# Modeling Input-Dependent Error Propagation in Programs

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

***Abstract*—Transient hardware faults are increasing in computer systems due to shrinking feature sizes. Traditional methods to mitigate such faults are through hardware duplication, which incurs huge overhead in performance and energy consumption. Therefore, researchers have explored software solutions such as selective instruction duplication, which require fine-grained analysis of instruction vulnerabilities to Silent Data Corruptions (SDCs). These are typically evaluated via Fault Injection (FI), which is often highly time-consuming. Hence, most studies confine their evaluations to a single input for each program. However, there is often significant variation in the SDC probabilities of both the overall program and individual instructions across inputs, which compromises the correctness of results with a single input.**

**In this work, we study the variation of SDC probabilities across different inputs of a program, and identify the reasons for the variations. Based on the observations, we propose a model, vTRIDENT, which predicts the variations in programs' SDC probabilities without any FIs, for a given set of inputs. We find that vTRIDENT is nearly as accurate as FI in identifying the variations in SDC probabilities across inputs. We demonstrate the use of vTRIDENT to bound overall SDC probability of a program under multiple inputs, while performing FI on only a single input.**

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

## I. Introduction

Transient hardware fault probabilities are predicted to increase in future computer systems due to growing system scales, progressive technology scaling, and lowering operating voltages [29]. While such faults were masked through hardware-only solutions such as redundancy and voltage guard bands in the past, 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 microprocessor design [5]. As a result, software needs to be able to tolerate hardware faults with low overheads.

Hardware faults can cause programs to fail by crashing, hanging or producing incorrect program outputs, also known as silent data corruptions (SDCs). SDCs are a serious concern in practice as there is no indication that the program failed, and hence the results of the program may be taken to be correct. Hence, developers must first evaluate the SDC probability of their programs, and if it does not meet their reliability target, they need to add protection to the program until it does.

Fault injections (FIs) are commonly used for evaluating and characterizing programs' resilience, and to obtain the overall SDC probability of a program. In each FI campaign, a single fault is injected into a randomly sampled instruction, and the program is executed till it crashes or finishes. FI therefore requires that the program is executed with a specific input. In practice, a large number of FI campaigns are usually required to achieve statistical significance, which can be extremely time-consuming. As a result, most prior work limits itself to a single program input or at most a small number of inputs. Unfortunately, the number of possible inputs can be large, and there is often significant variance in SDC probabilities across program inputs. For example, in our experiments, we find that the overall SDC probabilities of the same program (Lulesh) can vary by more than 42 times under different inputs. This seriously compromises the correctness of the results from FI. Therefore, there is a need to characterize the variation in SDC probabilities across multiple inputs, without expensive FIs.

We find that there are two factors determining the variation of the SDC probabilities of the program across its inputs (we call this the *SDC volatility*): (1) Dynamic execution footprint of each instruction, and (2) SDC probability of each instruction (i.e., error propagation behaviour of instructions). Almost all existing techniques [7], [8], [10] on quantifying programs' failure variability across inputs consider only the execution footprint of instructions. However, we find that the error propagation behavior of individual instructions often plays as important a role in influencing the SDC volatility (Section III). Therefore, all existing techniques experience significant inaccuracy in determining a program's SDC volatility.

In this paper, we propose an automated technique to determine the SDC volatility of a program across different inputs, that takes into account both the execution footprint of individual instructions, and their error propagation probabilities. Our approach consists of three steps. First, we perform experimental studies using FI to analyze the properties of SDC volatility, and identify the sources of the volatility. We then build a model, vTRIDENT, which predicts the SDC volatility of programs automatically without any FIs. vTRIDENT is built on our prior model, TRIDENT [11] for predicting error propagation, but sacrifices some accuracy for speed of execution. Because we need to run vTRIDENT for multiple inputs, execution speed is much more important than in the case of TRIDENT. The intuition is that for identifying the SDC volatility, it is more important to predict the relative SDC probabilities among inputs than the absolute probabilities. Finally, we use vTRIDENT to bound the SDC probabilities of a program across multiple inputs, while performing FI on only a single input. *To the best of our knowledge, we are the first to systematically study and model the variation of SDC probabilities in programs across inputs.*

The main contributions are as follows:

- We identify two sources of SDC volatility in programs, namely INSTRUCTION-EXECUTION-VOLATILITY that captures the variation of dynamic execution footprint of instructions, and INSTRUCTION-SDC-VOLATILITY that captures the variability of error propagation in instructions, and mathematically derive their relationship (Section III).
- To understand how SDC probabilities vary across inputs, we conduct a FI study using nine benchmarks with ten different program inputs for each benchmark, and quantify the relative contribution of INSTRUCTION-EXECUTION-VOLATILITY and INSTRUCTION-SDC-VOLATILITY (Section IV) to the overall SDC volatility.
- Based on the understanding, we build a model, VTRIDENT[1], on top of our prior framework for modeling error propagation in programs TRIDENT (Section V-B). VTRIDENT predicts the SDC volatility of instructions without any FIs, and also bounds the SDC probabilities across a given set of inputs.
- Finally, we evaluate the accuracy and scalability of VTRIDENT in identifying the SDC volatility of instructions (Section VI), and in bounding SDC probabilities of program across inputs (Section VII).

Our main results are as follows:

- Volatility of overall SDC probabilities is due to both the INSTRUCTION-EXECUTION-VOLATILITY and INSTRUCTION-SDC-VOLATILITY. Using only INSTRUCTION-EXECUTION-VOLATILITY to predict the overall SDC volatility of the program results in significant inaccuracies, i.e., an average of 7.65x difference with FI results (up to 24x in the worst case).
- We find that the accuracy of VTRIDENT is 87.81% when predicting the SDC volatility of individual instructions in the program. The average difference between the variability predicted by VTRIDENT and that by FI is only 1.26x (worst case is 1.29x).
- With VTRIDENT 78.89% of the given program inputs' overall SDC probabilities fall within the predicted bounds. With INSTRUCTION-EXECUTION-VOLATILITY alone, only 32.22% of the probabilities fall within the predicted bounds.
- Finally, the average execution time for VTRIDENT is about 15 minutes on an input of nearly 500 million dynamic instructions. *This constitutes a speedup of more than 8x compared with the* TRIDENT *model to bound the SDC probabilities, which is itself an order of magnitude faster than FI [11].*

## II. BACKGROUND

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

[1]VTRIDENT stands for "Volatility Prediction for TRIDENT".

### A. Fault Model

In this paper, we consider transient hardware faults that occur in the computational elements of the processor, including pipeline stages, flip-flops, and functional units. We do not consider faults in the memory or caches, as we assume that these are protected with ECC. Likewise, we do not consider faults in the processor's control logic as we assume it is protected. Neither do we consider faults in the instructions' encoding as these can be detected through other means such as error correcting codes. Finally, we assume that program does not jump to arbitrary illegal addresses due to faults during the execution, as this can be detected by control-flow checking techniques [27]. However, the program may take a faulty legal branch (execution path is legal but branch direction can be wrong due to faults propagating to it). Our fault model is in line with other work in the area [6], [9], [14], [24], [33].

### B. Terms and Definitions

- **Fault Occurrence:** The event corresponding to the occurrence of transient 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's 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 [9], [13], [21], [22], [31], [34].

### C. LLVM Compiler and LLVM Fault Injector

In this paper, we use the LLVM compiler [20] to perform the program analysis, 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 it to the source code. Secondly, 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 of different assembly language formats. Finally, LLVM IR has been shown to be accurate for doing FI studies [31], and there are many fault injectors developed for LLVM [1], [22], [28], [31]. Most of the papers we compare with in this study also use LLVM infrastructure [9], [21]. Therefore, in this paper, when we say instruction, we mean an instruction at the LLVM IR level. However, our methodology is not tied to LLVM.

