CSED601
Dependable Computing
Lecture 17

Jong Kim

Dept. of CSE

POSTECH

SW Reliability Paper

- “Modeling Soft-Error Propagation in Programs” DSN 18
- Related Issues

Introduction

- Prelude
  - Transient hardware faults (i.e., **soft errors**) are predicted to increase
  - 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 (energy constraint).
  - Researchers have postulated that future processors will expose hardware faults to the software and expect the software to tolerate them.
- Symptoms
  - incorrect program output and silent data corruptions (SDCs)
  - very difficult to detect and can have severe consequences
  - a small fraction of the program states are responsible for almost all the error propagations resulting in SDCs
  - important to estimate the SDC probability of a program – both in the aggregate, and on an individual instruction basis

SDC Probabilities

- FI and SDC
  - 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.
  - 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

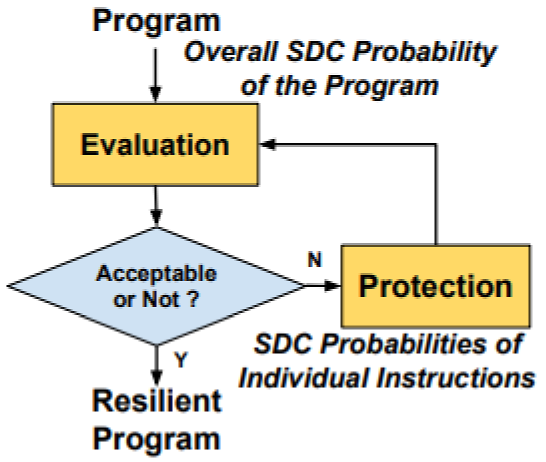
- Analytical Model and SDC
  - Attempted to analytically model error propagation to identify vulnerable instructions
  - Advantage: scalability, do not require FIs, and are fast to execute.
  - Disadvantage: suffer from a lack of accuracy due to modeling limitation.
    - Modeling faults in the normal (i.e., fault-free) control-flow path of the program.
    - A fault can propagate to not only the data-dependencies of an instruction, but also to the subsequent branches and memory locations that are dependent on it.
    - Tracking the deviation in control-flow and memory locations

      due to a fault often leads to state space explosion.

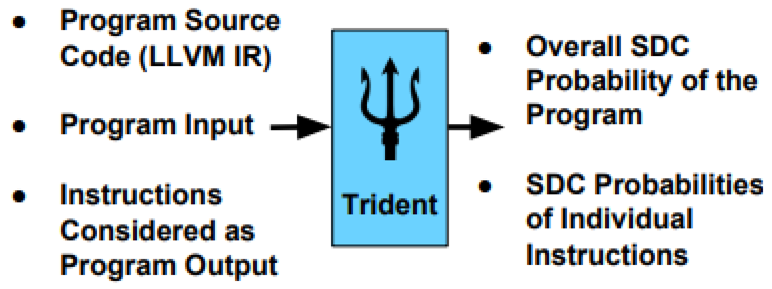
TRIDENT: A Model for Tracking E-P

- Key Idea of TRIDENT
  - Tracking error propagation in programs that addresses the two challenges.
    - **Tracking the deviation in control-flow and memory locations**
  - The key insight is that
    - error propagation in dynamic execution can be decomposed into a combination of individual modules
    - each can be abstracted into probabilistic events.
    - 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.
- Implementation and experiments
  - In the LLVM compiler
  - Evaluate its accuracy and scalability vis-a-vis FI
- Contribution
  - The first to propose a model to estimate the SDC probability of individual instructions and the entire program without performing any FIs

Development Process


Program
Overall SDC Probability of the Program
Evaluation
Acceptable or Not ?
N
Protection
Y
SDC Probabilities of Individual Instructions
Resilient Program

(a) Development Cycle


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

(b) Workflow of TRIDENT

Fig. 1: Development of Fault-Tolerant Applications

Background

- Fault model
  - Transient hardware faults occurring in the computational elements of the processor, including pipeline registers and functional units.
  - Do not consider faults in the memory or caches (protected with error correction code (ECC))
  - Do not consider faults in the processor’s control logic as we assume that it is protected.
  - Do not consider faults in the instructions’ encodings.
  - Assume that the program does not jump to arbitrary illegal addresses due to faults during the execution
  - 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).

