Configurable Detection of SDC-causing Errors in Programs

QINING LU, University of British Columbia
GUANPENG LI, University of British Columbia
KARTHIK PATTABIRAMAN, University of British Columbia
MEETA S. GUPTA, Unaffiliated
JUDE A. RIVERS, Unaffiliated

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- and resource-intensive. In this paper, 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 main difference between them is that *SDCTune* requires manual tuning, while *SDCAuto* is completely automated using machine-learning algorithms.

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 compared to fault-injection, for a fraction of the time taken. Further, 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.

CCS Concepts: •**Computer systems organization** → **Reliability;** •**Software and its engineering** → **Software fault tolerance;**

Additional Key Words and Phrases: fault tolerance, error detection, reliability, compiler, modeling

ACM Reference Format:
Qining Lu, Guanpeng Li, Karthik Pattabiraman, Meeta S. Gupta and Jude A. Rivers, 2015. Configurable Detection of SDC-causing Errors in Programs *ACM Trans. Embedd. Comput. Syst.* 9, 4, Article 39 (March 2010), 25 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Hardware transient errors are increasing due to shrinking feature sizes and diminishing voltage margins [Borkar 2005; Constantinescu 2008]. Conventional hardware-only solutions such as guard banding and hardware redundancy are challenging to apply due to power constraints. As a result, researchers have explored software techniques to tolerate hardware faults [Reis et al. 2005]. 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 important for embedded systems, which have to op-

Author's addresses: Q. Lu, (Current address) Department of Electrical and Computer Engineering, UBC, Vancouver, Canada G. Li, Department of Electrical and Computer Engineering, UBC, Vancouver, Canada K. Pattabiraman, Department of Electrical and Computer Engineering, UBC, Vancouver, Canada M. S. Gupta, Work performed when author was with IBM T.J. Watson Research J. A. Rivers, Work performed when author was with IBM T.J. Watson Research

erate 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, when a crash/hang occurs, there is an indication given so that the system can take appropriate recovery actions. When an SDC occurs, the program will fail without any indication of the failure. This can lead to the error propagating in the system and causing catastrophic effects. Therefore, we focus on SDCs in this paper.

SDCs are caused by errors that propagate to program outputs, and as such, can be prevented by inserting error detectors or checks in programs. A detector checks the consistency of a program variable, and if it fails, aborts the program. However, it is too expensive in practice to insert a check after every program variable, for most commodity applications. Commodity applications include those used in desktop or mobile contexts, as well as data center and high-performance computing (HPC) applications. Unlike mission-critical and life-critical applications such as those used in aerospace or banking, commodity applications typically have a fixed performance overhead budget for protection, and are usually willing to accept the risk of failures. For example, the cost of a failure in a data center application may be a financial penalty as outlined in its service level agreement (SLA), and as a result, the risk of failure needs to be balanced with the performance overhead incurred during normal operation. We focus on commodity applications in this paper, and hence our goal is to provide as high a fault coverage as possible subject to a given performance overhead specified by the user.

The main challenge then becomes how does one identify the program variables which, when protected, offer high SDC coverage within the performance overhead budget. One way to identify the variables to protect in an application is to use fault injection, where we systematically perturb the program to emulate hardware faults and study whether the fault leads to an SDC. While effective, this method requires a large number of fault injections to obtain statistically significant estimates, and is hence highly time-consuming. Another issue with fault injection is that there is significant manual effort involved in mapping the results of the injection back to the program, and making decisions about which variable(s) to protect for a given performance overhead.

In this paper, we propose two models, *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, thus achieving significant time-saving. Further, the models allow 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 requires considerable manual tuning for building the model, and second model, *SDCAuto*, as it builds the model automatically through a machine learning algorithm, thus requiring little to no effort on the part of the developer. *To the best of our knowledge, we are the first to propose models for performing configurable protection against SDC-causing errors in general-purpose applications without using fault injections*.

Our approach has three parts. We first identify heuristics that correlate with highly SDC-prone program variables. We then extract these heuristics using fault injection experiments on a small set of benchmark programs that we use for training purposes. Finally, we integrate the heuristics with manual analysis and machine learning algorithms to build our models. While the initial identification of the heuristics used in

SDCTune and *SDCAuto*, and the subsequent training require fault injection, we do not need fault injection to apply our models to new programs.

In this paper, we target transient hardware 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. We make the following contributions in this paper.

— We develop heuristics to identify SDC-prone variables based on an initial fault-injection study (Section 2). These heuristics are based on static analysis and dynamic profile information obtained at runtime (Section 2.5).
— We develop a manually tuned model, *SDCTune*, based on the heuristics developed to identify the relatively SDC-prone variables in a program. We also develop an automatically tuned model, *SDCAuto*, based on a machine learning algorithm [Quinlan 1993], which can automatically build a regression model from training data.
— We then propose an algorithm based on the models to derive error detectors that check the values of the SDC-prone variables at runtime, subject to a given performance overhead constraint specified by the programmer (Section 3).
— 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 ranks (i.e., relative SDC rates) ranges from 0.855 to 0.877 (Section 5).
— 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 78% to 165% higher efficiencies than full duplication (i.e., ratio of coverage provided to performance overhead incurred), while *SDCAuto* achieves 62% to 96% higher efficiencies (Section 5) than full duplication. Both models achieve much higher efficiencies than hot-path duplication.

2. INITIAL FAULT INJECTION STUDY

In this section, we empirically study how *SDC proneness* of instructions is influenced by the data dependency chains in the program. We first define some terms we will use in the paper 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 are discussed in Section 2.4, and Section 2.5 develops heuristics for estimating the SDC proneness of program variables based on the results.

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 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 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 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 find an instruction set $inst\ set$ that maximizes 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 (through detectors) 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, $P(SDC|I)$ at a statistically significant level. We then use Bayes' formula to obtain $P(I|SDC)$:

$$P(I|SDC) = \frac{P(SDC|I)P(I)}{P(SDC)} \quad (1)$$

where,

$$P(SDC) = \sum_{I \in prog} P(SDC|I)P(I) \quad (2)$$

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. We focus on transient faults because they occur more frequently than permanent errors [Siewiorek 1991]. Further, rates of transient errors are projected to increase significantly due to the effects of technology scaling [Shivakumar et al. 2002]. Extending our fault model to permanent errors is a direction for future research.