We use LLVM Fault Injector (LLFI) [31] to perform FI experiments. LLFI is found to be accurate in studying SDCs [31]. Since we consider transient errors that occur in computational components, we inject single bit flips in the return values of the target instruction randomly chosen at runtime. We consider single bit flips as this is the de-facto fault model for simulating transient faults in the literature [9], [21], [14]. Although there have been concerns expressed about the representativeness of using single-bit flip faults for FI to model soft errors [4], a recent study [28] has shown that there is very little difference in SDC probabilities due to single and multiple bit flips at the application level. Since we focus on SDCs, we use single bit flips in our evaluation.

## III. Volatilities and SDC

In this section, we explain how we calculate the overall SDC probability of a program under multiple inputs. Statistical FI is the most common way to evaluate the overall SDC probability of a program and has been used in other related work in the area [7], [10], [14], [15], [21]. It randomly injects a large number (usually thousands) of faults under a given program input, one fault per program execution, by uniformly choosing program instruction for injection from the set of all executed instructions.

Equation 1 shows the calculation of the overall SDC probability of the program, $P_{overall}$, from statistical FI. $N_{SDC}$ is the number of FI campaigns that result in SDCs among all the FI campaigns. $N_{total}$ is the total number of FI campaigns. Equation 1 can be expanded to the equivalent equations shown in Equation 2. $P_i$ is the SDC probability of each (static) instruction that is chosen for FI, $N_i$ is the amount of times that the static instruction is chosen for injection over all FI campaigns. $i$ to $n$ indicates all the distinct static instructions that are chosen for injection.

$$P_{overall} = N_{SDC}/N_{total} \tag{1}$$

$$= (\sum_{i=1}^{n} P_i * N_i)/N_{total} = \sum_{i=1}^{n} P_i * (N_i/N_{total}) \tag{2}$$

In Equation 2, we can see that $N_i/N_{total}$ and $P_i$ are the two relevant factors in the calculation of the overall SDC probability of the program. $N_i/N_{total}$ can be interpreted as the probability of the static instruction being sampled during the program execution. Because the faults are uniformly sampled during the program execution, $N_i/N_{total}$ is statistically equivalent to the ratio between the number of dynamic executions of the chosen static instruction, and the total number of dynamic instructions in the program execution. We call this ratio the dynamic execution footprint of the static instruction. The larger the dynamic execution footprint of a static instruction, the higher the chance that it is chosen for FI.

Therefore, we identify two kinds of volatilities that affect the variation of $P_{overall}$ when program inputs are changed from Equation 2: (1) Instruction-SDC-Volatility, and (2) Instruction-Execution-Volatility. Instruction-SDC-Volatility represents the variation of $P_i$ across the program inputs, Instruction-Execution-Volatility is equal to the variation of dynamic execution footprints, $N_i/N_{total}$, across the program inputs. We also define the variation of $P_{overall}$ as Overall-SDC-Volatility. As explained above, Instruction-Execution-Volatility can be calculated by profiling the number of dynamic instructions when inputs are changed, which is straight-forward to derive. However, Instruction-SDC-Volatility is difficult to identify as $P_i$ requires a large number of FI campaigns on every such instruction $i$ with different inputs, which becomes impractical when the program size and the number of inputs become large. As mentioned earlier, prior work investigating Overall-SDC-Volatility considers only the Instruction-Execution-Volatility, and ignores Instruction-SDC-Volatility [7], [10]. However, as we show in the next section, this can lead to significant inaccuracy in the estimates. Therefore, we focus on deriving Instruction-SDC-Volatility efficiently in this paper.

## IV. Initial FI Study

In this section, we design experiments to show how Instruction-SDC-Volatility and Instruction-Execution-Volatility contribute to Overall-SDC-Volatility, then explain the variation of Instruction-SDC-Volatility across programs.

TABLE I: Characteristics of Benchmarks

| Benchmark | Suite/Author | Description | Total Dynamic Instructions (Millions) |
|---|---|---|---|
| Libquantum | SPEC | Simulation of quantum computing | 6238.55 |
| Nw | Rodinia | A nonlinear global optimization method for DNA sequence alignments | 564.63 |
| Pathfinder | Rodinia | Use dynamic programming to find a path on a 2-D grid | 6.71 |
| Streamcluster | Rodinia | Dense Linear Algebra | 3907.70 |
| Lulesh | Lawrence Livermore National Laboratory | Science and engineering problems that use modeling hydrodynamics | 3382.79 |
| Clomp | Lawrence Livermore National Laboratory | Measurement of HPC performance impacts | 11324.17 |
| CoMD | Lawrence Livermore National Laboratory | Molecular dynamics algorithms and workloads | 17136.62 |
| FFT | Open Source | 2D fast Fourier transform | 6.37 |
| Graph | Open Source | Graph traversal in operational research | 0.15 |

### A. Experiment Setup

*1) Benchmarks:* We choose nine applications in total for our experiments. These are drawn from standard benchmark suites, as well as from real world applications. Note that there are very few inputs provided with the benchmark applications, and hence we had to generate them ourselves. We search the entire benchmark suites of Rodinia [3], SPLASH-2 [32], PARSEC [2] and SPEC [16], and choose applications based on two criteria: (1) Compatibility with our toolset (i.e., we could compile them to LLVM IR and work with LLFI), and (2) Ability to generate diverse inputs for our experiments. For the latter criteria, we choose applications that take numeric values as their program inputs, rather than binary files or files of unknown formats, since we cannot easily generate different inputs in these applications. As a result, there are only three applications in Rodinia and one application in SPEC meeting the criteria. To include more benchmarks, we pick three HPC applications (Lulesh, Clomp, and CoMD) from Lawrence Livermore National Laboratory [17], and two open-source projects (FFT [19] and Graph [18]) from online repositories. The nine benchmarks span a wide range of application domains from simulation to measurement, and are listed in Table I.

*2) Input Generation:* Since all the benchmarks we choose take numerical values as their inputs, we randomly generate numbers for their inputs. The inputs generated are chosen based on two criteria: (1) The input should not lead to any reported errors or exceptions that halt the execution of the program, as such inputs may not be representative of the application's behavior in production, And (2) The number of dynamic executed instructions for the inputs should not exceed 50 billion to keep our experimental time reasonable. We report the total number of dynamic instructions generated from the 10 inputs of each benchmark in Table I. The average number of dynamic instructions per input is 472.95 million, which is significantly larger than what have been used in most other prior work [9], [21], [24], [33], [34]. We consider large inputs to stress vTRIDENT and evaluate its scalability.

*3) FI methodology:* As mentioned before, we use LLFI [31] to perform the FI experiments. For each application, we inject 100 random faults for each static instruction of the application – this yields error bars ranging from 0.03% to 0.55% depending on the application for the 95% confidence intervals. Because we need to derive SDC probabilities of every static instruction, we have to perform multiple FIs on every static instruction in each benchmark. Therefore, to balance the experimental time with accuracy, we choose to inject 100 faults on each static instruction. This adds up to a total number of injections ranging from 26,000 to 2,251,800 in each benchmark, depending on the number of static instructions in the program.

## B. Results

*1)* INSTRUCTION-EXECUTION-VOLATILITY *and* OVERALL-SDC-VOLATILITY*:* We first investigate the relationship between INSTRUCTION-EXECUTION-VOLATILITY and OVERALL-SDC-VOLATILITY. As mentioned in Section III, INSTRUCTION-EXECUTION-VOLATILITY is straight-forward to derive based on the execution profile alone, and does not require performing any FIs. If it is indeed possible to estimate OVERALL-SDC-VOLATILITY on the basis of INSTRUCTION-EXECUTION-VOLATILITY alone, we can directly plug in INSTRUCTION-EXECUTION-VOLATILITY to $N_i$ and $N_{total}$ in Equation 2 when different inputs are used and treat $P_i$ as a constant (derived based on a single input) to calculate the overall SDC probabilities of the program with the inputs.

