SDCTune: A Model for Predicting the SDC Proneness of an Application for Configurable Protection

Qining Lu Karthik Pattabiraman
Department of Electrical and Computer Engineering, UBC
{qining, karthikp}@ece.ubc.ca

Meeta S. Gupta Jude A. Rivers
Reliability- and Power-Aware Microarchitectures, IBM T.J. Watson Research Center *
{meetasgupta,juderivers}@gmail.com

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 paper, we propose an empirical model to predict the SDC proneness of a program's data called *SDCTune*. *SDCTune* is based on static and dynamic features of the program alone, and does not require fault injections to be performed. We then develop an algorithm using *SDCTune* to selectively protect the most SDC-prone data in the program subject to a given performance overhead bound. Our results show that our technique is highly accurate at predicting the relative SDC rate of an application, and outperforms full duplication by a factor of 0.83 to 1.87x in efficiency of detection (i.e., ratio of SDC coverage provided to performance overhead).

Categories and Subject Descriptors C.4 [*Performance of Systems*]: Fault tolerance, Reliability, availability, and serviceability

Keywords Reliability, Compiler, Modeling

1. Introduction

Hardware errors are increasing due to shrinking feature sizes [3, 5]. 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 [19]. 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 [17, 24] has broadly focused on crashes and hangs; therefore 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 [9, 11, 23], 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 [9, 11]. Other work [23] 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. [21], but their technique does not distinguish between SDC causing errors and other failure causing errors.

In this paper, we propose *SDCTune*, a model 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* uses 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 [1].

The main novelty of our approach is in the identification of heuristics or features that correlate with highly SDC-prone program variables. 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 *SDCTune* to quantify the relative SDC proneness of a variable. While the initial identification of the heuristics used in *SDCTune* requires fault injection, we do not need fault injection to apply *SDCTune* to new programs.

* Meeta S. Gupta is currently on leave-of-absence from IBM Research, USA, working on a temporary full-time supplemental research position at IBM India Research Laboratory in Bangalore, India.

[1] We call our model, *SDCTune*, as it allows tuneable protection.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ESWEEK'14, October 12 - 17 2014, New Delhi, India.
Copyright © 2014 ACM 978-1-4503-3050-3/14/10. . . $15.00.
http://dx.doi.org/10.1145/2656106.2656127

In this paper, 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* 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.

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 profile information (Section 3).
- We develop a model called *SDCTune*, based on the heuristics developed to identify the relatively SDC-prone variables in a program. We then propose an algorithm based on *SDCTune* 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 (Section 4).
- We evaluate *SDCTune* by using it to predict the overall SDC proneness of a program relative to other programs. The results show that *SDCTune* is highly accurate at predicting the overall SDC proneness of a program relative to other programs. The rank correlation coefficient between the predicted and observed values ranges from 0.83 to 0.97 (Section 6).
- 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, and achieves 0.83 to 1.87x higher efficiencies than both full duplication and hot-path duplication (Section 6).

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 section, we empirically study how *SDC proneness* of instructions is influenced by the data dependency chains. 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 is discussed in Section 2.4, and will be used in Section 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 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 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 $P(I|SDC)$:

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

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. 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 [14] 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 [27]. 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 [8, 9, 23], 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 [25]. LLFI works at the intermediate representative (IR) level of LLVM compiler infrastructure [13], 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 [8, 9, 23]). 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 necessary for our analysis.

We use four benchmarks in this experiment, namely *Bzip2*, *IS*, *LU* and *Water-spatial*. They are from SPEC[10], NAS[1] and SPLASH-2[26] 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 [9]. However, this does not mean that all the hot-path instructions should be protected, as they incur high performance overhead when protected (as shown in Section 6.2). Further, there is considerable variation in SDC rates even among the top 10% most executed instructions as the example below shows.

Table 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 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 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 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 Section 3 by devising heuristics for finding highly SDC-prone instructions.

3. Heuristics

In this section, we formulate various heuristics for modelling error propagation in a program, and for estimating the SDC proneness of an instruction. These heuristics will be used in the next section to build our model: *SDCTune*.