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 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. We do not consider faults in the program's code or program counter, as such faults can be detected by control-flow checking techniques. Finally, as in other work [Hari et al. 2012a; Thomas and Pattabiraman 2013; Feng et al. 2010], 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 [Wei et al. 2014]. LLFI works at the intermediate representative (IR) level of LLVM compiler infrastructure [Lattner and Adve 2004], 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 [Hari et al. 2012a; Thomas and Pattabiraman 2013; Feng et al. 2010]).

We use four benchmarks in this experiment, namely *Bzip2*, *IS*, *LU* and *Water-spatial*. They are from SPEC[Henning 2000], NAS[Bailey et al. 1991] and SPLASH-2[Woo et al. 1995] 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% of the SDCs on average. This result is similar to that of prior work, which has also observed that a small fraction of static program instructions cause most SDCs [Hari et al. 2012a]. However, this does not mean that all the hot-path instructions should be protected, as they incur high performance overhead when protected (as we show later). Further, there is considerable variation in SDC rates even among the top 10% most executed instructions as the example below shows.

Table I 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 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, 5, 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 will not improve coverage by much, even while incurring high overheads. 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 I, *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 of

Table I: Example from Bzip2 to illustrate the variation of SDC proneness of highly executed instructions. Results obtained from fault injection.

Source code:

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

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

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* 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* 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 such instructions are usually used for pointer dereferences and are likely to cause segmentation faults which crash the application, when corrupted. 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. These form the basis of our heuristics.

2.5. Heuristics

In this section, we list the heuristics for modelling error propagation and for estimating the SDC proneness of instructions in Table II. Details of these heuristics can be found in the conference version of the paper [Lu et al. 2014]. Here we provide an overview. We focus on stores and comparison instructions as these are the ones that are directly responsible for SDCs. Note however that we calculate the SDC rates for all instructions based on the propagation model (Section 2.4), not just stores and compares. There are 4 categories of store and comparison instruction values as follows:

(1) Addr NoCmp: The stored value is used in calculating memory addresses but not comparison results, with SDC proneness 22.82% on average.
(2) Addr Cmp: The stored value is used in calculating both memory addresses and comparison results, with SDC proneness 48.17% on average.
(3) Cmp NoAddr: The stored value is used in calculating comparison results but not memory addresses, with SDC proneness 67.25% on average.

(4) NoCmp NoAddr: The stored value is neither used in memory address calculation nor comparison results, with SDC proneness 56.41% on average.

Table II: Heuristics extracted from initial fault injections

Class of Heuristics	Description
Fault propagation	The SDC proneness of an instruction will decrease if its result is used in either fault masking or crash prone instructions.
Store Operations	**Addr NoCmp** stored values have low SDC proneness in general.
	Addr Cmp stored values usually have higher SDC proneness than *Addr NoCmp*.
	The SDC proneness of **Addr NoCmp** and **Addr Cmp** stored values increase as their **Data width** decrease.
	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.
	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.
Comparison Operations	**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.
	Comparison operations that only affect **silent stores** have low SDC proneness.
	Comparisons that affect output-related store values have high SDC proneness.
Other Factors	**Memory allocation functions related** stored values and comparison operations have low SDC proneness.

3. APPROACH

In this section, we first extract program features based on the heuristics to describe each store and comparison instruction (Section 3.1). We then build both the *SDCTune* (Section 3.2) and *SDCAuto* models (Section 3.3) with 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 3.5), and the nature of the detectors we inserted to protect the program (Section 3.6)

3.1. Feature extraction

The first step of building our SDC-proneness estimation model is extracting features. Features are extracted according to our heuristics and also based on prior work [Feng et al. 2010; Thomas and Pattabiraman 2013; Cong and Gururaj 2011; Pattabiraman et al. 2005]. Note however that the features that are eventually selected to be used in either *SDCTune* or *SDCAuto* are determined in the model building phase, covered in Section 3.2 and Section 3.3 respectively. Therefore, we can be conservative at this stage and extract as many features as possible.

In total, 66 features are extracted for stored values and 67 for comparisons. The features can be classified into three broad categories. (1) *Execution time related features* pertain to dynamic counts of a program variable or those affected by it. (2) *Code structure related features* pertain to the position of an program variable in the code. (3) *Data usage related features* pertain to the usage of an program variable in the code. Table III shows an excerpt of all the features we extracted.

Table III: Some features extracted for Model Building

Feature group	Subgroup	Feature	Description
Common features	Execution time	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
		post_dominated_execution_time_ratio_bymax	post dominated dynamic counts of an instruction divided by the maximum of all instructions
	Code structure	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	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	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
		execution_time_required_for_addr	the dynamic counts required for computing the storing address
	Code structure	num_static_loads_ratio_bymax	the number of static load instructions divided by the maximum of all stored values
	Data usage	used_in_oef_func_call	whether the stored value is used in functions which have no side effect
Features for comparisons	Execution time	decision_entropy_execution_time	the entropy computed based on the probabilities of the comparison results
	Code structure	is_loop_terminator	whether the comparison result can break a loop execution
	Data usage	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

3.2. Manually Tuned Model: *SDCTune*

Both *SDCTune* and *SDCAuto* are built from fault injections over a set of training programs with the above features. We start building *SDCTune* 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 2.5). 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 data dependencies 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 extracted earlier 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 be covered by our heuristics. We then adopt those features to split our current group into several subgroups. We recursively split these subgroups with our heuristics until all the heuristics are utilized.

Different categories of stored values and comparison results have different categorical features (these are part of our heuristics) for determining their SDC proneness (e.g. *resilient comparisons or not* for *Cmp NoAddr* stored values, and *used in output*

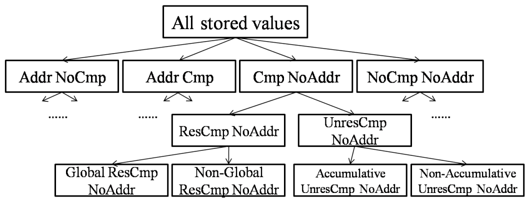


Fig. 1: Manually Tuned classification tree for stored values.

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.

An example tree is shown in Figure 1. In the tree, consider the *Cmp NoAddr* stored values category we introduced in Section 2.5. This constitutes one of the four partitions from the root node of all stored values, and we then split this group into two groups, namely *ResCmp NoAddr* and *UnresCmp NoAddr*. As the tree grows, *ResCmp NoAddr* will then be divided again based on whether the value is a *global variable*, while *UnresCmp NoAddr* will be split based on whether it is *accumulative computation*. Finally, we generate a tree that partitions all stored values into its leaf nodes.

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 4.1. 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.

