schroeder and Gibson (2007)) performance and storage overhead of frequent and multiple checkpoints which are required for successful recovery from SDCs Aupy *et al.* (2013). Note that on top of the checkpointing cost, the error detection overhead, which is in order of 2x for redundancy-based techniques has to be considered for calculating the overall cost of backward recovery. I believe that if the error handling phases, mainly detection and recovery, collaborate with each other, more effective and efficient fault tolerance is achievable. In this work, I present an in-application low-level recovery mechanism called InCheck (INtegrated CHECK-pointing and Recovery). It combines with a fine-grain instruction duplication-based detection strategy, to provide fast and effective error recovery. The key idea of Incheck is to preserve the state of error-free register file in the beginning of each basic block, and to back up every memory location before each update. If the error detector declares the manifestation of any error, the program is redirected to the recovery block. In recovery block, the state of memory and register file revert to the initial state that they had in the beginning of BB and the program execution resumes from there. In order to evaluate the effectiveness of Incheck, I performed about 10,000 system-wide microarchitectural fault injection experiments on Mibench programs compiled with -O3 optimization flag. The results prove that Incheck is extremely effective.

Chapter 2

MOTIVATION AND RELATED WORKS

Transient faults or soft errors are considered as one of the main reliability threats in future systems. The effects of soft error will get masked by various masking effects in most of the cases. However, in some situations soft errors will survive masking effects and lead to a failures such as `SDC` (Silent Data Corruption), segmentation fault, system hang. In contrast to other failure modes, SDCs are considered to be the most dangerous as there will be no user recognizable sign that the results are incorrect.

One way to provide complete fault tolerance is to adopt forward-recovery which is made possible by nMR (n-Module Redundancy) execution and voting strategy. There are no active error detection and recovery phases in forward-recovery strategy. They eliminate the effect of error by performing majority-voting between redundantly computed results. Software implementation of nMR can take place at multiple granularities ranging from coarse-grained program or module triplication Shye *et al.* (2009, 2007); Quinn *et al.* (2015); Döbel and Härtig (2014) to fine-grain, low-level, assembly instruction triplication Reis *et al.* (2007); Chang *et al.* (2006); Restrepo-Calle *et al.* (2013). In coarse-grain software implemented TMR, three independent versions of a program Quinn *et al.* (2015) or process Shye *et al.* (2009, 2007) with separate data and memory get executed.The majority-voting takes place at the end of program either between the redundantly computed-results Quinn *et al.* (2015) or between arguments at system calls' boundaries Shye *et al.* (2009, 2007); Döbel and Härtig (2014). Fine-grain forward-recovery techniques provide a more efficient way of dealing with soft errors by eliminating the need for memory triplication and placing checking/voting operations at strategic points of execution. For instance, SWIFT-R Reis *et al.* (2007); Chang

et al. (2006) an assembly-level forward-recovery technique, assumes ECC-protected memory and therefore excludes the memory subsystem from its sphere-of-protection. SWIFT-R transformation divides programmer-visible registers into three sets, and triplicates program's computational instructions. To make sure that the error-free values will transfer to/from the memory, SWIFT-R performs 2-of-3 majority-voting between three types of redundantly-computed values:

1. Memory(read/write) instruction's base address register
2. Memory write instruction's value register
3. Compare instruction's register operands

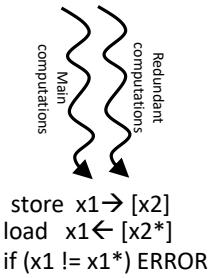

Figure 2.1: nZDC Data-Flow Protection Scheme

Software Implemented Fault Tolerance (SWIFT) Reis *et al.* (2005) is a double modular redundancy based technique that can provide soft-error detection. However it does not duplicate memory and control flow instructions. Research done in nZDCDidehban and Shrivastava (2016), shows that in SWIFT memory and control flow instructions (about 30% of a SWIFT-protected program) are susceptible to soft errors as they turn into single points of failure. SWIFT-R was an attempt to integrate the feature of forward recovery into SWIFT. SWIFT-R adds one redundancy on top of SWIFT's double modular redundancy and accomplishes recovery by means of majority voting between these three redundancies. However, similar to SWIFT, it suffers from

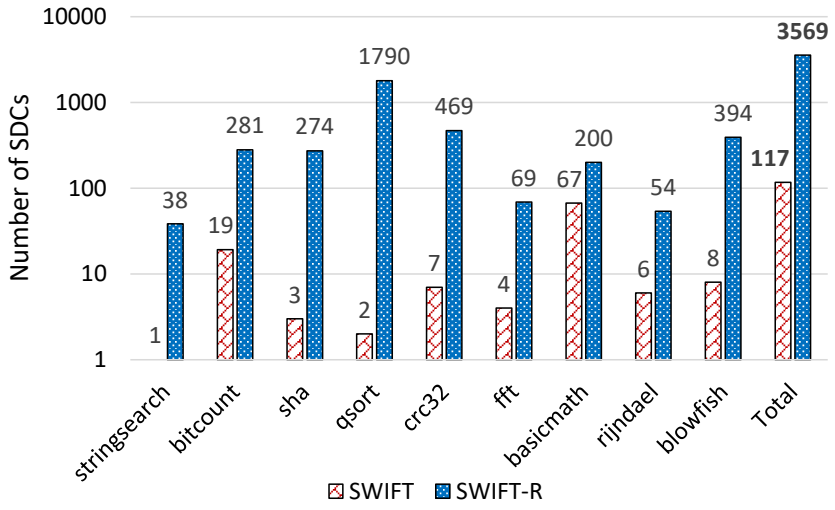

Figure 2.2: Total Number of SDCs in SWIFT vs SWIFT-R in Fault Injection Experiments Performed on Register File