We profiled INSTRUCTION-EXECUTION-VOLATILITY in each benchmark and use it to calculate the overall SDC probabilities of each benchmark across all its inputs. To show OVERALL-SDC-VOLATILITY, we calculate the differences between the highest and the lowest overall SDC probabilities of each benchmark, and plot them in Figure 1. In the figure, *Exec. Vol.* represents the calculation with the variation of INSTRUCTION-EXECUTION-VOLATILITY alone in Equation 2, treating $P_i$ as a constant, which are derived by performing FI on only one input. *FI* indicates the results derived from FI experiment with the set of all inputs of each benchmark. As can be observed, the results for individual benchmark with OVERALL-SDC-VOLATILITY estimated from *Exec. Vol.* alone are significantly lower than the FI results (up to 24x in *Pathfinder*). The average difference is 7.65x. *This shows that* INSTRUCTION-EXECUTION-VOLATILITY *alone is not sufficient to capture* OVERALL-SDC-VOLATILITY*, motivating the need for accurate estimation of* INSTRUCTION-SDC-VOLATILITY. This is the focus of our work.

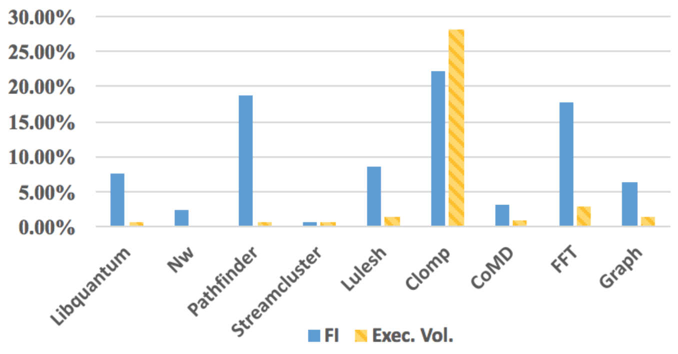
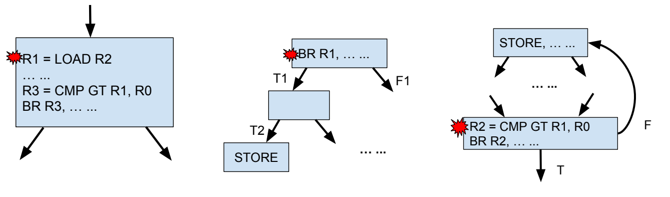


Fig. 1: OVERALL-SDC-VOLATILITY Calculated by INSTRUCTION-EXECUTION-VOLATILITY Alone (Y-axis: OVERALL-SDC-VOLATILITY, Error Bar: 0.03% to 0.55% at 95% Confidence)

*2) Code Patterns Leading to* INSTRUCTION-SDC-VOLATILITY*:* To figure out the root causes of INSTRUCTION-SDC-VOLATILITY, we analyze the FI results and their error propagation based on the methodology proposed in our prior work [11]. We identify three cases leading to INSTRUCTION-SDC-VOLATILITY.

*Case 1: Value Ranges of Operands of Instructions*

Different program inputs change the values that individual instructions operate with. For example, in Figure 2a, there are three instructions (LOAD, CMP and BR) on a straight-line code sequence. Assume that under some INPUT A, R1 is 16 and R0 is 512, leading the result of the CMP (R3) to be FALSE. Since the highest bit of 512 is the 9th bit, any bit-flip at the bit positions that are higher than 9 in R1 will modify R1 to a value that is greater than R0. This may in turn cause the result of the CMP instruction (R3) to be TRUE. In this case, the probability for the fault that occurred at R1 of the LOAD instruction to propagate to R3 is *(32-9)/32=71.88%* (assuming a 32-bit data width of R1). In another INPUT B, assume R1 is still 16, but R0 becomes 64 of which the highest bit is the 6th bit. In this case, the probability for the same fault to propagate to R3 becomes *(32-6)/32=81.25%*. In this example, the propagation probability increases by almost 10% for the same fault for a different input. In other words, the SDC volatility of the LOAD instruction in the example is changed by about 10%. We find that in the nine benchmarks, the proportion of instructions that fall into this pattern varies from 3.07% (*FFT*) to 15.23% (*Nw*) - the average is 6.98%. The instructions exhibit different error propagation even if the control flow does not change.



(a) Value Range (b) Execution Path (c) Size of Loop

Fig. 2: Patterns Leading to INSTRUCTION-SDC-VOLATILITY

*Case 2: Execution Paths and Branches*

Different program inputs may exercise different execution paths of programs. For example, in Figure 2b, there are three branch directions labeled with T1, F1 and T2. Each direction may lead to a different execution path. Assume that the execution probabilities of T1, F1 and T2 are 60%, 70% and 80% for some INPUT A. If a fault occurs at the BR instruction and modifies the direction of the branch from F1 to T1, the probability of this event is 70% as the execution probability of F1 is 70%. In this case, the probability for the fault to propagate to the STORE instruction under T2 is *70%*80%=56%*. Assuming there is another INPUT B which makes the execution probabilities of T1, F1 and T2, 10%, 90% and 30% respectively. The probability for the same fault to propagate to the STORE instruction becomes *90%*30%=27%*. Thus, the propagation probability of the fault decreases by 29% from INPUT A to INPUT B, and thus the SDC volatility of the BR instruction is 29%. In the nine benchmarks, we find that 43.28% of the branches on average exhibit variations of branch probabilities across inputs, leading to variation of SDC probability in instructions.

*Case 3: Number of Iterations of Loops*

The number of loop iterations can change when program inputs are changed, causing volatility of error propagation. For example, in Figure 2c, there is a loop whose termination is controlled by the value of R2. The CMP instruction compares R1 against R0 and stores it in R2. If the F branch is taken, the loop will continue, whereas if T branch is taken, the loop will terminate. Assume that under some INPUT A the value of R0 is 4, and that in the second iteration of the loop, a fault occurs at the CMP instruction and modifies R2 to TRUE from FALSE, causing the loop to terminate early. In this case, the STORE instruction is only executed twice whereas it should be executed 4 times in a correct execution. Because of the early termination of the loop, there are 2 STORE executions missing. Assume there is another INPUT B that makes R0 8, indicating there are 8 iterations of the loop in a correct execution. Now for the same fault in the second iteration, the loop terminates resulting in only 2 executions of the STORE whereas it should execute 8 times. 6 STORE executions are missing with INPUT B (*8-2=6*). If the SDC probability of the STORE instruction stays the same with the two inputs, INPUT B triples (6/2=3) the probability for the fault to propagate through the missing STORE instruction, causing the SDC volatility. In the nine benchmarks, we find that 90.21% of the loops execute different numbers of iterations when the input is changed.

## V. MODELING INSTRUCTION-SDC-VOLATILITY

We first explain how the overall SDC probability of a program is calculated using TRIDENT, which we proposed in our prior work [11]. We then describe vTRIDENT, an extension of TRIDENT to predict INSTRUCTION-SDC-VOLATILITY. The main difference between the two models is that vTRIDENT simplifies the modeling in TRIDENT to improve running time, which is essential for processing multiple inputs.

### *A.* TRIDENT

*1) How* TRIDENT *works:* TRIDENT [11] models error propagation in a program using static and dynamic analyses of the program. The model takes the code of the program and executes it with a (single) program input provided to analyze error propagation. It tracks error propagation at three levels, namely static instruction sequence, control-flow and memory dependency in the program execution (see appendix for details). The output of TRIDENT is the SDC probability of each individual instruction and the overall SDC probability of the program. TRIDENT *requires a single program input for its calculations, and consequently, the output of* TRIDENT *is specific to the program input provided.*

As mentioned in Section IV-B2, we identify three patterns leading to INSTRUCTION-SDC-VOLATILITY in programs. TRIDENT first tracks error propagation in static instructions, in which the propagation probability of each instruction is computed based on the profiled values and the mechanism of the instruction. The propagation probabilities of the instructions are used to compute the SDC probability of each straight-line code sequence. Since the profiling phase is per input, *Case 1* is captured and different values of instructions from different inputs can be factored into the computation. After tracking errors at the static instruction level, TRIDENT computes the probability leading to memory corruption if any control-flow divergence occurs. At this phase, branch probabilities and loop information are profiled for the computation, which are also input specific. Therefore, *Cases 2* and *3* are also captured by TRIDENT when different inputs are used.

