Configurable Detection of SDC-causing Errors in Programs

by

Qining Lu

B.E., Beihang University, 2012

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE

in

The Faculty of Graduate and Postdoctoral Studies
(Electrical and Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA
(Vancouver)

February 2015

© Qining Lu 2015

Abstract

Silent Data Corruption (SDC) is a serious reliability issue in many domains, including embedded systems. However, current protection techniques are brittle, and do not allow programmers to trade off performance for SDC coverage. Further, many of them require tens of thousands of fault injection experiments, which are highly time-intensive. In this thesis, we propose two empirical models, namely *SDCTune* and *SDCAuto*, to predict the SDC proneness of a program’s data. Both models are based on static and dynamic features of the program alone, and do not require fault injections to be performed. The difference between the two models is that *SDCTune* is built using a manual tuning process, while *SDCAuto* is built using a machine learning algorithm. We then develop an algorithm using both models to selectively protect the most SDC-prone data in the program subject to a given performance overhead bound. Our results show that both models are accurate at predicting the relative SDC rate of an application. And in terms of efficiency of detection (i.e., ratio of SDC coverage provided to performance overhead), our technique outperforms full duplication by a factor of 0.78x to 1.65x with *SDCTune* model, and 0.62x to 0.96x with *SDCAuto* model.

Preface

This thesis is based on a work conducted by myself in collaboration with Dr. Karthik Pattabiraman, Dr. Meeta S. Gupta and Dr. Jude A. Rivers. The work was published as a conference paper in the ESWeek 2014 International Conference on Compilers, Architectures and Synthesis of Embedded Systems (CASES) [21]. I was responsible for coming up with the solution and validating it, evaluating the solution and analyzing the results, and writing the paper. Karthik was responsible for guiding me with the solution reasoning, experiments design and results analysis, as well as editing and writing portions of the paper.

Qining Lu, Karthik Pattabiraman, Meeta S. Gupta and Jude A. Rivers, SDC Tune: A Model for Predicting the SDC Proneness of an Application for Configurable Protection, the ESWeek International Conference on Compilers, Architectures and Synthesis of Embedded Systems, 2014

Table of Contents

Abstract	ii
Preface	iii
Table of Contents	iv
List of Tables	vii
List of Figures	viii
Acknowledgements	x
Dedication	xi
1 Introduction	1
1.1 Motivation	1
1.2 Background	2
1.3 Proposed Solution	5
1.4 Contributions	6
2 Initial Fault Injection Study	8
2.1 Terminology and Protection Model	8

Table of Contents

2.2	Fault Model	10
2.3	Fault Injection Experiment	11
2.4	Injection Results	12
2.5	Summary	14
3	Heuristics	16
3.1	Heuristics for Fault Propagation	17
3.2	Heuristics for Store Operations	19
3.3	Heuristics for Comparison Operations	26
3.4	Heuristics of Other Factors	27
3.5	Summary	31
4	Approach	34
4.1	Feature Extraction	34
4.2	Manually Tuned Model: <i>SDCTune</i>	38
4.3	Automatically Tuned Model: <i>SDCAuto</i>	40
4.4	Model Usage	44
4.5	Choosing the Instructions	45
4.6	Detector Design	45
4.7	Summary	48
5	Experimental Setup	49
5.1	Benchmarks	49
5.2	Evaluation Method	50
5.3	Work Flow and Implementation	53
5.4	Summary	56

Table of Contents

6 Results	57
6.1 Time Taken by Models	57
6.2 Effect of Decision Tree Parameters	59
6.3 Estimation of Overall SDC Rates	60
6.4 SDC Coverage and Detection Efficiency	64
6.5 Summary	67
7 Discussion	70
7.1 Differences between Benchmarks	70
7.2 Differences between Models	72
7.3 Threats to Validity	73
8 Related Work	75
8.1 Duplication Based Techniques	75
8.2 Invariant Based Techniques	77
8.3 Application or Algorithm Specific Techniques	78
9 Conclusion and Future Work	80
Bibliography	82

List of Tables

2.1	Variation of SDC proneness of highly executed instructions. . .	14
3.1	Effects on SDC proneness of some operations	18
3.2	SDC decreasing rates of masking/crashing prone operations . .	18
3.3	Four major categories of stored values	21
4.1	Some features extracted for modeling building	35
4.1	Some features extracted for modeling building	36
4.1	Some features extracted for modeling building	37
5.1	Training programs	50
5.2	Testing programs	51
6.1	Time consumption of <i>SDCTune</i> and <i>SDCAuto</i>	58
6.2	Features adopted by the optimal decision trees for <i>SDCAuto</i> model	62
6.3	The SDC rates and ranks from fault injections and our models	64

List of Figures

3.1	Average SDC proneness observed across all studied programs	20
3.2	Example of <i>Addr NoCmp</i>	22
3.3	Example of <i>Addr Cmp</i>	22
3.4	Example of <i>Cmp NoAddr</i>	23
3.5	Example of <i>Cmp NoAddr</i>	23
3.6	Example of <i>NoCmp NoAddr</i>	24
3.7	Example of <i>NoCmp NoAddr</i>	24
3.8	Examples of comparison results.	28
3.9	Average SDC proneness of global ones and others in resilient- comparisons-used variables	30
3.10	Average SDC proneness of comparisons with different depths of loop	31
3.11	Average SDC proneness for cumulative variables and others .	32
3.12	Average SDC proneness for different fan-outs	32
4.1	Auto tuned decision tree for stored values	44
4.2	Example of inserted detectors and concatenating instructions	46
4.3	Example of inserted detectors and lazy checking	47

List of Figures

5.1	The workflow of building regression trees and exploring the parameter space for <i>SDCAuto</i>	54
5.2	The workflow of applying our models	55
6.1	Effect of <i>data point threshold</i> (y-axis) and <i>minimum size of leaves</i> (x-axis) on regression results	61
6.2	The correlation of overall SDC rates for all programs.	63
6.3	The overhead of full duplication and hot-path duplication . . .	67
6.4	The results of <i>SDCTune</i> model for different performance overhead bounds, hot-path duplication and full duplication. . . .	68
6.5	The results of <i>SDCAuto</i> model for different performance overhead bounds, hot-path duplication and full duplication. . . .	69

Acknowledgements

First of all, I would like to thank my advisor Dr. Karthik Pattabiraman for his support during the past two years. Karthik constantly motivated me to think out of the box independently, and at the same time he gave me guidance and suggestions at the appropriate time. Without his support, this thesis would not have been possible, and I would not have improved so much in creative and critical thinking. Karthik is also a mentor to me. His passion, enthusiasm and consistency is always something I am trying to learn.

I would like to thank my colleagues in the Computer Systems Reading Group (CSRG) for their feedbacks on my practice talks. The weekly paper discussion meeting is also a place where I got the big picture in this area. Many thanks to my labmates for making the lab an enjoyable place to work, and for their advice and suggestions on my work and paper writing.

I am grateful to my dear friends in Canada and China, for helping me with some real problems or to talk about life.

Finally, special thanks to my family. They do not know what I am working on exactly, but they have always provided me support. I would never be whom I am today without their constant support.

Dedication

To my parents

Chapter 1

Introduction

1.1 Motivation

Hardware errors are increasing due to shrinking feature sizes [3, 7]. Conventional hardware-only solutions such as guard banding and hardware redundancy are no longer feasible due to power constraints. As a result, researchers have explored software duplication techniques to tolerate hardware faults [28]. However, generic software solutions such as full duplication incur high power and performance overhead, and hence there is a compelling need for configurable, application-specific solutions for tolerating hardware faults. This is especially so for embedded systems, which have to operate under strict performance and/or power constraints, in order to meet system-wide timing and energy targets.

Hardware faults can affect the running software in three ways: (1) they may not have any effect on the application (benign/masked), (2) they may crash or hang the program, or (3) they may lead to incorrect outputs, also called Silent Data Corruption (SDCs). While crashes and hangs are important from an availability perspective, SDCs are important from a reliability perspective because they cause programs to fail without any indication of the failure. Prior work [24, 34] has broadly focused on crashes and hangs; there-

fore we focus on configurable techniques to reduce or eliminate the number of SDCs in programs.

Studies have shown that SDCs are caused by errors in a relatively small proportion of programs' data variables [11, 14, 33], and by selectively protecting these SDC-prone variables, one can achieve high coverage against SDCs. However, most prior work has identified SDC-prone variables using fault injection experiments, which are expensive for large applications [11, 14]. Other work [33] focuses on Egregious Data Corruptions (EDC), which are a subset of SDCs that cause unacceptable deviations in soft-computing applications, i.e., applications with relaxed correctness properties. For example, a single pixel being corrupted in a frame of a video processing application would be an SDC but not an EDC, while the entire frame being corrupted would be an EDC as it can cause an unacceptable deviation. *While their approach is useful for soft-computing applications, it does not apply to general-purpose applications.* Further, most of the prior approaches do not allow the user to trade-off performance for reliability by selectively protecting only a fraction of the SDC-prone variables to satisfy strict performance constraints, especially for embedded systems. The only exception that we are aware of is the work by Shafique et al. [30], but their technique does not distinguish between SDC causing errors and other failure causing errors.

1.2 Background

We adopt Lapire et al.'s definition of "fault-error-failure" chain [17] as follows.

System failure occurs when the delivered service deviates from the specified service. The failure occurred because the system was erroneous: an error is that part of the system state which is liable to lead to failure. The cause in its phenomenological sense of an error is a fault.

Faults can be classified into two groups based on their sources: *hardware faults* and *software faults*. (1) Hardware faults represent a dysfunction of one or more hardware components for a period of time. Hardware faults can be further classified according to their lifetime into *transient hardware faults*, *intermittent hardware faults* and *permanent hardware faults*. Transient hardware faults usually occur only once at a location, and last only for a short duration of time. Such faults are usually triggered by cosmic ray strikes, temperature variation or electronic noise. Prior studies have show that transient hardware faults are increasing due to shrinking feature sizes [3, 7]. Intermittent hardware faults usually recur at a location over a period of time. They are usually the results of timing violations in the chip, which are exacerbated by wear out. Permanent hardware faults occur continuously at a faulty location, and are usually the result of manufacturing defects or circuit aging. (2) Software faults are those rooted in software code which are usually caused by the programmer’s mistakes or oversights. Memory leak, dangling pointers and deadlocks can be considered as common software faults. In this work, we focus on the errors caused by transient hardware faults.

A fault becomes an error once it corrupts the state of the program (i.e., changing the result of an operation, accessing wrong memory spaces, etc.).

The error may or may not result in a failure. Errors that do not cause failures are known as benign errors. Other errors are classified according to their consequences as follows. (1) *Crash*, if the errors trigger system alerts like hardware exceptions, operating system panic, etc. (2) *SDC*, if the program returns a wrong output without throwing any exceptions or raising any alerts. (3) *Hang*, if the program never ends or ends after a considerably long time. In this thesis, we focus on detecting SDCs, which are important for the reliability of the program.

To study the program behaviour in the presence of faults, we apply *fault injection* to simulate transient hardware faults in our initial study. Fault injection is a procedure to introduce faults in a systematic, controlled manner to study the behaviour of the system under test. Fault injection techniques can be generally categorized into hardware-based and software-based techniques. In this work, we adopt software-based fault injection technique. Software-based fault injection techniques emulate the effects of hardware faults at the software level by corrupting the values of program data/instructions [5, 15]. The main limitations of software-based techniques are their limited coverage of potential fault locations and speed. However, software-based techniques offer a high level of configurability without invasive hardware modifications and sufficient emulation speed to repeat thousands of runs. Therefore, we use software fault injection for our study.

1.3 Proposed Solution

In this thesis, we propose two models, namely *SDCTune* and *SDCAuto*, to quantify the SDC proneness of program variables, and develop a model-based technique to selectively protect highly SDC-prone variables in the program. An SDC prone variable is one in which a fault is highly likely to result in an SDC, and hence needs to be protected. *SDCTune* and *SDCAuto* use only static and dynamic analysis to identify the SDC-prone variables in a program, without requiring any fault injections to be performed. Further, it allows users to configure the amount of protection depending on the amount of performance overhead they are willing to tolerate. We call our first model, *SDCTune*, as it allows tunable protection, and second model, *SDCAuto*, as it builds the model automatically through a machine learning algorithm.