many single-point-of-failures. In fact, in case of SWIFT-R these should be bigger because it imposes more register pressure than SWIFT, resulting in more spilling code (memory operations). Apart from this SWIFT-R replaces compare operations used by SWIFT's error detection routines with majority voting before every memory operation, thereby adding vulnerability to the already vulnerable parts of SWIFT programs. In-order to quantify the vulnerability added by SWIFT-R on top of SWIFT I performed 90,000 fault injection experiments on Register File while running 10 benchmarks from MIbench on a gem5 simulator based ARM processor(10,000 faults on each benchmark). As in shown in Figure 2.2, it was observed that adding SWIFT-R upon a given SWIFT program considerably increased number of SDCs in all the 10 benchmarks.

Recovery can also be implemented by means of checkpointing and rollback. Since the program replays certain number of instructions after resuming from most recent checkpoint, this mode of recovery is often referred to as Backward Recovery. Check-

pointing can be done either at application level (coarse-grained) (or) at instruction level (fine-grained).

Coarse-grained/offline check-pointing approaches Hargrove and Duell (2006); Sankaran *et al.* (2005) pause the execution of a program, save the memory and program states and then resume execution. So they need more storage for saving the state.

In this work InCheck, a fine-grained recovery approach that integrates check-pointing into the program at instruction level to accomplish fast and efficient error recovery is presented. InCheck implements recovery on top of a given fine-grained detection scheme without compromising the detection's fault coverage.

Recent works like Xu *et al.* (2013); Kadav *et al.* (2013) demonstrated the use of fine-grained checkpointing for error recovery. FASER Xu *et al.* (2013), is a fine-grain backward recovery approach implemented on top of SWIFT. The process of checkpointing in FASER is flawed. Check-pointing is performed only on store-free basic blocks. Therefore, if a store operation makes an erroneous change to memory, it cannot be undone as it does not make memory checkpoints. As a result, the recovery operation performed by restoring the values of registers from the previous checkpoint, can use the erroneous memory location after recovery and could lead to an SDC. SWIFT detection mechanism cannot detect such errors because it does not duplicate memory instructions. There are similar flaws in many other existing fine-grained backward recovery approaches. However, through InCheck I could perform a flawless recovery from almost every error caught by the detection mechanism.

The fault coverage offered by a fine-grained backward recovery approach highly depends on the detection mechanism they are built upon. These techniques cannot recover from errors undetected by their underlying detection scheme. Even if the recovery methodology is flawless, the coverage numbers could still be the same or even worse if the recovery is integrated into an improper detection scheme. Therefore,

prime importance was given to two factors while designing InCheck.

1. Choosing a detection technique that has very good coverage.

2. Implementing the recovery scheme without losing the coverage offered by detection technique.

Amongst all the recent literature on Fine-grained soft-error detection techniques we considered to implement InCheck, nZDC Didehban and Shrivastava (2016), could provide a near-zero SDC with an acceptable performance overhead. So it has been chosen as a candidate for implementing InCheck on top of it.

2.1 nZDC: A Detection Technique for near Zero SDC

nZDC is a fine-grain error detection technique which duplicates all computational,logical, memory read and compare instructions in a program and checks the results of memory write and conditional branch instructions after their execution. Figure 2.1 illustrates key ideas of nZDC data-flow error-detection. As it shows, the nZDC technique, loads-back the stored value (`x1`) from the memory and compares that against the redundantly-computed value(`x1*`). Therefore, nZDC is not only able to detect the effect of soft error on the operands of memory write instructions, but also errors which impact the execution of memory write instructions themselves.
nZDC introduced a new control-flow scheme which is able to detect wrong-direction and wrong-address control flow errors. Wrong-direction control flow errors are those errors which can directly or indirectly cause a branch direction to alter from taken to not-taken or vise-versa. For instance, if error happens during the computation of operand(s) of a compare instruction, it can change the result of compare instruction and, indirectly alter the direction of conditional branch instruction. In another instance, the error can directly affect the op-code of a conditional branch instruction,

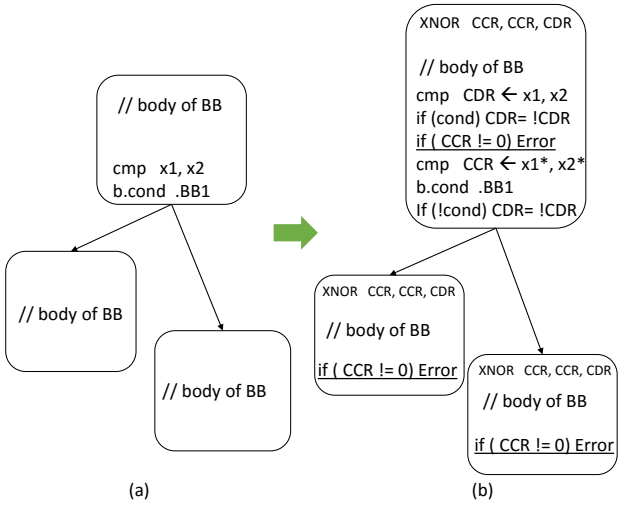

Figure 2.3: nZDC Wrong-Direction CF Error Detection

and result in a wrong-direction control flow error. Figure 2.3 demonstrates wrong-direction control flow detection part of nZDC control-flow checking technique. nZDC CF scheme also employees static signatures to detect wrong address control flow errors in which an unexpected jump will change the control flow of a program. The reasons of wrong address branches can be a fault on PC (Program Counter), errors which change the op-code of a non-branch instruction to a branch instruction, or the errors during the computation of target address of a taken-branch instruction. However, as study Shrivastava *et al.* (2014) shows that wrong direction control flow errors are dominant control flow errors and wrong target control flow errors are rare.