3.3. 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 [Quinlan 1993]. Our goal here is to demonstrate that machine learning can be used to learn the model automatically, and not to necessarily find the best approach for machine learning. We choose the CART algorithm due to three 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 (ANNs), 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 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, while some other features are numerical, e.g., loop_depth, dominated_execution_time and

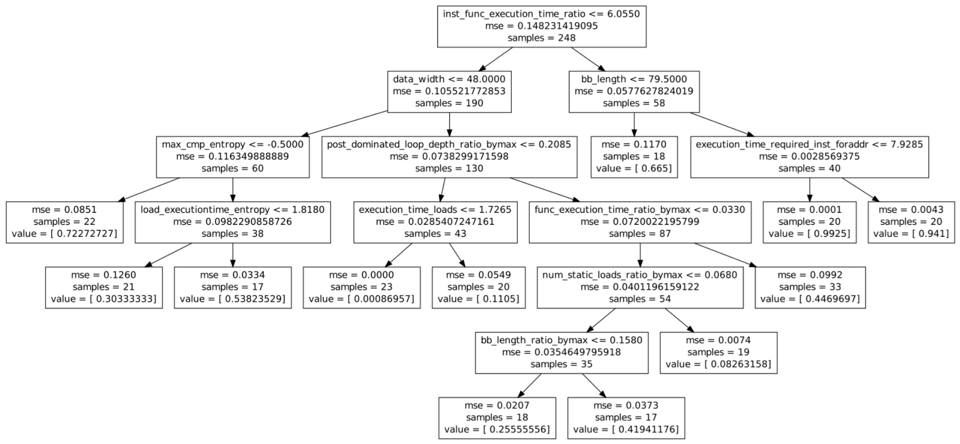


Fig. 2: Auto tuned decision tree for stored values

data_width. Other regression algorithms may not support a mix of categorical data and numerical data.

(3) CART tree model is one of the decision tree models that requires little data preparation [Breiman et al. 1984]. Other regression models, such as 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.

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 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 per application for stored values and comparisons. Store and comparison instructions will be ranked in the decreasing order of their dynamic counts, and only the top ones are incorporated. This is because highly executed instructions are more valuable for SDC proneness estimation (Equation 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. The goal is to minimize the MSE of the tree, which represents the information gain. 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 find an optimal threshold for splitting the node according to the feature which maximizes the MSE reduction.

In the above algorithm, the growth of the trees is controlled by the parameters: *minimum size of leaves* and *data point threshold*. We study the influence of these parameters later. Figure 2 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*.

3.4. Model usage

Once the trees are built from training dataset, we can use them 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 slices to derive the SDC proneness of each instruction with fault masking or crashing rate considered. 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 4.3.

3.5. 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 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 [Martello and Toth 1990].

3.6. Detector Design

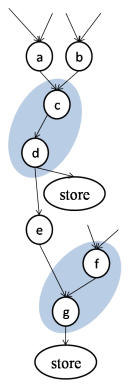
Once we identify a set of instructions to protect, the next step is to insert error detectors at these instructions. Our detectors are based on duplicating the backward slices of the instructions to protect, similar to prior work [Feng et al. 2010]. We insert a check immediately after the instructions 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 3b shows a conceptual example of our detector for a given set of instructions to be protected in Figure 3a.

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 simultaneously. Therefore, any transient error in the computation performed by the chosen instruction will be detected by the corresponding error detector.

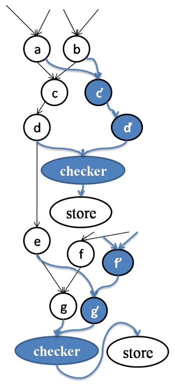
A naive implementation of our detectors can result in prohibitive performance overhead. Therefore, we develop two optimizations to lower the detector overhead. First, we *concatenate* adjacent duplicated pieces of code by adding the instructions between them to the protection set so that we can combine their detectors. Figure 3c shows how this optimization works for the conceptual example. 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. This optimization is effective for long running loops, where the cost of running the check in every loop iteration is high.

4. EXPERIMENTAL SETUP

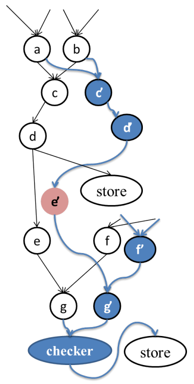
In this section, we empirically evaluate both the *SDCTune* and *SDCAuto* models for configurable SDC protection through fault injection experiments. All the experiments and evaluations are conducted on a Intel i7 4-core machine with 8GB memory running Debian Linux. Section 4.1 presents the details of benchmarks and Section 4.2 presents our evaluation metrics. Section 4.3 presents our methodology and workflow for performing the experiments.



(a) Data dependency of detector-free code. The shaded portion shows the instructions need protection.



(b) Basic detector instrumented. The shaded nodes shows the duplicated instructions and the detector inserted at the end of the two dependency chains.



(c) concatenate duplicated instructions. One instruction is added to be protect(node *e'*) that concatenates the two dependency chains and save one checker

Fig. 3: Example of inserted detectors and concatenating instructions

4.1. Benchmarks

We choose a total of 12 applications from a wide variety of domains for training and testing both of our models. The applications are drawn from the SPEC [Henning 2000], SPLASH2 [Woo et al. 1995], NAS parallel [Bailey et al. 1991], PARSEC [Bienia et al. 2008] and Parboil [Stratton et al. 2012] benchmark suites. We randomly divide the 12 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 IV and Table V respectively. We also show the inputs used for running them. Later we will swap the testing and training benchmarks. 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 work only with single-threaded programs. This is not a fundamental limitation of our technique though, but rather a limitation of our implementation.

4.2. Evaluation Method

We evaluate our SDC proneness estimation model from three perspectives as follows.

1. 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 3.3, there are two parameters controlling the tree building process: (1)*minimum size of leaves*, and (2)*data point threshold*. To explore this two-dimensional parameter space, we vary the *minimum size of leaves* from 1 to 120 points per leaf and *data point threshold* from 10 to 120 data points per program. Average squared errors for both training and testing dataset are calculated for each point in our exploration space. Two optimal pairs of parameters are picked out and used for following evaluations for stored value decision tree and comparison decision tree respectively. We also present the features that are adopted by the optimal trees as these are the features that show strong correlations with SDC proneness.