The main novelty of our approach is in the identification of heuristics that correlate with highly SDC-prone program variables and then integrating them in our model to quantify the SDC proneness of a variable. We extract these heuristics using fault injection experiments on a small set of benchmark programs that we use for training purposes. We integrate the heuristics in our models with automated program analysis and machine learning algorithms. While the initial identification of the heuristics used in *SDCTune* and *SDCAuto* requires fault injection, we do not need fault injection to apply our models to new programs.

In this thesis, we target transient errors, and hence we focus on error detection rather than recovery (as the program can be restarted from a checkpoint to recover from a transient error). We use *SDCTune* and *SDCAuto* to

identify SDC-prone variables in the program, and to derive error detectors for the variables, subject to a given performance overhead. Our detectors recompute the value of the chosen variable(s) by duplicating their backward slice(s), and compare the recomputed value with the original one. Any deviation between the two values is treated as a successful error detection.

1.4 Contributions

We make the following contributions in this thesis:

- We develop heuristics to identify SDC-prone variables based on an initial fault-injection study (Chapter 2). These heuristics are based on static analysis and profile information (Chapter 3).
- We first develop a manually-tuned model, *SDCTune*, based on the heuristics developed to identify the relatively SDC-prone variables in a program. We then propose an algorithm based on the model to derive error detectors that check the values of the SDC-prone variables at runtime, subject to a performance overhead constraint specified by the programmer (Chapter 4).
- We also develop a automatically tuned model, *SDCAuto*, based on the decision tree machine learning algorithm [27] which can automatically build a regression model from training data.
- We evaluate *SDCTune* and *SDCAuto* by using them to predict the overall SDC proneness of a program relative to other programs. The results show that both *SDCTune* and *SDCAuto* are highly accurate at predicting the overall SDC proneness of a program relative to other programs.

The correlation coefficient between the predicted and observed overall SDC rates ranges from 0.855 to 0.877 (Chapter 6) depending on the model.

- We evaluate the detectors inserted by our algorithm by performing fault-injection experiments on six *different* programs from those used in our model extraction, for performance overhead bounds ranging from 10% to 30%. The results show that our detectors can achieve high detection coverage for SDC-causing errors, for the given performance overhead. *SDCTune* achieves 0.78x to 1.65x higher efficiencies (i.e., ratio of SDC coverage provided to performance overhead) than both full duplication and hot-path duplication, *SDCAuto* achieves 0.62x to 0.96x higher efficiencies (Chapter 6).

Chapter 2

Initial Fault Injection Study

Because SDC failures are caused by faults that propagate to the program’s output, the SDC proneness of an instruction depends on how it propagates a fault, which in turn is determined by its data dependencies. In this chapter, we empirically study how *SDC proneness* of instructions is influenced by the data dependency chains. We first define some terms we will use in this thesis and formalize the protection problem. We then present our fault model in Section 2.2 and describe our fault injection experiment in Section 2.3. The results of the experiment is discussed in Section 2.4, and will be used in Chapter 3 to develop heuristics for estimating the SDC proneness of program variables.

2.1 Terminology and Protection Model

We first define the following terms in this paper:

Overall SDC rate: This is the overall probability that a transient hardware fault leads to an SDC in the program. We denote this by $P(SDC)$.

SDC coverage of an instruction: We define the SDC coverage of an instruction I to be the probability that an SDC failure is caused by a transient hardware fault in instruction I ’s result and thus can be detected

by protecting instruction I with a detector. This is denoted as $P(I|SDC)$.

SDC proneness per instruction: This is the probability that a transient hardware fault in instruction I leads to an SDC. This is denoted as $P(SDC|I)$.

Dynamic count ratio: This is the ratio of the number of dynamic instances of instruction I executed to the total number of dynamic instructions in the program. This is denoted as $P(I)$.

Our overall goal is to selectively protect instructions with detectors, to maximize the SDC detection coverage for a given performance cost budget. The SDC detection coverage of an instruction, $P(I|SDC)$, represents the "fraction of SDCs" that can be detected by protecting instruction I , and thus directly represents the importance of the instruction I . Therefore, our goal is to maximize the $\sum_{I \in inst\ set} P(I|SDC)$ subject to a certain $\sum_{I \in inst\ set} P(I)$ specified by the user. $\sum_{I \in inst\ set} P(I|SDC)$ is the coverage of SDC causing faults by protecting the instructions in set: *inst set* while $\sum_{I \in inst\ set} P(I)$ is the number of dynamic instances of protected instructions and is proportional to the protection overhead.

As mentioned above, it is important to understand how $P(I|SDC)$ varies for each instruction in the program. One way to do this is to perform random fault injection into the program and measure $P(I|SDC)$ for each instruction. However, it is difficult to directly measure this probability for each instruction by random fault injection as each instruction may not be injected sufficient number of times to obtain statistically significant estimates. Instead, we perform a fixed number of fault injections into individual instructions to measure their SDC proneness, $P(SDC|I)$. We then use Bayes' formula to

obtain $P(I|SDC)$:

$$P(I|SDC) = \frac{P(SDC|I)P(I)}{P(SDC)} \quad (2.1)$$

where,

$$P(SDC) = \sum_{I \in prog} P(SDC|I)P(I) \quad (2.2)$$

2.2 Fault Model

We consider transient hardware faults that occur in processors and corrupt program data. Such faults are usually caused by electrical noise, cosmic rays or temperature variation. These faults are exacerbated by decreases in feature sizes and supply voltages. More specifically, we focus on the faults that occur in processors' functional units and registers, (i.e., the ALUs, LSUs, GPRs, etc.) which generally result in a corruption of the program data. However, we do not consider the faults in caches or control logic. Architectural solutions [19] such as ECC or parity can protect the chip from the faults in the caches, while faults in the control logic usually trigger hardware exceptions [37]. We do not consider faults in the program's code or program counter, as such faults can be detected by control-flow checking techniques.

As in other work [10, 11, 33], we assume that at most one fault occurs during a program's execution. This is because transient faults are rare relative to the execution times of typical programs.

2.3 Fault Injection Experiment

The goal of our fault injection experiment is to understand the reasons for SDCs when faults are injected into the program. In other words, we want to study the SDC proneness of instructions in the program, and understand how it varies by instruction.

The fault injection experiment is conducted using LLFI, a program level fault injection tool, which has been shown to be accurate for measuring SDCs in programs [35]. LLFI works at the intermediate representative (IR) level of LLVM compiler infrastructure [18], and enables the user to inject faults into the LLVM IR instructions. Using LLFI, we inject into the result of a random dynamic instruction to emulate the effect of a computational error in the program. Specifically, we corrupt the instruction’s destination register by flipping a single bit in it (similar to what prior work has done [10, 11, 33]). The main advantage of using LLFI is that it allows us to map the faults back to the program’s IR and trace its propagation in the program. This is necessary for our analysis.

Please note that LLVM applies Static Single Assignment (SSA) form in its IR code, so that each instruction is represented as its own result. This makes a program variable equivalent to the instruction that is computing it when considering program data in LLVM IR code. Therefore, we consider instructions and program variables as interchangeable in this thesis.

We use four benchmarks in this experiment, namely *Bzip2*, *IS*, *LU* and *Water-spatial*. They are from SPEC[13], NAS[1] and SPLASH-2[36] benchmark suites respectively. Note that these benchmarks are only used for the

initial fault-injection study - we later derive and validate the model with a larger set of programs. We choose a limited set of benchmarks in this study to balance representativeness with time efficiency for fault injections.

We classify the outcome into four categories: (1) Crash, meaning that the program threw an exception, (2) SDC, which means the program’s output deviated from the fault-free outcome, (3) Hang, which means the program took significantly longer to execute than a fault-free run, and (4) benign, which means the program completed successfully and its output matched the fault-free outcome. The above outcomes are mutually exclusive and exhaustive.

2.4 Injection Results

The results of our fault injection experiments show that the top 10% most executed instructions, or those on the hot paths of the program, are responsible for 85% SDC failures on average. This result is similar to that of prior work, which has also observed that a small fraction of static instructions cause most SDCs [11]. However, this does not mean that all the hot-path instructions should be protected, as they incur high performance overhead when protected. Further, there is considerable variation in SDC rates even among the top 10% most executed instructions as the example below shows.

Table 2.1 shows an excerpt from the *Bzip2* program on its hot path. The principle described here is observed across all four benchmarks we studied, but we focus on this (single) basic block for simplicity. The excerpt contains instructions from the LLVM IR, into which we inject faults. Although the

2.4. Injection Results

original code is in the LLVM IR form, we use C source-like semantics for simplicity. For each instruction in the table, we report its SDC proneness measured by fault injection. It can be observed from the table that some of the instructions have low SDC proneness, even in this highly executed block, e.g., *instruction 4-6*. This means even if a fault occurs in the result of these instructions, it is unlikely to result into an SDC, and hence protecting such instructions is unlikely to improve coverage by much. Therefore, we need to find factors other than execution time that influence the SDC proneness of an instruction.

After investigating further, we found that SDC proneness is highly influenced by data dependencies among the instructions. For example, in Table 2.1, *instruction 4-8* constitute a data dependency chain whose final result is stored in *instruction 10*. *Instruction 8* is the end of this data dependency chain and has an SDC proneness = 71%. The result of *instruction 7* is used in *instruction 8* so a fault may propagate from *instruction 7* to *instruction 8*. But, the execution of *instruction 8: or* can mask the faulty bit from *instruction 7* if the corresponding bit of the result of *instruction 2* is 1. This explains why the SDC proneness for *instruction 7* is slightly lower than that of *instruction 8*. The operation of *instruction 7: shift left* can mask the fault in high bit positions of the second source operand due to architectural wrapping implementation of these shifting operations. The consequence of this masking effect is the low SDC proneness of *instruction 4-6*. In addition to the arithmetic operations, our results show that address calculation operations such as *instructions 1, 3* and *9* ("getelementptr" instructions in LLVM) have low SDC proneness. This is because the results of

2.5. Summary

Table 2.1: Variation of SDC proneness of highly executed instructions.

Source code:

```
1  s->bsBuff |= (v << (32 - s->bsLive - n));
```

Basic block	ID	Instruction	SDC proneness
bsW()-bb2	1	$t_1 = \&s + \text{OFFSET}(\text{bsBuff})$	21%
	2	$t_2 = \text{load } t_1$	47%
	3	$t_3 = \&s + \text{OFFSET}(\text{bsLive})$	21%
	4	$t_4 = \text{load } t_3$	13%
	5	$t_5 = 32 - t_4$	12%
	6	$t_6 = t_5 - n$	12%
	7	$t_7 = v \ll t_6$	58%
	8	$t_8 = t_2 \mid t_7$	71%
	9	$t_9 = \&s + \text{OFFSET}(\text{bsBuff})$	26%
	10	store t_8, t_9	-

such instructions are usually used for pointer dereferences and are likely to cause segmentation faults which crash the application.

Thus, we see that to calculate the SDC proneness of an instruction and determine whether it should be protected, one needs to take into account the *fault propagation* and *SDC proneness of the end point* of its data dependency chain. We will examine this in more detail in Chapter 3 by devising heuristics for finding highly SDC-prone instructions.

2.5 Summary

This chapter defined the fault model adopted in our technique in Section 2.2. It also defined the core problem of building a configurable SDC

detection technique in Section 2.1 namely, estimating the SDC proneness of an instruction. It then presented the results of fault injection experiments (Section 2.3) and found that *fault propagation* and *SDC proneness of data dependency end points* are the two major factors required to estimate SDC proneness (Section 2.4). We will examine the two factors in detail and formulate heuristics for them in Chapter 3, and then propose our approach for configurable SDC detection in Chapter 4

Chapter 3

Heuristics

In this chapter, we formulate various heuristics for modelling error propagation in a program, and for estimating the SDC proneness of an instruction. We first propose heuristics as hypotheses, and validate them with our experimental data. These heuristics will be used in the next chapter to extract program features that are required to build both manually tuned and automatically tuned model.

