Modeling Soft-Error Propagation in Programs

Guanpeng Li, Karthik Pattabiraman
University of British Columbia
{gpli, karthikp}@ece.ubc.ca

Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai
NVIDIA
{shari, misullivan, timothyt}@nvidia.com

***Abstract*—As technology scales to lower feature sizes, devices become more susceptible to soft errors. Soft errors can lead to silent data corruptions (SDCs), seriously compromising the reliability of a system. Traditional hardware-only techniques to avoid SDCs are energy hungry, and hence not suitable for commodity systems. Researchers have proposed selective software-based protection techniques to tolerate hardware faults at lower costs. However, these techniques either use expensive fault injection or inaccurate analytical models to determine which parts of a program must be protected for preventing SDCs. In this work, we construct a three-level model, TRIDENT, that captures error propagation at the static data dependency, control-flow and memory levels, based on empirical observations of error propagations in programs. TRIDENT is implemented as a compiler module, and it can predict both the overall SDC probability of a given program and the SDC probabilities of individual instructions, without fault injection. We find that TRIDENT is nearly as accurate as fault injection and it is much faster and more scalable. We also demonstrate the use of TRIDENT to guide selective instruction duplication to efficiently mitigate SDCs under a given performance overhead bound.**

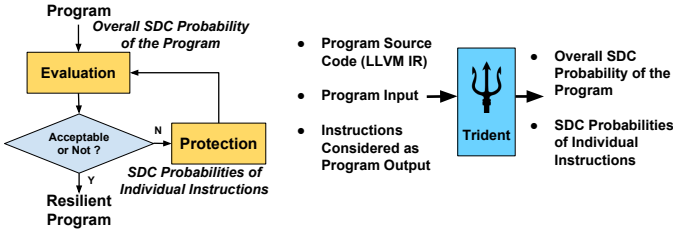
***Keywords*—Error Propagation, Soft Error, Silent Data Corruption, Error Resilience, Program Analysis**

I. INTRODUCTION

Transient hardware faults (i.e., soft errors) are predicted to increase in future computer systems due to growing system scale, progressive technology scaling, and lowering operating voltages [26]. In the past, such faults were masked through hardware-only solutions such as redundancy and voltage guard bands. However, these techniques are becoming increasingly challenging to deploy as they consume significant amounts of energy, and as energy is becoming a first-class constraint in processor design [6]. Therefore, researchers have postulated that future processors will expose hardware faults to the software and expect the software to tolerate them [24].

One consequence of such hardware errors is incorrect program output, or silent data corruptions (SDCs), which are very difficult to detect and can hence have severe consequences [26]. Studies have shown that a small fraction of the program states are responsible for almost all the error propagations resulting in SDCs, and so one can selectively protect these states to meet the target SDC probability while incurring lower energy and performance costs than full duplication techniques [10], [27]. Therefore, in the development of fault-tolerant applications (Figure 1a), it is important to estimate the SDC probability of a program – both in the aggregate, and on an individual instruction basis - to decide whether protection is required, and if so, to selectively protect the SDC-causing states of the program. This is the goal of our work.

Fault Injection (FI) has been commonly employed to estimate the SDC probabilities of programs. FI involves perturbing the program state to emulate the effect of a hardware fault and executing the program to completion to determine if the fault caused an SDC. However, real-world programs may consist of billions of dynamic instructions, and even a single execution of the program may take a long time. Performing thousands of FIs to get statistically meaningful results for each instruction takes too much time to be practical [13], [14]. As a result, researchers have attempted to analytically model error propagation to identify vulnerable instructions [10], [21], [27]. The main advantage of these analytical models is scalability, as the models usually do not require FIs, and they are fast to execute. However, most existing models suffer from a lack of accuracy, as they are limited to modeling faults in the normal (i.e., fault-free) control-flow path of the program. Since program execution is dynamic in nature, a fault can propagate to not only the data-dependencies of an instruction, but also to the subsequent branches (i.e., control flow) and memory locations that are dependent on it. This causes deviation from the predicted propagation, leading to inaccuracies. Unfortunately, tracking the deviation in control-flow and memory locations due to a fault often leads to state space explosion.



(a) Development Cycle (b) Workflow of TRIDENT

Fig. 1: Development of Fault-Tolerant Applications

This paper proposes a model, TRIDENT, for tracking error propagation in programs that addresses the above two challenges. The key insight in TRIDENT is that error propagation in dynamic execution can be decomposed into a combination of individual modules, each of which can be abstracted into probabilistic events. TRIDENT can predict both the overall SDC probability of a program and the SDC probability of individual instructions based on dynamic and static analysis of the program without performing FI. We implement TRIDENT in the LLVM compiler [17] and evaluate its accuracy and scalability vis-a-vis FI. *To the best of our knowledge, we are the first to propose a model to estimate the SDC probability of individual instructions and the entire program without performing any FIs.*

Our main contributions in this paper are as follows:

- Propose TRIDENT, a three-level model for tracking error propagation in programs. The levels are static-instruction, control-flow and memory levels, and they build on each other. The three-level model abstracts the data-flow of programs in the presence of faults.
- Compare the accuracy and scalability of TRIDENT with FI, to predict the SDC probability of individual instructions and that of the entire program.
- Demonstrate the use of TRIDENT to guide selective instruction duplication for configurable protection of programs from SDCs under a performance overhead.

The results of our experimental evaluation are as follows:

- The predictions of SDC probabilities using TRIDENT are statistically indistinguishable from those obtained through FI, both for the overall program and for individual instructions. On average, the overall SDC probability predicted by TRIDENT is 14.83% while the FI measured value is 13.59% across 11 programs.
- We also create two simpler models to show the importance of modeling control-flow divergence and memory dependencies - the first model considers neither, while the second considers control-flow divergence but not memory dependencies. The two simpler models predict the average SDC probabilities across programs as 33.85% and 23.76% respectively, which is much higher than the FI results.
- Compared to FI, whose cost is proportional to the number of injections, TRIDENT incurs a fixed cost, and a small incremental cost for each instruction sampled in the program. For example, TRIDENT takes about 16 minutes to calculate the *individual* SDC probabilities of about 1,000 static instructions, which is significantly faster than the corresponding FI experiments (which often take hours or even days).
- Using TRIDENT to guide selective instruction duplication reduces the overall SDC probability by 65% and 90% at 11.78% and 23.31% performance overheads, respectively (these represent $1/3$rd and $2/3$rd of the full-duplication overhead for the programs respectively). These reductions are higher than the corresponding ones obtained using the simpler models.

II. BACKGROUND

In this section, we first present our fault model, then define the terms we use and the compiler infrastructure we work with.

A. Fault Model

In this paper, we consider transient hardware faults that occur in the computational elements of the processor, including pipeline registers and functional units. We do not consider faults in the memory or caches, as we assume that these are protected with error correction code (ECC). Likewise, we do not consider faults in the processor's control logic as we assume that it is protected. Neither do we consider faults in the instructions' encodings. Finally, we assume that the program does not jump to arbitrary illegal addresses due to faults during the execution, as this can be detected by control-flow checking techniques [23]. However, the program may take a faulty legal branch (the execution path is legal but the branch direction can be wrong due to faults propagating to it). Our fault model is in line with other work in the area [7], [10], [13], [21].

B. Terms and Definitions

Fault Occurrence: The event corresponding to the occurrence of a hardware fault in the processor. The fault may or may not result in an error.

Fault Activation: The event corresponding to the manifestation of the fault to the software, i.e., the fault becomes an error and corrupts some portion of the software state (e.g., register, memory location). The error may or may not result in a failure (i.e., SDC, crash or hang).

Crash: The raising of a hardware trap or exception due to the error, because the program attempted to perform an action it should not have (e.g., read outside its memory segments). The OS terminates the program as a result.

Silent Data Corruption (SDC): A mismatch between the output of a faulty program run and that of an error-free execution of the program.

Benign Faults: Program output matches that of the error-free execution even though a fault occurred during its execution. This means either the fault was masked or overwritten by the program.

Error propagation: Error propagation means that the fault was activated, and has affected some other portion of the program state, say 'X'. In this case, we say the fault has propagated to state X. We focus on the faults that affect the program state and therefore consider error propagation at the application level.

SDC Probability: We define the SDC probability as the probability of an SDC given that the fault was activated – other work uses a similar definition [10], [14], [18], [30].