Table IV: Training programs: These are used for training *SDCTune* and *SDCAuto*

Program	Description	Input
NAS-IS	Integer sorting	N/A
SPLASH2-LU	Linear algebra	-p1 -t
SPEC-Bzip2	Compression	dryer.jpg
PARSEC-Swaptions	Price portfolio of swaptions	-ns 32 -sm 10000 -nt 1
SPLASH2-Water	Molecular dynamics	NMOL=512 NumProcs=1
Parboil-Lbm	Fluid dynamics	20 output.dat 2 1 100_100_130_cf_a.of

Table V: Testing programs: These are used for evaluating *SDCTune* and *SDCAuto*

Program	Description	Benchmark suite
SPEC-Gzip	Compression	input.compressed
SPLASH2-Ocean	Large-scale ocean movements	-n130 -p1 -e1e-7 -r20000.0 -t28800.0 -o
NAS-CG	Conjugate gradient	N/A
Parboil-Bfs	Breadth-First search	1M/graph_input.dat
SPEC-Mcf	Combinatorial optimization	inp.in
SPEC-Libquantum	Quantum computing	33 5

Note that we perform brute force search of the space above to fully characterize the space of the model's parameters. One can also employ techniques such as gradient descent to speed up the traversal process at the cost of having less coverage of the space.

2. Estimation of overall SDC rates:. We perform a statistical 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 use the same fault injection setup as described in Section 2.3. We also measure the correlation between our estimated SDC rates and SDC rates from fault injection. The correlation coefficient shows the capability of using our models in comparing SDC rates among different applications.

3. 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 3.5) 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, and measure the percentage(s) of SDCs detected[1]. We also compare our results with those of full duplication, i.e., when every instruction is duplicated in the program, and with that of hot-path duplication, i.e., when the top 10% most executed instructions 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 metric defined in prior work [Shafique et al. 2013]. 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 provided by the user, and compare it with the corresponding efficiencies of full duplication and hot-path duplication (these correspond to 10% of the paths that ac-

[1]In this experiment, we do not stop our application when an error is detected, but allow it to continue after logging the detection. If the error ultimately results in an SDC, we consider it as a detected SDC.

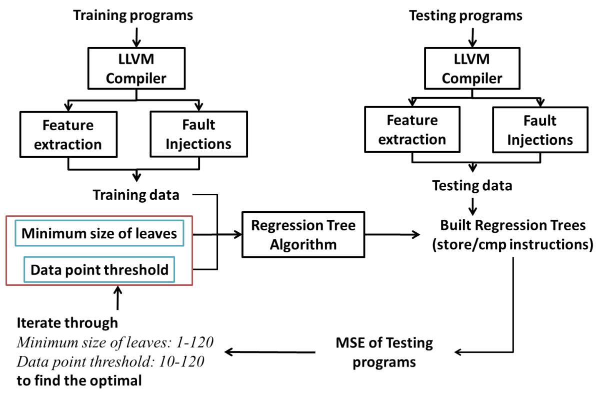


Fig. 4: The workflow of building regression trees and exploring the parameter space for *SDCAuto*.

count for 90% of the execution time). The SDC coverage of full duplication is assumed to be a hundred percent as it can detect all single errors [Reis et al. 2005]. Note that full duplication techniques cannot be configured to be selective, and hence we cannot consider different overhead bounds as we do for the other techniques. Further, full duplication can detect some errors much earlier, as it checks after every instruction, while we do not. However, we do not consider detection latency in our experiments.

4.3. Work Flow and Implementation

Finding optimal parameters of decision trees. Figure 4 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 consists 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 [Mason et al. 2009] 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 the models only - utilizing the built models does not require fault injection. Once the training data and testing data are ready, 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 measure the influences of the two parameters. The values of *minimum size of leaves* and *data point threshold* with minimum MSE of testing data are selected as optimal parameters of regression trees.

Measuring overall SDC estimation and coverage. Figure 5 shows the workflow for estimating the overall SDC rates and providing configurable protection using either the *SDCTune* model or *SDCAuto* model. The workflow requires the following inputs from the user: (1) source code for the program, (2) a set of representative input(s) for executing the application, and (3) output function calls that generate the output data that are important to the user in terms of SDC failures (as mentioned before,

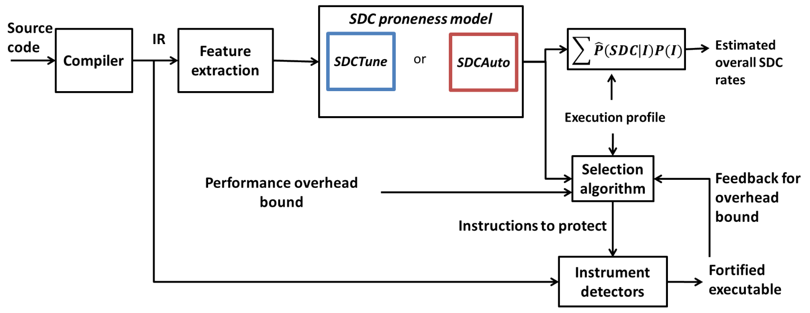


Fig. 5: The workflow of applying our models for two usage cases: (1) *estimate the overall SDC failure rate* and (2) *selectively protect the SDC-prone variables subject to a performance overhead*.

not all output data in an application is important from the perspective of SDCs, for example, statistical or timing information in the output - we therefore exclude these from consideration). In addition, it requires the user to specify the maximum allowable performance overhead that may be incurred by the detectors inserted by our technique for the given set of representative inputs.

As before, we first compile the source code and extract features from the compiled IR. Then, we select to run the extracted features through either the *SDCTune* or the *SDCAuto* model built in Section 3, 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 inserted by another LLVM pass we wrote. We use the representative inputs provided by the user to execute the program for obtaining its execution time with the detectors[2]. 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 and repeat the process. This is a heuristic as the problem of choosing an optimal set of detectors for a given overhead is NP-hard (it is an instance of the well known zero-one knapsack problem). Other heuristics are also possible. Finally, we generate the executable fortified with the detectors using the LLVM backend, and use it to measure its performance overhead and fault coverage by executing it on the target hardware.

5. RESULTS

This section 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* models, and (3) apply configurable protection to maximize detection coverage under different performance overhead bound.

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. Most of the time is spent on the *Feature extraction* and *Selection algorithm*, which require one to forty five minutes (average 10.08 minutes) and five seconds to forty nine minutes (average 14.67 minutes), respectively. On the contrary, fault injection alone requires anywhere from a few hours to a few days to generate the SDC rates

[2]Alternatively, we can use a performance model to predict the overheads of the detectors. However, such a model is non-trivial to obtain, and hence we use direct measurements on the target platform

for each application. Further, estimating the SDC-prone locations in a program using fault injection requires even more fault injection experiments to achieve statistical significance. Further, it requires significant manual effort to map the results of the fault injection back to the program's code, which is necessary for placing detectors.