Table 1: Example from Bzip2 to illustrate the variation of SDC proneness of highly executed instructions. Results obtained from fault injection. Source code:

```
1  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	-

Table 2: 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

In the previous section, 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 subsection, 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 1, the fault masking effect of *instruction 7* results in *instruction 6* having a low SDC proneness.

Table 2 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, 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 apply back-propagation starting from the store and comparison instructions through the data dependency chains of the program. The SDC

Table 3: 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%

Table 4: 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%

proneness of the result of an instruction will propagate to its source operands unless it is one of the operations listed in Table 2, in which case, the SDC proneness of the source operands will decrease by a certain extent, as listed in Table 3 to model the effect of masking. The values in Table 3 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 subsections.

3.2 Heuristics for Store Operations

In this subsection, 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, 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 4. 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 4. 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 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.

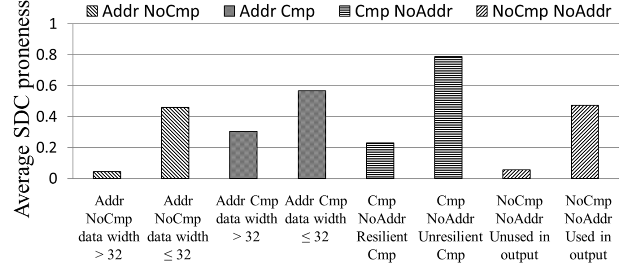
> *HS1:* **Addr NoCmp** *stored values have low SDC proneness in general, as shown in Table 4.*

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 1a).

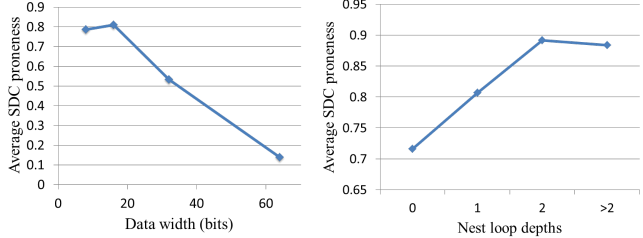
Figure 2a shows an example of this category, where a fault in the destination register of *i-3* in (line 3) results in a system crash upon pointer dereference.

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

As shown in Figure 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



(a) Effects of major related features for each of the four major categories of stored values.



(b) Effect of data width for address computation related stored values

(c) Effect of nest loop depths for loop terminating comparisons

Figure 1: Average SDC proneness observed across all studied programs

increasing the SDC proneness compared to the *Addr NoCmp* category. As an example of this category from *Bzip2* is shown in Figure 2b.

> *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 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 2c 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 2d, illustrates a case where a fault in the comparison operator, *selectorMtf[i]=j*(line 3) will affect the number of loop iterations, thus making it highly SDC prone. 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 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 af-

```
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 }
```

(a) 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 }
```

(b) Example of *Addr Cmp* from Bzip2. The fault occurs at line 2 may not propagate to the load at line 10 because of the control flow deviation at line 3

```
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  }
```

(c) 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  }
```

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

```
1  void main (...) {
2    ...
3    (start) = (unsigned long)(
         FullTime.tv_usec +
         FullTime.tv_sec *
         1000000);
4    ...
5    Global−>starttime = start;
6    printf (..., Global−>
         starttime);
7  }
8
9
```

(e) 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[ii][jj];
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 }
```

(f) Example of *NoCmp NoAddr* from *LU* with high SDC proneness.

Figure 2: Examples of stored values. The fault propagation is highlighted in red.

fect pointers or branches. Figure 2e and Figure 2f 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 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 1c.*

Figure 3a shows an example from *Bzip2*. Both *nHeap>1* and *weight[tmp]<weight[heap[zz»1]]* are used in determining the loop exit conditions for the outer and inner loops respectively.

```
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*. The comparison result for the outer loop has a higher SDC proneness than the comparison result for the inner loop

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

(b) An example from *Bzip2* that the comparison result: *a2update<BZ_NOVERSHOOT* (line 4) has a low SDC proneness. It only affects a silent store instruction