C. LLVM Compiler

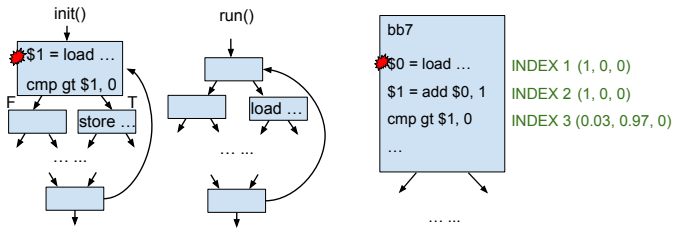
In this paper, we use the LLVM compiler [17] to perform our program analysis and FI experiments and to implement our model. Our choice of LLVM is motivated by three reasons. First, LLVM uses a typed intermediate representation (IR) that can easily represent source-level constructs. In particular, it preserves the names of variables and functions, which makes source mapping feasible. This allows us to perform a fine-grained analysis of which program locations cause certain failures and map them to the source code. Second, LLVM IR is a platform-neutral representation that abstracts out many low-level details of the hardware and assembly language. This greatly aids in portability of our analysis to different architectures and simplifies the handling of the special cases in different assembly language formats. Finally, LLVM IR has been shown to be accurate for doing FI studies [30], and there are many fault injectors developed for LLVM [3], [20], [25], [30]. Many of the papers we compare our technique with in this paper also use LLVM infrastructure [9], [10]. Therefore, in this paper, when we say instruction, we mean an instruction at the LLVM IR level.

III. The Challenge

We use the code example in Figure 2a to explain the main challenge of modeling error propagation in programs. The code is from *Pathfinder* [5], though we make minor modifications for clarity and remove some irrelevant parts. The figure shows the control-flow graphs (CFGs) of two functions: *init()* and *run()*. There is a loop in each function: the one in *init()* updates an array, and the one in *run()* reads the array for processing. The two functions *init()* and *run()* are called in order at runtime. In the CFGs, each box is a basic block and each arrow indicates a possible execution path. In each basic block, there is a sequence of statically data-dependent instructions, or a *static data-dependent instruction sequence*.

Assume that a fault occurs at the instruction writing to *$1* in the first basic block in *init()*. The fault propagates along its *static data-dependent instruction sequence* (from *load* to *cmp*). At the end of the sequence, if the fault propagates to the result of the comparison instruction, it will go beyond the static data dependency and cause the control-flow of the program to deviate from the fault-free execution. For example, in the fault-free execution, the *T* branch is supposed to be taken, but due to the fault, the *F* branch is taken. Consequently, the basic blocks under the *T* branch including the store instruction will not be executed, whereas subsequent basic blocks dominated by the *F* branch will be executed. This will load the wrong value in *run()*, and hence the fault will continue to propagate and it may reach the program's output resulting in an SDC.

We identify the following three challenges in modeling error propagation: (1) Statically modeling error propagation in dynamic program execution requires a model that abstracts the program data-flow in the presence of faults. (2) Due to the random nature of soft errors, a fault may be activated at any dynamic branch and cause control-flow divergence in execution from the fault-free execution. In any divergence, there are numerous possible execution paths the program may take, and tracking all of these paths is challenging. One can emulate all possible paths among the dynamic executions at every dynamic branch and figure out which fault propagates where in each case. However, this rapidly leads to state space explosion. (3) Faults may corrupt memory locations and hence continue to propagate through memory operations. Faulty memory values can be read by (multiple) load instructions at runtime and written to other memory locations as execution progresses. There are enormous numbers of store and load instructions in a typical program execution, and tracing error propagations among these memory dependencies requires constructing a huge data dependency graph, which is very expensive.



(a) Example of Propagation (b) Propagation in f_s

Fig. 2: Running Example

As we can see in the above example, if we do not track error propagations beyond the static data dependencies and instead stop at the comparison instruction, we may not identify all the cases that could lead to SDCs. Moreover, if control-flow divergence is ignored when modeling, tracking errors in memory is almost impossible, as memory corruptions often hide behind control-flow divergence, as shown in the above example. Existing modeling techniques capture neither of these important cases, and their SDC prediction accuracies suffer accordingly. In contrast, TRIDENT captures both the control-flow divergences and the memory corruptions that potentially arise as a result of the divergence.

IV. TRIDENT

In this section, we first introduce the inputs and outputs of our proposed model, TRIDENT, and then present the overall structure of the model and the key insights it leverages. Finally we present the details of TRIDENT using the running example.

A. Inputs and Outputs

The workflow of TRIDENT is shown in Figure 1b. We require the user to supply three inputs: (1) The program code compiled to the LLVM IR, (2) a program input to execute the program and obtain its execution profile (similar to FI methods, we also require a single input to obtain runtime information), and (3) the output instruction(s) in the program that are used for determining if a fault resulted in an SDC. For example, the user can specify *printf* instructions that are responsible for the program's output and used to determine SDCs. On the other hand, *printf*s that log debugging information or statistics about the program execution can be excluded as they do not typically determine SDCs. Without this information, all the output instructions are assumed to determine SDCs by default.

TRIDENT consists of two phases: (1) Profiling and (2) inferencing. In the profiling phase, TRIDENT executes the program, performing dynamic analysis of the program to gather information such as the count and data dependency of instructions. After collecting all the information, TRIDENT starts the inferencing phase which is based on static analysis of the program. In this phase, TRIDENT automatically computes (1) the SDC probabilities of individual instructions, and (2) the overall SDC probability of the program. In the latter case, the user needs to specify the number of sampled instructions when calculating the overall SDC probability of the program, in order to balance the time for analysis with accuracy.

B. Overview and Insights

Because error propagation follows program data-flow at runtime, we need to model program data-flow in the presence of faults at three levels: (1) Static-instruction level, which corresponds to the execution of a *static data-dependent instruction sequence* and the transfer of results between registers. (2) Control-flow level, when execution jumps to another program location. (3) Memory level, when the results need to be transferred back to memory. TRIDENT is divided into three sub-models to abstract the three levels, respectively, and we use f_s , f_c and f_m to represent them. The main algorithm of TRIDENT tracking error propagation from a given location to the program output is summarized in Algorithm 1.

Static-Instruction Sub-Model (f_s): First, f_s is used to trace error propagation of an arbitrary fault activated on a *static data-dependent instruction sequence*. It determines the propagation probability of the fault from where it was activated to the end of the sequence. For example, in Figure 2b, the model computes the probability of the fault propagating to the result of the comparison instruction given that the fault is activated at the load instruction (Line 4 in Algorithm 1). Previous models trace error propagation in data dependant instructions based on the dynamic data dependency graph (DDG) which records the output and operand values of each dynamic instruction in the sequence [9], [27]. However, such detailed DDGs are very expensive to generate and process, and hence the models do not scale. f_s avoids generating detailed dynamic traces and instead computes the propagation probability of each static instruction based on its average case at runtime to determine the error propagation in a *static data-dependent instruction sequence*. Since each static instruction is designed to manipulate target bits in a pre-defined way, the propagation probability of each static instruction can be derived. We can then aggregate the probabilities to calculate the probability of a fault propagating from a given instruction to another instruction within the same *static data-dependent instruction sequence*.

Control-Flow Sub-Model (f_c): As explained, a fault may propagate to branches and cause the execution path of the program to diverge from its fault-free execution. We divide the propagation into two phases after divergence: The first phase, modeled by f_c , attempts to figure out which dynamic store instructions will be corrupted at what probabilities if a conditional branch is corrupted (Lines 3-5 in Algorithm 1). The second phase traces what happens if the fault propagates to memory, and is modeled by f_m . The key observation is that error propagation to memory through a conditional branch that leads to control-flow divergence can be abstracted into a few probabilistic events based on branch directions. This is because the probabilities of the incorrect executions of store instructions are decided by their execution paths and the corresponding branch probabilities. For example, in the function *init()* in Figure 2a, if the comparison instruction takes the *F* branch, the store instruction is not supposed to be executed, but if a fault modifies the direction of the branch to the *T* branch, then it will be executed and lead to memory corruption. A similar case occurs where the comparison instruction is supposed to take the *T* branch. Thus, the store instruction is corrupted in either case.

Memory Sub-Model (f_m): f_m tracks the propagation from corrupted store instructions to the program output, by tracking memory dependencies of erroneous values until the output of the program is reached. During the tracking, other sub-models are recursively invoked where appropriate. f_m then computes the propagation probability from the corrupted store instruction to the program output (Lines 7-9 in Algorithm 1). A memory data-dependency graph needs to be generated for tracing propagations at the memory level because we have to know which dynamic load instruction reloads the faulty data previously written by an erroneous store instruction (if any). This graph can be expensive to construct and traverse due to the huge number of the dynamic store and load instructions in the program. However, we find that the graph can be pruned by removing redundant dependencies between symmetric loops, if there are any. Consider as an example the two loops in *init()* and *run()* in Figure 2a. The first loop updates an array, and the second one reads from the same array. Thus, there is a memory dependence between every pair of iterations of the two loops. In this case, instead of tracking every dependency between dynamic instructions, we only track the aggregate dependencies between the two loops. As a result, the memory dependence graph needs only two nodes to project the dependencies between the stores and loads in their iterations.