Chapter 3

InCheck: THE INTEGRATED RECOVERY METHODOLOGY

The goal of my thesis work is to provide an Integrated recovery scheme for nZDC-protected codes, without losing the coverage offered by nZDC. However, the main challenge in recovery from nZDC detected errors is its late error detection. nZDC detects errors by checking the results of store (after performing store instruction) or detects the control flow errors once the program is in the wrong basic block. Therefore, in each case, the recovery scheme should be able to return the state of the program (registers, memory, PC) to an error-free state by eliminating the possible side effects of error.

3.1 Recovery from Data Flow Errors

InCheck's strategy for recovery from errors that nZDC detects after a memory write instruction is presented in this section.

3.1.1 Safe Two-Phase Register-File Checkpointing

The main idea behind the InCheck is to make frequent built-in checkpoints at BB (Basic Block) granularity by preserving the live error-free registers in a specific part of memory (check-pointing storage) and restore them in the case of error detection. However, it is very crucial that the saved registers should be error-free. Therefore, the register preservation process requires some extra checking instructions (on the top of nZDC error checks) to make sure that the error-free registers are saved to the check-pointing storage. For that purpose, inCheck strategy reserves some spaces in memory which is as big as nZDC master registers (if the nZDC master instructions are using 15

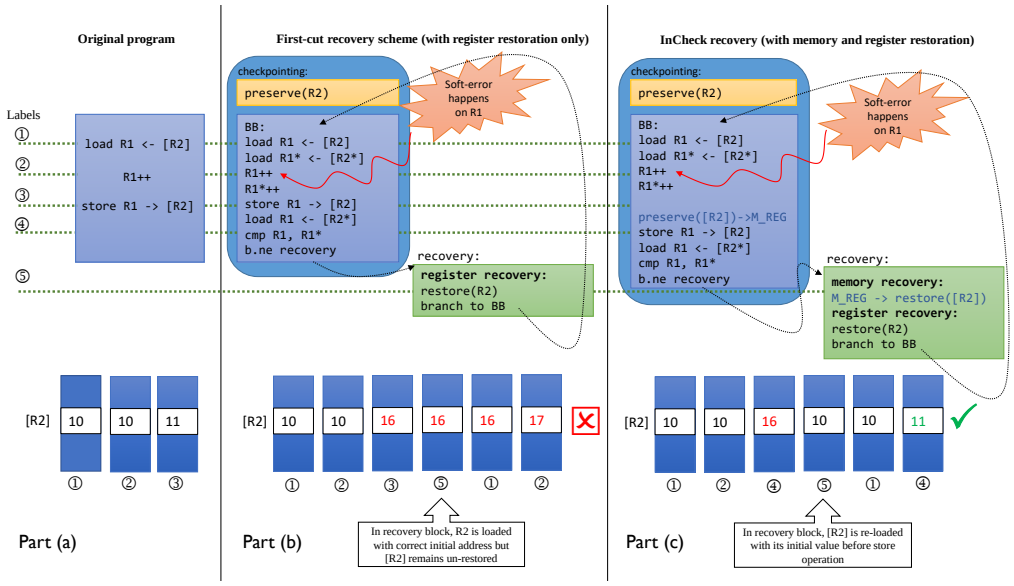

Figure 3.1: InCheck Recovery Strategy from Data-Flow Errors

64-bit wide programmer available registers, then checkpointing-storage size in memory should be 2 x 15 x 8 = 240 Bytes). InCheck partitions checkpointing-storage into two halves, and, periodically uses one segment to save the live registers of every alternate basic-block. The reason is that if the manifestation of error in a register pair gets detected, after the process of making a new checkpoint, the program can use the last safe checkpoint (which is preserved in the other segment of checkpointing-storage) to restore the register value and repeat the computation.

3.1.2 Need for Memory Checkpointing

Unfortunately, because of the late error detection strategy of nZDC, register file checkpointing alone cannot guarantee a successful recovery. For instance, consider the code snippet shown in figure 3.1. Part(a) of figure shows the original code, in which a variable reads some data from the memory, increments its value and saves it back to the same location of the memory. Part-(b) shows the first-cut solution for recovery. As it shows, the value of `R2` register was saved to check-pointing storage in the beginning

of BB and the body of BB is protected by nZDC error detection scheme. Now consider an error that alters the value of `R1` register. The store instruction then saves the faulty-value of `R1` into the correct memory location. However, nZDC checks after memory write instruction discover a mismatch between the stored value (`faulty-R1`) and the redundantly-computed value(`R1*`) and declare the manifestation of this error. The first-cut recovery technique loads the safely saved value of `R2` from check-pointing storage and transfers control flow of the program to first instruction in BB. However, the program loads the faulty value (`faulty-R1`) from the memory and increments it twice. Finally, program saves the `[R1-faulty++]` to the memory and since this value now matches with the redundant-computed ones, the program execution proceeds with wrong data.