*2) Drawbacks of* TRIDENT*:* Even though TRIDENT is orders of magnitude faster than FI and other models in measuring SDC probabilities, it can sometimes take a long time to execute depending on the program input. Further, when we want to calculate the variation in SDC probabilities across inputs, we need to execute TRIDENT once for each input, which can be very time-consuming. For example, if TRIDENT takes 30 minutes on average per input for a given application (which is still considerably faster than FI), it would take more than 2 days (50 hours) to process 100 inputs. This is often unacceptable in practice. Further, because TRIDENT tracks memory error propagation in a fine-grained manner, it needs to collect detailed memory traces. In a few cases, these traces are too big to fit into memory, and hence we cannot run TRIDENT at all. This motivates vTRIDENT, which does not need detailed memory traces, and is hence much faster.

## *B.* vTRIDENT

As mentioned above, the majority of time spent in executing TRIDENT is in profiling and traversing memory dependencies of the program, which is the bottleneck in scalability. vTRIDENT extends TRIDENT by pruning any repeating memory dependencies from the profiling, and keeping only distinct memory dependencies for tracing error propagation. The intuition is that if we equally apply the same pruning to all inputs in each program, similar scales of losses in accuracy will be experienced across the inputs. Therefore, the relative SDC probabilities across inputs are preserved. Since volatility depends only on the relative SDC probabilities across inputs, the volatilities will also be preserved under pruning.

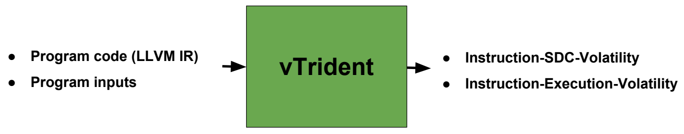


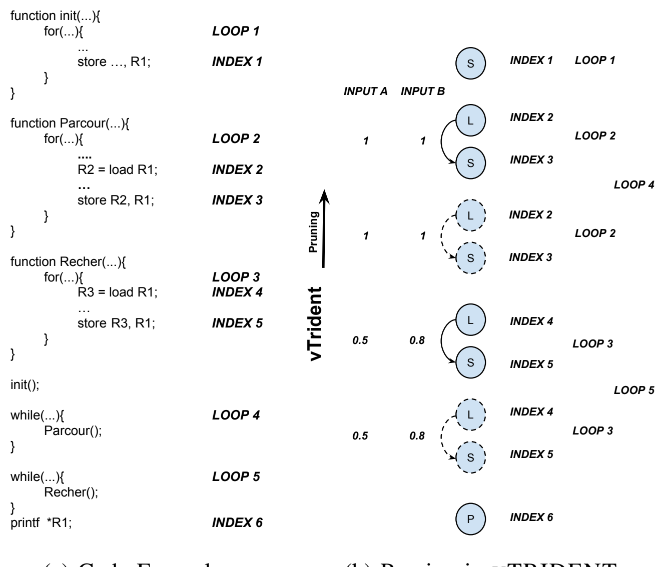
Fig. 3: Workflow of vTRIDENT

*1) Workflow:* Figure 3 shows the workflow of vTRIDENT. It is implemented as a set of LLVM compiler passes which take the code of the program (compiled into LLVM IR) and a set of inputs of the program. The output of vTRIDENT is the INSTRUCTION-SDC-VOLATILITY and INSTRUCTION-EXECUTION-VOLATILITY of the program across all the inputs provided, both at the aggregate level and per-instruction level. Based on Equation 2, OVERALL-SDC-VOLATILITY can be computed using INSTRUCTION-SDC-VOLATILITY and INSTRUCTION-EXECUTION-VOLATILITY.

vTRIDENT executes the program with each input provided, and records the differences of SDC probabilities predicted between inputs to generate INSTRUCTION-SDC-VOLATILITY. During each execution, the program's dynamic footprint is also recorded for the calculation of INSTRUCTION-EXECUTION-VOLATILITY. The entire process is fully automated and requires no intervention of the user. Further, no FIs are needed in any part of the process.

*2) Example:* We use an example from *Graph* in Figure 4a to illustrate the idea of vTRIDENT and its differences from TRIDENT. We make minor modifications for clarity and remove some irrelevant parts in the example. Although vTRIDENT works at the level of LLVM IR, we show the corresponding C code for clarity. We first explain how TRIDENT works for the example, and then explain the differences with vTRIDENT.

In Figure 4a, the C code consists of three functions, each of which contains a loop. In each loop, the same array is manipulated symmetrically in iterations of the loops and transferred between memory back and forth. So the load and store instructions in the loops (*LOOP 1, 2* and *3*) are all memory data-dependent. Therefore, if a fault contaminates any of them, it may propagate through the memory dependencies of the program. *init()* is called once at the beginning, then *Parcour()* and *Recher()* are invoked respectively in *LOOP 4* and *5*. *printf (INDEX 6)* at the end is the program's output. In the example, we assume *LOOP 4* and *5* execute two iterations each for simplicity. Therefore, the fault leads to an SDC if the fault propagates to the instruction.


```
function init(...){
    for(...){                  LOOP 1
        ...
        store ..., R1;         INDEX 1
    }
}

function Parcour(...){
    for(...){                  LOOP 2
        ....
        R2 = load R1;          INDEX 2
        ...
        store R2, R1;          INDEX 3
    }
}

function Recher(...){
    for(...){                  LOOP 3
        R3 = load R1;          INDEX 4
        ...
        store R3, R1;          INDEX 5
    }
}

init();

while(...){                    LOOP 4
    Parcour();
}

while(...){                    LOOP 5
    Recher();
}
printf *R1;                    INDEX 6
```

(a) Code Example (b) Pruning in vTRIDENT

Fig. 4: Example of Memory Pruning

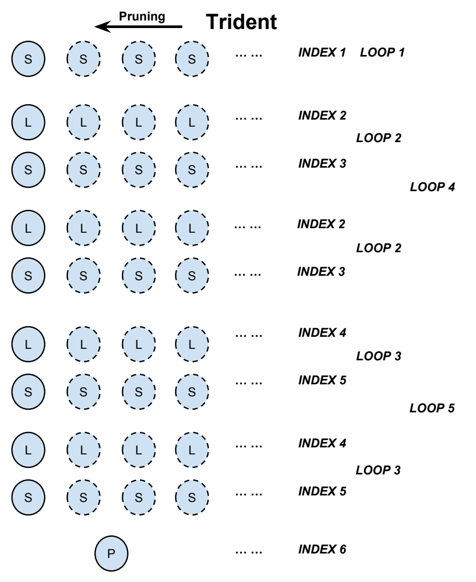


Fig. 5: Memory Dependency Pruning in TRIDENT

To model error propagation via memory dependencies of the program, a similar memory dependency graph is created in Figure 5. Each node represents either a dynamic load or store instruction of which indices and loop positions of their static instructions are marked on their right. In the figure, each column of nodes indicates data-dependent executions of the instructions - there is no data flowing between columns as the array of data are manipulated by *LOOP 1, 2* and *3* symmetrically. In this case, TRIDENT finds the opportunity to prune the repeated columns of nodes to speed up its modeling time as error propagations are similar in the columns. The pruned columns are drawn with dashed border in the figure, and they indicate the pruning of the inner-most loops. TRIDENT applies this optimization for memory-level modeling, resulting in significant acceleration compared with previous modeling techniques [11]. However, as mentioned, the graph can still take significant time to construct and process.