Algorithm 1: The Core Algorithm in TRIDENT

```
1 sub-models f_s , f_c , and f_m ;
  Input  : I: Instruction where the fault occurs
  Output: P_SDC: SDC probability
2 p_s = f_s (I);
3 if inst. sequence containing I ends with branch I_b then
4 |   // Get the list of stores corrupted and their prob.
5 |   [<I_c, p_c>, ...] = f_c (I_b);
6 |   // Maximum propagation prob. is 1
7 |   Foreach(<I_c, p_c>): P_SDC += p_s * p_c * f_m (I_c);
8 else if inst. sequence containing I ends with store I_s
  then
9 |   P_SDC = p_s*f_m (I_s);
```

C. Details: Static-Instruction Sub-Model (f_s)

Once a fault is activated at an executed instruction, it starts propagating on its *static data-dependent instruction sequence*. Each sequence ends with a store, a comparison or an instruction of program output. In these sequences, the probability that each instruction masks the fault during the propagation can be determined by analyzing the mechanism and operand values of the instruction. This is because instructions often manipulate target bits in predefined ways.

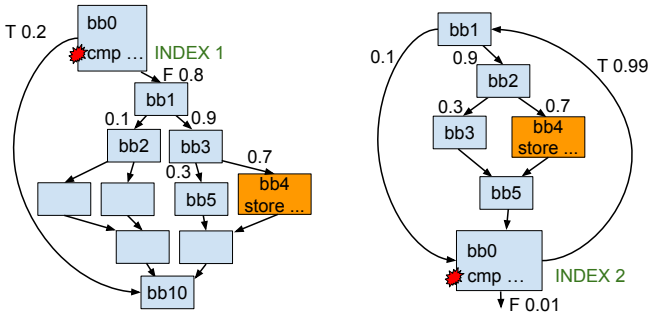
Given a fault that occurs and is activated on an instruction, f_s computes the probability of error propagation when the execution reaches the end of the static computation sequence of the instruction. We use a code example in Figure 2b to explain the idea. The code is from *Pathfinder* [5], and shows a counter being incremented until a positive value is reached. In Figure 2b, *INDEX 1-3* form a *static data-dependent instruction sequence*, which an error may propagate along. Assuming a fault is activated at *INDEX 1* and affects *$1*, the goal of f_s is to tell the probabilities of propagation, masking and crash after the execution of *INDEX 3*, which is the last instruction on the sequence. f_s traces the error propagation from *INDEX 1* to *INDEX 3* by aggregating the propagation probability of each instruction on the sequence. We use a tuple for each instruction to represent its probabilities which are shown in the brackets on the right of each instruction in Figure 2b. There are three numbers in each tuple, which are the probabilities of propagation, masking and crash respectively, given that an operand of the instruction is erroneous (we explain how to compute these later). For example, for *INDEX 3*, *(0.03, 0.97, 0)* means that the probability of the error continuing to propagate when *INDEX 3* is corrupted is 0.03, whereas 0.97 is the probability that the error will be masked and not propagate beyond *INDEX 3*. Finally, the probability of a crash at *INDEX 3*, in this case, is 0. Note that the probabilities in each tuple should sum to 1.

After calculating the individual probabilities, f_s aggregates the propagation probability in each tuple of *INDEX 1, 2* and

3 to calculate the propagation probability from *INDEX 1* to *INDEX 3*. That is given by *1*1*0.03=3%* for the probability of propagation, and the probabilities of masking and crash are *97%* and *0%* respectively. Thus, if a fault is activated at *INDEX 1*, there is a 3% of probability that the branch controlled by *INDEX 3* will be flipped, causing a control-flow divergence.

We now explain how to obtain the tuple for each instruction. Each tuple is approximated based on the mechanism of the instruction and/or the profiled values of the instruction's operands. We observe that there are only a few types of instructions that have non-negligible masking probabilities: they are comparisons (e.g., *CMP*), logic operators (e.g., *XOR*) and casts (e.g., *TRUNC*). We assume the rest of instructions neither move nor discard corrupted bits - this is a heuristic we use for simplicity (we discuss its pros and cons in Section VII-A).

In the example in Figure 2b, the branch direction will be modified based on whether *INDEX 3* computes a positive or negative value. In either case, only a flip of the sign bit of *$1* will modify the branch direction. Hence, the error propagation probability in the tuple of *INDEX 3* is $1/32 = 0.03$, assuming a 32-bit data width. We derive crash probabilities in the tuples for instructions accessing memory (i.e., load and store instructions). We consider crashes that are caused by program reading or writing out-of-bound memory addresses. Their probabilities can be approximated by profiling memory size allocated for the program (this is found in the */proc/* filesystem in Linux). Prior work [9] has shown that these are the dominant causes of crashes in programs due to soft errors.



(a) Example of NLT (b) Example of LT

Fig. 3: NLT and LT Examples of the CFG

D. Details: Control-Flow Sub-Model (f_c)

Recall that the goal of f_c is to figure out which dynamic store instructions will be corrupted and at what probabilities, if a conditional branch is corrupted. We classify all comparison instructions that are used in branch conditions into two types based on whether they terminate a loop. The two types are (1) *Non-Loop-Terminating cmp* (NLT), and (2) *Loop-Terminating cmp* (LT). Figure 3 shows two Control Flow Graphs (CFGs), one for each case. We also profile the branch probability of each branch and mark it beside each corresponding branch for our analysis purpose. For example, if a branch probability is 0.2, it means during the execution there is 20% probability the branch is taken. We will use the two examples in Figure 3 to explain f_c in each case.

1) Non-Loop-Terminating CMP (NLT): If a comparison instruction does not control the termination of a loop, it is NLT. In Figure 3a, *INDEX 1* is a NLT, dominating a store instruction in *bb4*. There are two cases for the store considered as being corrupted in f_c : (1) The store is not executed while it should be executed in a fault-free execution. (2) The store is executed while it should *not* be executed in a fault-free execution. Combining these cases, the probability of the store instruction being corrupted can be represented by Equation 1.

$$P_c = P_e / P_d \quad (1)$$

In the equation, P_c is the probability of the store being corrupted, P_e is the execution probability of the store instruction in fault-free execution, and P_d is the branch probability of which direction dominates the store.

We illustrate how to derive the above equation using the example in Figure 3a. There are two legal directions a branch can take. In the first case, the branch of *INDEX 1* is supposed to take the *T* branch at the fault-free execution (20% probability), but the *F* branch is taken instead due to the corrupted *INDEX 1*. The store instruction in *bb4* will be executed when it is not supposed to be executed and will hence be corrupted. The probability that the store instruction is executed in this case is calculated as $0.2 * 0.9 * 0.7 = 0.126$ based on the probabilities on its execution path (*bb0-bb1-bb3-bb4*). In the second case, if the *F* branch is supposed to be taken in a fault-free execution (80% probability), but the *T* branch is taken instead due to the fault, the store instruction in *bb4* will not be executed, while it is supposed to have been executed in some execution path in the fault-free execution under the *F* branch. For example, in the fault-free execution, path *bb0-bb1-bb3-bb4* will trigger the execution of the store. Therefore, the probability of the store instruction being corrupted in this case is $0.8 * 0.9 * 0.7 = 0.504$. Therefore, adding the two cases together, we get f_c in this example as $0.126 + 0.504 = 0.63$. The Equation 1 is simplified by integrating the terms in the calculations. In this example, in Equation 1, P_e is $0.8 * 0.9 * 0.7$ (*bb0-bb1-bb3-bb4*), P_d is 0.8 (*bb0-bb1*), thus P_c is $0.8 * 0.9 * 0.7 / 0.8 = 0.63$. Note that if the branch immediately dominates the store instruction, then the probability of the store being corrupted is 1, as shown by the example in Figure 2.