3.1.3 InCheck Memory Preservation and Restoration

To solve the problem of memory checkpointing in an effective way, I first transform the nZDC-protected BB to safe-recovery basic blocks in which at most one memory instruction is allowed. Then, InCheck preserves a backup of memory location that is about to get overwritten by the store instruction by inserting a load instruction right before the program store instructions. It then saves the loaded value in a specially reserved register, called `MemReg`. Now, assume that the value of `x1` has changed to `faulty-x1` because of an error. The backup-load preserves the state of memory before store instruction, and InCheck Recovery routine first restores the memory state to last fault-free state by writing the memory-backed-up data, preserved in `MemReg`, to memory. Then, it restores the registers from check-pointing storage and transfers the control-flow of program to first non-checkpointing instruction of the basic block. Now the program can resume its execution without any erroneous data.

3.2 Recovery from Control-Flow Errors

Soft errors can change the control flow of a program in two ways,

- Wrong target
- Wrong direction

A wrong target control flow arises when, for instance, an error alters the target address of a taken branch instruction (if the branch is not-taken the error would get masked) or the error changes opcode of a non-branch instruction to a branch instruction, or even errors directly altering the value of program counter register. A wrong target control flow error gets detected with signature-part of nZDC control flow checking mechanism. A wrong direction control flow, on the other hand, occurs when a branch direction changes from taken to not-taken or vice-versa, which can be caused by errors affecting (directly or in-directly) `cmp` instructions (operands as well as opcode) or `branch` instruction's opcode or even program status flag register. Since the wrong direction control flow errors are considerably more frequent than wrong target errors Shrivastava *et al.* (2014), in this work, we provide the recovery from such errors, and, leave wrong target control flow errors as detected/unRecoverable.

The main challenge in recovery from control flow errors is that by the time nZDC detects the presence of a control-flow error, the register state saved in storage-checkpoint for the recovery purpose may not be the correct one due to several register preservation operations. For instance, consider an nZDC-protected basic block (nZDC-BB) which is a fan-in basic block (has more than one predecessors) and contains at least one memory write instruction and control-flow error-check is positioned close to the end of the basic block by nZDC technique. Since there is a store in the middle of basic block, InCheck transformation converts the nZDC-BB in two consecutive basic-blocks (InCheckBB1 and IncheckBB2), which are separated immediately

after memory write instructions. Assume that at run time the control flow of the program reaches to InCheckBB1 from the predecessor1 in which InCheck preserves the live registers of predecessor1 into the first segment of checkpointing-storage. Once the control-flow of program reaches to InCheckBB1, the register-file preservation takes place first and the live registers get saved into the second segment of checkpointing-storage and no error is detected because the nZDC control-flow error detector is placed at the end of InCheckBB2 basic block. Once the program control reaches to the InCheckBB2, InCheck checkpointing mechanism saves the live registers into first segment of checkpointing-storage (the register backup for the predecessor1 is gone). Finally, in the end of InCheckBB2, nZDC error detector discovers a control-flow error. However, the recovery is not possible because the back-up data for the predecessor1 block is already overwritten. To solve this problem, InCheck also adds an extra control-flow detection check after the register-file preservation at the beginning of all fan-in basic blocks. The idea of InCheck error recovery has been shown in Figure 3.2.

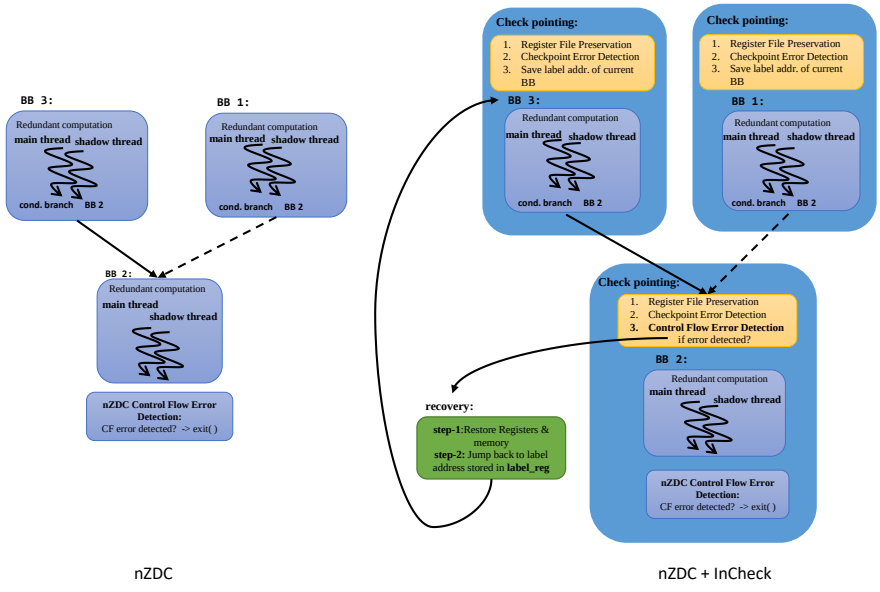

Figure 3.2: InCheck Recovery Strategy from Control-Flow Errors

3.2.1 Unrecoverable Errors

InCheck's recovery mechanism has been developed to always perform correct recovery. The diagnosis mechanism built into the recovery modules is responsible to decide whether a fault is recoverable or unrecoverable. If the diagnosis routine declares a fault as unrecoverable, InCheck stops the process of recovery and by deeming the fault as detected. This strategy of InCheck avoids the possibility if SDCs that can happen due to improper recovery.