To address this issue, vTRIDENT further prunes memory dependency by tracking error propagations only in distinct dependencies to speed up the modeling. Figure 4b shows the idea: The graph shown in the figure is pruned to the one by TRIDENT in Figure 5. Arrows between nodes indicate propagation probabilities in the straight-line code. Because there could be instructions leading to crashes and error masking in straight-line code, the propagation probabilities are not 1. The propagation probabilities marked beside the arrows are aggregated to compute SDC probabilities for INPUT A and INPUT B respectively. For example, if a fault occurs at *INDEX 1*, the SDC probability for the fault to reach program output (*INDEX 6*) is calculated as $1*1*0.5*0.5 = 25\%$ for INPUT A, and $1*1*0.8*0.8 = 64\%$ for INPUT B. Thus, the variation of the SDC probability is 39% for these two inputs. vTRIDENT prunes the propagation by removing repeated dependencies (their nodes are drawn in dashed border in Figure 4b). The calculation of SDC probability for the fault that occurred at *INDEX 1* to *INDEX 6* becomes *1*0.5 = 50%* with INPUT A, and *1*0.8 = 80%* with INPUT B. The variation between the two inputs thus becomes 30%, which is 9% lower than that computed by TRIDENT (i.e., without any pruning).

We make two observations from the above discussion: (1) If the propagation probabilities are 1 or 0, the pruning does not result in loss of accuracy (e.g., *LOOP 4 in Figure 4b*). (2) The difference with and without pruning will be higher if the numbers of iterations become very large in the loops that contain non-1 or non-0 propagation probabilities (i.e., *LOOP 5 in Figure 4b*). This is because more terms will be removed from the calculation by vTRIDENT. We find that about half (55.39%) of all faults propagating in the straight-line code have either *all* 1s or at least one 0 as the propagation probabilities, and thus there is no loss in accuracy for these faults. Further, the second case is rare because large iterations of aggregation on non-1 or non-0 numbers will result in an extremely small value of the overall SDC probability. This is not the case as the average SDC probability is 10.74% across benchmarks. Therefore, the pruning does not result in significant accuracy loss in vTRIDENT.

## VI. Evaluation of vTRIDENT

In this section, we evaluate the accuracy and performance of vTRIDENT in predicting Instruction-SDC-Volatility across multiple inputs. We use the same benchmarks and experimental procedure as before in Section IV. The code of vTRIDENT can be found in our GitHub repository.[2]

### A. Accuracy

To evaluate the ability of vTRIDENT in identifying Instruction-SDC-Volatility, we first classify all the instructions based on their Instruction-SDC-Volatility derived by FI and show their distributions – this serves as the ground truth. We classify the differences of the SDC probabilities of each measured instruction between inputs into three categories based on their ranges of variance ($<$10%, 10%−20% and $>$20%), and calculate their distribution based on their dynamic footprints. The results are shown in Figure 6. As can be seen in the figure, on average, only 3.53% of instructions across benchmarks exhibit variance of more than 20% in the SDC probabilities. Another 3.51% exhibit a variance between 10% and 20%. The remaining 92.93% of the instructions exhibit within 10% variance across inputs.

We then use vTRIDENT to predict the Instruction-SDC-Volatility for each instruction, and then compare the predictions with ground truth. These results are also shown in Figure 6. As can be seen, for instructions that have Instruction-SDC-Volatility less than 10%, vTRIDENT gives relatively accurate predictions across benchmarks. On average, 97.11% of the instructions are predicted to fall into this category by vTRIDENT, whereas FI measures it as 92.93%. Since these constitute the vast majority of instructions, vTRIDENT has high accuracy overall.

On the other hand, instructions that have Instruction-SDC-Volatility of more than 20% are significantly underestimated by vTRIDENT, as vTRIDENT predicts the proportion of such instructions as 1.84% whereas FI measures it as 3.53% (which is almost 2x more). With that said, for individual benchmarks, vTRIDENT is able to distinguish the sensitivities of Instruction-SDC-Volatility in most of them. For example, in *Pathfinder* which has the largest proportion of instructions that have Instruction-SDC-Volatility greater than 20%, vTRIDENT is able to accurately identify that this benchmark has the highest proportion of such instructions relative to the other programs. However, we find vTRIDENT is not able to well identify the variations that are greater than 20% as mentioned above. This case can be found in *Nw, Lulesh, Clomp and FFT.* We discuss the sources of inaccuracy in Section VIII-A. Since these instructions are relatively few in terms of dynamic instructions in the programs, this underprediction does not significantly affect the accuracy of vTRIDENT.

We then measure the overall *accuracy* of vTRIDENT in identifying Instruction-SDC-Volatility. The accuracy is defined as the number of correctly predicted variation categories of instructions over the total number of instructions being predicted. We show the accuracy of vTRIDENT in Figure 7. As can be seen, the highest accuracy is achieved in Streamcluster (99.17%), while the lowest accuracy is achieved in Clomp (67.55%). The average accuracy across nine benchmarks is 87.81%, indicating that vTRIDENT is able to identify most of the Instruction-SDC-Volatility.

Finally, we show the accuracy of predicting Overall-SDC-Volatility using vTRIDENT, and using Instruction-Execution-Volatility alone (as before) in Figure 8. As can be seen, the average difference between vTRIDENT and FI is only 1.26x. Recall that the prediction using Instruction-Execution-Volatility alone (*Exec. Vol.*) gives an average difference of 7.65x (Section IV). The worst case difference when considering only *Exec. Vol.* was 24.54x, while it is 1.29x (in *Pathfinder*) when Instruction-SDC-Volatility is taken into account. Similar trends are observed in all other benchmarks. This indicates that the accuracy of Overall-SDC-Volatility prediction is significantly higher when considering both Instruction-SDC-Volatility and

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

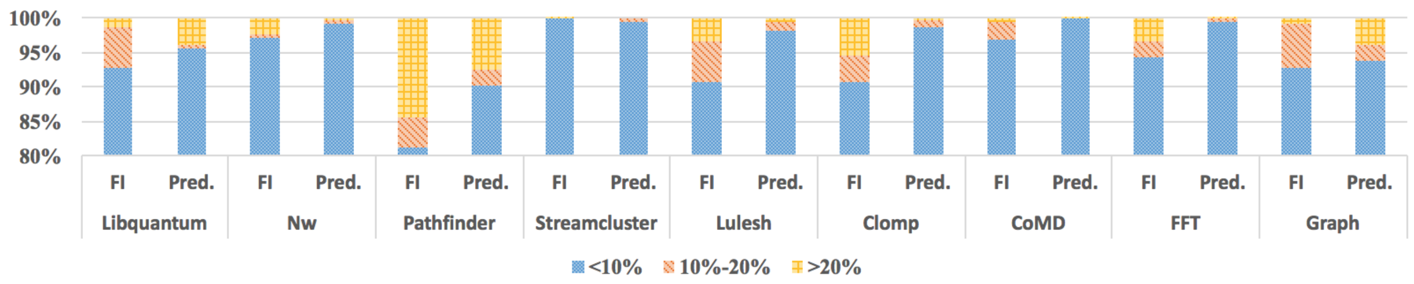


Fig. 6: Distribution of INSTRUCTION-SDC-VOLATILITY predictions by vTrident Versus Fault Injection Results (Y-axis: Percentage of instructions, Error Bar: 0.03% to 0.55% at 95% Confidence)

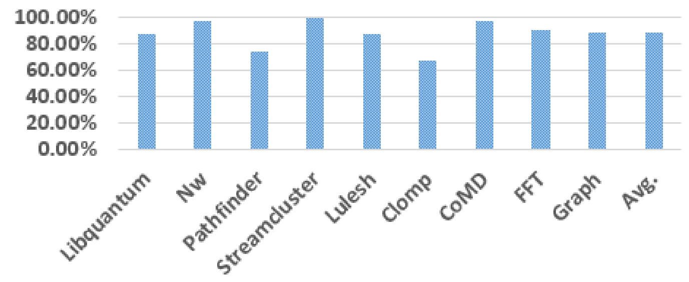


Fig. 7: Accuracy of VTRIDENT in Predicting INSTRUCTION-SDC-VOLATILITY Versus FI (Y-axis: Accuracy)

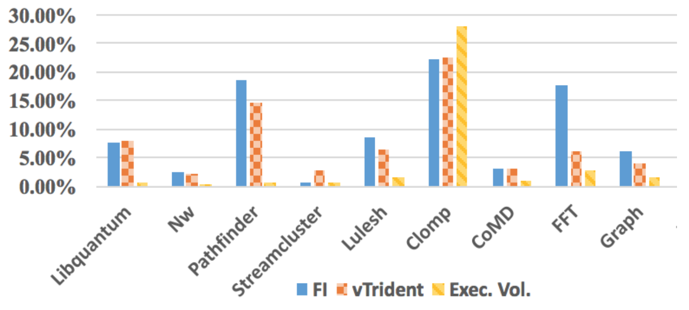


Fig. 8: OVERALL-SDC-VOLATILITY Measured by FI and Predicted by VTRIDENT, and INSTRUCTION-EXECUTION-VOLATILITY alone (Y-axis: OVERALL-SDC-VOLATILITY, Error Bar: 0.03% to 0.55% at 95% Confidence)

INSTRUCTION-EXECUTION-VOLATILITY rather than just using INSTRUCTION-EXECUTION-VOLATILITY.