```
1  daxpy(double* a, double* b ,...) {
2    long l;
3    for(i=0;i<n;i++){ // terminates the loop too early
4      a[i]+=alpha*b[i]; //skipped due to loop termination
5    }
6  }
7  bmodd(double* a, double* c ,...) {...
8    daxpy(&a[k+1+j*stride_c],&a[k+1+j*stride_a],dimi−k−1,alpha);...
9    // the content of a[] is corrupted
10 }
11 lu() {...
12   A=&a[K+j*nblocks]; // fault propagates to a[] through the call of
         bmodd()
13   bmodd(D,A,strK,strJ,strK,strK);... // content of A[] is corrupted
14 }
15 CheckResult(...,double* a ,...)  {... // called by main()
16   printf (..., max_diff ,...)  ;... // corrupted because of corrupted a[]
17 }
```

(c) An example from *LU* that a faulty comparison result: *i<n*(line 3) will change the control flow to elide the store operation *a[i]+=alpha*b[i]*(line 4). The value is used in the calculation of the output in function *lu()* and finally in *CheckResult()*. The fault propagation trace is highlighted in red.

Figure 3: Examples of comparison results. Fault propagation is highlighted in red.

Since the outer loop covers more program data than the inner loop, an extra or missing iteration of the outer loop caused by a faulty comparison result has a higher chance to cause the program to deviate from its correct execution. Prior work has made a similar observation in the context of soft-computing applications [23].

> *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 3b. A flip in the comparison *a2update<BZ_NOVERSHOOT*(line 5) can cause the store operation *quadrant[a2update+nblock]=qVal*(line 6) 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 3c 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 branch and store operations, 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 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%.

Other Features: In addition to the above features, we consider other program features considered in prior work, such as *global variable* [8], *the loop depth* [23], *accumulative computation* [4], and *fan-out of variable* [17]. We do not explain them due to space constraints, however.

4. Approach

In the previous section, we examined various heuristics for identifying SDC-prone variables in a program. In this section, we first quantify the estimation of SDC proneness using the *SDC-Tune* model obtained from empirical data (Section 4.1). We then present our approach for choosing the SDC-prone locations subject to a maximum performance overhead using *SDCTune*(Section 4.2). Finally, we describe the nature of the detectors we inserted to protect the program (Section 4.3).

4.1 Model Building

Our model, *SDCTune*, for predicting the SDC proneness of a variable is built from fault injections over a set of training programs, and incorporates the heuristics defined in the previous section. We start by modelling the SDC proneness of store and branch instructions in the program. The SDC proneness of these instructions depends on discrete features such as *resilient comparisons* and on continuous features such as *data width* (Section 3.2 and Section 3.3). We use *classification* to model the discrete features, and *linear regression* to model the continuous ones. Once we determine the SDC proneness of the store and branch instructions, we use the back 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 discrete features that we observed before to categorize the stored values or comparison results into different groups so that we can apply the continuous features (or arithmetic means) to quantify the SDC proneness of each group. As shown in Sections 3.2 and 3.3, different categories of stored values and comparison results have different discrete 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.

For example, consider the *Cmp NoAddr* stored values category we introduced in Section 3.2. 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* based on heuristics *HS4* (Section 3.2). As the tree grows, *ResCmp NoAddr* will then be divided again based on whether the value is a *global variable* (Section 3.4), while *UnresCmp NoAddr* will be split based on whether it is *accumulative computation* (Section 3.4). The other types of stored values: *Addr NoCmp*, *Addr Cmp* and *NoCmp NoAddr* will also be partitioned in a similar way but with different features. Finally, we generate a tree that partitions all stored values into its leaf nodes. The tree generated for this example is shown in Figure 4.

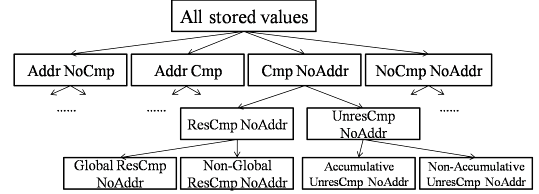


Figure 4: The example of the classification tree for stored values.

Regression is applied upon the leaf nodes of the classification tree to factor in the effects of continuous 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 continuous features, we take the arithmetic means as the estimation of their SDC proneness.

4.2 Choosing the Instructions

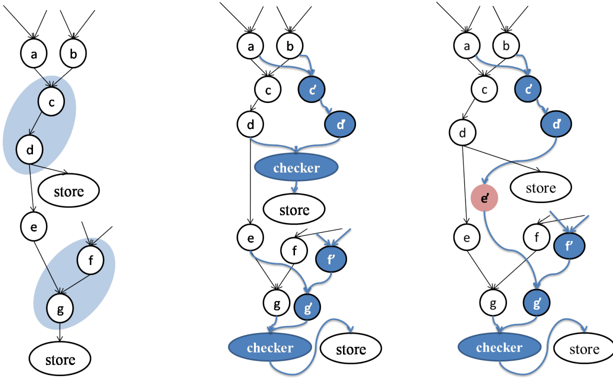
As shown in Section 2, we can calculate the *SDC coverage* of protecting an instruction if we know the *SDC proneness* of that instruction using Equation 1 in Section 2.1. We apply *SDCTune* to estimate the SDC proneness of each instruction in the program that we want to protect. We also obtain the dynamic 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), using a standard dynamic programming algorithm [9].

4.3 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 [8]. 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 5b shows a conceptual example of our detector for a given set of instructions to be protected in Figure 5a.

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 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 5c 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 6 illustrates. This optimization is effective for long running loops.



(a) Data dependency of detector-free code

(b) Basic detector instrumented

(c) *concatenate* duplicated instructions

Figure 5: The shaded portion of (a) shows the instructions need protection. (b) shows the duplicated instructions (the shaded nodes) and the detector inserted at the end of the two dependency chains. (c) shows one added instruction to protect(node *e'*) that concatenates the two dependency chains and save one checker