The necessity to declare a fault as detected/unrecoverable comes when the store instruction corrupts a data element in memory that was not backed up by the load that performed memory preservation. Figure 3.3 depicts a code snippet from an nZDC+InCheck basic block. When a fault happens on register `x2` after the first load instruction and before the execution of store, there are chances that store instruction is going to corrupt a memory location that we did not backup into `M_REG`. In order to prevent that from happening, we tried to be conservative in designing InCheck's recovery process. Any error that changed the memory that we haven't backed up is deemed as Detected/Un-Recoverable error.

```
load   M_REG ← [x2]
store  x1→ [x2]
load   x1← [x2*]
compare     (x1 , x1*)
if(x1≠x1*) → recovery
```

Figure 3.3: Unrecoverable Faults

Chapter 4

EXPERIMENTAL RESULTS

4.1 Compilation and Simulation Framework

In order to evaluate the effectiveness of nZDC+inCHECK fault tolerant technique, I implemented SWIFTR and nZDC+inCHECK techniques as late back-end passes in LLVM 3.7 infrastructure for an ARMv8-a ISA. This implementation enabled me to take advantage of all the advanced compiler optimizations. I compiled 6 benchmarks from Mibench benchmark suite Guthaus *et al.* (2001) with *-O3* compiler optimization flag. For each benchmark three versions, Original , nZDC+inCheck, SWIFT-R were generated. It should be noted I did not modify the standard library functions and therefore am excluding them from all of fault injection and performance overhead estimation results shown in this work. Extensive fault injection experiments were performed on different hardware components of a modern, high-performance low-power, ARM cortex A-53 like microprocessor simulated in gem5Binkert *et al.* (2011) a cycle accurate simulator. Table 4.1 shows the details of the processor configuration.

4.2 Fault Model and Injection Set-Up

Fault model and Fault sites:

I used single bit-flip per execution as the fault model in this work. Faults were injected 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. Processor-wide fault injection enables us to

Parameter	Value
CPU Model	ARM64 bit in-order processor
Pipeline	Two way/4-stage
NUmber of FUs	2Int, 1Mul, 1Div, 1Float, 1Mem
L1 D/I-Cache	64KB (2-way) / 32KB (2-way)
TLB size	512 entries
Integer registerfile	32 registers (64-bit width)
Store buffer size	5 entries

Table 4.1: Simulator Parameters.

estimate the microprocessor-level recovery capability of InCheck. In fact as Cho *et al.* (2013) pointed out and our experimental results also confirm, injecting single bit-flip just in register file is not a true representative for the whole system, because it can not capture all effects of errors.

Number of fault injections experiments and outcome classification: To make sure that I covered almost all cases in my experimental results we randomly inject 400 faults for each version of a program. Thus, we injected 1600 faults in four hardware components per version of program. Overall, we performed 9600 fault injection experiments. According to Leveugle *et al.* (2009), these extensive fault injection experiments provide us a 1% error margin with 95% confidence interval in our results. It is worth to mention that similar researchers Didehban and Shrivastava (2016); Mitropoulou *et al.* (2014); Feng *et al.* (2010); Reis *et al.* (2007); Khudia *et al.* (2012) usually assume 95% confidence interval with 1% error which is achievable by just injecting about 400 fault injection experiments per component. For each fault injection experiment, a target component and a (bit, cycle) are randomly selected before the simulation run starts. Once the simulator reaches the target fault injection cycle,

simulation is paused, the selected bit is inverted, and then, the simulation run resumes its execution till simulation terminates or the allowable simulation time gets completed. The result of each simulation run is classified into one of the following category:

Masked: Program terminates and the output is correct.

Failed: Program terminates normally, but, the output is incorrect.

Detected/Recovered: Since the goal of this work is to prevent a program from producing any incorrect output and provide the recovery from the detected faults, we also count the number of Detected/Recovered faults.

Others: Program terminates by generating some symptoms such as segmentation fault or simulation time is over.

Chapter 5

EVALUATION AND ANALYSIS

5.1 Fault Injection Results

Figure 5.1 depicts the absolute number of injected faults which lead to SDCs in different hardware components. Fault injection during the execution of nZDC+InCheck-protected programs never resulted in an SDC. This implies that,

1. Error detection is able to detect all errors
2. Diagnosis routine distinguished all recoverable errors from unrecoverable errors,
3. If it was recoverable error, the recovery routine succeeded every single time.

The nZDC+inCHECK error detection is able to detect all errors because it takes place after the execution of critical instructions rather than before, and it checks