2) Loop-Terminating CMP (LT): If a comparison instruction controls the termination of a loop, it is LT. For example, in Figure 3b, the back-edge of *bb0* forms a loop, which can be terminated by the condition computed by *INDEX 2*. Hence, *INDEX 2* is a LT. We find that the probability of the store instruction being corrupted can be represented by Equation. 2.

$$P_c = P_b * P_e \quad (2)$$

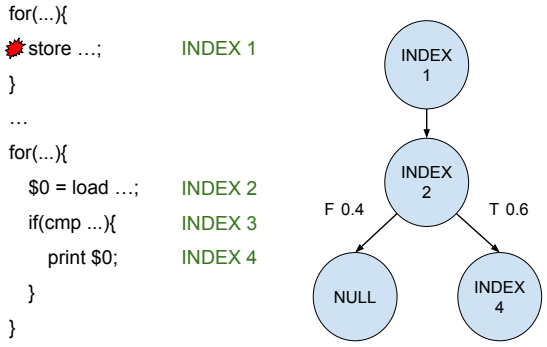
P_c is the probability that a dynamic store instruction is corrupted if the branch is modified, P_b is the execution probability of the back-edge of the branch, and P_e is the execution probability of the store instruction dominated by the back-edge.

We show the derivation of the above equation using the example in Figure 3b. In the first case, if the *T* branch (the loop back-edge) is supposed to be taken in a fault-free execution (99% probability), the store instruction in *bb4* may or may not execute, depending on the branch in *bb2*. But if a fault modifies the branch of *INDEX 2*, the store will certainly not execute. So we need to omit the probabilities that the store is not executed in the fault-free execution to calculate the corruption probability of the store. They are $0.99 * 0.9 * 0.3 = 0.27$ for the path *bb0-bb1-bb2-bb3* and

$0.99 * 0.1 = 0.099$ for *bb0-bb1-bb0*. Hence, the probability of a corrupted store in this case is $0.99 - 0.27 - 0.099 = 0.62$. In the second case where the *F* branch should be taken in a fault-free execution (1% probability), if the fault modifies the branch, the probability of a corrupted store instruction is $0.01 * 0.9 * 0.7 = 0.0063$. Note that this is usually a very small value which can be ignored. This is because the branch probabilities of a loop-terminating branch are usually highly biased due to the multiple iterations of the loop. So the total probability in this example is approximated to be 0.62, which is what we calculated above. Equation 2 is simplified by integrating and cancelling out the terms in the calculations. In this example, P_b is 0.99 (*bb0-bb1*), P_e is 0.7*0.9 (*bb1-bb2-bb4*), and thus P_c is $0.99 * 0.7 * 0.9 = 0.62$.

E. Details: Memory Sub-Model (f_m)

Recap that f_m reports the probability for the error to propagate from the corrupted memory locations to the program output. The idea is to represent memory data dependencies between the load and store instructions in an execution, so that the model can trace the error propagation in the memory.



(a) Code Example (b) Dependency Graph

Fig. 4: Examples for Memory Sub-model

We use the code example in Figure 4a to show how we prune the size of the memory dependency graph in f_m by removing redundant dependencies (if any). There are two inner loops in the program. The first one executes first, storing data to an array in memory (*INDEX 1*). The second loop executes later, loading the data from the memory (*INDEX 2*). Then the program makes some decision (*INDEX 3*) and decides whether the data should be printed (*INDEX 4*) to the program output. Note that the iterations between loops are symmetric in the example, as both manipulate the same array (one updates, and the other one reloads). This is often seen in programs because they tend to manipulate data in blocks due to spatial locality. In this example, if one of the dynamic instructions of *INDEX 1* is corrupted, one of the dynamic instructions of *INDEX 2* must be corrupted too. Therefore, instead of having one node for every dynamic load and store in the iterations of the loop executions, we need only two nodes in the graph to represent the dependencies. The rest of the dependencies in the iterations are redundant, and hence can be removed from the graph as they share the same propagation. The dependencies between dynamic loads and stores are tracked at runtime with their static indices and operand memory addresses recorded. The redundant dependencies are pruned when repeated static load and store pairs are detected.

We show the memory data dependency graph of f_m for the code example in Figure 4b. Assume each loop is invoked once with many iterations. We create a node for the store (*INDEX 1*), load (*INDEX 2*) and printf (*INDEX 3*, as program output) in the graph. We draw an edge between nodes to present their dependencies. Because *INDEX 3* may cause divergence of the dependencies and hence error propagation, we weight the propagation probability based on its execution probability. We place a *NULL* node as a placeholder indicating masking if *F* branch is taken in *INDEX 3*. Note that an edge between nodes may also represent a *static data-dependent instruction sequence*, e.g., the edge between *INDEX 2* and *INDEX 4*. Therefore, f_s is recursively called every time a *static data-dependent instruction sequence* is encountered. We then aggregate the propagation probabilities starting from the node of *INDEX 1* to each leaf node in the graph. Each edge may have different propagation probabilities to aggregate – it depends on what f_s outputs if a *static data-dependent instruction sequence* is present on the edge. In this example, assume that f_s always outputs 1 as the propagation probability for each edge. Then, the propagation probability to the program output (*INDEX 4*), if one of the store (*INDEX 1*) in the loop is corrupted, is $1*1*1*0.6/(0.4+0.6)+1*1*0*0.4/(0.4+0.6) = 0.6$. The zero in the second term represents the masking of the *NULL* node. As an optimization, we memoize the propagation results calculated for store instructions to speed up the algorithm. For example, if later the algorithm encounters *INDEX 1*, we can use the memoized results, instead of recomputing them. We will evaluate the effectiveness of the pruning in Section V-C.

Floating Point: When we encounter any floating point data type, we apply an additional masking probability based on the output format of the floating point data. For example, in benchmarks such as *Hotspot*, the *float* data type is used. By default, *Float* carries 7-digit precision, but in (many) programs' output, a "%g" parameter is specified in *printf* which prints numbers with only 2-digit precision. Based on the specification of IEEE-754 [1], we assume that only the mantissa bits (23 bits in *Float*) may affect the 5 digits that are cut off in the precision. This is because bit-flips in exponential bits likely cause large deviations in values, and so cutting-off the 5 digits in the precision is unlikely to mask the errors in the exponent. We also assume that each mantissa bit has equal probability to affect the missing 5 digits of precision. In that way, we approximate the propagation probability to be *((32-23)+23*(2/7))/32 = 48.66%*. We apply this masking probability on top the propagation probabilities, for *Float* data types used with the non-default format of *printf*.

V. Evaluation

In this section, we evaluate TRIDENT in terms of its accuracy and scalability. To evaluate accuracy, we use TRIDENT to predict overall SDC probabilities of programs as well as the SDC probabilities for individual instructions, and compare them with those obtained using FI and the simpler models. To evaluate scalability, we measure the time for executing TRIDENT, and compare it with the time taken by FI. We first present the experimental setup and then the results. We also make TRIDENT and the experimental data publicly available[1].

[1] https://github.com/DependableSystemsLab/Trident

A. Experimental Setup

1) Benchmarks: We choose eleven benchmarks from common benchmark suites [4], [5], [15], and publicly available scientific programs [2], [16], [29] — they are listed in Table I. Our benchmark selection is based on three criteria: (1) Diversity of domains and benchmark suites, (2) whether we can compile with our LLVM infrastructure, and (3) whether fault injection experiments of the programs can finish within a reasonable amount of time. We compiled each benchmark with LLVM with standard optimizations (-O2).

TABLE I: Characteristics of Benchmarks

Benchmark	Suite/Author	Area	Program Input
Libquantum	SPEC	Quantum computing	33 5
Blackscholes	Parsec	Finance	in_4.txt
Sad	Parboil	Video encoding.	reference.bin frame.bin
Bfs	Parboil	Graph traversal	graph_input.dat
Hercules	Carnegie Mellon University	Earthquake simulation	scan simple_case.e
Lulesh	Lawrence Livermore National Laboratory	Hydrodynamics modeling	-s 1 -p
PuReMD	Purdue University	Reactive molecular dynamics simulation	geo ffield control
Nw	Rodinia	DNA sequence optimization	2048 10 1
Pathfinder	Rodinia	Dynamic programming	1000 10
Hotspot	Rodinia	Temperature and power simulation	64 64 1 1 temp_64 power_64
Bfs	Rodinia	Graph traversal	graph4096.txt

2) FI Method: We use LLFI [30] which is a publicly available open-source fault injector to perform FIs at the LLVM IR level on these benchmarks. LLFI has been shown to be accurate in evaluating SDC probabilities of programs compared to assembly code level injections [30]. We inject faults into the destination registers of the executed instructions to simulate faults in the computational elements of the processor as per our fault model. Further, we inject single bit flips as these are the de-facto model for emulating soft errors at the program level, and have been found to be accurate for SDCs [25]. There is only one fault injected in each run, as soft errors are rare events with respect to the time of execution of a program. Our FI method ensures that all faults are activated, i.e., read by an instruction of the program, as we define SDC probabilities based on the activated instructions (Section II). The FI method is in line with other papers in the area [3], [9], [18], [30].