```
1  for(i=0;; i++){
2    // loop body
3    flag = i<n?1:0;
4    if(flag == 1)
5      break;
6    //decompose exit
           predication
           to
           simulate
           instruction
           —level
           behaviour.
7  }
8
```

(a) Detector-free code

```
1  i=0;
2  // duplication of i
3  dup_i=0;
4  for (;;) {
5    //loop body
6    flag = i<n?1:0;
7    dup_flag = dup_i
          <n?1:0;
8    if(flag != dup_flag)
9      Assert();
10     // inconsistent
11   if(flag == 1)
12     break;
13 }
```

(b) Basic detector instrumented

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

(c) *Lazy checking* applied

Figure 6: (b) shows how the loop index *i* in original code (a) is protected with bold code as check. (c) shows how we move the check out of the loop body

5. Experimental Setup

In this section, we empirically evaluate *SDCTune* 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 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 12 applications from a wide variety of domains for training and testing *SDCTune*. The applications are drawn from the SPEC [10], SPLASH2 [26], NAS parallel [1], PARSEC [2] and Parboil [22] benchmark suites. We randomly divide the 12 applications into two groups of 6 applications each, 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 and Table 6 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 *SDCTune* works only with single-threaded programs.

5.2 Evaluation Method

To gauge the accuracy of *SDCTune*, we use it for estimating the overall SDC rate of an application, as well as the SDC coverage(s) for different performance overhead bounds. The former is used for

Table 5: Training programs: These are used for training *SDCTune*

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
CG	Conjugate gradient	NAS	default	32	97

Table 6: Testing programs: These are used for evaluating *SDCTune*

Program	Description	Benchmark suite	Input	Stores	Comparisons
Lbm	Fluid dynamics	Parboil	short	71	34
Gzip	Compression	SPEC	test	251	399
Ocean	Large-scale ocean movements	SPLASH2	test	322	813
Bfs	Breadth-First search	Parboil	1M	36	57
Mcf	Combinatorial optimization	SPEC	test	87	158
Libquantum	Quantum computing	SPEC	test	39	136

comparing the SDC rates of different applications, while the latter is used to insert detectors for configurable protection. We use the same experimental setup for fault injection as that described in Section 2.3.

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 *SDCTune* with that obtained from the fault injection experiment. We also compare the relative SDC rate of an application with respect to other applications (i.e., its rank) estimated by *SDCTune* with that obtained from fault injection.

SDC coverages for different performance overhead bounds: The SDC coverage is defined as the fraction of SDC causing errors detected by our detectors. We use *SDCTune* to predict the SDC coverage for different instructions to satisfy the performance overhead bounds provided by the user. Our selection algorithm(Section 4.2) 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 then 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 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 defined in prior work [21]. 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 [19].

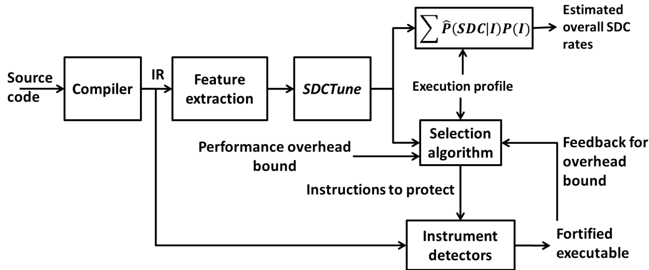


Figure 7: The workflow of applying *SDCTune* for two usage cases: (1) *estimate the overall SDC failure rate* and (2) *selectively protect the SDC-prone variables subject to a performance overhead.*

5.3 Work Flow and Implementation

Figure 7 shows the workflow for estimating the overall SDC rates and providing configurable protection using *SDCTune*. 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 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.