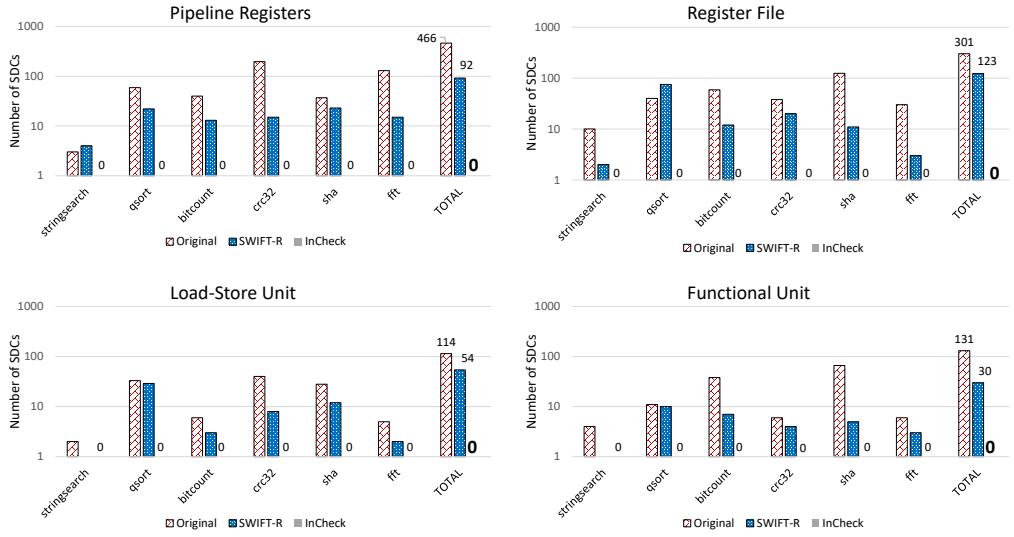

Figure 5.1: SDC Distribution in Component-Wise Fault Injection Experiments

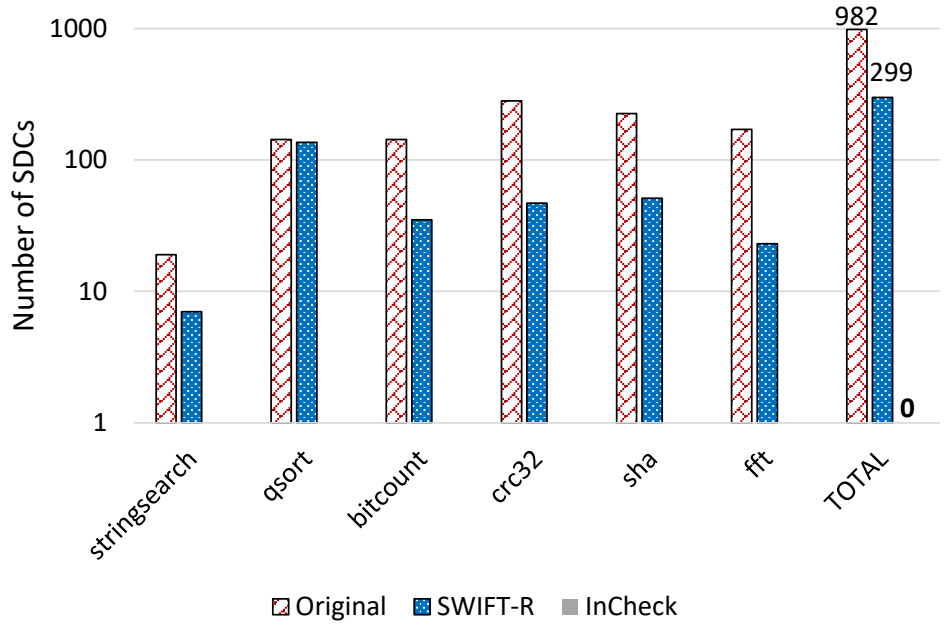

Figure 5.2: Total Number of SDCs from Processor-Wide Single Bit-Flip Fault Injection Experiments

for silent-store and missing-memory update vulnerable cases. However, SWIFTR transformation can reduce the percentage of SDCs from 16.4%, 8.8%, 9.1% and 3.8% to 8.7%, 4.1%, 1.7% and 1.4% for register file, pipeline registers, functional units and load-store unit, respectively. Surprisingly, for some programs such as qsort SWIFTR transformation actually doubles the number of failures! For a SWIFT-R protected program, the SDCs in Register file mainly occur when error happens between the time elapsed from voter providing its final output to the critical instruction reading it. In case of qsort, voter single-point-of-failures before frequent library calls are the reason for growth in SDC count. The main reason of SDCs from fault injection on pipeline register and functional units in SWIFTR-protected programs is that the injected fault affects the computation of an unprotected instruction, i,e. memory or compare operations. Figure 5.2 refers to the total number of SDCs that occurred during the Processor level Fault Injection campaign that was performed on Original, SWIFT-

R and nZDC+InCheck protected versions of the programs in MiBench Benchmark. SWIFT-R could bring down the total number of SDCs in Original program from 982 to 299. However InCheck could bring this number down to zero!

5.2 Detected and Unrecoverable Errors

As specified in 5.3, InCheck was unable to recover from about 5% of the faults detected by SDC. However, nZDC+InCheck caught all those unrecoverable faults and deemed them as detected. Restart/Retry can be employed as recovery in these scenarios.

Out of all the fault injection experiments performed on 4 major hardware components of the processor, InCheck showed 5 detected/unrecoverable faults in Register File, 61 in Load Store Queue, 2 in Pipeline Registers and 4 in Functional Unit Registers. Overall, InCheck had a successful recovery rate of about 96% throughout the Fault Injection Campaign.