5.1. Effect of decision tree parameters

We explored the parameter spaces for *stored value* decision tree and *comparison instruction* decision tree. Figure 6 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. From the figure, we can observe that overfitting occurs as we expected when *minimum size of leaves* is too small and incomplete learning occurs when the value 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 6d, while a small value of this parameter can hinder the tree splitting process and decrease the accuracy, as shown in Figure 6c and Figure 6a. Based on the above graphs, we consider *minimum size of leaves = 17*, *data point threshold = 80* as optimal for stored values, and *minimum size of leaves = 57*, *data point threshold = 40* as optimal for comparison instructions.

5.2. 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 VI 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.

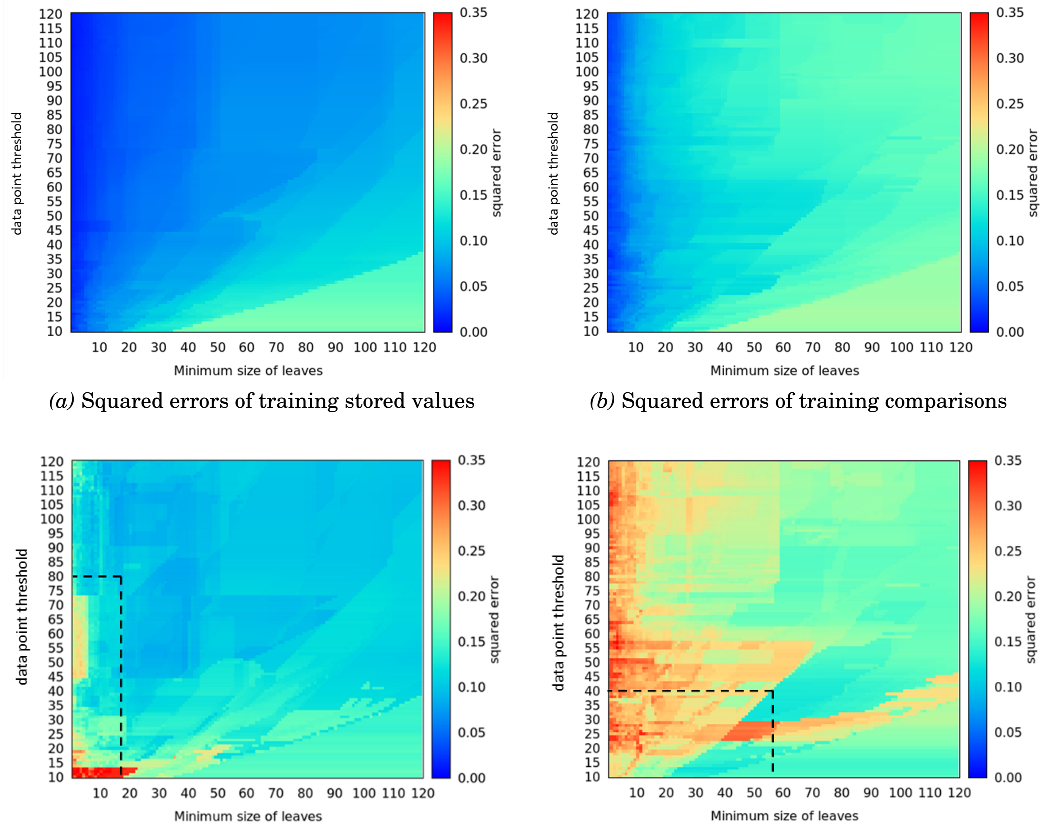
From Table VI, it can be observed that the absolute values of the estimated SDC rates do not match with the observed ones accurately. For example, for *bzip2* the actual SDC rate is 24.47%, while *SDCTune* and *SDCAuto* estimate it to be 17.88% and 19.78% respectively. However, when we consider the *ranks* of the SDC rates predicted by the model, the accuracy is high. Figure 7 plots the estimated SDC ranks versus the observed ranks for both the *SDCTune* and *SDCAuto* models. The Pearson's correlation coefficient is 0.8770 for our *SDCTune* model, and 0.8545 for *SDCAuto* model, showing a strong positive correlation for both models with regard to the SDC ranks.

Thus, our models are highly accurate in comparing SDC rates of applications relative to others. However, they are not accurate at predicting the absolute rates of SDCs. There are two reasons for this inaccuracy. First, our estimation of SDC rates is conservative, and sometimes 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*.

From our results, we find that despite the inaccuracy in predicting absolute SDC rates, our models can guide detector placement to obtain high coverage at low performance overheads. This is because we are more interested in comparing sets of variables with each other when deciding which ones to choose for detector placement, rather than estimate absolute SDC rates.

5.3. SDC Coverage and Detection Efficiency

We use both of our models for inserting error detectors into the applications to maximize SDC coverage under a given performance overhead. Figure 9a shows the SDC



(a) Squared errors of training stored values

(b) Squared errors of training comparisons

(c) Squared errors of testing stored values

(d) Squared errors of testing comparisons

Fig. 6: Effect of *data point threshold*(y-axis) and *minimum size of leaves*(x-axis) on regression results. Lower values of Mean Square Error (MSE) are better.

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

Group	**Benchmark**	$P(SDC)$ **from injections**	$\hat{P}(SDC)$ **from *SDCTune***	$\hat{P}(SDC)$ **from *SDCAuto***
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%

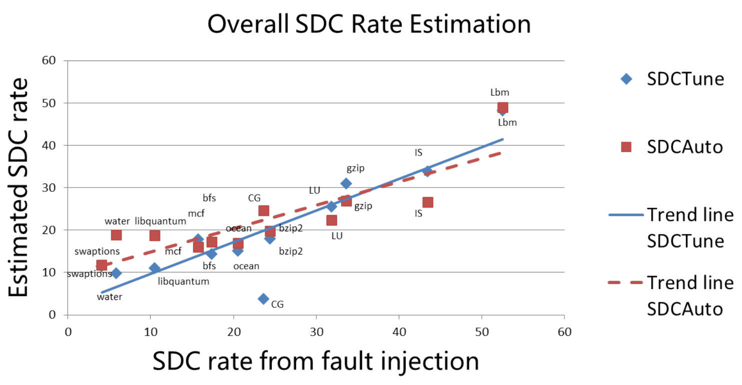


Fig. 7: The correlation of overall SDC rates for all programs. The x-axis is the overall SDC rates from 3000 random fault injections, the y-axis is the estimated overall SDC rates using either *SDCTune* or *SDCAuto*.

coverage obtained by the *SDCTune* model for each benchmark under three different performance overhead bounds: 10%, 20% and 30%. For the training programs, the geometric means[3] 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. Recall that we assumed the coverage of full duplication to be 100%.