B. Accuracy

We design two experiments to evaluate the accuracy of TRIDENT. The first experiment examines the prediction of overall SDC probabilities of programs, and the second examines predicted SDC probabilities of individual instructions. In the experiments, we compare the results derived from TRIDENT with those from the two simpler models and FI. As described earlier, TRIDENT consists of three sub-models in order: f_s , f_c and f_m . We create two simpler models to (1) understand the accuracy gained by enabling each sub-model and (2) as a proxy to investigate other models, which often lack modeling beyond static data dependencies (Section VII-C performs a more detailed comparison with prior work). We first disable f_m in TRIDENT, leaving the two sub-models f_s and f_c enabled, to create a model: $f_s + f_c$. We then further remove f_c to create the second simplified model which only has f_s enabled, which we represent as f_s .

1) Overall SDC probability: To evaluate the overall SDC probability of a given program, we use statistical FI. We measure error bars for statistical significance at the 95% confidence level. We randomly sample 3,000 dynamic instructions for FIs (one fault per run) as these yield tight error bars at the 95% confidence level ($\pm$0.07% to $\pm$1.76%) - this is in line with other work that uses FI. We calculate SDC probability of each program based on how many injected faults result in SDC. We then use TRIDENT, as well as the two simpler models, to predict the SDC probability of each program, and compare the results with those from FI. To ensure fair comparison, we sample 3,000 instructions in our models as well (Section IV-A).

The results are shown in Figure 5. We use *FI* to represent the FI method, TRIDENT for our three-level model, and *fs+fc* and *fs* for the two simpler models. We find TRIDENT prediction matches the overall SDC probabilities obtained through FI, with a maximum difference of 14.26% in *Sad*, and a minimum difference of 0.11% in *Blackscholes*, both in percentage points. This gives a *mean absolute error* of 4.75% in overall SDC prediction. On the other hand, $f_s + f_c$ and f_s have a *mean absolute error* of 19.56% and 15.13% respectively compared to FI – *more than 4 and 3 times higher than those obtained using the complete three-level model.* On average, $f_s + f_c$ and f_s predict the overall SDC probability as 33.85% and 23.76% across the different programs, whereas TRIDENT predicts it to be 14.83%. The SDC probability obtained from FI is 13.59%, which is much more in line with the predictions of TRIDENT.

We observe that in *Sad, Lulesh* and *Pathfinder*, TRIDENT encounters relatively larger differences between the prediction and the FI results (14.26%, 7.48% and 8.87% respectively). The inaccuracies are due to a combination of gaps in the implementation, assumptions, and heuristics we used in TRIDENT. We discuss them in Section VII-A.

To compare the results more rigorously, we use a paired T-test experiment [28] to determine how similar the predictions of the overall SDC probabilities by TRIDENT are to the FI results.[2] Since we have 11 benchmarks, we have 11 sets of paired data with one side being FI results and the other side being the prediction values of TRIDENT. The null hypothesis is that there is no statistically significant difference between the results from FIs and the predicted SDC probabilities by TRIDENT in the 11 benchmarks. We calculate the p-value in the T-test as 0.764. By the conventional criteria (p-value$>$0.05), we fail to reject the null hypothesis, indicating that the predicted overall SDC probabilities by TRIDENT are not statistically different from those obtained by FI.

We find that the model $f_s + f_c$ always over-predicts SDCs compared with TRIDENT. This is because an SDC is assumed once an error propagates to store instructions, which is not always the case, as it may not propagate to the program output. On the other hand, f_s may either over-predict SDCs

[2]We have verified visually that the differences between the two sides of every pair are approximately normally distributed in all the T-test experiments we conduct, which is the requirement for validity of the T-test.

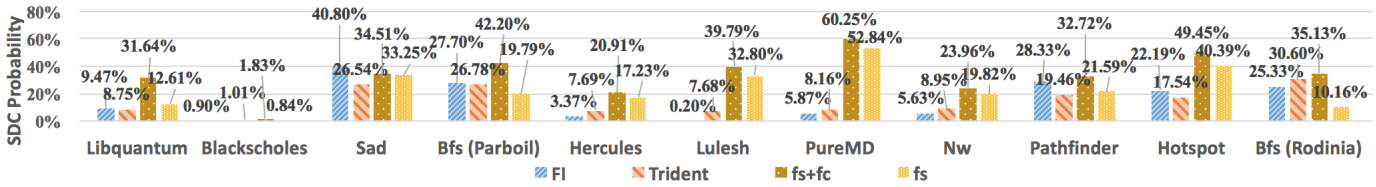


Fig. 5: Overall SDC Probabilities Measured by FI and Predicted by the Three Models (Margin of Error for FI: ±0.07% to ±1.76% at 95% Confidence)

(e.g., Libquantum, Hercules) because an SDC is assumed once an error directly hits any *static data-dependent instruction sequence* ending with a store, or under-predict them (e.g., Bfs, Blackscholes) because error propagation is not tracked after control-flow divergence.

2) SDC Probability of Individual Instructions: We now examine the SDC probabilities of individual instructions predicted by TRIDENT and compare them to the FI results. The number of static instructions per benchmark varies from 76 to 4,704, with an average of 944 instructions. Because performing FIs into each individual instruction is very time-consuming, we choose to inject 100 random faults per instruction to bound our experimental time. We then input each static instruction to TRIDENT, as well as the two simpler models ($f_s + f_c$ and f_s), to compare their predictions with the FI results. As before, we conduct paired T-test experiments [28] to measure the similarity (or not) of the predictions to the FI results. The null hypothesis for each of the three models in each benchmark is that there is no difference between the FI results and the predicted SDC probability values in each instruction.

TABLE II: p-values of T-test Experiments in the Prediction of Individual Instruction SDC Probability Values ($p > 0.05$ indicates that we are not able to reject our null hypothesis – the counter-cases are shown in red)

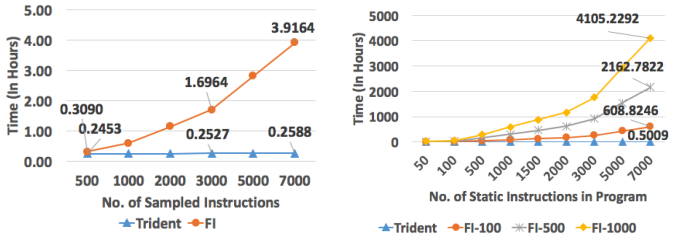
Benchmark	TRIDENT	fs+fc	fs
Libquantum	0.602	0.000	0.000
Blackscholes	0.392	0.173	0.832
Sad	0.000	0.003	0.000
Bfs (Parboil)	0.893	0.000	0.261
Hercules	0.163	0.000	0.003
Lulesh	0.000	0.000	0.000
PureMD	0.277	0.000	0.000
Nw	0.059	0.000	0.000
Pathfinder	0.033	0.130	0.178
Hotspot	0.166	0.000	0.000
Bfs (Rodinia)	0.497	0.001	0.126
No. of rejections	**3/11**	**9/11**	**7/11**

The p-values of the experiments are listed in the Table II. At the 95% confidence level, using the standard criteria ($p > 0.05$), we are not able to reject the null hypothesis in 8 out of the 11 benchmarks using TRIDENT in the predictions. This indicates that the predictions of TRIDENT are shown to be statistically indistinguishable from the FI results in most of the benchmarks we used. The three outliers for TRIDENT again are *Sad, Lulesh* and *Pathfinder*. Again, even though the individual instructions' SDC probabilities predicted are statistically distinguishable from the FI results, these predicted values are still reasonably close to the FI results. In contrast, when using $f_s + f_c$ and f_s to predict SDC probabilities for each individual instruction, there are only 2 and 4 out of the 11 benchmarks having p-values greater than 0.05, indicating that the null hypotheses cannot be rejected for most of the benchmarks. *In other words, the predictions from the simpler models for individual instructions are (statistically) significantly different from the FI results.*

C. Scalability

In this section, we evaluate the scalability of TRIDENT to predict the overall SDC probabilities of programs and the SDC probabilities of individual instructions, and compare it to FI. By scalability, we mean the ability of the model to handle large numbers of instruction samples in order to obtain tighter bounds on the SDC probabilities. In general, the higher the number of sampled instructions, the higher the accuracy and hence the tighter are the bounds on SDC probabilities for a given confidence level (e.g., 95% confidence). This is true for both TRIDENT and for FI. The number of instructions sampled for FI in prior work varies from 1,000 [30] to a few thousands [9], [10], [20]. We vary the number of samples from 500 to 7,000. The number of samples is equal to the number of FI trials as one fault is injected per trial.