## B. Performance

We evaluate the performance of VTRIDENT based on its execution time, and compare it with that of TRIDENT. We do not consider FI in this comparison as FI is orders of magnitude slower than TRIDENT [11]. We measure the time taken by executing VTRIDENT and TRIDENT in each benchmark, and compare the speedup achieved by VTRIDENT over TRIDENT. 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 that there is no parallelization for the purpose of comparison in the case of TRIDENT and VTRIDENT, though both TRIDENT and VTRIDENT can be parallelized. Therefore, the speedup can be computed by measuring their wall-clock time.

We also measure the time per input as both TRIDENT and VTRIDENT experience similar slowdowns as the number of inputs increase (we confirmed this experimentally). The average execution time of VTRIDENT is 944 seconds per benchmark per input (a little more than 15 minutes). Again, we emphasize that this is due to the considerably large input sizes we have considered in this study (Section IV).

The results of the speedup by VTRIDENT over TRIDENT are shown in Figure 9. We find that on average VTRIDENT is 8.05x faster than TRIDENT. The speedup in individual cases varies from 1.09x in *Graph* (85.16 seconds versus. 78.38 seconds) to 33.56x in *Streamcluster* (3960 seconds versus. 118 seconds). The variation in speedup is because applications have different degrees of memory-boundedness: the more memory bounded an application is, the slower it is with TRIDENT, and hence the larger the speedup obtained by VTRIDENT (as it does not need detailed memory dependency traces). For example, *Streamcluster* is more memory-bound than computation-bound than *Graph*, and hence experiences much higher speedups.

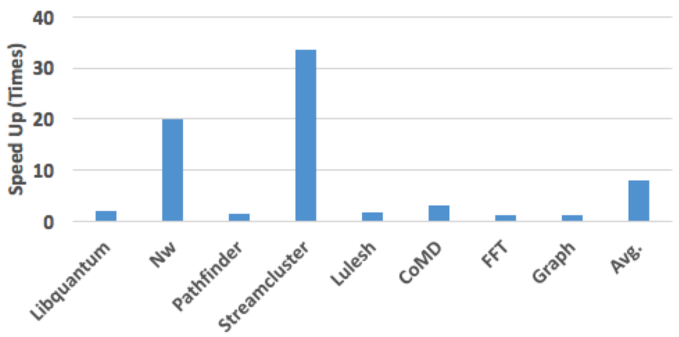


Fig. 9: Speedup Achieved by VTRIDENT over TRIDENT. Higher numbers are better.

Note that we omit *Clomp* from the comparison since *Clomp* consumes more than 32GB memory in TRIDENT, and hence crashes on our machine. This is because *Clomp* generates a huge memory-dependency trace in TRIDENT, which exceeds the memory of our 32GB-memory machine (in reality, it experiences significant slowdown due to thrashing, and is terminated by the OS after a long time). On the other hand, VTRIDENT prunes the memory dependency and incurs only 21.29MB memory overhead when processing *Clomp*.

## VII. Bounding Overall SDC Probabilities with vTRIDENT

In this section, we describe how to use vTRIDENT to bound the overall SDC probabilities of programs across given inputs by performing FI with only one selected input. We need FI because the goal of vTRIDENT is to predict the variation in SDC probabilities, rather than the absolute SDC probability which is much more time-consuming to predict (Section V-B). Therefore, FI gives us the absolute SDC probability for a given input. However, we only need to perform FI on a single input to bound the SDC probabilities of any number of given inputs using vTRIDENT, which is a significant savings as FI tends to be very time-consuming to get statistically significant results.

For a given benchmark, we first use vTRIDENT to predict the Overall-SDC-Volatility across all given inputs. Recall that Overall-SDC-Volatility is the difference between the highest and the lowest overall SDC probabilities of the program across its inputs. We denote this range by $R$. We then use vTRIDENT to find the input that results in the median of the overall SDC probabilities predicted among all the given inputs. This is because we need to locate the center of the range in order to know the absolute values of the bounds. Using inputs other than the median will result in a shifting of the reference position, but will not change the boundaries being identified, which are more important. Although vTRIDENT loses some accuracy in predicting SDC probabilities as we mentioned earlier, most of the rankings of the predictions are preserved by vTRIDENT. Finally, we perform FI on the selected input to measure the true SDC probability of the program, denoted by $S$. Note that it is possible to use other methods for this estimation (e.g., TRIDENT [11]). The estimated lower and upper bounds of the overall SDC probability of the program across all its given inputs is derived based on the median SDC probability measured by FI, as shown below.

$$[(S - R/2), (S + R/2)] \tag{3}$$

We bound the SDC probability of each program across its inputs using the above method. We also use Instruction-Execution-Volatility alone for the bounding as a point of comparison. The results are shown in Figure 10. In the figure, the triangles indicate the overall SDC probabilities with the ten inputs of each benchmark measured by FI. The overall SDC probability variations range from 1.54x (*Graph*) to 42.01x (*Lulesh*) across different inputs. The solid lines in the figure bound the overall SDC probabilities predicted by vTRIDENT. The dashed lines bound the overall SDC probabilities projected by considering only the Instruction-Execution-Volatility.

*On average, 78.89% of the overall SDC probabilities of the inputs measured by FI are within the bounds predicted by* vTRIDENT. *For the inputs that are outside the bounds, almost all of them are very close to the bounds.* The worst case is *FFT*, where the overall SDC probabilities of two inputs are far above the upper bounds predicted by vTRIDENT. The best cases are *Streamcluster* and *CoMD* where almost every input's SDC probability falls within the bounds predicted by vTRIDENT (Section VIII-A explains why).

On the other hand, Instruction-Execution-Volatility alone bounds only 32.22% SDC probabilities on average. This is a sharp decrease in the coverage of the bounds compared with vTRIDENT, indicating the importance of considering Instruction-SDC-Volatility when bounding overall SDC probabilities. The only exception is Streamcluster where considering Instruction-Execution-Volatility alone is sufficient in bounding SDC probabilities. This is because Streamcluster exhibits very little SDC volatility across inputs (Figure 6).

In addition to coverage, tight bounds are an important requirement, as a loose bounding (i.e., a large $R$ in Equation 3) trivially increases the coverage of the bounding. To investigate the tightness of the bounding, we examine the results shown in Figure 8. Recall that Overall-SDC-Volatility is represented by $R$, so the figure shows the accuracy of $R$. As we can see, vTRIDENT computes bounds that are comparable to the ones derived by *FI* (ground truth), indicating that the bounds obtained are tight.

## VIII. Discussion

In this section, we first summarize the sources of inaccuracy in vTRIDENT, and then we discuss the implications of vTRIDENT for error mitigation techniques.

### A. Sources of Inaccuracy

Other than the loss of accuracy from the coarse-grain tracking in memory dependency (Section V-B), we identify three potential sources of inaccuracy in identifying Instruction-SDC-Volatility by vTRIDENT. They are also the sources of inaccuracy in TRIDENT, which vTRIDENT is based on. We explain how they affect identifying Instruction-SDC-Volatility here.