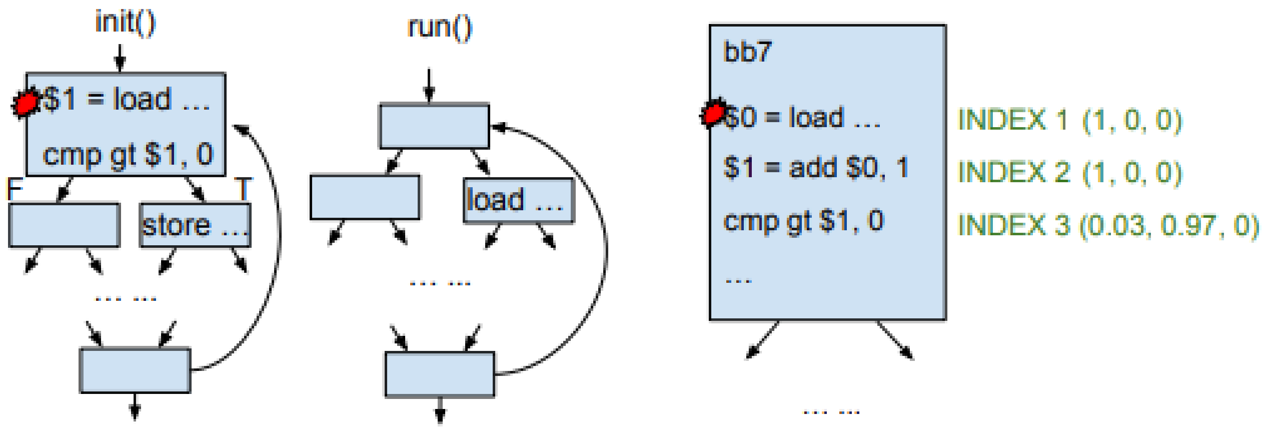
Background

- Other terminology
  - Fault Occurrence: the occurrence of a hardware fault in the processor
  - Fault Activation: 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).
  - 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: the fault was activated, and has affected some other portion of the program state, say ’X’.
  - SDC Probability: the probability of an SDC given that the fault was activated

Background

- LLVM Compiler
  - Three reasons of using LLVM.
    - LLVM uses a typed intermediate representation (IR)
      - easily represent source-level constructs
      - preserves the names of variables and functions, which makes source mapping feasible.
      - allows us to perform a fine-grained analysis of which program locations cause certain failures and map them to the source code.
    - LLVM IR is a platform-neutral representation
      - abstracts out many low-level details of the hardware and assembly language.
      - aids in portability of our analysis to different architectures
      - simplifies the handling of the special cases in different assembly language formats.
    - LLVM IR has been shown to be accurate for doing FI studies, and there are many fault injectors developed for LLVM.
      - Many of the papers we compare our technique also use LLVM infrastructure

Running Example



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

- Two functions: init() and run(): a loop in each function, the one in init() updates an array, and the one in run() reads the array for processing.
- 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

Challenges

- Three challenges in modelling error propagation
  - Statically modeling error propagation in dynamic program execution requires a model that **abstracts the program data-flow in the presence of faults.**
  - 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: **state space explosion**.
  - Faults may corrupt memory locations and hence continue to propagate through memory operations. Tracing error propagations among these memory dependencies requires constructing **a huge data dependency graph,** which is very expensive.

TRIDENT: Inputs and Outputs

- Inputs and Outputs
  - User-supplied three inputs
    - The program code compiled to the LLVM IR
    - A program input to execute the program, its execution profile
    - The output instruction(s) in the program that are used for determining if a fault resulted in an SDC
  - Two working phases
    - Profiling: executes the program, performing dynamic analysis of the program to gather information such as the count and data dependency of instructions.
    - Inferencing: automatically computes
      - (1) the SDC probabilities of individual instructions
      - (2) the overall SDC probability of the program
  - For inference, 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.