In the previous chapter, we found that the SDC proneness of a variable depends on (1) the fault propagation in its data dependency chain, and (2) the SDC proneness of the end point of that chain. An end point can be a branch instruction, a store instruction or a function call instruction (in LLVM, function calls are represented by instructions). This is because stores and branches do not have destination registers, and function call instructions create a new stack frame, thereby terminating their dependency chains. However, function calls are not considered in our work, as LLVM aggressively inlines functions, and hence there are few instances of such instructions. Further, because branch instructions depend on the results from comparison instructions to determine the direction of the branch, we consider the results of comparison instructions as the end points of their dependency

chains. Therefore, we consider only comparison and store instructions for the SDC proneness of end points of dependency chains.

3.1 Heuristics for Fault Propagation

In this section, we study how faults propagate along dependency chains, and how to estimate the SDC proneness of an instruction based on the SDC proneness of the store or comparison instructions that the instruction depends on, directly or indirectly.

HP1: The SDC proneness of an instruction will decrease if its result is used in either fault masking or crash prone instructions.

Fault propagation can be stopped by an instruction either masking the fault, or by crashing the program. Both masking and crashing decrease the probability of an SDC resulting from the instruction that propagates its data to the other crashing/masking instruction, as a result of which its SDC proneness is lowered. For example, in Table 2.1, the fault masking effect of *instruction 7* results in *instruction 6* having a low SDC proneness.

Table 3.1 shows instructions that have high probability of masking/crashing the program, thus lowering the SDC proneness. We derived this table from the initial fault injection study in Section 2.4, based on general trends across the applications. Note that these are conservative, as other instructions may also mask fault propagation in specific circumstances depending on the values of their operands.

To estimate SDC proneness of all instructions, we simulate backward fault propagation from the store and comparison instructions through the data dependency chains of the program. The SDC proneness of the result

3.1. Heuristics for Fault Propagation

Table 3.1: Effects on SDC proneness of some operations

Operation	Description	Effect
getelementptr	address calculation	Crash
trunc	truncate data size	Mask due to truncation
lshr	logical shift right	Mask due to Wrapping
ashr	arithmetic shift right	Mask due to Wrapping
shl	shift left	Mask due to Wrapping

Table 3.2: SDC decreasing rates of masking/crashing prone operations

Operation	Involved source operands	Decrease by
getelementptr	all operands	75%
trunc	variable needs truncation	50%
lshr	shift bit variable	85%
ashr	shift bit variable	85%
shl	shift bit variable	85%

of an instruction will propagate to its source operands unless it is one of the operations listed in Table 3.1, in which case, the SDC proneness of the source operands will decrease by a certain extent, as listed in Table 3.2 to model the effect of masking. The values in Table 3.2 are based on our fault injection experiments.

Then, the question left is how to estimate the SDC proneness of store and comparison instructions. This is addressed in the following two sections.

3.2 Heuristics for Store Operations

In this section, we examine the SDC proneness of store instructions, as this is one of the two categories of instructions used to estimate the SDC proneness of every instruction in the program. Through our fault injection study in Section 2.4, we found the SDC proneness of store instructions depends on how the stored value is used in the program. Therefore, we categorized the stores into four types according to their usage in memory addresses and comparisons, as shown in Table 3.3. For each of the categories, we found that the SDC proneness is dependent on a specific feature of that category, which is also shown in Table 3.3. For example, in the *Cmp NoAddr* category, the SDC proneness of the store is determined by whether the value results in the comparison result being flipped, thus causing the wrong fork of the branch to be taken. Figure 3.1a shows the average SDC proneness of the four categories, and the associated feature for each of the categories.

We now examine each of the four categories in detail.

*HS1: **Addr NoCmp** stored values have low SDC proneness in general, as shown in Table 3.3.*

3.2. Heuristics for Store Operations

Figure 3.1: Average SDC proneness observed across all studied programs

3.2. Heuristics for Store Operations

Table 3.3: Four major categories of stored values

Category	Description	Major related features	Average SDC proneness
Addr NoCmp	The stored value is used in calculating memory addresses but not comparison results	Data width	22.82%
Addr Cmp	The stored value is used in calculating both memory addresses and comparison results	Data width and control flow deviation	48.17%
Cmp NoAddr	The stored value is used in calculating comparison results but not memory addresses	Resilient or Unresilient comparison	67.25%
NoCmp NoAddr	The stored value is neither used in memory address calculation nor comparison results	Used in output or not	56.41%

This is because faults in such values are highly likely to propagate to addresses of other loads and stores, which would likely result in the application crashing due to a segmentation fault, especially for those values that are wider than 32 bits (see Figure 3.1a).

Figure 3.2 shows an example of this category, where a fault in the destination register of *i-3* in (line 3) results in a system crash upon pointer dereference.

HS2: Addr Cmp stored values usually have higher SDC proneness than Addr NoCmp.

As shown in Figure 3.1a, by propagating the fault to the comparison instruction, *Addr Cmp* values may change the control flow and elide the pointer dereference, which would have crashed the application otherwise. This decreases the probability of a crash, thereby increasing the SDC prone-

3.2. Heuristics for Store Operations

```
1 static void mainSort (...) {
2     for (; i >= 3; i -= 4)
3         {... ptr[j] = i-3; } // corrupted
4 }
5 static void mainSimpleSort (...) {
6     while (mainGtU(ptr[j-h]+d, ...))
7         {...}
8 }
9 static Bool mainGtU(UInt32* i1, ...) {
10    c1 = block[i1]; ... i1++; c1 = block[i1]; // load operation
11 }
```

Figure 3.2: Example of *Addr NoCmp* from Bzip2.

```
1 static void mainSort (...) {
2     Int32 lo = ftab[sb] & CLEARMASK; // corrupted
3     if (hi > lo) { // control flow changed
4         mainQsort3(lo, ...);
5     }
6 }
7 void mainQSort3(Int32 loSt, ...) {
8     mpush(loSt, ...); ... mpop(lo, ...);
9     med = (Int32) mmed3(block[ptr[lo]+d], ...); // load avoided
10 }
```

Figure 3.3: Example of *Addr Cmp* from Bzip2. The fault occurs at line 2 may not propagate to the load at line 9 because of the control flow deviation at line 3

3.2. Heuristics for Store Operations

```
1 Bool copy_input_until_stop(Estate* s){
2     ...
3     while(True){
4         ...
5         s->strm->total_in_lo32++;
6         if(s->strm->total_in_lo32==0)
7             s->strm->total_in_hi32++;
8     }
9 }
```

Figure 3.4: Example of *Cmp NoAddr* from Bzip2. A resilient comparison operations(line 6) that masks the fault that occurs at line 5.

```
1 static void sendMTFValues(Estate* s){
2     for(i=0;i<nSelectors;i++){
3         s->selectorMtf[i]=j;
4     }...
5     for(i=0;i<nSelectors;i++){
6         for(j=0;j<s->selectorMtf[i];j++)
7             bsW(s,1,1);
8     }
9 }
```

Figure 3.5: Example of *Cmp NoAddr* from Bzip2. An unresilient comparison usage of the stored value at *s->selectorMtf[i]=j*(line 3).

3.2. Heuristics for Store Operations

```
1 void main (...) {
2     ...
3     (start) = (unsigned long)(FullTime.tv_usec + FullTime.tv_sec *
4         1000000);
5     ...
6     Global->starttime = start;
7     printf(..., Global->starttime);
8 }
9 //The value stored in
10 //Global->starttime is not used
11 //as the output of the program
```

Figure 3.6: Example of *NoCmp NoAddr* from *IS* with zero SDC proneness.

```
1 void InitA(double* rhs){
2     for(j=0;j<n;j++){
3         for(i=0;i<n;i++){
4             rhs[i] += a[i][j];
5         }
6     }
7 }
8 void CheckResult(..., double* rhs){
9     for(j=0;j<n;j++){y[j]=rhs[j];}...
10    for(j=0;j<n;j++){diff=y[j]-1.0;...}
11    max_diff=diff
12    printf(..., max_diff);
13 }
```

Figure 3.7: Example of *NoCmp NoAddr* from *LU* with high SDC proneness.

3.2. Heuristics for Store Operations

ness compared to the *Addr NoCmp* category. As an example of this category from *Bzip2* is shown in Figure 3.3.

*HS3: The SDC proneness of **Addr NoCmp** and **Addr Cmp** stored values increase as their **Data width** decrease.*

Data width is the number of bits in values, and is a major feature affecting the SDC proneness of stored values used in address computation (i.e., *Addr NoCmp* and *Addr Cmp*). Figure 3.1b shows the average SDC proneness of the stored values used in address computations, for different data width values. For values used in address computation, a wider data width means more bits are crash-prone, and hence the value as a whole has lower SDC proneness.

*HS4: The SDC proneness of **Cmp NoAddr** stored values depends on the resilience of the comparison operation to which the value propagates i.e., how likely it is to change the result of the comparison given a faulty data operand.*

We illustrate the above heuristic with an example from the *Bzip2* application. Figure 3.4 shows an example of a resilient comparison operation in line 6. In this case, the equality is not satisfied in the majority of executions (obtained through profiling the program), and hence the branch is highly biased toward the not-equal fork. Therefore, a fault in the variable *total_in_lo32*(line 5) which feeds into the comparison operation is unlikely to result in the equality being true, and hence the control flow of the program does not change from a fault-free execution. We call such comparisons as *resilient*. On the other hand, the code in the right of Figure 3.5, illustrates a case where a fault in the comparison operator, *selectorMtff[i]=j*(line 3) will affect the number of loop iterations, thus making it highly SDC prone.

3.3. Heuristics for Comparison Operations

We call such comparisons as *unresilient*. A key factor in deciding the SDC proneness of **Cmp NoAddr** stored values is whether the comparison using the stored value is resilient (Figure 3.1a).

*HS5: The SDC proneness of **NoCmp NoAddr** stored values depend on the probability of a fault in them propagating to the program's output, and whether the output is important to the program.*

NoCmp NoAddr stored values are used neither in computing memory addresses nor in comparison instructions, and do not affect pointers or branches. Figure 3.6 and Figure 3.7 show two excerpts from *IS* and *LU* respectively. The faulty stored value in *IS* only affects the time statistics while the one in *LU* may affect the output of the application. This explains the difference of their SDC proneness. Also in Figure 3.1a, we can see the average SDC proneness for the stored values that do not propagate to program output is much lower than the SDC proneness of those values that do.

3.3 Heuristics for Comparison Operations

Comparison instructions are the other category of instructions whose SDC proneness determines the SDC proneness of every instruction in the program. We find that the SDC proneness of comparison instructions depends on three features, as follows:

*HC1: **Nested loop depths** affect the SDC proneness of loops' comparison operations, as the SDC proneness of comparison operations in inner loops are generally lower than the comparison operations in outer loops, as shown in Figure 3.1c.*

Figure 3.8a shows an example from *Bzip2*. Both *nHeap* > 1 and *weight[tmp]*

3.4. Heuristics of Other Factors

$< \text{weight}[\text{heap}[\text{zz} \gg 1]]$ are used in determining the loop exit conditions for the outer and inner loops respectively.

HC2: Comparison operations that only affect silent stores have low SDC proneness.

A silent store is a store whose stored value is not subsequently used by the program. Therefore, the comparison operation has a low likelihood of affecting the program's output. An example from *Bzip2* is shown in Figure 3.8b. A flip in the comparison $a2update < BZ_NOVERSHOOT$ (line 4) can cause the store operation $quadrant[a2update + nblock] = qVal$ (line 5) to be elided. However, this is a silent store, and hence does not result in an SDC.

HC3: Comparisons that affect output-related store values have high SDC proneness.

A fault in these comparisons has a high probability of resulting in a corrupted program output. Figure 3.8c shows an example from the *LU* benchmark. A faulty comparison result at $i < n$ (line 3) may terminate the loop too early and elide the store operation $a[i] += alpha * b[i]$ (line 4) whose stored value is used in calculating the output. This results in a high SDC proneness of $i < n$ (line 3).