*Source 1: Manipulation of Corrupted Bits*

We assume only instructions such as comparisons, logical operators and casts have masking effects, 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, repeated division operations such as *fdiv* may also average out corrupted bits in the mantissa of floating point numbers, and hence mask errors. The dynamic footprints of such instructions may be different across inputs hence causing them to have different masking probabilities, so vTRIDENT does not capture the volatility from such cases. For instance, in *Lulesh*, we observe that the number of *fdiv* may differ by as much as 9.5x between inputs.

*Source 2: Memory Copy*

vTRIDENT does not handle bulk memory operations such as memmove and memcpy. Hence, we may lose track of error propagation in the memory dependencies built via such operations. Since different inputs may diversify memory dependencies, the diversified dependencies via the bulk memory operations may not be identified either. Therefore, vTRIDENT may not be able to identify Instruction-SDC-Volatility in these cases.

*Source 3: Conservatism in Determining Memory Corruption*

We assume all the store instructions that are dominated by the faulty branch are corrupted when control-flow is corrupted,

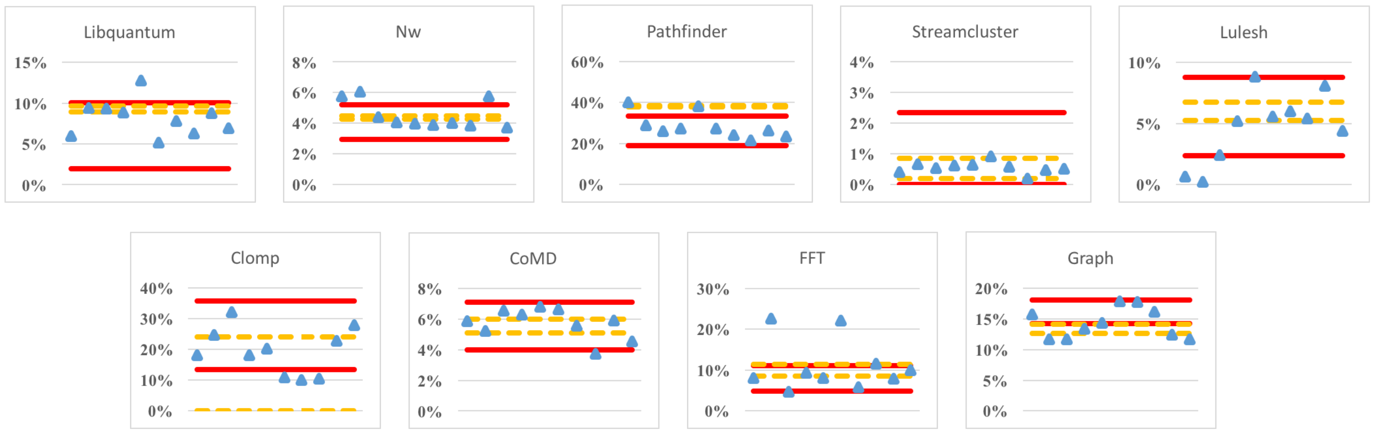


Fig. 10: Bounds of the Overall SDC Probabilities of Programs (Y-axis: SDC Probability; X-axis: Program Input; Solid Lines: Bounds derived by vTRIDENT; Dashed Lines: Bounds derived by INSTRUCTION-EXECUTION-VOLATILITY alone, Error Bars: 0.03% to 0.55% at the 95% Confidence). Triangles represent FI results.

similar to the examples in Figure 2b and Figure 2c. 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* in prior work [6]. When inputs change, the number of *lucky loads* may also change due to the changes of the distributions of such zeros in memory, possibly causing volatility in SDC. vTRIDENT does not identify *lucky loads*, so it may not capture the volatility from such occasions.

### B. Implication for Mitigation Techniques

Selective instruction duplication is an emerging mitigation technique that provides configurable fault coverage based on performance overhead budget [9], [21], [23], [24]. The idea is to protect only the most SDC-prone instructions in a program so as to achieve high fault coverage while bounding performance overheads. The problem setting is as follows: Given a certain performance overhead $C$, what static instructions should be duplicated in order to maximize the coverage for SDCs, $F$, while keeping the performance overhead below $C$. Solving the above problem involves finding two factors: (1) $P_i$: The SDC probability of each instruction in the program, to decide which set of instructions should be duplicated, and (2) $O_i$: The performance overhead incurred by duplicating the instructions. Then the problem can be formulated as a classical 0-1 knapsack problem [26], where the objects are the instructions and the knapsack capacity is represented by $C$, the maximum allowable performance overhead. Further, object profits are represented by the estimated SDC probability (and hence selecting the instruction means obtaining the coverage $F$), and object costs are represented by the performance overhead of duplicating the instructions.

Almost all prior work investigating selective duplication confines their study to a single input of each program in evaluating $P_i$ and $O_i$ [9], [21], [23], [24]. Hence, the protection is only optimal with respect to the input used in the evaluation. Because of the INSTRUCTION-SDC-VOLATILITY and INSTRUCTION-EXECUTION-VOLATILITY incurred when the protected program executes with different inputs, there is no guarantee on the fault coverage $F$ the protection aims to provide, compromising the effectiveness of the selective duplication. To address this issue, we argue that the selective duplication should take both INSTRUCTION-SDC-VOLATILITY and INSTRUCTION-EXECUTION-VOLATILITY into consideration. One way to do this is solving the knapsack problem based on the average cases of each $P_i$ and $O_i$ across inputs, so that the protection outcomes, $C$ and $F$, are optimal with respect to the average case of the executions with the inputs. This is a subject of future work.

## IX. RELATED WORK

There has been little work investigating error propagation behaviours across different inputs of a program. Czek et al. [7] were among the first to model the variability of failure rates across program inputs. They decompose program executions into smaller unit blocks (i.e., instruction mixes), and use the volatility of their dynamic footprints to predict the variation of failure rates, treating the error propagation probabilities as constants in their unit blocks across different inputs. Their assumption is that similar executions (of the unit blocks) result in similar error propagations, so the propagation probabilities within the unit blocks do not change across inputs. Thus, their model is equivalent to considering just the execution volatility of the program (Section III), which is not very accurate as we show in Section IV.

Folkesson et al. [10] investigate the variability of the failure rates of a single program (*Quicksort*) with its different inputs. They decompose the variability into the execution profile, and its data usage profile. The latter requires the identification of critical data and its usage within the program - it is not clear how this is done. They consider limited propagation of errors across basic blocks, but not within a single block. This results in their model significantly underpredicting the variation of error propagation. Finally, it is difficult to generalize their results as they consider only one (small) program.

Di Leo et al. [8] investigate the distribution of failure types under hardware faults when the program is executed with different inputs. However, their study focuses on the measurement of the volatility in SDC probabilities, rather than on predicting it. They also attempt to cluster the variations and correlate the clusters with the program's execution profile. However, they do not propose a model to predict the variations, nor do they consider sources of variation beyond the execution profile - again, this is similar to using only the execution volatility to explain the variation of SDC probabilities. Tao et al. [30] propose efficient detection and recovery mechanisms for iterative methods across different inputs. Mahmoud et al. [25] leverage software testing techniques to explore input dependence for approximate computing. However, neither of them focus on hardware faults in generic programs. Gupta et al. [12] measure the failure rate in large-scale systems with multiple program inputs during a long period, but they do not propose techniques to bound the failure rates. In contrast, our work investigates the root causes behind the SDC volatility under hardware faults, and proposes a model to bound it in an accurate and scalable fashion.