Note that the total computation is proportional to both the time and power required to run each approach. Parallelization will reduce the time spent, but not the power consumed. We assume there is no parallelization for the purpose of comparison in the case of TRIDENT and FI, though both TRIDENT and FI can be parallelized. Therefore, the computation can be measured by the wall-clock time.



(a) Overall SDC Probability (b) Instruction SDC Probability

Fig. 6: Computation Spent to Predict SDC Probability

1) Overall SDC Probability: The results of the time spent to predict the overall SDC probability of program are shown in Figure 6a. The time taken in the figure is projected based on the measurement of one FI trial (averaged over 30 FI runs). As seen, the curve of FI time versus number of samples is much steeper than that of TRIDENT, which is almost flat. TRIDENT is 2.37 times faster than the FI method at 1,000 samples, it is 6.7 times faster at 3,000 samples and 15.13 times faster at 7,000 samples. From 500 to 7,000 samples, the time taken by TRIDENT increases only 1.06 times (0.2453 to 0.2588), whereas it increases 14 times (0.2453 to 3.9164) for FI - an exact linear increase. The profiling phase of TRIDENT takes 0.24 hours (or about 15 minutes) on average. This is a fixed cost incurred by TRIDENT regardless of the number of sampled instructions. However, once the model is built, the incremental cost of calculating the SDC probability of a new instruction is minimal (we only calculate the SDC probabilities on demand to save time). FI does not incur a noticeable fixed cost, but its time rapidly increases as the number of sampled instructions

increase. This is because FI has to run the application from scratch on each trial, and hence ends up being much slower than TRIDENT as the number of samples increase.

2) Individual Instructions: Figure 6b compares the average time taken by TRIDENT to predict SDC probabilities of individual instructions with FI, for different numbers of static instructions. We consider different numbers of samples for each static instruction chosen for FI: 100, 500 and 1,000 (as mentioned in Section IV-A, TRIDENT does not need samples for individual instructions' SDC probabilities). We denote the number of samples as a suffix for the FI technique. For example, *FI-100* indicates 100 samples are chosen for performing FI on individual instructions. We also vary the number of static instructions from 50 to 7,000 (this is the X-axis). As seen from the curves, the time taken by TRIDENT as the number of static instructions vary remains almost flat. On average, it takes 0.2416 hours at 50 static instructions, and 0.5009 hours at 7,000 static instructions, which is only about a 2X increase. In comparison, the corresponding increases for *FI-100* is 140X, which is linear with the number of instructions. Other FI curves experience even steeper increases as they gather more samples per instruction.

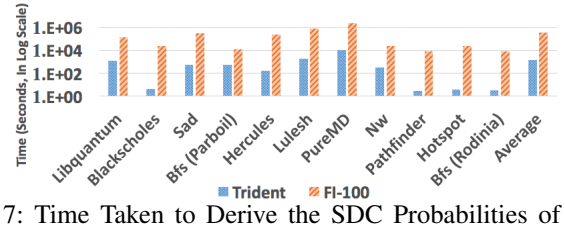


Fig. 7: Time Taken to Derive the SDC Probabilities of Individual Instructions in Each Benchmark

Figure 7 shows the time taken by TRIDENT and *FI-100* to derive the SDC probabilities of individual instructions in each benchmark (due to space constraints, we do not show the other FI values, but the trends were similar). As can be seen, there is wide variation in the times taken by TRIDENT depending on the benchmark program. For example, the time taken in *PureMD* is 2.893 hours, whereas it is 2.8 seconds in *Pathfinder*. This is because the time taken by TRIDENT depends on factors such as (1) the total number of static instructions, (2) the length of *static data-dependent instruction sequence*, (3) the number of dynamic branches that require profiling, and (4) the number of redundant dependencies that can be pruned. The main reason for the drastic difference between *PureMD* and *Pathfinder* is that we can prune only 0.08% of the redundant dependencies in the former, while we can prune 99.83% of the dependencies in the latter. On average, 61.87% of dynamic load and store instructions are redundant and hence removed from the memory dependency graph.

VI. USE CASE: SELECTIVE INSTRUCTION DUPLICATION

In this section, we demonstrate the utility of TRIDENT by considering a use-case of selectively protecting a program from SDC causing errors. The idea is to protect only the most SDC-prone instructions in a program so as to achieve high coverage while bounding performance costs. We consider instruction duplication as the protection technique, as it has been used in prior work [9], [10], [21]. The problem setting is as follows: given a certain performance overhead P, what static instructions should be duplicated in order to maximize the coverage for SDCs while keeping the overhead below P.

Solving the above problem involves finding the SDC probability of each instruction in the program in order to decide which set of instructions should be duplicated. It also involves calculating the performance overhead of duplicating the instructions. We use TRIDENT for the former, namely, to estimate the SDC probability of each instruction, without using FI. For the latter, we use the dynamic execution count of each instruction as a proxy for the performance overhead incurred by it. We then formulate the problem as a classical 0-1 knapsack problem [22], where the objects are the instructions and the knapsack capacity is represented by P, the maximum allowable performance overhead. Further, object profits are represented by the estimated SDC probability (and hence selecting the instruction means obtaining the coverage), and object costs are represented by the dynamic execution count of the instruction. Note that we assume that the SDC probability estimates of the instructions are independent of each other – while this is not necessarily true in practice, it keeps the model tractable, and in the worst case leads to conservative protection (i.e., over-protection). We use the dynamic programming algorithm for the 0-1 knapsack problem - this is similar to what prior work did [21].

For the maximum performance overhead P, we first measure the overhead of duplicating all the instructions in the program (i.e., full duplication) and set this as the baseline as it represents the worst-case overhead. The overheads are measured based on the wall-clock time of the actual execution of the duplicated programs (averaged on 3 executions each). We find that full duplication incurs an overhead of 36.18% across benchmarks. We consider 2 overhead bound levels, namely the $1/3$rd and $2/3$rd of the full duplication overheads, which are (1) 11.78% and (2) 23.31% respectively.

For each overhead level, our algorithm chooses the instructions to protect using the knapsack algorithm. The chosen instructions are then duplicated using a special pass in LLVM we wrote, and the duplication occurs at the LLVM IR level. Our pass also places a comparison instruction after each instruction protected to detect any deviations of the original computations and duplicated computations. If protected instructions are data dependent on the same *static data-dependent instruction sequence*, we only place one comparison instruction at the latter protected instruction to reduce performance overhead. This is similar to what other related work did [9], [21]. For comparison purposes, we repeat the above process using the two simpler models ($f_s + f_c$ and f_s). We then use FI to obtain the SDC probabilities of the programs protected using the different models at different overhead levels. Note that FI is used only for the evaluation and not for any of the models.

Figure 8 shows the results of the SDC probability reduction at different protection levels. Without protection, the average SDC probability of the programs is 13.59%. At the 11.78% overhead level, after protection based on TRIDENT, $f_s + f_c$ and f_s the corresponding SDC probabilities are 5.50%, 5.53%, 9.29% respectively. On average, the protections provided by the three models reduce the SDC probabilities by 64%, 64% and 40% respectively. At the 23.31% overhead level, after the protections based on TRIDENT, $f_s + f_c$ and f_s respectively, the average SDC probabilities are 1.55%, 2.00% and 4.04%. This corresponds to a reduction of 90%, 87% and 74% of the SDC probability in the baseline respectively. Thus, on average,

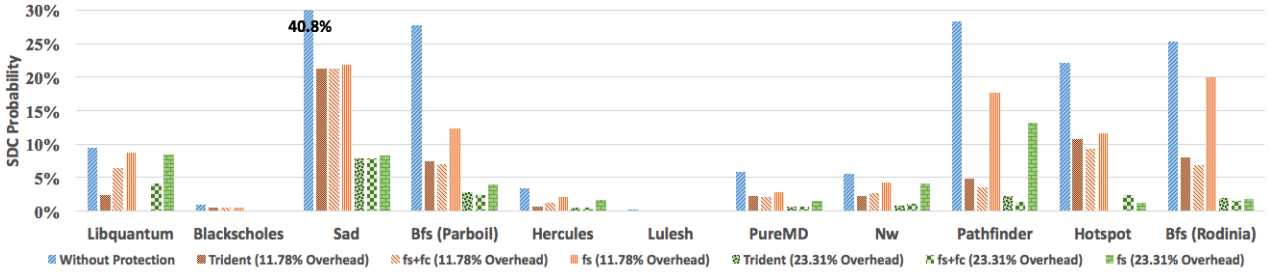


Fig. 8: SDC Probability Reduction with Selective Instruction Duplication at 11.78% and 23.31% Overhead Bounds (Margin of Error: $\pm$0.07% to $\pm$1.76% at 95% Confidence)

TRIDENT provides a higher SDC probability reduction for the same overhead level compared with the two simpler models.