3.4 Heuristics of Other Factors

In addition to the specific features for comparison and store operations we observed in our experiment, the following factors also affect the SDC proneness of an instruction.

HO1: Memory allocation functions related stored values and comparison operations have low SDC proneness.

Memory allocation functions related stored values or comparison

3.4. Heuristics of Other Factors

```

1 void BZ2_hbMakeCodeLengths
  (...) {
2   while (nHeap>1){ //outer loop
3     ...
4     while (weight[tmp]<weight[
        heap[zz>>1]]){
5       //inner loop
6       Heap[zz]=heap[zz>>1];
7       zz>>1;
8     }
9   }
10 }

```

(a) An excerpt from *Bzip2*. Outer loop comparisons have higher SDC proneness than inner loop comparisons

```

1 mainSort (...) {
2   for (j=bbSize-1; j>=0; j--){
3     ...
4     if (a2update
        <BZ_NOVERSHOOT)
5       quadrant[a2update+nblock
        ]=qVal;
6     //Not used in future
7     ...
8   }
9   ...
10 }

```

(b) An example from *Bzip2* that the comparison result only affects a silent store instruction

```

1 daxpy(double* a, double* b,...) {
2   long l;
3   for (i=0;i<n;i++){
4     a[i]+=alpha*b[i]; //skipped due to loop termination}
5   }
6 bmodd(double* a, double* c,...) {...
7   daxpy(&a[k+1+j*stride_c],&a[k+1+j*stride_a],dimi-k-1,alpha);...
8   //the content of a[] is corrupted
9   }
10 lu() {...
11   A=&a[K+j*nblocks]; //fault propagates to a[] through the call of
        bmodd()
12   bmodd(D,A,strK,strJ,strK,strK);... //content of A[] is corrupted
13 }
14 CheckResult(...,double* a,...) {... //called by main()
15   printf(...,max_diff,...);... //corrupted because of corrupted a[]
16 }

```

(c) An example from *LU* that a faulty comparison result: $i < n$ (line 3) will change the control flow and finally affect *CheckResult()*. The fault propagation trace is highlighted in red.

Figure 3.8: Examples of comparison results.

3.4. Heuristics of Other Factors

operations can directly affect memory allocation functions such as `malloc()`, `valloc()`, `palloc()`, and hence faults in the instructions are very likely to trigger memory exceptions. This results in having low SDC proneness. We observe that the average SDC proneness for memory allocation related store or comparison operations is 12.42%, which is considerably lower than the average of other store and comparison operations, which is 42.58%.

In addition to the above features, we consider other program features considered in prior work, such as *global variable* [10], *the loop depth* [33], *cumulatively calculation* [6], and *fan-out of variable* [24].

*HO2: For variables that are used in resilient comparisons, **global variables** in this group have higher SDC proneness than others variables that heading to resilient comparisons.*

As mentioned in prior work [10], global variables are more likely to store the global states of a program, so a fault in these variables is likely to live longer and affect more program data. In our initial study experiments, we found this effect can be helpful in estimating SDC proneness for variables that are used in resilient comparisons. As presented in Section 3.2 HS4, resilient comparisons can mask faults in their backward slices so that variables used in such comparisons may have lower SDC proneness. However, faults in global variables have longer life time and are hence less likely to be masked. Figure 3.9 shows the average SDC proneness of global variables and others in the group of variables leading to resilient comparisons.

*HO3: Comparisons that are in higher **loop depths** exhibit higher SDC proneness.*

Comparisons that are in higher **loop depths** have higher SDC proneness on average. This heuristic is opposite to the conclusion of prior work [33].

3.4. Heuristics of Other Factors

Figure 3.9: Average SDC proneness of global ones and others in resilient-comparisons-used variables

The reason for this difference is that the prior work focuses on Egregious Data Corruptions(EDCs), which are a subset of SDCs that cause significant deviation in the program’s output. A deeper loop structure usually implies a core computation in a program, and faults in comparisons from these deep loops are prone to corrupt a small but critical portion of program data therefore corrupt the output. However, corrupting such a small portion of program data may not lead to a large deviation in the program output, and are hence not considered important in terms of EDC detection. Figure 3.10 shows the average SDC proneness of comparisons with different depths of loop in our initial fault injection experiment.

*HO4: Variables that are **cumulatively calculated** have higher SDC proneness.*

Similar to the result of prior work [6], cumulatively calculated variables may have higher SDC proneness. This is because faults in such variables may accumulate along with program execution and thus less likely to be masked. Figure 3.11 shows the average SDC proneness for cumulative variables and

3.5. Summary

Figure 3.10: Average SDC proneness of comparisons with different depths of loop

others.

*HO5: Variables with higher **fan-out** are prone to be low in SDC proneness.*

Pattabiraman et al. [24] have found that the fan-out, or the dynamic number of uses of a variable, can be a good measure of the crash-proneness of a program variable. In our initial injection experiment, we also found that variables with high fan-out usually have lower SDC proneness when $fanout < 4$. However, when $fanout \geq 4$, we observed a higher average SDC proneness. Figure 3.12 shows the average SDC proneness for different fan-outs in our initial injection experiment.

3.5 Summary

In Chapter 2, we identified two major factors that contribute to the estimation of SDC proneness: *fault propagation* and *SDC proneness of the end points of data dependency chains*. In this chapter we formulated heuristics

Figure 3.11: Average SDC proneness for cumulative variables and others

Figure 3.12: Average SDC proneness for different fan-outs

3.5. Summary

to model both of these two factors. We first extracted instructions that have shown high fault masking or crash proneness in all the programs in our initial study, we then formulated heuristics for *fault propagation* based on the result in Section 3.1. For the *SDC proneness of the end points of data dependency chains*, we classified the end points into two groups: *store operation* and comparison operation, and formulated heuristics for both groups respectively (Section 3.2 and Section 3.3). Finally, we incorporated some other program features based on our experiments, and some heuristics proposed in prior work to complement the heuristics for the two groups.

Chapter 4

Approach

In the previous chapter, we examined various heuristics for identifying SDC-prone variables in a program. In this chapter, we first extract program features based on the heuristics to describe each store and comparison instruction (Section 4.1). We then build the *SDCTune* (Section 4.2) and *SDCAuto* models (Section 4.3) with the extracted features, to quantify the estimation of SDC proneness based on empirical data. Finally, we present our approach for choosing the SDC-prone locations subject to a maximum performance overhead using *SDCTune* and *SDCAuto* (Section 4.5), and the nature of the detectors we inserted to protect the program (Section 4.6)

4.1 Feature Extraction

The first step of building our SDC-proneness estimation model is in extracting features. As shown in Chapter 3, features are extracted according to our heuristics and also those proposed in prior work [6, 10, 24, 33]. There is a one to many mapping between heuristics and features. In other words, a heuristic may correspond to multiple features. For example, stored values of *Addr NoCmp* group will be identified using two features, namely *address computing* and *comparison used*. However, the features that are eventually

4.1. Feature Extraction

selected to be used in either *SDCTune* or *SDCAuto* are determined in the model building phase, covered in Section 4.2 and Section 4.3 respectively.

We extract features through static analysis and dynamic profiling of the programs. These features describe the stored values and comparison results through three perspectives. (1) *Execution time related features* contain features about dynamic counts of or affected by an program variable. (2) *Code structure related features* contain features about the position of an program variable in the code. (3) *Data usage related features* contain features relevant to the usage of an program variable.

Table 4.1 shows an excerpt of all the features we extracted. In total, 66 features are extracted for stored values and 67 for comparisons.

Table 4.1: Some features extracted for modeling building

Feature group	Subgroup	Feature	Description
Common features	Execution time related	inst func execution time ratio	dynamic counts of the specific instruction divided by the dynamic counts of the function it belongs to
		inst execution time ratio bymax	dynamic counts of an instruction divided by the maximum dynamic counts of all instructions
		dominated execution time ratio bywhole	dominated dynamic counts of an instruction divided by the dynamic counts of all instructions

4.1. Feature Extraction

Table 4.1: Some features extracted for modeling building

Feature group	Subgroup	Feature	Description
Common features	Execution time related	post dominated execution time ratio bymax	post dominated dynamic counts of an instruction divided by the maximum of all instructions
	Code structure related	bb length	the number of static instructions in the basic block that contains the specific instructions
		bb length ratio bymax	bb length divided by the maximum of all instructions
		post dominated loop depth ratio bymax	post-dominated loop depth of an instruction divided by the maximum of all instructions
	Data usage related	data width	the number of bits of the result of the specific instruction
		in global	whether the specific instruction changes a globally defined value
Features for stored values	Execution time related	execution time loads	the dynamic counts of the stored value being loaded
		load execution time entropy	the entropy computed based on the probabilities of a stored value being loaded by different load instructions

4.1. Feature Extraction

Table 4.1: Some features extracted for modeling building

Feature group	Subgroup	Feature	Description
Features for stored values		execution time required for addr	the dynamic counts required for computing the storing address
	Code structure related	num static loads ratio bymax	the number of static load instructions divided by the maximum of all stored values
	Data usage related	used in oef func call	whether the stored value is used in functions which have no side effect
Features for comparisons	Execution time related	decision entropy execution time	the entropy computed based on the probabilities of the comparison results
	Code structure related	is loop terminator	whether the comparison result can break a loop execution
	Data usage related	is icmp	whether the comparison is made between integers
		is fcmp	whether the comparison is made between float point values
		cmp with zero	whether the comparison is made with zero

Along with these features, we also need the SDC proneness of the stored values and comparisons as training data. We conduct fault injection experiments upon these variables to gather the SDC proneness.

4.2 Manually Tuned Model: *SDCTune*

Both our manually tuned model (*SDCTune*) and automatically tuned model (*SDCAuto*), for predicting the SDC proneness of a variable, are built from fault injections over a set of training programs, with program features extracted before which incorporate the heuristics defined in the previous section.

We start building *SDCTune* model by modelling the SDC proneness of store and comparison instructions in the program. The SDC proneness of these instructions depends on categorical features such as *resilient comparisons* and on numerical features such as *data width* (Section 3.2 and Section 3.3). We manually apply *classification* to model the categorical features, and *linear regression* to model the numerical ones. Once we determine the SDC proneness of the store and branch instructions, we use the fault propagation procedure outlined in Section 3.1 for estimating the SDC proneness of other instructions. We explain the classification and regression methods below.

Classification The goal of classification is to use the categorical features that we observed before to classify the stored values or comparison results into different groups so that we can apply the numerical features (or arithmetic means) to quantify the SDC proneness of each group. This classification is done manually according to our empirical data. For each division in the model, we first select features that can describe our heuristics, and then we adopted those features to split our current group into several sub-

groups. We recursively split these subgroups with our heuristics until all the heuristics are utilized. As shown in Sections 3.2 and 3.3, different categories of stored values and comparison results have different categorical features for determining their SDC proneness (e.g. *resilient comparison or not* for *Cmp NoAddr* stored values and *used in output or not* for *NoCmp NoAddr* ones). Therefore, we apply tree-structured classification so that different features can be used in different categories. The features are arranged hierarchically in the form of a tree, starting from a root node, and partitioning the nodes based on different features recursively until all the data in a leaf node belongs to a single category.

Regression is applied upon the leaf nodes of the classification tree to factor in the effects of numerical features such as *data width*. For example, consider a leaf node of stored values: *Addr NoCmp->Not Used in Masking Operations*. We find that the SDC proneness of stored values in this node satisfy the following equation: $\hat{P}(SDC|I) = -0.012 * data\ width + 0.878$. This expression was derived using linear regression based on the results from fault injection over a set of training programs in Section 5.1. The reason for the negative correlation in this equation is that the higher bit positions of stored values in leaf *Addr NoCmp->Not Used in Masking Operations* are very likely to cause application crash if they are corrupted. Since values with larger *data width* have a higher probability of being corrupted in higher bit positions, faults that occur in those values are less likely to cause SDCs as they are more likely to cause the program to crash. For the leaf nodes that do not exhibit a correlation with numerical features, we take the arithmetic

means as the estimation of their SDC proneness.

4.3 Automatically Tuned Model: *SDCAuto*

Unlike *SDCTune*, our automatically tuned model, *SDCAuto*, is built automatically using a machine learning approach known as the *Classification and Regression Tree (CART)* algorithm [27].