Figure 10a shows the SDC coverage obtained by the *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 at the 10%, 20% and 30% performance overhead bounds.

Figure 8 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 overheads are higher than the 30% overhead bound we considered with our detectors.

We also calculate the detection efficiency of the detectors we inserted, and that of hot-path duplication based on their overhead and SDC coverages (Section 4.2). Figure 9b and Figure 10b show the SDC detection efficiency of our detectors with the three overhead bounds, and the efficiency of hot-path duplication. The efficiencies are normalized to that of full duplication, which has a baseline efficiency of 1. An efficiency value close to 1 means that it is not much better than full duplication.

For detectors inserted using the *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. On the other hand, detectors inserted using the *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. We observe

[3]We use geometric means as we are summarizing ratios, i.e., coverage is a ratio.

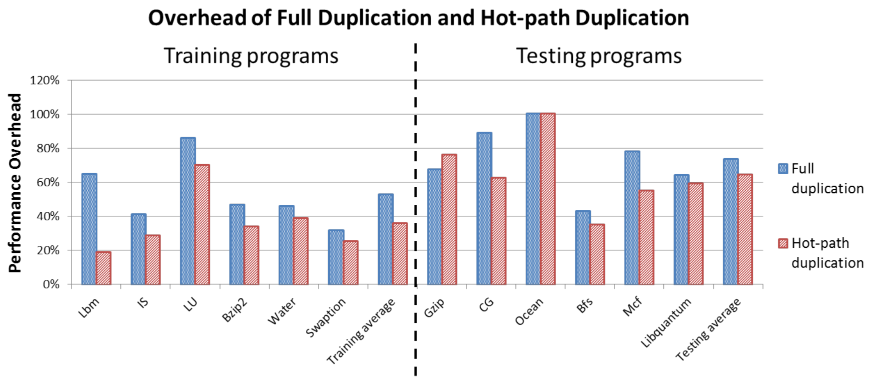


Fig. 8: The overhead of full duplication and hot-path duplication

higher detection efficiencies for our testing programs because the full duplication overheads of the testing programs are often higher than the training programs. This results in a lower baseline for the testing programs in terms of detection efficiencies. More details are provided in Section 6. 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.

We also observe no gain in efficiency with hot-path duplication compared to full duplication in spite of its higher coverage than our technique, as it incurs correspondingly higher overhead (as mentioned in Section 2.4). Thus, placing detectors on the hot-paths of the application is not much better than full duplication in terms of efficiency. In contrast, our technique significantly outperforms both full-duplication and hot-path duplication in providing better detection efficiency, for much lower performance overhead bounds.

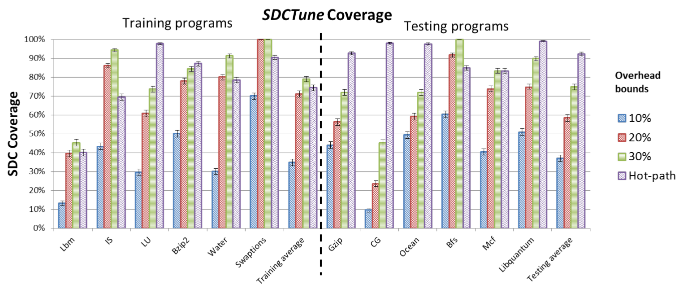
To get a better sense of how dependent are the results on the choices of the testing and training sets, we repeated our experiments on our *SDCAuto* model with the swapped training and testing set. The results show a similar trend in general, but with better coverages and efficiencies for the original testing programs, which are used as training programs now. Figure 11a and Figure 11b show the SDC detection coverage results and the normalized SDC detection efficiency results respectively. The overall coverage and efficiency values of the testing sets are similar in both cases, showing that the model is not highly sensitive to the choices of these two sets.

6. DISCUSSION

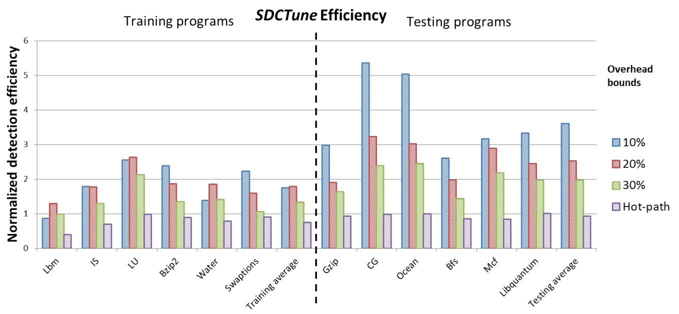
In this section, we first discuss the differences between benchmarks when using *SDCTune* model (Section 6.1). Then we discuss the reasons of different results between *SDCTune* model and the auto tuned model: *SDCAuto* (Section 6.2).

6.1. Differences between benchmarks for *SDCTune*

There are two main reasons for the differences in the detection efficiency among benchmarks. 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 VI, programs such as *Libquantum*, *Bfs*, *Mcf*, *Bzip2*, and *Ocean* fall into this category. Generally, these programs benefit the most



(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%



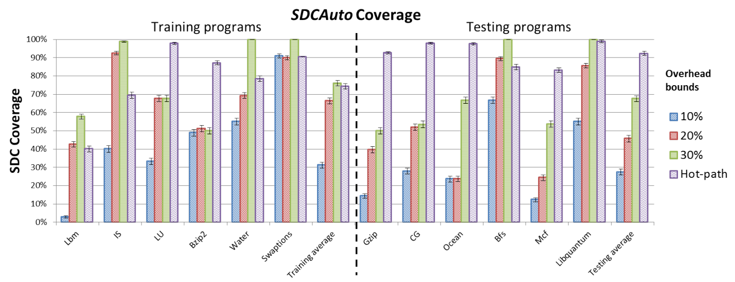
(b) The normalized detection efficiency of *SDCTune* model. Full duplication is the baseline and has detection efficiency = 1. (Detection efficiency is the ratio of SDC coverage and performance overhead)

Fig. 9: The results of *SDCTune* model for different performance overhead bounds, hot-path duplication and full duplication.