Taking a closer look, the protection based on $f_s + f_c$ achieves comparable SDC probability reductions with TRIDENT. This is because the relative ranking of SDC probabilities between instructions plays a more dominant role in the selective protection than the absolute SDC probabilities. The ranking of the SDC probabilities of individual instructions derived by $f_s + f_c$ is similar to that derived by TRIDENT. Adding f_m boosts the overall accuracy of the model in predicting the absolute SDC probabilities (Figure 5), but not the relative SDC probabilities – the only exception is *Libquantum*. This shows the importance of modeling control-flow divergence, which is missing in other existing techniques [9], [10], [27].

VII. DISCUSSION

We first investigate the sources of inaccuracy in TRIDENT based on the experimental results (Section V). We then examine some of the threats to the validity of our evaluation. Finally, we compare TRIDENT with two closely related prior techniques, namely PVF and ePVF.

A. Sources of Inaccuracy

Errors in Store Address: If a fault modifies the address of a store instruction, in most cases, an immediate crash would occur because the instruction accesses memory that is out of bounds. However, if the fault does not cause a crash, it can corrupt an arbitrary memory location, and may eventually lead to SDC. It is difficult to analyze which memory locations may be corrupted as a result of such faults, leading to inaccuracy in the case. In our fault injection experiments, we observe that on average about 5.05% of faults affect addresses in store instructions and survive from crashes.

Memory Copy: Another source of inaccuracy in TRIDENT is that we do not handle bulk memory operations such as memmove and memcpy, which are represented by special instructions in the LLVM IR. We find such operations in benchmark such as *Sad, Lulesh, Hercules* and *PureMD*, which makes our technique somewhat inaccurate for these programs.

Manipulation of Corrupted Bits: As mentioned in Section IV-C, we assume only instructions such as comparisons, logical operators and casts have masking effects to simplify our calculations, and that none of the other instructions mask the corrupted bits. However, this is not always the case as other instructions may also cause masking. For example, division operations such as *fdiv* may also average out corrupted bits in the mantissa of floating point numbers, and hence mask errors. We find that 1% of the faults affect *fdiv* in program such as *Lulesh*, thereby leading to inaccuracies.

Conservatism in Determining Memory Corruption: Recall that when control-flow divergence happens, we assume all the store instructions that are dominated by the faulty branch are corrupted (Section IV). This is a conservative assumption, as some stores may end up being coincidentally correct. For example, if a store instruction is supposed to write a zero to its memory location, but is not executed due to the faulty branch, the location will still be correct if there was a zero already in that location. These are called *lucky loads* [7], [9].

B. Threats to Validity

Benchmarks: As mentioned in Section V-A1, we choose 11 programs to encompass a wide variety of domains rather than sticking to just one benchmark suite (unlike performance evaluation, there is no standard benchmark suite for reliability evaluation). Our results may be specific to our choice of benchmarks, though we have not observed this to be the case. Other work in this domain makes similar decisions [9], [21].

Platforms: In this work, we focus on CPU programs for TRIDENT. Graphic Processing Units (GPU) are another important platform for reliability studies. We have attempted to run TRIDENT on GPU programs, but were crippled by the lack of automated tools for code analysis and fault injection on GPUs. Our preliminary results in this domain using small CUDA kernels (instrumented manually) confirm the accuracy of TRIDENT. However, more rigorous evaluation is needed.

Program Input: As the high-fidelity fault injection experiments take a long time (Section V-C), we run each program only under 1 input. This is also the case for almost all other studies we are aware of in this space [9], [10]. Di Leo et at. [8] have found SDC probabilities of programs may change under different program inputs. We plan to consider multiple inputs in our future work.

Fault Injection Methodology: We use LLFI, a fault injector that works at the LLVM IR level, to inject single bit flips. While this method is accurate for estimating SDC probabilities of programs [30], [25], it remains an open question as to how accurate it is for other failure types. That said, our focus in this paper is SDCs, and so this is an appropriate choice for us.

C. Comparison with ePVF and PVF

ePVF (enhanced PVF) is a recent modeling technique for error propagation in programs [9]. It shares the same goal with TRIDENT in predicting the SDC probability of a program, both at the aggregate level and instruction level. ePVF is

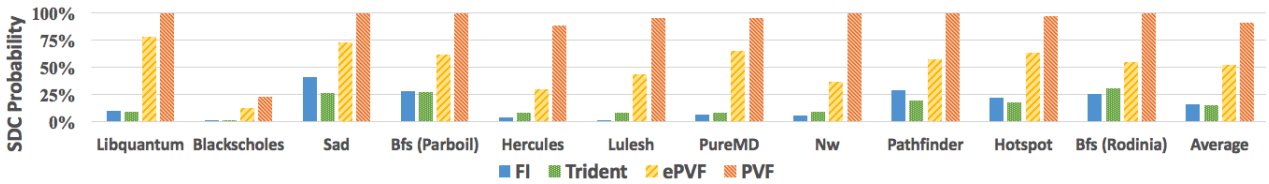


Fig. 9: Overall SDC Probabilities Measured by FI and Predicted by TRIDENT, ePVF and PVF (Margin of Error: ±0.07% to ±1.76% at 95% Confidence)

based on PVF [27], which stands for *Program Vulnerability Factor*. The main difference is that PVF does not distinguish between crash-causing faults and SDCs, and hence its accuracy of SDC prediction is poor [9]. ePVF improves the accuracy of PVF by removing most crashes from the SDC prediction. Unfortunately, ePVF cannot distinguish between benign faults and SDCs, and hence its accuracy suffers accordingly [9]. This is because ePVF only models error propagation in *static data-dependent instruction sequence* and in memory if the *static data-dependent instruction sequence* ends with a store instruction, ignoring error propagation to control-flow and other parts of memory. Both ePVF and PVF, like TRIDENT, require no FI in their prediction of SDC, and can be implemented at the LLVM IR level[3]. We implement both techniques using LLVM, and compare their results with TRIDENT's results.

Since crashes and SDCs are mutually exclusive, by removing the crash-causing faults, ePVF computes a relatively closer result to SDC probability than PVF [9]. However, the *crash propagation model* proposed by ePVF in identifying crashes requires a detailed DDG of the entire program's execution, which is extremely time-consuming and resource hungry. As a result, ePVF can be only executed in programs with a maximum of a million dynamic instructions in practice [9]. To address this issue and reproduce ePVF on our benchmarks and workloads (average 109 million dynamic instructions), we modify ePVF by replacing its *crash propagation model* with the measured results from FI. In other words, we assume ePVF identifies 100% of the crashes accurately, which is higher than the accuracy of the ePVF model. Hence, this comparison is conservative as it overestimates the accuracy of ePVF.

We use TRIDENT, ePVF and PVF to compute the SDC probabilities of the same benchmarks and workloads, and then compare them with FI which serves as our ground truth. The number of randomly sampled faults are 3,000. The results are shown in Figure 9. As shown, ePVF consistently overestimates the SDC probabilities of the programs with a *mean absolute error* of 36.78% whereas it is 4.75% in TRIDENT. PVF results in an even larger *mean absolute error* of 75.19% as it does not identify crashes. The observations are consistent with those reported by Fang et al. [9]. The average SDC probability measured by FI is 13.59%. ePVF and PVF predict it as 52.55% and 90.62% respectively, while TRIDENT predicts it as 14.83% and is significantly more accurate as a result.

VIII. RELATED WORK

There is a significant body of work on estimating the error resilience of a program either through FI [7], [11], [13], [14], [20], [30], or through modeling error propagation in programs [9], [10], [27]. The main advantage of FI is that it is simple, but it has limited predictive power. Further, its long running time often limits the FI approach from deriving program vulnerabilities at finer granularity (i.e., SDC probabilities of individual instructions). The main advantage of modeling techniques is that they have predictive power and are significantly faster, but existing techniques suffer from poor accuracy due to important gaps in the models. The main question we answer in this paper is that whether we can combine the advantages of the two approaches by constructing a model that is both accurate and scalable.