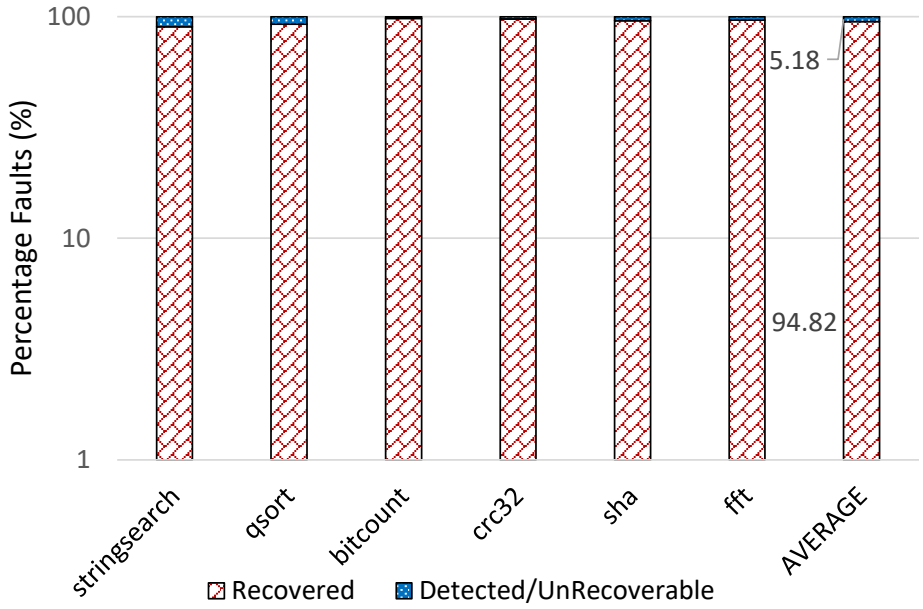

Figure 5.3: Percentage Recoverable vs Detected/Unrecoverable Faults with InCheck

5.3 Performance Overhead

Figure 5.4 shows the execution overheads of InCheck+nZDC and SWIFT-R protected programs normalized to Original Program. It can be see that on an average, an nZDC+InCheck version of a program can run 105% faster than its SWIFT-R equivalent. The performance overhead reported in this work may seem higher than similar works because of two main reasons.

1. Unlike common practice in related works Feng *et al.* (2010), numbers reported in this work are based on the cycles that a program spends in the user function where the protection scheme was applied. I excluded the cycles for executing unmodified library calls.

2. Similar works usually select an aggressively OoO processor which in protected version of the program can best utilize the hardware.

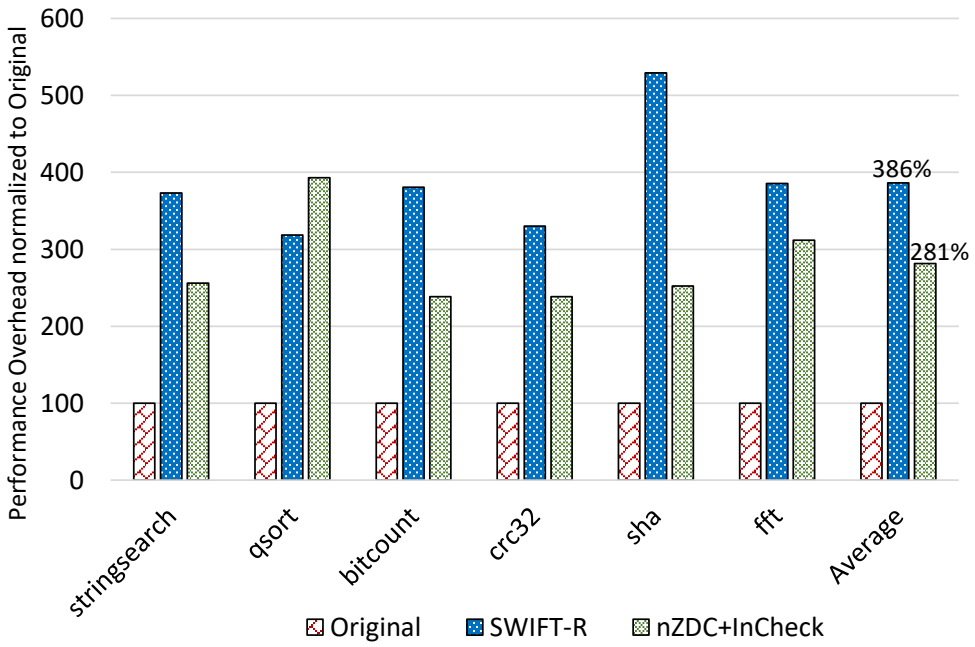

Figure 5.4: Execution Overhead of InCheck Normalized to nZDC

In Figure 5.5, the breakdown of dynamic instructions committed by InCheck and nZDC fractions of the executed program is presented. It shows that, on an average nZDC+InCheck program has about 20% more dynamic instructions compared to its nZDC counterpart.

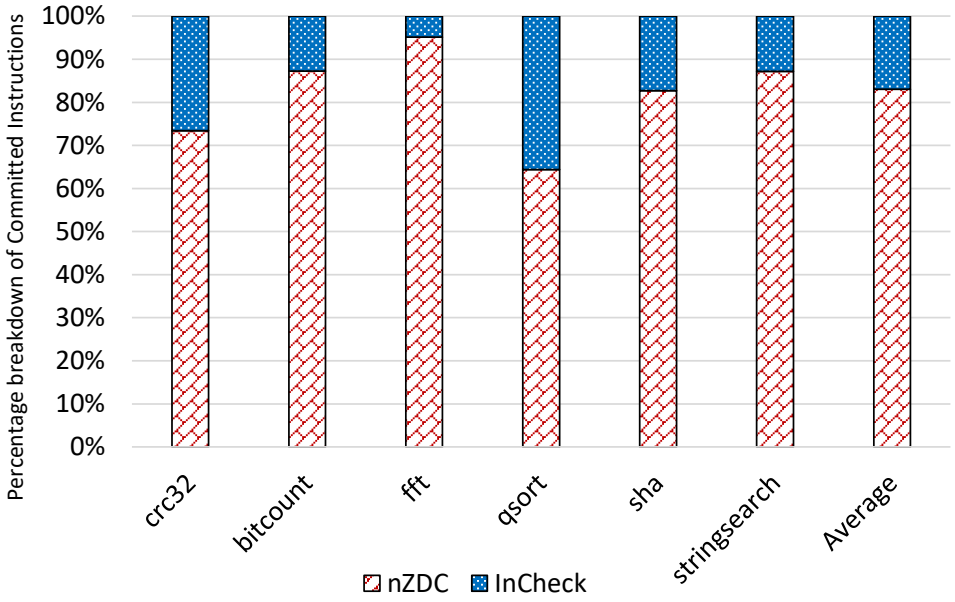

Figure 5.5: Dynamic Instruction Count of nZDC and InCheck

Chapter 6

LIMITATIONS OF InCheck

Though not encountered in our extensive FI experiments, there is a very rare possibility that InCheck's recovery could cause SDC. If error happens on InCheck's Store instruction (register file preservation) and changes the effective address of such instructions in a way that it overwrites useful data of a critical memory location being used by the program, it could cause an SDC. That being said, it can be avoided by adding load back and compare instructions inside checkpointing routine before every register preservation. Considering the fact that it hasn't happened in my experiments, I saved on execution time by making this trade-off.