Other papers that investigate error propagation confine their studies to a single input of each program. For example, Hari et al. [14], [15] group similar executions and choose the representative ones for FI to predict SDC probabilities given a single input of each program. Li et al. [21] find patterns of executions to prune the FI space when computing the probability of long-latency propagating crashes. Lu et al. [24] characterize error resilience of different code patterns in applications, and provide configurable protection based on the evaluation of instruction SDC probabilities. Feng et al. [9] propose a modeling technique to identify likely SDC-causing instructions. Our prior work. TRIDENT [11], which vTRIDENT is based on, also restricts itself to single inputs. These papers all investigate program error resilience characteristics based on static and dynamic analysis, without large-scale FI. However, their characterizations are based on the observations derived from a single input of each program, and hence their results may be inaccurate for other inputs.

## X. Conclusion

Programs can experience Silent Data Corruptions (SDCs) due to soft errors, and hence we need fault injection (FI) to evaluate the resilience of programs to SDCs. Unfortunately, most FI studies only evaluate a program's resilience under a single input or a small set of inputs as FI is very time consuming. In practice however, programs can exhibit significant variations in SDC probabilities under different inputs, which can make the FI results inaccurate.

In this paper, we investigate the root causes of variations in SDCs under different inputs, and we find that they can occur due to differences in the execution of instructions as well as differences in error propagation. Most prior work has only considered the former factor, which leads to significant inaccuracies in their estimations. We propose a model vTRIDENT to incorporate differences in both execution and error propagation across inputs. We find that vTRIDENT is able to obtain achieve higher accuracy and closer bounds on the variation of SDC probabilities of programs across inputs compared to prior work that only consider the differences in execution of instructions. We also find vTRIDENT is significantly faster than other state of the art approaches for modeling error propagation in programs, and is able to obtain relatively tight bounds on SDC probabilities of programs across multiple inputs, while performing FI with only a single program input.

## 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] 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.

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

[3] 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.

[4] Hyungmin Cho, Shahrzad Mirkhani, Chen-Yong Cher, Jacob A Abraham, and Subhasish Mitra. Quantitative evaluation of soft error injection techniques for robust system design. In *Proceedings of the 50th Annual Design Automation Conference*, page 101. ACM, 2013.

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

[6] 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.

[7] Edward W. Czeck and Daniel P. Siewiorek. Observations on the effects of fault manifestation as a function of workload. *IEEE Transactions on Computers*, 41(5):559–566, 1992.

[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] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. Shoestring: probabilistic soft error reliability on the cheap. In *ACM SIGARCH Computer Architecture News*, volume 38, page 385. ACM, 2010.

[10] Peter Folkesson and Johan Karlsson. The effects of workload input domain on fault injection results. In *European Dependable Computing Conference*, pages 171–190, 1999.

[11] Guanpeng Li, Karthik Pattabiraman, Siva Kumar Sastry Hari, Michael Sullivan and Timothy Tsai. Modeling soft-error propagation in programs. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018.

[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, and Helia Naeimi. Low-cost program-level detectors for reducing silent data corruptions. In *International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE, 2012.

[14] 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 *ACM SIGARCH Computer Architecture News*, volume 40, page 123. ACM, 2012.

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

[16] John L Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer*, 33(7):28–35, 2000.

[17] https://asc.llnl.gov/CORAL-benchmarks/. Coral benchmarks.

[18] https://github.com/coExp/Graph. Github.

[19] https://github.com/karimnaaji/fft. Github.

[20] 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.

[21] 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.

[22] 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.

[23] Guanpeng Li, Karthik Pattabiraman, Chen-Yong Cher, and Pradip Bose. Experience report: An application-specific checkpointing technique for minimizing checkpoint corruption. In *26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 141–152. IEEE, 2015.

[24] 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.

[25] Abdulrahman Mahmoud, Radha Venkatagiri, Khalique Ahmed, Sarita V. Adve, Darko Marinov, and Sasa Misailovic. Leveraging software testing to explore input dependence for approximate computing. *Workshop on Approximate Computing Across the Stack (WAX)*, 2017.

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

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

[28] 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.

[29] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, Andrew A Chien, Paul Coteus, Nathan A DeBardeleben, Pedro C Diniz, Christian Engelmann, Mattan Erez, Saverio Fazzari, Al Geist, Rinku Gupta, Fred Johnson, Sriram Krishnamoorthy, Sven Leyffer, Dean Liberty, Subhasish Mitra, Todd Munson, Rob Schreiber, Jon Stearley, and Eric Van Hensbergen. Addressing failures in Exascale computing. *The International Journal of High Performance Computing Applications*, 28(2):129–173, 2014.

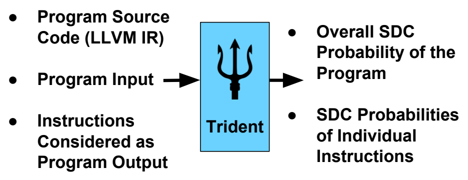
[30] Dingwen Tao, Shuaiwen Leon Song, Sriram Krishnamoorthy, Panruo Wu, Xin Liang, Eddy Z Zhang, Darren Kerbyson, and Zizhong Chen. New-sum: A novel online abft scheme for general iterative methods. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 43–55. ACM, 2016.

[31] 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.

[32] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *22nd Annual International Symposium on Computer Architecture*, pages 24–36. IEEE, 1995.

[33] Keun Soo Yim, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. Quantitative analysis of long-latency failures in system software. In *Pacific Rim International Symposium on Dependable Computing(PRDC)*, pages 23–30. IEEE, 2009.

[34] Keun Soo Yim, Cuong Pham, Mushfiq Saleheen, Zbigniew Kalbarczyk, and Ravishankar Iyer. Hauberk: Lightweight silent data corruption error detector for GPGPU. In *International Parallel & Distributed Processing Symposium (IPDPS)*, page 287. IEEE, 2011.



Program Source Code (LLVM IR)
Program Input
Instructions Considered as Program Output
Trident
Overall SDC Probability of the Program
SDC Probabilities of Individual Instructions

Fig. 11: Workflow of TRIDENT

## XI. Appendix

In this appendix, we summarize how TRIDENT [11] models error propagation in programs. This is provided for completeness, and is based on the material in our earlier paper [11].

### A. *Overview*

The overall workflow of TRIDENT is shown in Figure 11. The inputs of TRIDENT are the program's source code compiled with LLVM IR, and an input of the program. The outputs of TRIDENT are the SDC probabilities of each program instruction, and the overall SDC probability of the program with the given input.

TRIDENT consists of two phases: (1) Profiling, and (2) Inferencing. In the profiling phase, TRIDENT executes the program under the input provided, and gathers information such as instruction dependencies, and branch execution counts. These are used for constructing the model. Once the model is constructed, TRIDENT is ready for the inferencing phase, where a location of fault activation is given to TRIDENT to compute the SDC probability of the given fault. There are three sub-models in TRIDENT which model error propagation at three levels: (1) Static-instruction level, (2) Control-flow level, and (3) Memory level. The results from the three sub-models are aggregated to calculate the SDC probability of the given instruction where the fault is activated.

### B. *Static-Instruction Sub-Model*

A fault activated first propagates on its static data-dependent instruction sequence. A static data-dependent instruction sequence is a sequence of statically data-dependent instructions that are usually contained in the same basic block. TRIDENT computes a propagation probability for each instruction in the sequence, and then aggregates the probabilities to compute the probability for the fault from the activation location to the end of the sequence.

### C. *Control-Flow Sub-Model*

A static data-dependent instruction sequence usually ends with either a branch or store instruction. If it ends with a branch instruction, the fault may propagate to the branch instruction and modify the direction of the branch, leading to control-flow divergence. Consequently, the store instructions dominated by the branch may not be executed correctly, causing corruptions in memory. Control-flow sub-model identifies which store instructions are affected by the control-flow divergence, and at what probabilities.

### D. *Memory Sub-Model*

Once a store instruction is corrupted, the fault continues to propagate in memory and may finally reach the program output. Knowing which store instructions are corrupted, memory sub-model further tracks the error propagation via memory dependency. A memory dependency graph is constructed based on the memory addresses profiled from all load and store instructions, and the sub-model computes the probability for the fault in the corrupted store instruction to propagate to the program output via the memory dependency.