We first compile the application using LLVM into its IR form. We then extract the features that our model 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 [16] for analyzing load/store dependencies. Third, we run the parameters through *SDCTune* built in Section 4.1, to generate a estimated SDC proneness for each instruction. Fourth, we use the results from *SDCTune* 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. 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.

6. Results

This section presents the results of our experiments to use *SDCTune* for (1) estimating the overall SDC rate of an application and (2) for configurable protection to maximize detection coverage under different performance overhead bound. In our experiments, *SDCTune* requires 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. 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.

6.1 Estimation of Overall SDC Rates

We estimate the overall SDC rate of the application using *SDCTune*, and compare it with the SDC rate obtained through 3000 random fault injections per benchmark. Table 7 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 0.49%(CG), at the 95% confidence intervals.

Table 7: The SDC rates and ranks from fault injections and *SDCTune*

Group	Benchmark	$P(SDC)$ **(rank) from injections**	$\hat{P}(SDC)$ **(rank) from *SDCTune***
Training	IS	43.46% (1)	33.75% (1)
	LU	31.9% (2)	25.43% (2)
	Bzip2	24.47% (3)	17.88% (3)
	Water	5.9% (4)	9.75% (5)
	Swaptions	4.1% (5)	11.46% (4)
	CG	1.89% (6)	3.54% (6)
Testing	Lbm	52.53% (1)	48.11% (1)
	Gzip	33.67% (2)	32.46% (2)
	Ocean	20.6% (3)	14.75% (4)
	Bfs	17.37% (4)	14.27% (5)
	Mcf	15.76% (5)	17.84% (3)
	Libquantum	10.5% (6)	10.9% (6)

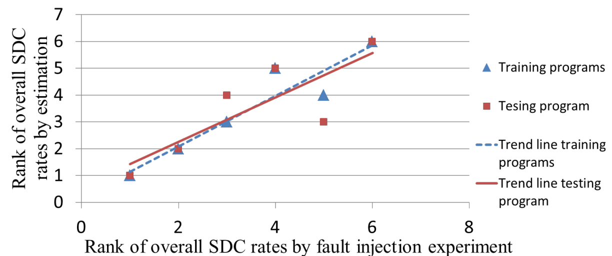


Figure 8: The rank correlation for both training and testing programs. The x-axis is the rank of overall SDC rates from 3000 random fault injections, the y-axis is the rank of estimated overall SDC rates using *SDCTune*. The rank correlaion coefficients are 0.9714(training) and 0.8286(testing).

From Table 7, it can be observed that the absolute values of the estimated SDC rates do not match with the observed ones. However, the ranks of the SDC rates estimated by the model closely match those observed in reality. Figure 8 plots the estimated SDC ranks versus the observed ranks for both the training and testing programs. The Spearman's rank correlation coefficient is 0.9714 for training programs (p-value=0.00694), and 0.8286 for testing programs (p-value = 0.0125), showing a strong positive correlation for both sets of programs.

Thus, *SDCTune* is highly accurate in predicting SDC ranks of applications relative to other applications. 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 back-propagation 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*. This inaccuracy in absolute SDC rate prediction may lead to inadequate protection, and additional overhead. However, our results show that despite the inaccuracy, *SDCTune* can guide detector placement to obtain high coverage at low performance overheads.

6.2 SDC Coverage and Detection Efficiency

We use *SDCTune* for inserting error detectors into the applications to maximize SDC coverage under a given performance overhead. Figure 9a shows the SDC coverage obtained by our technique 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 44.8%, 78.6% and 86.8%, respectively. For the testing programs, the corresponding geometric means are 39.0%,

63.7% and 74.9% 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 79.5% and 87.6% on average for training and testing programs respectively.