We choose this algorithm due to the following reasons:

1. The built tree model is simple to understand and to interpret. On the contrary, results from other models such as artificial neural networks, may be more difficult to interpret. The generated CART tree can help us gain a better understanding of the relation between SDC proneness and program features by picking out the features showing strong correlation with SDC proneness.
2. CART tree model, as one of decision tree model requires little data preparation [4]. Other regression models, like support vector machine (SVM) and Gaussian process, rely on an appropriate normalization method of input data, which needs delicate tuning based on the application scenarios. However in our case, the orders of magnitude may be very different for different features (e.g., execution time related features and code structure related features), and are hence very difficult to normalize. Therefore, we prefer models like *CART Tree* which require little data preparation.
3. CART tree is able to handle both numerical and categorical data.

Many of the features we extracted are categorical data, e.g., `is_global`, `is_integer` and `is_fcmp`. At the same time, other features are numerical, e.g., `loop_depth`, `dominated_execution_time` and `data_width`. Many other regression algorithms may not support a mix of categorical data and numerical data.

However, one disadvantage of CART algorithm is that the tree may grow to be biased if some classes of data dominate. In our case, the tree may be biased towards some training programs because of the large number of data points from them while ignoring other training programs. To balance the dataset between different training programs, we define *data point threshold* as a parameter to constrain the maximum number of data points allowed in the growth of the trees for stored values and comparisons. Store and comparison instructions will be ranked decreasingly according to their dynamic counts, and we incorporate the top ones as the highly executed instructions are more valuable for SDC proneness estimation (Equation 2.2). The number of instructions we incorporate from each program is limited by the *data point threshold*.

Our decision tree is built based on the Mean Squared Error (MSE) criteria, and used as regression tree to estimate the SDC proneness. The algorithm splits the training dataset recursively to divide the data points into multiple groups until the divided groups have data points fewer than a threshold value, namely *minimum size of leaves*. The end groups are known as *leaves* and the average value (i.e., SDC proneness) are assigned as the value of each leaf. For each split, the decision tree algorithm will select a

feature and splitting threshold among all possible positions to maximize the reduction of MSE which represents the information gain.

Algorithm 1 shows the pseudo code of the CART algorithm to build a decision tree. The algorithm takes a set of n data points: $\langle X_i, y_i \rangle, i = 1, \dots, n$ as the input data. In our case, the size of this set of data points are controlled by *data point threshold*. For each data point, there is a target value: y , SDC proneness in our case, and an input vector: X with in total D dimensions as each dimension represents one feature that we extract. The algorithm will first test if the current set of n data points can be split into two subsets (line 1). The precondition here is that every leaf should have at least min_{leaf} data points. If the current set of data points cannot be split, the algorithm will create a leaf node and assign it with the average y value of its data points as its estimation value (line 2-4). If the set of data points can be split, the algorithm will traverse all the D dimensions of X and all the possible splitting values of the data points to find a split that can maximize the information gain which, in our case, is represented as a minimum Mean Squared Error (MSE) (line 7-18). Once a split is done, we recursively call the routine on the two new subsets until no more divisions can be done (line 19-20). Finally, the algorithm will return the root node of the tree.

In the above algorithm, the growth of the trees is controlled by parameter: *minimum size of leaves* and we control the input data through *data point threshold*. We study the influence of them and present our results in Section 2.4. Figure 4.1 shows an example of our built decision tree for stored values with 17 points as *minimum size of leaves* and 80 instructions as *data point threshold*.

input : 1) A set of n data points: $\langle X_i, y_i \rangle, i = 1, \dots, n$;
 2) minimum size of leaves: min_{leaf}
output: A regression tree

```

1 if  $n < 2 * min_{leaf}$  then
2   No split can be made to create two leaves with more than
    $min_{leaf}$  data points;
3   Create Leaf Node and assign it the average  $y$  value of the  $n$ 
   data point;
4   Return Leaf Node;
5 end
6 else
7    $MSE_{total}$  = The MSE of  $y$  values of all  $n$  data points;
8   for dimension  $d$  in all dimensions of  $X$  do
9      $\langle x[d]_i \rangle, i = p, \dots, q$  = all possible variables in
      $\langle x[d]_i, i = 1, \dots, n \rangle$  that can split  $n$  points into two
     groups with more than  $min_{leaf}$  points in each;
10    for variable  $v$  in all possible variables  $\langle x[d]_i \rangle, i = p, \dots, q$ 
    do
11       $left\ group, right\ group$  = Split at dimension:  $d$  with
      value:  $v$ ;
12       $new\ MSE_{total} = MSE_{left\ group} \times \frac{Num_{left\ group}}{n} +$ 
       $MSE_{right\ group} \times \frac{Num_{right\ group}}{n}$ ;
13      if  $new\ MSE_{total} < MSE_{total}$  then
14         $MSE_{total} = new\ MSE_{total}$ ;
15        cache split( $d, v$ );
16      end
17    end
18  end
19   $left\ leaf, right\ leaf$  = split data points  $\langle X_i, y_i \rangle$  with last
  cached split( $d_{last}, v_{last}$ );
20  Call recursively upon created  $left\ leaf, right\ leaf$ ;
21 end
22 Return Root Node;
```

Algorithm 1: CART Algorithm to Build a Decision Tree

4.4. Model Usage

Figure 4.1: Auto tuned decision tree for stored values

4.4 Model Usage

Once the trees are built from training dataset, we can utilize it to estimate the SDC proneness of the stored values and comparison results of the testing programs. The estimated SDC proneness of those end points of data dependency chains will be back propagated along their backward slice to derive the SDC proneness of each instruction with fault masking or crashing rate considered (section 2.4). Then the SDC proneness of each instruction will be used to calculate the importance of the instruction and guide the selection of instructions to duplicate and check under a specific overhead bound as described in Section 5.3.

4.5 Choosing the Instructions

As shown in Section 2.4, we can calculate the *SDC coverage* of protecting an instruction if we know the *SDC proneness* of that instruction using Equation 2.1 in Section 2.1. We apply either *SDCTune* model or *SDCAuto* model to estimate the SDC proneness of each instruction in the program that we want to protect. We also obtain the dynamic count of each instruction in the program by profiling it with representative inputs. We then attempt to choose instructions to maximize the SDC coverage subject to a given performance overhead (Section 2.4), using a standard dynamic programming algorithm [22].

4.6 Detector Design

Once we identify a set of instruction to protect, the next step is to insert error detectors for instructions. Our detectors are based on duplicating the backward slices of the instructions to protect, similar to prior work [10]. We insert a check immediately after the instructions to be protected, which compares the original value computed by the instruction with the value computed by the duplicated instructions. Any difference in these values is deemed to be an error detection and the program is stopped. Figure 4.2b shows a conceptual example of our detector for a given set of instructions to be protected in Figure 4.2a.

Note that we assume that there is a single transient fault in the program (Section 2.2), and hence it is not possible for both the detector and the chosen instruction to be erroneous. Therefore, any error in the computation

Figure 4.2: Example of inserted detectors and concatenating instructions

performed by the chosen instruction will be detected by the corresponding error detector.

A naive implementation of our detectors can result in prohibitive performance overhead. Therefore, we develop two optimizations to lower the detector overhead. First, we *concatenate* adjacent duplicated pieces of code by adding the instructions between them to the protection set so that we can combine their detectors. Figure 4.2c shows how this optimization works. This optimization provides benefits when the cost of the saved detector is higher than the cost due to the added instructions. Second, we perform *lazy checking*, in which detectors for cumulative computations in loops are moved out of the loop bodies, as the example in Figure 4.3 illustrates. This optimization is effective for long running loops.

4.6. Detector Design

```
1 for(i=0;; i++){
2     //loop body
3     flag = i<n?1:0;
4     if(flag == 1) break; //decompose exit predication
                           to simulate instruction-level behaviour.
5 }
```

(a) Detector-free code

```
1 i=0;
2 dup_i=0; //duplication of i
3 for(;;) {
4     //loop body
5     flag = i<n?1:0;
6     dup_flag = dup_i<n?1:0;
7     if(flag != dup_flag)
8         Assert(); //inconsistent
9     if(flag == 1) break;
10 }
```

(b) Basic detector instrumented. This shows how the loop index i in original code (a) is protected with bold code as check.

```
1 i=0;
2 dup_i=0; //duplication of i
3 for(;;) {
4     //loop body
5     flag = i<n?1:0;
6     dup_flag = dup_i<n?1:0;
7     if(flag == 1) break;
8 }
9 if(flag != dup_flag)
10 Assert();
11 //inconsistent
```

(c) *Lazy checking* applied. This shows how we move the check out of the loop body

Figure 4.3: Example of inserted detectors and lazy checking

4.7 Summary

This chapter described the approach of building our configurable SDC detection technique based on the heuristics from Chapter 3. Section 4.1 presented the extracted features for building both the manually tuned model: *SDCTune* and automatically tuned model: *SDCAuto*. Section 4.2 presented the building of *SDCTune* and Section 4.3 presented the building of *SDCAuto* with the CART algorithm. Section 4.4 presented the usage of the two models to estimate SDC proneness, and Section 4.5 presented how to utilize the estimated SDC proneness to guide detector placement. The last section presented our SDC detector and two optimizations to reduce its overhead. In the next chapter, we will evaluate the models we built with the algorithms and use them to guide the placement of error detectors in the application.

Chapter 5

Experimental Setup

In this chapter, we present our experimental setup to evaluate both the *SDCTune* and *SDCAuto* models for configurable SDC protection. All the experiments and evaluations are conducted on a Intel i7 4-core machine with 8GB memory running Debian Linux. Section 5.1 presents the details of benchmarks and Section 5.2 presents our evaluation metrics. Section 5.3 presents our methodology and workflow for performing the experiments.

5.1 Benchmarks

We choose a total of twelve applications from a wide variety of domains for training and testing both of our models. The applications are drawn from the SPEC [13], SPLASH2 [36], NAS parallel [1], PARSEC [2] and Parboil [32] benchmark suites. We randomly divide the twelve applications into two groups, one group for training and the other for testing. The four benchmarks used in Section 2.3 to derive the heuristics are drawn from the training group. The details of these training and testing benchmarks are shown in Table 5.1 and Table 5.2 respectively. All the applications are compiled and linked into native executables with -O2 optimization flags and run in a single threaded mode, as our current implementation of both *SDCTune* and *SDCAuto* models

5.2. Evaluation Method

Table 5.1: Training programs

Program	Description	Benchmark suite	Input	Stores	Comparisons
IS	Integer sorting	NAS	default	21	20
LU	Linear algebra	SPLASH2	test	41	110
Bzip2	Compression	SPEC	test	681	646
Swaptions	Price portfolio of swaptions	PARSEC	Sim-large	36	101
Water	Molecular dynamics	SPLASH2	test	187	224
Lbm	Fluid dynamics	Parboil	short	71	34

work only with single-threaded programs.

5.2 Evaluation Method

We evaluate our SDC proneness estimation model from three perspectives. First we evaluate the regression results of CART algorithm with different parameters, the optimal parameters will be selected for *SDCAuto*. We then use both model for estimating the overall SDC rate of applications, as well as the SDC coverage(s) for different performance overhead bounds. The estimation of overall SDC rates are used for comparing the SDC rates of different applications free from fault injection, while the coverage(s) show the capability of configurable protection of our technique. We use the same experimental setup for fault injection as that described in Section 2.3.

5.2. Evaluation Method

Table 5.2: Testing programs

Program	Description	Benchmark suite	Input	Stores	Comparisons
Gzip	Compression	SPEC	test	251	399
Ocean	Large-scale ocean movements	SPLASH2	test	322	813
CG	Conjugate gradient	NAS	default	32	97
Bfs	Breadth-First search	Parboil	1M	36	57
Mcf	Combinatorial optimization	SPEC	test	87	158
Libquantum	Quantum computing	SPEC	test	39	136

Regression results from decision tree model To evaluate the regression results, we calculate the average squared errors for both training and testing dataset. As shown in Section 4.3, there are two parameters controlling the tree building process: (1)*minimum size of leaves* and (2)*data point threshold*.