TRIDENT: Overview

- 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: Overview

- Static Instruction Sub-model (f_s)
  - used to trace error propagation of an arbitrary fault activated on a static data-dependent instruction sequence.
  - determines the propagation probability of the fault from where it was activated to the end of the sequence.
  - Previous models trace error propagation in data dependant instructions based on the dynamic data dependency graph (DDG)
  - detailed DDGs are very expensive to generate and process, and hence the models do not scale.
  - fs 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

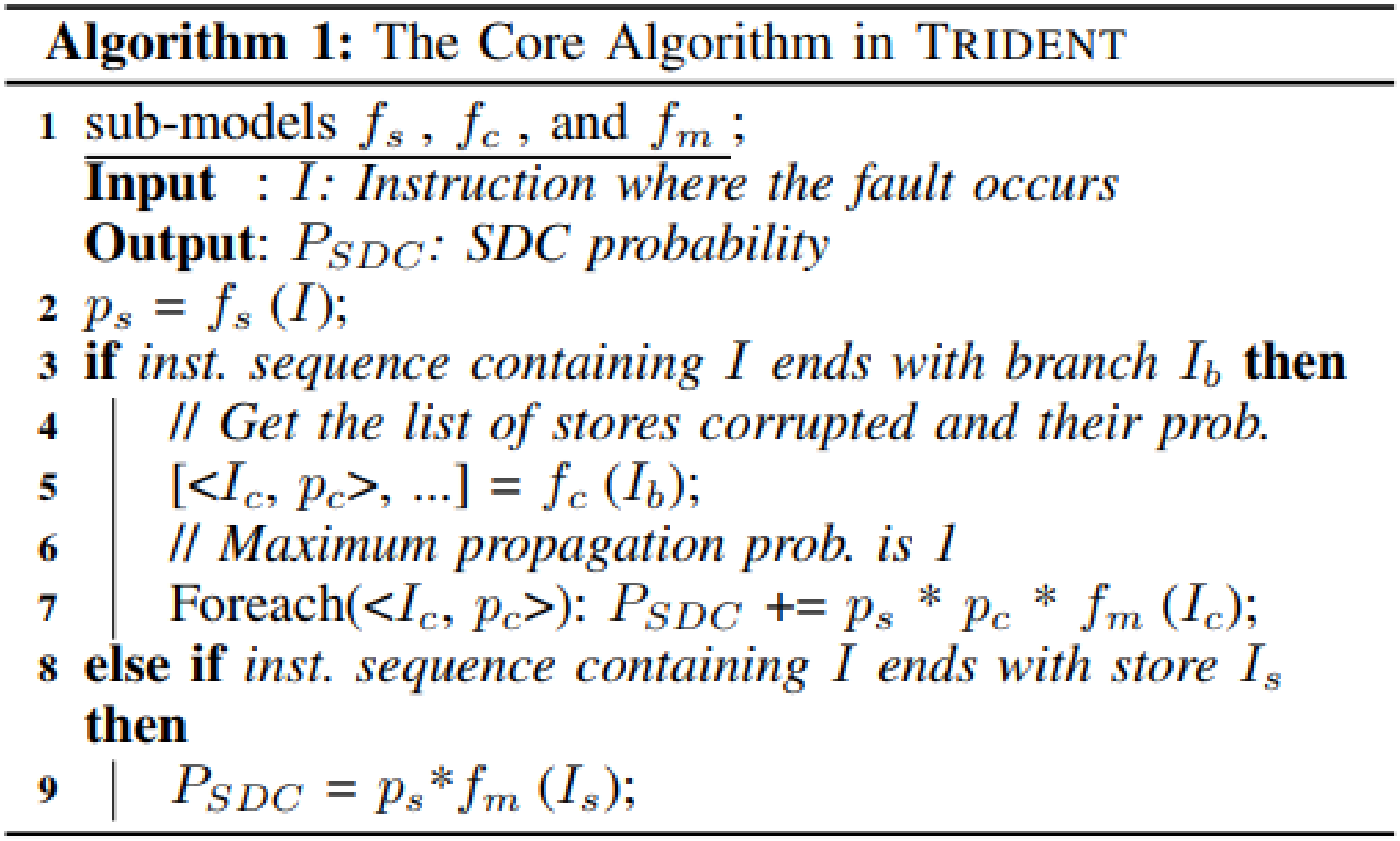
TRIDENT: Overview

- Control-flow Sub-model (f_c)
  - A fault may cause the execution path to diverge from its fault-free execution.
  - 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
  - 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.

TRIDENT: Overview

- 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 computes the propagation probability from the corrupted store instruction to the program output
  - A memory data-dependency graph needs to be generated for tracing propagations at the memory level
  - This graph can be expensive to construct and traverse due to the huge number of the dynamic store and load instructions in the program.
  - We find that the graph can be pruned by removing redundant dependencies between symmetric loops.