Figure 9b shows the performance overhead of full duplication and hot-path duplication. The overhead of full duplication is 53.7% on average for the training programs, while it is 73.6% on average for the testing programs. Hot-path duplication has an overhead of 43.5% for the training programs, and 57.6% for the testing programs. Note that both of these are considerably higher than the 30% overhead bound we considered with our detectors.

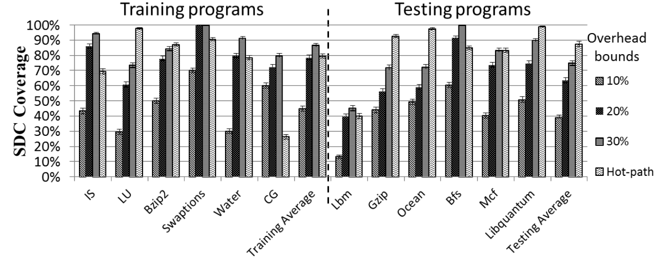
We also calculate the detection efficiency of the detectors we inserted, and for hot-path duplication based on their overhead and SDC coverages (Section 5.2). Figure 9c shows the SDC detection efficiency for our detectors with the three overhead bounds, and for hot-path duplication. The efficiencies 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.

For our detectors, we observe SDC detection efficiencies of 2.38x, 2.09x and 1.54x for the training programs, and 2.87x, 2.34x and 1.84x for the testing programs, at the 10%, 20% and 30% performance overhead bounds respectively. The reason that the efficiencies 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 high coverage, as it incurs correspondingly higher overhead (as mentioned in Section 2).

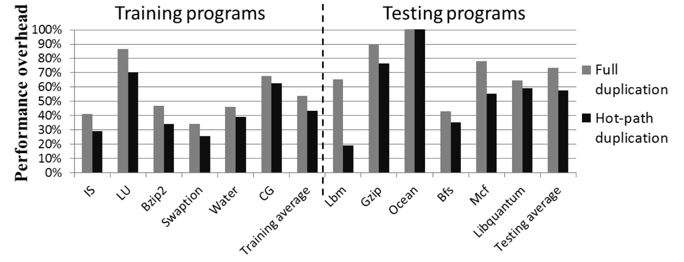
We find that there is considerable variation in detector efficiency among benchmarks. There are two reasons for this variation. First, for our technique to be efficient, it needs to protect instructions with high SDC proneness, but with low dynamic execution count. We observed that applications which have such instructions experience moderate SDC rates, which are neither too high nor too low. From Table 7, programs such as *Gzip*, *Libquantum* and *Ocean* fall into this category. These programs benefit the most from our technique (Figure 9c). 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*. An exception to this is the *CG* benchmark, which has only a few SDC-prone instructions that are highly executed. Therefore, protecting these instructions is sufficient to obtain high coverage, and this can be done with relatively low overhead (compared to full duplication). Second, if all instructions of a program are low in SDC proneness (e.g., *water*), our technique does not do as well, since no instruction provides higher benefit when protected than others.

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 9b. 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.

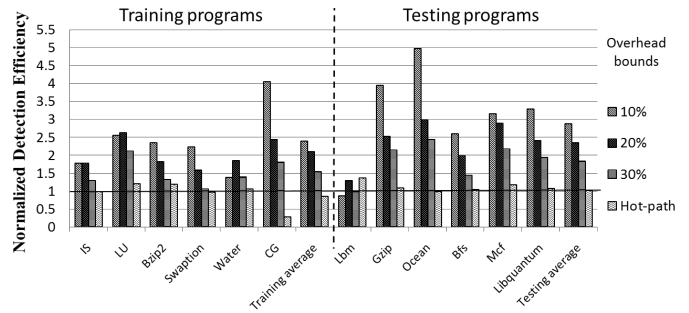
In summary, our technique significantly outperforms both full-duplication and hot-path duplication in providing better detection efficiency, for much lower performance overhead bounds.



(a) The SDC coverages with error bars at the 95% confidence interval. 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 overhead of full duplication and hot-path duplication



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

Figure 9: The results for different performance overhead bounds, hot-path duplication and full duplication. The X-axis shows the training and testing programs.

7. Related Work

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

Duplication based techniques: SWIFT [19] 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 9c, *SDCTune* 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. [8], and Khudia et al. [12] 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 [4, 6, 14, 15, 23] 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. [23] 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. [21] 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* with their technique.

Invariant based techniques [7, 18, 20] 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. Conclusion