We explore this two-dimensional parameter space, we vary the *minimum size of leaves* from 1 to 120 points per leaf, and the *data point threshold* from 10 to 120 data points for each program. These values were chosen based on our empirical measurements of the numbers of stores and comparison instructions in the program. We explore this two-dimensional parameter space and calculate the mean squared errors for both training and testing dataset for each point in our exploration space. We then present the two

optimal pairs of parameters for stored value decision tree and comparison decision tree respectively. We also present the features that are adopted by the optimal trees and compare them with the manually selected features in *SDCTune* model.

Estimation of overall SDC rates: We perform a random fault injection experiment to determine the overall SDC rate of the application. We then compare the SDC rate estimated by both of our models with that obtained from the fault injection experiment. We also estimate the correlation between our estimated SDC rates and SDC rates from fault injection. A high positive correlation implies the usefulness of our models in comparing SDC rates among different applications.

SDC coverages for different performance overhead bounds: The SDC coverage is defined as the fraction of SDC causing errors detected by our detectors. We apply both *SDCTune* model and *SDCAuto* model to predict the SDC coverage for different instructions to satisfy the performance overhead bounds provided by the user. Our selection algorithm(Section 4.5) starts with the instructions providing the highest coverage, and iteratively expands the set of instructions until the performance overhead bounds are met. We then perform fault injection experiments on the program instrumented with our detectors for these instructions, and measure the percentage(s) of SDCs detected. We also compare our results with those of full duplication, i.e., when every instruction is duplicated in the program, and that of hot-path duplication, i.e., when the top 10% most executed instruc-

tions are duplicated in the program.

To ensure a fair comparison among these techniques, we use a metric called the ***SDC detection efficiency***, which is similar to the efficiency defined in prior work by Shafique et al. [30]. We define the SDC detection efficiency as the ratio between SDC coverage and performance overhead for a detection technique. We calculate the SDC detection efficiency of each benchmark under a given performance overhead bound, and compare it with the corresponding efficiencies of full duplication and hot-path duplication. The SDC coverage of full duplication is assumed to be a hundred percent [28].

5.3 Work Flow and Implementation

Measuring regression results of decision trees Figure 5.1 shows the workflow for selecting parameters and measuring the regression results of the decision trees which are parts of *SDCAuto* model. The workflow explores the parameter space which is consist of *minimum size of leaves* and *data point threshold* to test their influences on the regression results.

We first compile the application using LLVM into its IR form. We then extract the features that *SDCAuto* needs to estimate the SDC proneness of stored values and comparison results. This is done using an automated compiler pass we wrote in LLVM, and the LAMPView tool [23] for analyzing load/store dependencies. We also need initial SDC proneness data for each stored value and comparison instructions to build our decision tree model. This is obtained by fault injections. However, the fault injections are done for building *SDCAuto*; using the built model does not require fault injection.

Figure 5.1: The workflow of building regression trees and exploring the parameter space for *SDCAuto*.

Once the training data and testing data are obtained, we build regression trees for stored values and comparison instructions with *minimum size of leaves* iterating from 1 to 120 and *data point threshold* iterating from 10 to 120. For each combination of *minimum size of leaves* and *data point threshold*, we calculate MSE for both training data and testing data to show the influences of the two parameters. The values of *minimum size of leaves* and *data point threshold* with minimum MSE of testing data will be selected as the optimal parameters of the regression trees.

Measuring overall SDC estimation and coverage Figure 5.2 shows the workflow for estimating the overall SDC rates and providing configurable protection using either *SDCTune* model or *SDCAuto* model. The workflow requires the following inputs from the user: (1) source code for the program,

5.3. Work Flow and Implementation

Figure 5.2: The workflow of applying our models for (1) *estimate the overall SDC failure rate* and (2) *selectively protect the SDC-prone variables subject to a performance overhead*.

(2) a set of representative input(s) for executing the application, and (3) output function calls that generate the output data that we care about in terms of SDC failures (as mentioned before, not all output data in an application is important from the perspective of SDCs, for example, statistical or timing information in the output). In addition, it requires the user to specify the maximum allowable performance overhead that may be incurred by the detectors inserted by our technique.

Similar to how we did the initial study, we first compile the source code and extract features from the compiled IR. Then, we run the extracted features through either the *SDCTune* or the *SDCAuto* model built in Chapter 4, to generate an estimated SDC proneness for each instruction. We then use the results from our model to estimate the overall SDC rate of the application, and for inserting detectors into the program for protecting the most SDC-prone instructions within the given overhead bound. The detectors are automatically inserted into the program by another LLVM pass we wrote.

We use the representative inputs provided by the user to execute the pro-

gram for obtaining its execution time with the detectors. The above process of choosing instructions to protect is repeated iteratively until the designated performance overhead bound is fulfilled. If we exceed the performance overhead bound, we backtrack and remove the most recently inserted detectors. Finally, we use the program fortified with the detectors to measure its performance overhead and fault coverage.

5.4 Summary

In this chapter, we described the experimental setup for evaluating our *SDCTune* and *SDCAuto* models. Section 5.1 presented the benchmark programs for evaluating our models and described the characteristics of the programs. Section 5.2 presented the of evaluation methods and experiment design. Section 5.3 presented the workflow and implementation details of evaluating our technique. In the next chapter, we will present the results of our evaluation.

Chapter 6

Results

This chapter presents the results of our experiments to: (1) explore the parameter space for our decision tree model for *SDCAuto*, (2) estimate the overall SDC rate of an application with both *SDCTune* and *SDCAuto* model and (3) apply configurable protection to maximize detection coverage under different performance overhead bound. We first present the results of the times taken by both the *SDCTune* and *SDCAuto* models.

6.1 Time Taken by Models

In our experiments, both *SDCTune* and *SDCAuto* models require five to fifty minutes (average of 24 minutes) depending on the application, to estimate the overall SDC rate and to generate a fortified executable protected with detectors for a given performance overhead. As shown in Table 6.1, most of the time are spent on *Feature extraction* and *Instruction selection*, which require one to forty five minutes (average 10.08 minutes) and five seconds to forty nine minutes (average 14.34 minutes), respectively.

The time taken by the *Feature Extraction* phase is spent in profiling the program and recording the store-load dependencies with LAMPView [23]. Therefore, programs with longer execution time and more dynamic counts

6.1. Time Taken by Models

Table 6.1: Time consumption of *SDCTune* and *SDCAuto*

Group	Benchmark	Feature extraction (minutes)	Instruction Selection (minutes)	Total (minutes)
Training	Lbm	23	9	32
	IS	9	1	10
	LU	3	0.083	3.083
	Bzip2	9	49	58
	Water	3.5	16	19.5
	Swaptions	9	9	18
Testing	Gzip	8	27	35
	CG	45	3	48
	Ocean	6	35	41
	Bfs	1	4	5
	Mcf	2	9	11
	Libquantum	2.5	10	4.167
Mean		10.083	14.340	23.729

for memory operations requires much longer time for profiling than other programs. In this study, *Lbm* and *CG* require longest time on feature extraction, and they also require a obviously longer time for a normal run and have higher dynamic counts of store/load operations.

We found that the time taken by *Instruction Selection* phase is mainly affected by the number of stores and comparisons of a program. This is shown in the number of stores and comparisons in Table 5.1, Table 5.2 and time spent on *Instruction Selection* in Table 6.1. The reason that programs with many stores and comparisons usually have a large number of static instructions. And more static instructions will polynomially increase the

search space of finding an optimal set of instructions to protect in our technique [22].

We also found that merely applying our models to estimate SDC proneness causes nearly no time consumption. This matches our intuition as using the tree to classify the store/comparison instructions and then back propagating the SDC proneness has $O(n)$ time complexity, where n represents the number of instructions. Even several tens of thousands static instructions can be processed in just tens of milliseconds on today’s desktop computers.

On the contrary, fault injection alone requires anywhere from a few hours to a few days to generate the SDC rates for each application. Further, estimating the SDC-prone locations in a program using fault injection requires even more fault injections and significant effort to map the results of the fault injection back to the program’s code, which is necessary for placing detectors.

6.2 Effect of Decision Tree Parameters

We explored the parameter spaces for *stored value* decision tree and *comparison instruction* decision tree. Figure 6.1 shows the mean squared errors (MSE) for the decision trees under different *minimum size of leaves* and *data point threshold* for training and testing dataset. Recall that our goal is to choose parameters for the models to minimize the MSE.

From Figure 6.1, we can observe that overfitting occurs (as expected) when *minimum size of leaves* is too small, while incomplete learning occurs when it is too large. At the same time, a large *data point threshold* may

introduce imbalance in the training dataset and worsen the regression result, as shown in Figure 6.1d, while too small a value can hinder the tree splitting process then decrease the accuracy, as shown in Figure 6.1c and Figure 6.1a.

As shown in Figure 6.1c and Figure 6.1d, we found *minimum size of leaves* = 17, *data point threshold* = 80 has lowest MSE for testing stored values, and so is selected as optimal for tree of stored values. Similarly, *minimum size of leaves* = 57, *data point threshold* = 40 has lowest MSE for comparison instructions, and so is selected for tree of comparisons. Based on this configuration, we rank the features according to their importance in Table 6.2. We further discuss these features in Section 7.2.

6.3 Estimation of Overall SDC Rates

We estimate the overall SDC rates of the applications using *SDCTune* model and *SDCAuto* model, then compare them with the SDC rates obtained through 3000 random fault injections per benchmark. Table 6.3 shows the overall SDC rates ($P(SDC)$) from the fault injections and the estimated overall SDC rates ($\hat{P}(SDC)$) for both training programs and testing programs. The SDC rates are statistically significant with an error bar ranging from 1.78%(Lbm) to 0.71%(Swaptions), at the 95% confidence intervals.

From Table 6.3, it can be observed that the absolute values of the estimated SDC rates do not match with the observed ones accurately. However, the results show high positive correlation between the SDC ranks estimated by our models and those observed in reality, where rank represents the relative SDC rate. Figure 6.2 plots the estimated SDC ranks versus the observed

6.3. Estimation of Overall SDC Rates

Figure 6.1: Effect of *data point threshold*(y-axis) and *minimum size of leaves*(x-axis) on regression results

6.3. Estimation of Overall SDC Rates

Table 6.2: Features adopted by the optimal decision trees for *SDCAuto* model

Tree	Features used by decision trees	Importance
Stored values	inst func execution time ratio	0.4828
	bb length	0.1507
	data width	0.1501
	post dominated loop depth ratio bymax	0.0706
	post dominated execution time ratio bymax	0.0286
	in global	0.0269
	load execution time entropy	0.0257
	num static loads ratio bymax	0.0219
	bb length ratio bymax	0.0185
	dominated execution time ratio bywhole	0.0119
	execution time required inst foraddr	0.0075
	used in oef func call	0.0045
	execution time loads	0.0003
Comparisons	inst func execution time ratio	0.6058
	inst execution time ratio bymax	0.3942

ranks for both the *SDCTune* and *SDCAuto* models. The x-axis shows the overall SDC rates from 3000 random fault injections, while the y-axis shows the estimated overall SDC rates using either *SDCTune* or *SDCAuto*. The correlation coefficient is 0.8770 for our *SDCTune* model, and 0.8545 for *SDCAuto* model, showing a strong positive correlation for both models.