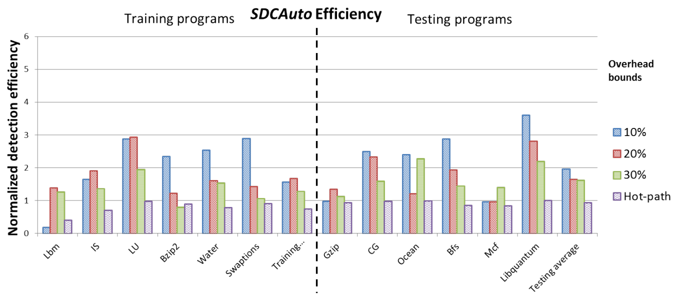
from *SDCTune* model (Figure 9b). However, the detectors inserted in *Mcf* and *Ocean* have higher overhead so that the SDC coverage of these two benchmarks are lower in general under the same performance overhead bound. *Mcf* has a large amount of comparison operations for branches at runtime so that many more check instructions need to be inserted (Section 3.6), which incur higher performance overheads. For *Ocean*, many of its dynamic instances are floating point operations, which cause higher overhead when duplicated because processors usually have very limited ALU resources for float point operations.

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, which also incur high overheads. 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 8. 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 the detection coverage for the benchmarks *IS*, *Bfs* and *Swaptions* is reasonable, their detection efficiency is not very high. In *Lbm*, our detectors have a lower detection efficiency compared to full duplication. This is because nearly all SDC prone



(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)

Fig. 10: The results of *SDCAuto* model for different performance overhead bounds, hot-path duplication and full duplication.

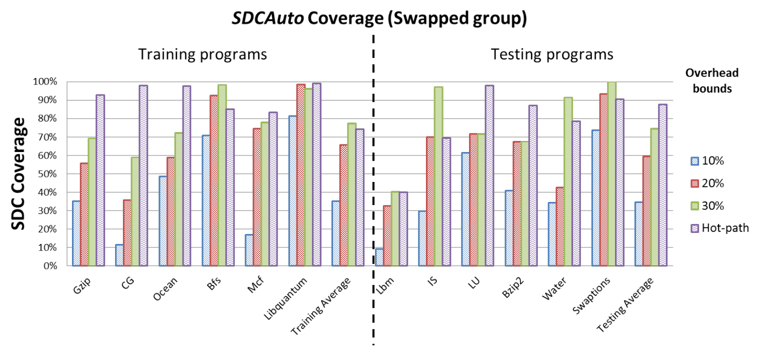
instructions in the program have high execution counts, and hence the performance overhead bounds cannot be satisfied if they are selected for protection. Therefore, this benchmark has low SDC coverage with our technique.

6.2. Differences between the *SDCTune* and *SDCAuto* models

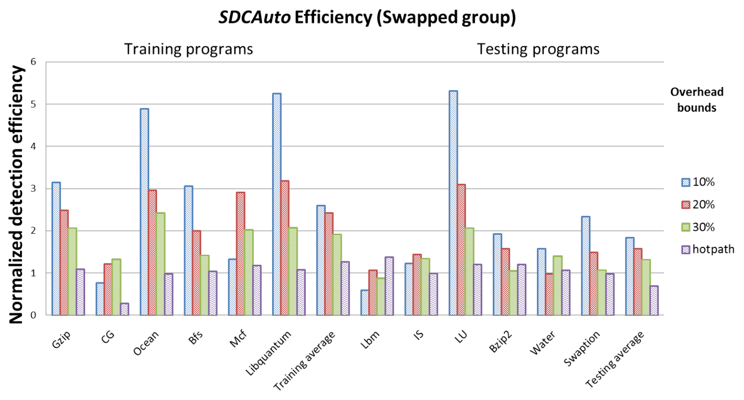
Compared with the results of *SDCTune* model, *SDCAuto* model performs worse across benchmarks. This suggests that the regression trees built by *SDCAuto* are not as robust as the ones built by manual analysis in *SDCTune*.

In *SDCTune* model, the classification depends on the data dependencies among the comparison results. These variables are directly grouped according to their usage and side effects. However, categorizing comparison instructions requires applying multiple features at the same time, e.g., *is_loop_terminator* and *nest_loop_depth*, which is not supported in the decision tree algorithm of *SDCAuto*. This makes the tree unable to split the training data according to the usage of instructions, as a result of which it fails to estimate the SDC proneness of comparison instructions accurately.

In addition, as shown in Figure 2, the regression tree for stored values also fails in categorizing the training data according to the four major usage groups(Section 2.5). All of these four groups need further division to distinguish instructions with high SDC proneness from those with low SDC proneness. Since our decision tree algorithm



(a) The SDC coverages with swapped training/testing programs for *SDCAuto* model.



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

Fig. 11: The results of *SDCAuto* model for different performance overhead bounds, hot-path duplication and full duplication with swapped training/testing set.

determines the splitting points based on MSE criteria, the trees are not likely to split at these features.

Thus, the *SDCAuto* model performs worse than the *SDCTune* model, but still manages to outperform full duplication and hot-path duplication in terms of efficiency (by 62% to 96%). Further, the *SDCAuto* model does not require any programmer effort to build and apply, making it a much more practical proposition for developers.

7. RELATED WORK

We classify related work into three categories, namely (1) duplication based techniques, (2) invariant based techniques, and (3) application specific techniques.

Duplication based techniques: SWIFT [Reis et al. 2005] is a compiler based technique that uses full duplication to detect faults in program data. However, full duplication can have significant performance overhead, especially in embedded systems which do not have as many idle resources to mask the overhead of duplication. As shown in Figure 9b and Figure 10b, *SDCTune* and *SDCAuto* both outperform full duplication in terms of SDC detection efficiency, and also enable configurability to protect programs from SDC causing errors under various given performance overheads. One of the earliest papers on identifying critical variables in programs, and selectively protecting them is by Pattabiraman et al. [Pattabiraman et al. 2005]. 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.

Feng et al. [Feng et al. 2010], and Khudia et al. [Khudia et al. 2012] 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 [Lee et al. 2009; Cong and Gururaj 2011; Thomas and Pattabiraman 2013; de Kruijf et al. 2010; Liu et al. 2011] 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, which are not as error resilient.

Finally, in recent work, Shafique et al. [Shafique et al. 2013] 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 detection 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.

Invariant based techniques [Sahoo et al. 2008; Ernst et al. 1999; Pattabiraman et al. 2006] 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 (which may differ from the test imputs). 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 hence incur no false positives as static analysis is conservative.