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 model *SDCTune* based on the heuristics. We then use *SDCTune* to guide the selection of instructions to be protected with error detectors under a given performance overhead. Our results show that *SDCTune* is highly accurate at predicting the relative SDC rates of applications, and the detectors inserted using our technique outperform both full duplication and hot-path duplication by a factor of 0.83 to 1.87x in detection efficiency.

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 that of the U.S. Government. We thank our shepherd Muhammad Shafique, for his insightful comments and suggestions.

References

[1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, et al. The NAS parallel benchmarks. *HPCA*, 1991.
[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, 2008.
[3] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *MICRO*, 2005.
[4] J. Cong and K. Gururaj. Assuring application-level correctness against soft errors. In *ICCAD*, 2011.
[5] C. Constantinescu. Intermittent faults and effects on reliability of integrated circuits. In *RAMS*, 2008.
[6] M. de Kruijf, S. Nomura, and K. Sankaralingam. Relax: An architectural framework for software recovery of hardware faults. In *ISCA.*, 2010.
[7] M. D. Ernst, J. Cockrell, W. G Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *Software Engineering, IEEE Transactions on*, 2001.
[8] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: probabilistic soft error reliability on the cheap. In *ASPLOS*, 2010.
[9] S. K. S. Hari, S. V. Adve, and H. Naeimi. Low-cost program-level detectors for reducing silent data corruptions. In *DSN*, 2012.
[10] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer*, 2000.
[11] M. Hiller, A. Jhumka, and N. Suri. On the placement of software mechanisms for detection of data errors. In *DSN*, 2002.
[12] D. S. Khudia, G. Wright, and S. Mahlke. Efficient soft error protection for commodity embedded microprocessors using profile information. In *LCTES*, 2012.
[13] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO.*, 2004.
[14] K. Lee, A. Shrivastava, I. Issenin, N. Dutt, and N. Venkatasubramanian. Partially protected caches to reduce failures due to soft errors in multimedia applications. *IEEE Transactions on VLSI*, 2009.
[15] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. Flikker: Saving DRAM refresh-power through critical data partitioning. In *ASPLOS*, 2011.
[16] T. Mason et al. LAMPVIEW: A loop-aware toolset for facilitating parallelization. *Master's thesis, Dept. of Electrical Engineeringi, Princeton University*, 2009.
[17] K. Pattabiraman, Z. Kalbarczyk, and R. K. Iyer. Application-based metrics for strategic placement of detectors. In *PRDC.*, 2005.
[18] 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 *EDCC.*, 2006.
[19] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In *CGO*, 2005.
[20] S. K. Sahoo, M.-L. Li, P. Ramachandran, S. V. Adve, V. S. Adve, and Y. Zhou. Using likely program invariants to detect hardware errors. In *DSN*, 2008.
[21] M. Shafique, S. Rehman, P. V. Aceituno, and J. Henkel. Exploiting program-level masking and error propagation for constrained reliability optimization. In *DAC*, 2013.
[22] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012.
[23] A. Thomas and K. Pattabiraman. Error detector placement for soft computation. In *DSN*, 2013.
[24] L. Wang, Z. Kalbarczyk, and R. 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.
[25] J. Wei, A. Thomas, G. Li, and K. Pattabiraman. Quantifying the accuracy of high-level fault injection techniques for hardware faults. In *DSN*, 2014.
[26] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA*, 1995.
[27] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *MICRO*, 1996.