Shoestring [10] was one of the first papers to attempt to model the resilience of instructions without using fault injection. Because Shoestring is not publicly available, we cannot directly compare it with our TRIDENT. However, Shoestring stops tracing error propagations after control-flow divergence, and assumes that any fault that propagates to a store instruction leads to an SDC. Hence, it is similar to removing f_c and f_m in our model and considering only f_s , which we show is not very accurate. Further, Shoestring does not quantify SDC probabilities of programs and instructions, unlike TRIDENT.

Gupta et al. [12] investigate the resilience characteristics of different failures in large-scale systems. However, they do not propose automated techniques to predict failure rates. Lu et al. [21], Li et al. [18] identify vulnerable instructions by characterizing different features of instructions in programs. While they develop efficient heuristics in finding vulnerable instructions in programs, their techniques do not quantify error propagation, and hence cannot accurately pin-point SDC probabilities of individual instructions.

Sridharan et al. [27] introduce PVF, an analytical model which eliminates microarchitectural dependency from architectural vulnerability to approximate SDC probabilities of programs. While the model requires no FIs and is hence fast, it has poor accuracy in determining SDC probabilities as it does not distinguish between crashes and SDCs. Fang et al. [9] introduce ePVF, which derives tighter bounds on SDC probabilities than PVF, by omitting crash-causing faults from the prediction of SDCs. However, both techniques focus on modeling the static data dependency of instructions, and do not consider error propagation beyond control-flow divergence, which leads to large gaps in the predictions of SDCs (as we showed in Section VII-C).

Finally, Hari et al. [13], [14] propose a technique to obtain a comprehensive resilience profile of programs without needing exhaustive FIs. They prune the FI space by leveraging the similarity in executions to identify similar error propagations in programs. While they reduce the number of FIs by orders of magnitude, this approach still requires many FIs to obtain the resilience profile, requiring several hours on a 200 node cluster. TRIDENT offers a significantly faster solution, requiring no FIs.

[3]ePVF was originally implemented using LLVM, but not PVF.

IX. Conclusion

In this paper, we proposed TRIDENT, a three-level model for soft error propagation in programs. TRIDENT abstracts error propagation at static instruction level, control-flow level and memory level, and does not need any fault injection (FI). We implemented TRIDENT in the LLVM compiler, and evaluated it on 11 programs. We found that TRIDENT achieves comparable accuracy as FI, but is much faster and scalable both for predicting the overall SDC probabilities of programs, and the SDC probabilities of individual instructions in a program. We also demonstrated that TRIDENT can be used to guide selective instruction duplication techniques, and is significantly more accurate than simpler models.

As future work, we plan to extend TRIDENT to consider (1) Multiple inputs of a program [19], and (2) Platforms other than CPUs, such as GPUs or special-purpose accelerators.

Acknowledgement

This research was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) through the Discovery Grants and Strategic Project Grants (SPG) Programmes. We thank the anonymous reviewers of DSN'18 for their insightful comments and suggestions.

References

[1] IEEE standard for floating-point arithmetic. https://standards.ieee.org/findstds/standard/754-2008.html, 2008. IEEE Std 754-2008.

[2] Hasan Metin Aktulga, Joseph C Fogarty, Sagar A Pandit, and Ananth Y Grama. Parallel reactive molecular dynamics: Numerical methods and algorithmic techniques. *Parallel Computing*, 38(4):245–259, 2012.

[3] Rizwan A Ashraf, Roberto Gioiosa, Gokcen Kestor, Ronald F DeMara, Chen-Yong Cher, and Pradip Bose. Understanding the propagation of transient errors in hpc applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 72. ACM, 2015.

[4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *17th International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81. ACM, 2008.

[5] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *International Symposium on Workload Characterization (IISWC 2009)*, pages 44–54. IEEE, 2009.

[6] Cristian Constantinescu. Intermittent faults and effects on reliability of integrated circuits. In *Reliability and Maintainability Symposium*, page 370. IEEE, 2008.

[7] Jeffrey J Cook and Craig Zilles. A characterization of instruction-level error derating and its implications for error detection. In *International Conference on Dependable Systems and Networks(DSN)*, pages 482–491. IEEE, 2008.

[8] Domenico Di Leo, Fatemeh Ayatolahi, Behrooz Sangchoolie, Johan Karlsson, and Roger Johansson. On the impact of hardware faults–an investigation of the relationship between workload inputs and failure mode distributions. *Computer Safety, Reliability, and Security*, pages 198–209, 2012.

[9] Bo Fang, Qining Lu, Karthik Pattabiraman, Matei Ripeanu, and Sudhanva Gurumurthi. ePVF: An enhanced program vulnerability factor methodology for cross-layer resilience analysis. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 168–179. IEEE, 2016.

[10] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. Shoestring: Probabilistic soft error reliability on the cheap. In *Architectural Support for Programming Languages and Operating Systems*, pages 385–396, 2010.

[11] Weining Gu, Zbigniew Kalbarczyk, Ravishankar K Iyer, and Zhenyu Yang. Characterization of linux kernel behavior under errors. In *International Conference on Dependable Systems and Networks(DSN)*, page 459. IEEE, 2003.

[12] Saurabh Gupta, Tirthak Patel, Christian Engelmann, and Devesh Tiwari. Failures in large scale systems: long-term measurement, analysis, and implications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 44. ACM, 2017.

[13] Siva Kumar Sastry Hari, Sarita V Adve, Helia Naeimi, and Pradeep Ramachandran. Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults. In *Architectural Support for Programming Languages and Operating Systems*, pages 123–134, 2012.

[14] Siva Kumar Sastry Hari, Radha Venkatagiri, Sarita V Adve, and Helia Naeimi. Ganges: Gang error simulation for hardware resiliency evaluation. In *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 61–72. IEEE, 2014.

[15] John L Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer*, 33(7):28–35, 2000.

[16] I Karlin. Lulesh programming model and performance ports overview. https://codesign.llnl.gov/pdfs/lulesh_Ports.pdf. [Accessed Apr. 2016].

[17] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, page 75. IEEE, 2004.

[18] Guanpeng Li, Qining Lu, and Karthik Pattabiraman. Fine-grained characterization of faults causing long latency crashes in programs. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 450–461. IEEE, 2015.

[19] Guanpeng Li and Karthik Pattabiraman. Modeling input-dependent error propagation in programs. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2018.

[20] Guanpeng Li, Karthik Pattabiraman, Chen-Yang Cher, and Pradip Bose. Understanding error propagation in GPGPU applications. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 240–251. IEEE, 2016.

[21] Qining Lu, Guanpeng Li, Karthik Pattabiraman, Meeta S Gupta, and Jude A Rivers. Configurable detection of sdc-causing errors in programs. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(3):88, 2017.

[22] George B Mathews. On the partition of numbers. *Proceedings of the London Mathematical Society*, 1(1):486–490, 1896.

[23] Nahmsuk Oh, Philip P Shirvani, and Edward J McCluskey. Control-flow checking by software signatures. *Transactions on Reliability*, 51(1):111–122, 2002.

[24] Vijay Janapa Reddi, Meeta S Gupta, Michael D Smith, Gu-yeon Wei, David Brooks, and Simone Campanoni. Software-assisted hardware reliability: abstracting circuit-level challenges to the software stack. In *Design Automation Conference*, pages 788–793. IEEE, 2009.

[25] Behrooz Sangchoolie, Karthik Pattabiraman, and Johan Karlsson. One bit is (not) enough: An empirical study of the impact of single and multiple bit-flip errors. In *International Conference on Dependable Systems and Networks (DSN)*, pages 97–108. IEEE, 2017.

[26] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, et al. Addressing failures in exascale computing. *Institute for Computing in Science (ICiS). More infor*, 4:11, 2012.

[27] Vilas Sridharan and David R Kaeli. Eliminating microarchitectural dependency from architectural vulnerability. In *15th International Symposium on High Performance Computer Architecture*.

[28] Student. The probable error of a mean. *Biometrika*, pages 1–25, 1908.

[29] Ricardo Taborda and Jacobo Bielak. Large-scale earthquake simulation: computational seismology and complex engineering systems. *Computing in Science & Engineering*, 13(4):14–27, 2011.

[30] Jiesheng Wei, Anna Thomas, Guanpeng Li, and Karthik Pattabiraman. Quantifying the accuracy of high-level fault injection techniques for hardware faults. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 375–382. IEEE, 2014.