Thus, our models are highly accurate in predicting the SDC rates of applications relative to others. However, it is not accurate at predicting the absolute rates of SDCs. There are two reasons for this inaccuracy. First, our estimation of SDC rates using fault propagation is conservative, and some-

Figure 6.2: The correlation of overall SDC rates for all programs.

times may overestimate the SDC proneness of variables in the presence of application-specific masking. Second, our load-store dependence analysis is performed using the LAMPView tool, which does not handle some library functions such as *memcpy*. This is also the reason for the large deviation of the SDC rate of CG when using *SDCTune* model. A large portion of the output of CG comes from *memcpy* and *memset*. This prevents LAMPView to trace the data flow so that cause large deviations in estimating the SDC proneness for instructions. This inaccuracy in absolute SDC rate prediction may lead to inadequate protection, and additional overhead. However, our results show that despite the inaccuracy, our models can guide detector placement to obtain high coverage at low performance overheads. This is because detector placement mainly relies on estimating the SDC proneness of an instruction compared with other instructions, or the relative SDC proneness. Even though the absolute SDC proneness is not estimated accurately, our selection algorithm is still able to choose correct instructions if the rel-

6.4. SDC Coverage and Detection Efficiency

Table 6.3: The SDC rates and ranks from fault injections and our models

Group	Benchmark	$P(SDC)$ from injections	$\hat{P}(SDC)$ from <i>SDCTune</i>	$\hat{P}(SDC)$ from <i>SDCAuto</i>
Training	Lbm	52.53%	48.11%	48.89%
	IS	43.46%	33.75%	26.57%
	LU	31.9%	25.43%	22.36%
	Bzip2	24.47%	17.88%	19.78%
	Water	5.9%	9.75%	18.85%
	Swaptions	4.1%	11.46%	11.74%
Testing	Gzip	33.67%	32.46%	26.88%
	CG	23.67%	3.75%	24.58%
	Ocean	20.6%	14.75%	16.8%
	Bfs	17.37%	14.27%	17.19%
	Mcf	15.76%	17.84%	15.89%
	Libquantum	10.5%	10.9%	18.64%

ative SDC proneness is correctly estimated by the models. However, this is not sufficient for the estimation of overall SDC rates, as the estimated overall SDC rates are actually a sum of absolute SDC proneness of all instructions.

6.4 SDC Coverage and Detection Efficiency

We use both of the models for inserting error detectors into the applications to maximize SDC coverage under a given performance overhead. Figure 6.4a shows the SDC coverage obtained by *SDCTune* model for each benchmark under three different performance overhead bounds: 10%, 20% and 30%. For the training programs, the geometric means of the SDC coverage for the 10%, 20% and 30% overhead bounds are 34.8%, 71.1% and

78.9%, respectively. For the testing programs, the corresponding geometric means are 37.0%, 58.4% and 74.8% respectively, which are somewhat lower than the training programs' averages, as expected. We also measured the SDC coverage obtained with hot-path duplication, and found it to be 74.28% and 92.33% on average for training and testing programs respectively.

Figure 6.5a shows the SDC coverage obtained by *SDCAuto* model. The geometric means of the SDC coverage are 31.14%, 66.32% and 76.03% respectively for the training programs. For the testing programs, the geometric means are 27.37%, 45.70% and 67.63% respectively.

Figure 6.3 shows the performance overhead of full duplication and hot-path duplication. The overhead of full duplication is 50.16% on average for the training programs, while it is 71.37% on average for the testing programs. Hot-path duplication has an overhead of 33.19% for the training programs, and 61.76% for the testing programs. Note that both of these are considerably higher than the 30% overhead bound we considered with our detectors.

We also calculate the detection efficiency of the detectors we inserted, and for hot-path duplication based on their overhead and SDC coverages (Section 5.2). Figure 6.4b and Figure 6.5b show the SDC detection efficiency for our detectors with the three overhead bounds, and for hot-path duplication. The efficiencies are normalized to that of full duplication, which has a baseline efficiency of 1. A value close to 1 means that no improvement is achieved over full duplication.

With *SDCTune* model, we observe SDC detection efficiencies of 1.75x, 1.78x and 1.32x for the training programs, and 2.65x, 2.09x and 1.78x for the

testing programs, at the 10%, 20% and 30% performance overhead bounds respectively. We have higher detection efficiencies for our testing programs because the full duplication overheads of the testing programs are commonly higher than the training programs. This results into a lower baseline for the testing programs in terms of detection efficiencies. More details are provided in Chapter 7. The reason that the efficiencies generally decrease as overhead increase is that some of the instructions protected at higher overhead are not as SDC prone. As the performance overhead of the detectors approaches that of full duplication, the detection efficiencies will drop to 1.

Detectors inserted using *SDCAuto* model have detection efficiencies of 1.56x, 1.67x and 1.27x over full duplication for the training programs, and 1.96x, 1.64x and 1.62x over full duplication for the testing programs. Thus, we find that there is considerable variation in detector efficiency among benchmarks and between *SDCTune* and *SDCAuto* model. We explain the reasons in the next chapter.

We also observe no gain in efficiency with hot-path duplication compared to full duplication in spite of its high coverage, as it incurs correspondingly higher overhead (as mentioned in Section 2.4). In summary, our technique significantly outperforms both full-duplication and hot-path duplication in providing better detection efficiency, for much lower performance overhead bounds.

Figure 6.3: The overhead of full duplication and hot-path duplication

6.5 Summary

In this chapter, we presented the results of the evaluation of the models. We found that *SDCTune* and *SDCAuto* are both accurate in predicting the SDC rates of applications relative to other applications. However, both the two models did not predict the absolute SDC rates accurately for some applications, as shown in Table 6.3 (Section 6.3). When applying our models for guiding detector placement, we found that our technique improved detection efficiency of full duplication by a factor of 0.78x to 1.65x, compared with full duplication and hot-path duplication, for the *SDCTune* model, and 0.62x to 0.96x for the *SDCAuto* model. This means that our technique provides more efficient SDC detection for much lower performance overhead bounds compared with full duplication like approaches.

6.5. Summary

(a) The SDC coverages with error bars at the 95% confidence interval for *SDCTune* model. The error bars are less than 2%, and obtained from 3000 random fault injections per benchmark. The SDC coverage of full duplication is considered as 100%

(b) The normalized detection efficiency of *SDCTune* model. Full duplication is the baseline and has detection efficiency = 1. (Detection efficiency is the ratio of SDC coverage and performance overhead)

Figure 6.4: The results of *SDCTune* model for different performance overhead bounds, hot-path duplication and full duplication.

6.5. Summary

(a) The SDC coverages with error bars at the 95% confidence interval for *SDCAuto* model. The error bars are less than 2%, and obtained from 3000 random fault injections per benchmark. The SDC coverage of full duplication is considered as 100%

(b) The normalized detection efficiency of *SDCAuto* model. Full duplication is the baseline and has detection efficiency = 1. (Detection efficiency is the ratio of SDC coverage and performance overhead)

Figure 6.5: The results of *SDCAuto* model for different performance overhead bounds, hot-path duplication and full duplication.

Chapter 7

Discussion

In this chapter, we discuss the reasons of the variation in detection coverage and efficiency among benchmarks and also the differences between the two models. We first present the differences between benchmarks when using *SDCTune* model (Section 7.1). Then we discuss the reasons of different results between *SDCTune* model and *SDCAuto* (Section 7.2). Finally we discuss the threats to the validity of our approach (Section 7.3).

7.1 Differences between Benchmarks

There are two main reasons for the differences of the detection efficiency.

First, for our technique to be efficient, it needs to protect instructions with high SDC proneness, but with low dynamic execution count. We observed that applications which have such instructions experience moderate SDC rates, which are neither too high nor too low. From Table 6.3, programs such as *Libquantum*, *Bfs*, *Mcf*, *Bzip2*, and *Ocean* fall into this category. Generally, these programs benefit the most from *SDCTune* model (Figure 6.4b). But the detectors inserted in *Mcf* and *Ocean* have higher overhead so that the SDC coverage of these two benchmarks are lower in general under same performance overhead bound. For *Mcf* the reason is that it has a large

amount of comparison operations for branches at runtime so that much more check instructions are needed to be inserted (Section 4.6) which cost more performance overhead budget. For *Ocean*, many of its dynamic instances are float point operations which means duplicating these instructions may cause higher overhead because processors usually have very limited ALU resources for float point operations. Higher overhead for detectors prevents our technique from protecting larger amount of instructions so that these two benchmarks have relatively low SDC coverage. However, The high detection efficiencies of these two benchmarks also benefit a lot with our technique.

On the other hand, if the benchmark has highly SDC prone instructions that are also highly executed, our technique does not do as well since the overhead limit prevents our technique from selecting those SDC prone instructions. Examples of these programs are *Lbm*, and *IS*.

The second reason for the variation in efficiency among benchmarks relative to full duplication, is that the overhead of full duplication is not uniform, as shown in Figure 6.3. This is because of benchmark-specific reasons such as the distribution of integer and floating point operations. In general, processors have abundant integer computation units but not as many floating point units, so the higher the fraction of floating point operations, the higher is the overhead due to duplication. We found that for some benchmarks such as *IS*, *Bfs*, and *Bzip2*, the full duplication overhead is only about 40%. This means that the detection efficiency improvement over full duplication is unlikely to be very high for these benchmarks. For example, even though *IS*, *Bfs* and *Swaptions* have reasonable SDC coverage, their detection efficiency is not very high. In one of the benchmarks, *Lbm*, our detectors have a lower

detection efficiency compared to full duplication. This is because nearly all SDC prone instructions in the program have high execution counts, and hence the performance overhead bounds cannot be satisfied if they are selected for protection. Therefore, this benchmark has low SDC coverage with our technique.

7.2 Differences between Models

Comparing the coverage provided by the detectors of *SDCTune* model and the *SDCAuto* model, the latter performs much worse on three programs, namely *Bzip2*, *Gzip*, *Ocean* and *Mcf*. The reason is the regression tree built for comparison operations. As shown in Table 6.2, only two features are utilized in the optimal regression tree by *SDCAuto* which turns out to have only three leaves. Such a result means an incomplete learning and failure in classifying the extracted features correctly.

This is because our CART model cannot utilize multiple features for one split, while categorizing comparison instructions usually requires to do so. For example, *is_loop_terminator* and *nest_loop_depth* should be considered at the same time, as *nest_loop_depth* show strong correlation for loop terminating comparisons, which are labeled with *is_loop_terminator* = True. However, CART may not discover this, as splitting on the individual features, *is_loop_terminator* or *nest_loop_depth* along does not reduce the total MSE. So the *SDCAuto* is not likely to select these features and use them correctly, and hence fails to build an optimal tree for comparison instructions.

In addition, as shown in Figure 4.1 and Table 6.2, the regression tree for stored values also failed in categorizing the training data according to the four major usage groups (Section 3.2). However, in contrast to usage-based classification, the decision tree of stored values selects many *code structure related* (e.g. `bb_length`, `post_dominated_loop_depth_ratio_bymax`) and *common execution time related* (e.g. `inst_func_execution_time_ratio`, `dominated_execution_time`) features in the model building phase. This means that from the decision tree algorithm’s perspective, these easy-to-extract features also have strong correlations with SDC proneness and are worthy of further study for SDC proneness estimation.

In short, our *SDCTune* model illustrates an upper bound on the SDC detection capability of a model, while our *SDCAuto* model presents an example of applying an existing machine learning algorithm to build such a model. Although our results show that *SDCTune* outperforms *SDCAuto* in both estimating overall SDC rates and guiding detector placement, we note that the gap is mainly caused by the limitation of the CART algorithm. A different automatically tuned model with a more appropriate algorithm and more training data may be able to match the manually tuned model. This is a subject of future investigation.

7.3 Threats to Validity

Internal Threats First, the heuristics presented in Chapter 3 are formulated based on our observation of the fault injection experiments in our initial study (Chapter 2) and we can not guarantee that these heuristics can be al-

ways held for all programs. Second, we cannot guarantee that the CART algorithm is the best for building our automatically tuned model of estimating SDC proneness. Other machine learning algorithms such as neural network and hidden Markov models may work better. Third, our *SDCAuto* model is not built with K-fold cross validation. This is because the parameter data point threshold limits the total number of data points available in our training set so that we do not have enough data points for K-fold cross validation.

External Threats A major external threat comes from the training programs and testing programs of our study. In this thesis, we randomly selected six training programs and six testing programs from standard benchmark suites. However, there may not be sufficient for training and testing a solid predictor. We partially mitigate this threat by choosing programs from a variety of standard benchmark suites rather than confining ourselves to a single suite. Another external threat is that both of our models are architecture and operating system dependent, which means the models may only work for programs that run on a specific operating system with a specific architecture. This is because in our study, the consequences of fault injection experiments depend on the underlying architecture and operating system. Shifting to another architecture or operating system may vary the results of the faults so that our models may not be able to predict them successfully. We partially mitigate this threat by using an x86 platform running the Linux operating system, which represent common choices running in many commodity platforms.