In benchmarks with very large number of stores, the checkpointing overhead might be considerably high. This will result in large execution overheads. However, other recovery approaches also suffer from similar problems when the program has large number of store instructions.

Chapter 7

CONCLUSION AND FUTURE WORK

A backward recovery methodology called InCheck has been proposed in this work. It has been integrated with nZDC, a fine-grain duplication based SE detection technique. It could recover from almost all faults detected by nZDC. The fault injection results proved that the nZDC+InCheck protected programs could accomplish high recovery rates without compromising the fault coverage given by nZDC.

Since all our experiments were performed on a gem5 based processor simulator, there are chances that some of the processor's architectural bits might have not been modeled accurately inside it. So we are planning to perform RTL level fault injection experiments to test the soft-error resilience of InCheck.

The process of verification is never complete. There will always be certain corner cases that the random fault injection campaigns may not have covered. That being said, it is also impractical to go for a comprehensive (exhaustive) fault injection assessment. Therefore to understand any possible flaws in my approach that were not evident from my fault injection experiments, I am planning to consider doing a formal analysis of InCheck for a simple processor architectural model.

REFERENCES

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), 2013 IEEE 19th Pacific Rim International Symposium on", pp. 11–20 (IEEE, 2013).

Binkert, N. *et al.*, "The gem5 simulator", ACM SIGARCH Computer Architecture News **39**, 2 (2011).

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

Cho, H. *et al.*, "Quantitative evaluation of soft error injection techniques for robust system design", in "Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE", (IEEE, 2013).

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).

Döbel, B. and H. Härtig, "Can we put concurrency back into redundant multithreading?", in "Proceedings of the 14th International Conference on Embedded Software", p. 19 (ACM, 2014).

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, 97–108 (2004).

Feng, S. *et al.*, "Shoestring: probabilistic soft error reliability on the cheap", in "ACM SIGARCH Computer Architecture News", vol. 38 (ACM, 2010).

Guthaus, M. R. *et al.*, "Mibench: A free, commercially representative embedded benchmark suite", in "Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on", (IEEE, 2001).

Hargrove, P. H. and J. C. Duell, "Berkeley lab checkpoint/restart (blcr) for linux clusters", in "Journal of Physics: Conference Series", vol. 46, p. 494 (IOP Publishing, 2006).

Kadav, A., M. J. Renzelmann and M. M. Swift, "Fine-grained fault tolerance using device checkpoints", in "ACM SIGARCH Computer Architecture News", vol. 41, pp. 473–484 (ACM, 2013).

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

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, X., M. C. Huang, K. Shen and L. Chu, "A realistic evaluation of memory hardware errors and software system susceptibility", in "Proc. USENIX Annual Technical Conference (ATC10)", pp. 75–88 (2010).

Mitropoulou, K. *et al.*, "Drift: Decoupled compiler-based instruction-level fault-tolerance", in "Languages and Compilers for Parallel Computing", (Springer, 2014).

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

Oh, N. *et al.*, "Ed 4 i: error detection by diverse data and duplicated instructions", Computers, IEEE Transactions on **51**, 2 (2002b).

Quinn, H., Z. Baker, T. Fairbanks, J. L. Tripp and G. Duran, "Software resilience and the effectiveness of software mitigation in microcontrollers", IEEE Transactions on Nuclear Science **62**, 6, 2532–2538 (2015).

Reis, G. A. *et al.*, "Swift: Software implemented fault tolerance", in "Proceedings of the international symposium on Code generation and optimization", (IEEE, 2005).

Reis, G. A. *et al.*, "Automatic instruction-level software-only recovery", IEEE micro **27**, 1 (2007).

Rennels, D. A. and R. Hwang, "Recovery in fault-tolerant distributed microcontrollers", in "Dependable Systems and Networks, 2001. DSN 2001. International Conference on", pp. 475–480 (IEEE, 2001).

Restrepo-Calle, F., A. Martínez-Álvarez, S. Cuenca-Asensi and A. Jimeno-Morenilla, "Selective swift-r", Journal of Electronic Testing **29**, 6, 825–838 (2013).

Sankaran, S., J. M. Squyres, B. Barrett, V. Sahay, A. Lumsdaine, J. Duell, P. Hargrove and E. Roman, "The lam/mpi checkpoint/restart framework: System-initiated checkpointing", International Journal of High Performance Computing Applications **19**, 4, 479–493 (2005).

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

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., T. Moseley, V. J. Reddi, J. Blomstedt and D. A. Connors, "Using process-level redundancy to exploit multiple cores for transient fault tolerance", in "37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)", pp. 297–306 (IEEE, 2007).

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).

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", pp. 244–258 (IEEE Computer Society, 2007).

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).

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 and B. Chin, "Ibm experiments in soft fails in computer electronics (1978–1994)", IBM journal of research and development **40**, 1, 3–18 (1996).