Application specific techniques: Hari et al. [Hari et al. 2012a] 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 [Hari et al. 2012b], they still need to perform tens of thousands of injections. In contrast, we use our model to determine the SDC prone locations without needing any 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.

8. CONCLUSION

As hardware errors increase with technology scaling, 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 an automatically tuned model, *SDCAuto*, based on the heuristics and a decision tree algorithm. We then use our models to guide the selection of instructions to be protected with error detectors. Our results show that the detectors inserted using *SDCTune* outperform full duplication by 78% to 165% in detection efficiency, while those inserted using *SDCAuto* outperform full duplication by a factor of 62% to 96%.

ACKNOWLEDGMENTS

This work is sponsored, in part, by the Natural Science and Engineering Research Council of Canada (NSERC), Defense Advanced Research Projects Agency (DARPA), Microsystems Technology Office (MTO), under contract no. HR0011-13-C-0022. The views expressed are those of the authors and do not reflect the official policy or position of the NSERC, Department of Defense or the U.S. Government. We thank Muhammad Shafique for his comments on an earlier draft of this paper.

REFERENCES

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. 1991. The NAS parallel benchmarks. In *ACM/IEEE Conference on Supercomputing (Supercomputing '91)*. ACM, New York, NY, USA, 158–165. http://doi.acm.org/10.1145/125826.125925

Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. ACM, New York, NY, USA, 72–81. http://doi.acm.org/10.1145/1454115.1454128

S. Borkar. 2005. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *Micro, IEEE* 25, 6 (Nov 2005), 10–16.

L. Breiman, J. Friedman, R. Olshen, and C. Stone. 1984. *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA.

J. Cong and K. Gururaj. 2011. Assuring application-level correctness against soft errors. In *Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference on*. 150–157.

C. Constantinescu. 2008. Intermittent faults and effects on reliability of integrated circuits. In *Reliability and Maintainability Symposium, 2008. RAMS 2008.* 370–374.

Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. 2010. Relax: An Architectural Framework for Software Recovery of Hardware Faults. In *International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 497–508. http://doi.acm.org/10.1145/1815961.1816026

Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 1999. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *International Conference on Software Engineering (ICSE '99)*. ACM, New York, NY, USA, 213–224. http://doi.acm.org/10.1145/302405.302467

Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. 2010. Shoestring: Probabilistic Soft Error Reliability on the Cheap. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, USA, 385–396. http://doi.acm.org/10.1145/1736020.1736063

Siva Kumar Sastry Hari, Sarita V. Adve, and Helia Naeimi. 2012a. Low-cost Program-level Detectors for Reducing Silent Data Corruptions. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) (DSN '12)*. IEEE Computer Society, Washington, DC, USA, 1–12. http://dl.acm.org/citation.cfm?id=2354410.2355132

Siva Kumar Sastry Hari, Sarita V. Adve, Helia Naeimi, and Pradeep Ramachandran. 2012b. Relyzer: Exploiting Application-level Fault Equivalence to Analyze Application Resiliency to Transient Faults. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, New York, NY, USA, 123–134. http://doi.acm.org/10.1145/2150976.2150990

John L. Henning. 2000. SPEC CPU2000: measuring CPU performance in the New Millennium. *Computer* 33, 7 (Jul 2000), 28–35.

Daya Shanker Khudia, Griffin Wright, and Scott Mahlke. 2012. Efficient Soft Error Protection for Commodity Embedded Microprocessors Using Profile Information. In *International Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES '12)*. ACM, New York, NY, USA, 99–108. http://doi.acm.org/10.1145/2248418.2248433

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

Kyoungwoo Lee, A. Shrivastava, I. Issenin, N. Dutt, and N. Venkatasubramanian. 2009. Partially Protected Caches to Reduce Failures Due to Soft Errors in Multimedia Applications. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 17, 9 (Sept 2009), 1343–1347.

Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. 2011. Flikker: Saving DRAM Refresh-power Through Critical Data Partitioning. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 213–224. http://doi.acm.org/10.1145/1950365.1950391

Qining Lu, Karthik Pattabiraman, Meeta S. Gupta, and Jude A. Rivers. 2014. SDCTune: A Model for Predicting the SDC Proneness of an Application for Configurable Protection. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '14)*. ACM, New York, NY, USA, Article 23, 10 pages. http://doi.acm.org/10.1145/2656106.2656127

Silvano Martello and Paolo Toth. 1990. *Knapsack problems*. Wiley New York.

Thomas Mason and others. 2009. LAMPVIEW: A Loop-Aware Toolset for Facilitating Parallelization. *Master's thesis, Dept. of Electrical Engineeringi, Princeton University* (2009).

K. Pattabiraman, Z. Kalbarczyk, and R.K. Iyer. 2005. Application-based metrics for strategic placement of detectors. In *Dependable Computing. Pacific Rim International Symposium on*. 8 pp.–.

K. Pattabiraman, G.P. Saggese, D. Chen, Z. Kalbarczyk, and R.K. Iyer. 2006. Dynamic Derivation of Application-Specific Error Detectors and their Implementation in Hardware. In *Dependable Computing Conference, European (EDCC)*. 97–108.

John Ross Quinlan. 1993. *C4. 5: programs for machine learning*. Vol. 1. Morgan kaufmann.

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

S.K. Sahoo, Man-Lap Li, P. Ramachandran, S.V. Adve, V.S. Adve, and Yuanyuan Zhou. 2008. Using likely program invariants to detect hardware errors. In *Dependable Systems and Networks, IEEE International Conference on*. 70–79.

M. Shafique, S. Rehman, P.V. Aceituno, and J. Henkel. 2013. Exploiting program-level masking and error propagation for constrained reliability optimization. In *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE*. 1–9.

Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. 2002. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic *(DSN)*. 389–398.

D.P. Siewiorek. 1991. Architecture of fault-tolerant computers. *Proceedings of IEEE* (1991), 79–91.

John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and W-m Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* (2012).

A. Thomas and K. Pattabiraman. 2013. Error detector placement for soft computation. In *Dependable Systems and Networks (DSN), IEEE/IFIP International Conference on*. 1–12.

Jiesheng Wei, A. Thomas, Guanpeng Li, and K. Pattabiraman. 2014. Quantifying the Accuracy of High-Level Fault Injection Techniques for Hardware Faults. In *Dependable Systems and Networks (DSN), IEEE/IFIP International Conference on*. 375–382.

Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 Programs: Characterization and Methodological Considerations. *SIGARCH Comput. Archit. News* (1995), 13. http://doi.acm.org/10.1145/225830.223990