Chapter 8

Related Work

In this section, we present prior work related to the detection of silent data corruption with software technique. We classify related work into three categories, namely (1) duplication based techniques, (2) invariant based techniques, and (3) application or algorithm specific techniques.

8.1 Duplication Based Techniques

One of the earliest papers on identifying critical variables in programs, and selectively protecting them is by Pattabiraman et al. [24]. Unlike our work, they focus mostly on crash-causing errors, which are relatively easy to detect compared to SDCs. Further, they do not provide configurable protection in their work.

SWIFT [28] is a compiler based technique that uses full duplication to detect faults in program data. However, full duplication can have significant performance overhead, especially on embedded systems which do not have an abundant idle resources to mask the overhead of duplication. As shown in Figure 6.4b and Figure 6.5b, *SDCTune* and *SDCAuto* outperforms full duplication in terms of SDC detection efficiency, and also enables configurability to protect programs from SDC causing errors under various given

performance overheads.

Feng et al. [10], and Khudia et al. [16] have attempted to reduce the overhead of full duplication by only duplicating "high-value" instructions (and variables), where a fault is unlikely to be detected by other techniques and hence lead to SDCs. Unlike our work however, they do not provide a mechanism to configure the protection for a given performance overhead bound. This is especially important for embedded systems where the system has to satisfy strict performance constraints.

Another branch of work [6, 8, 19, 20, 33] has focused on protecting soft-computing applications from soft errors, by duplicating only critical instructions or data in the program. Examples of soft-computing applications are those used in media processing and machine learning, which can tolerate a certain amount of errors in their outputs. These papers exploit the resilience of soft computing applications to come up with targeted protection mechanisms. However, they cannot be applied in general purpose applications.

Thomas et al. [33] propose a technique to protect soft-computing applications from Egregious Data Corruptions (EDCs), which are errors that cause unacceptable deviations in the program's output. Similar to our work, they formulate program-level heuristics to identify EDC prone data in the program. However, there are two main differences between their work and ours. First, the heuristics they propose are based on how much program data is affected by an error. While this is important for EDC-causing errors, this is not so for SDC-causing errors as even a small deviation in the output can be an SDC. Therefore, we need a more complex set of heuristics to predict SDC prone data in a program. Secondly, EDCs constitute only 2 to 10%

of a program’s faulty outcomes. In comparison, SDCs constitute up to 50% of a program’s faulty outcomes, and hence need much more heavyweight protection.

Finally, in recent work, Shafique et al. [30] propose a technique for exploiting fault masking in applications to provide efficient detection. Similar to our work, they rank the vulnerability of instructions in the program, and allow the user to specify performance overhead bounds to selectively choose instructions to protect. However, our work differs from theirs in two ways. First, they consider all failures as equally bad, including crashes and hangs. However, we focus exclusively on SDC-causing faults, which are the most insidious of faults. Therefore, we can achieve higher efficiency for protecting against SDC-causing faults. Secondly, their work employs three metrics to determine the instructions to protect, all of which are estimated by performing a static analysis of the application’s control and data flow graph, which is conservative by nature. In contrast, our work uses empirical data to build the model for estimating the SDC proneness of different instructions, and is hence relatively less conservative. Since Shafique et al. do not provide a breakdown of their coverage among SDC failures, crashes and hangs, we cannot quantitatively compare the coverage of *SDCTune* and *SDCAuto* with their technique.

8.2 Invariant Based Techniques

Invariant-based techniques [9, 25, 29] detect errors by extracting likely invariants in programs through runtime profiling and dependency analysis.

Those likely invariants are used as assertions to check abnormal behaviours or data out-of-bounds to detect errors. Invariant based techniques typically have lower overhead than duplication-based techniques, as the assertions consist of much fewer instructions than the entire backward slice of the variables. However, an important limitation of this class of techniques is that they incur false positives, i.e., they can detect an error even when none occurs. This is because they all learn invariants from *testing* inputs, and these invariants may not hold when the program is running with real inputs in production. While our work also learns the model for SDC proneness based on training applications, it uses static analysis to actually derive the detectors from the backward slices, and has no false positives as static analysis is conservative.

8.3 Application or Algorithm Specific Techniques

Hari et al. [11] proposes a set of detectors for detecting SDCs using program-level detectors. Similar to our work, they also come up with a method to choose variables to protect for maximizing the SDC coverage under a given performance overhead bound. However, there are two differences between our work and theirs. First, they require fault injections to find the highly SDC prone variables in the program, which is time consuming. Although they reduce the fault injection space using their Relyzer technique [12], they still need to perform tens of thousands of injections. In contrast, we use our model to determine the SDC prone locations without needing fault-injections. Secondly, their detector derivation is done manually

based on understanding of the program. Further, some of their detectors are application-specific and cannot be generalized across programs, as they rely on specific algorithmic properties. In contrast, we use generic duplication-based detectors which are automatically derived for any application.

Sloan et al. [26, 31] propose an algorithm specific approach to enhance the fault detection for sparse linear algebra applications. Algorithmic solutions can achieve high coverage while keeping the performance overhead low. However, they are not general solutions and cannot be easily applied to other application types.

Chapter 9

Conclusion and Future Work

As hardware errors increase with technology scaling, Silent Data Corruptions (SDCs) are becoming more serious for a wide class of systems. Generic solutions such as full duplication incur high performance overhead as they do not prioritize protecting against SDC-causing errors.

This paper proposes a configurable protection technique for SDC-causing errors that allows users to trade-off performance for reliability. We develop heuristics for estimating the SDC proneness of instructions and build a manually tuned model, *SDCTune*, and a auto tuned model, *SDCAuto*, based on the heuristics and decision tree algorithm. We then use our models to guide the selection of instructions to be protected with error detectors under a given performance overhead. Our results show that our models are highly accurate at predicting the relative SDC rates of applications. The detectors inserted using *SDCTune* model outperform both full duplication and hot-path duplication by a factor of 0.78x to 1.65x in detection efficiency. And with *SDCAuto* model, our detectors outperform full duplication by a factor of 0.62x to 0.96x.

We plan to explore two directions as future work. First, while *SDCTune* and *SDCAuto* have high accuracy in predicting the relative SDC rate of

an application, they are not as accurate in predicting the absolute SDC rates. We plan to work on improving their absolute accuracy. Second, we will explore parallelizing the detectors to lower their performance overhead further.

Bibliography

- [1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks—summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA, 1991. ACM.
- [2] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.
- [3] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *Micro, IEEE*, 25(6):10–16, Nov 2005.
- [4] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA, 1984.
- [5] Joao Carreira, Henrique Madeira, João Gabriel Silva, et al. Xception:

- Software fault injection and monitoring in processor functional units. *Dependable Computing and Fault Tolerant Systems*, 10:245–266, 1998.
- [6] J. Cong and K. Gururaj. Assuring application-level correctness against soft errors. In *Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference on*, pages 150–157, Nov 2011.
- [7] C. Constantinescu. Intermittent faults and effects on reliability of integrated circuits. In *Reliability and Maintainability Symposium, 2008. RAMS 2008. Annual*, pages 370–374, Jan 2008.
- [8] Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. Relax: An architectural framework for software recovery of hardware faults. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 497–508, New York, NY, USA, 2010. ACM.
- [9] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 213–224, New York, NY, USA, 1999. ACM.
- [10] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. Shoestring: Probabilistic soft error reliability on the cheap. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 385–396, New York, NY, USA, 2010. ACM.

- [11] Siva Kumar Sastry Hari, Sarita V. Adve, and Helia Naeimi. Low-cost program-level detectors for reducing silent data corruptions. In *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.
- [12] Siva Kumar Sastry Hari, Sarita V. Adve, Helia Naeimi, and Pradeep Ramachandran. Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 123–134, New York, NY, USA, 2012. ACM.
- [13] John L. Henning. Spec cpu2000: measuring cpu performance in the new millennium. *Computer*, 33(7):28–35, Jul 2000.
- [14] M. Hiller, A. Jhumka, and N. Suri. On the placement of software mechanisms for detection of data errors. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 135–144, 2002.
- [15] Ang Jin, Jianhui Jiang, Jiawei Hu, and Jungang Lou. A pin-based dynamic software fault injection system. In *Young Computer Scientists, 2008. ICYCS 2008. The 9th International Conference for*, pages 2160–2167. IEEE, 2008.
- [16] Daya Shanker Khudia, Griffin Wright, and Scott Mahlke. Efficient soft error protection for commodity embedded microprocessors using profile

- information. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES '12, pages 99–108, New York, NY, USA, 2012. ACM.
- [17] Jean-Claude Laprie. Dependable computing and fault-tolerance. *Digest of Papers FTCS-15*, pages 2–11, 1985.
- [18] Chris Lattner and Vikram Adve. Llm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.
- [19] Kyoungwoo Lee, A. Shrivastava, I. Issenin, N. Dutt, and N. Venkatasubramanian. Partially protected caches to reduce failures due to soft errors in multimedia applications. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 17(9):1343–1347, Sept 2009.
- [20] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. Flickr: Saving dram refresh-power through critical data partitioning. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 213–224, New York, NY, USA, 2011. ACM.
- [21] Qining Lu, Karthik Pattabiraman, Meeta S Gupta, and Jude A Rivers. Sdctune: a model for predicting the sdc proneness of an application for

- configurable protection. In *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, page 23. ACM, 2014.
- [22] Silvano Martello and Paolo Toth. *Knapsack problems*. Wiley New York, 1990.
- [23] Thomas Mason et al. LAMPVIEW: A loop-aware toolset for facilitating parallelization. *Master's thesis, Dept. of Electrical Engineeringi, Princeton University*, 2009.
- [24] K. Pattabiraman, Z. Kalbarczyk, and R.K. Iyer. Application-based metrics for strategic placement of detectors. In *Dependable Computing, 2005. Proceedings. 11th Pacific Rim International Symposium on*, pages 8 pp.–, Dec 2005.
- [25] K. Pattabiraman, G.P. Saggese, D. Chen, Z. Kalbarczyk, and R.K. Iyer. Dynamic derivation of application-specific error detectors and their implementation in hardware. In *Dependable Computing Conference, 2006. EDCC '06. Sixth European*, pages 97–108, Oct 2006.
- [26] J.S. Plank, Youngbae Kim, and J.J. Dongarra. Algorithm-based diskless checkpointing for fault tolerant matrix operations. In *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, pages 351–360, June 1995.
- [27] John Ross Quinlan. *C4. 5: programs for machine learning*, volume 1. Morgan kaufmann, 1993.

- [28] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D.I. August. Swift: software implemented fault tolerance. In *Code Generation and Optimization, 2005. CGO 2005. International Symposium on*, pages 243–254, March 2005.
- [29] S.K. Sahoo, Man-Lap Li, P. Ramachandran, S.V. Adve, V.S. Adve, and Yuanyuan Zhou. Using likely program invariants to detect hardware errors. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 70–79, June 2008.
- [30] M. Shafique, S. Rehman, P.V. Aceituno, and J. Henkel. Exploiting program-level masking and error propagation for constrained reliability optimization. In *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE*, pages 1–9, May 2013.
- [31] Joseph Sloan, Rakesh Kumar, and Greg Bronevetsky. Algorithmic approaches to low overhead fault detection for sparse linear algebra. In *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.
- [32] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and W-m Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012.

- [33] A. Thomas and K. Pattabiraman. Error detector placement for soft computation. In *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, pages 1–12, June 2013.
- [34] Long Wang, Z. Kalbarczyk, and R.K. Iyer. Formalizing system behavior for evaluating a system hang detector. In *Reliable Distributed Systems, 2008. SRDS '08. IEEE Symposium on*, pages 269–278, Oct 2008.
- [35] Jiesheng Wei and K. Pattabiraman. Blockwatch: Leveraging similarity in parallel programs for error detection. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–12, June 2012.
- [36] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture, ISCA '95*, pages 24–36, New York, NY, USA, 1995. ACM.
- [37] Kenneth C. Yeager. The mips r10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.