TRIDENT: The Core Algorithm



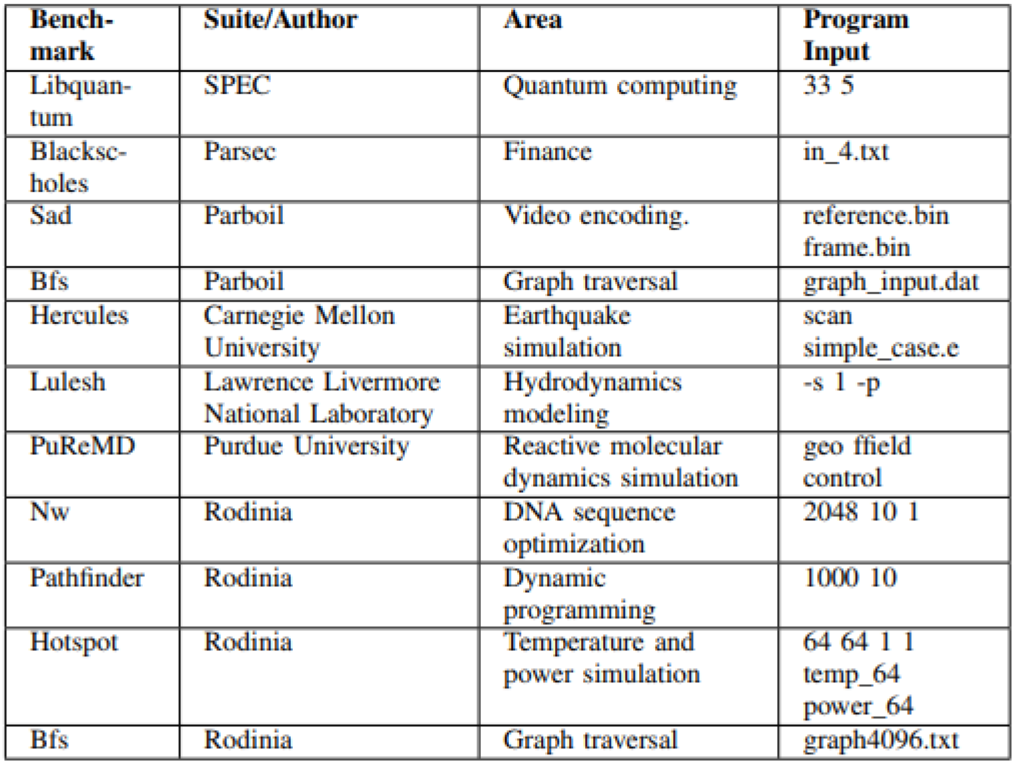
Algorithm 1: The Core Algorithm in TRIDENT

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

Evaluation

- Experiment Setup
  - Benchmarks

TABLE I: Characteristics of Benchmarks



Bench-mark	**Suite/Author**	**Area**	**Program Input**
Libquan-tum	SPEC	Quantum computing	33 5
Blacksc-holes	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

Evaluation

- Experiment Setup
  - FI Method: LLFI: a publicly available open-source fault injector to perform FIs at the LLVM IR level on these benchmarks
- Accuracy
  - The first experiment examines the prediction of overall SDC probabilities of programs
  - The second examines predicted SDC probabilities of individual instructions

Accuracy

- Overall SDC probability

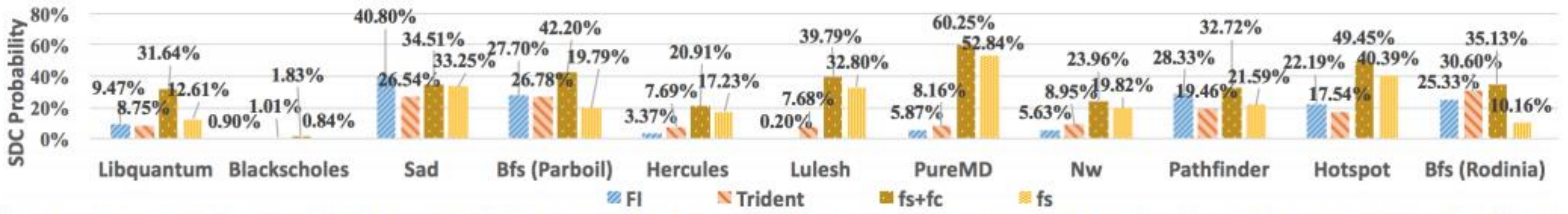
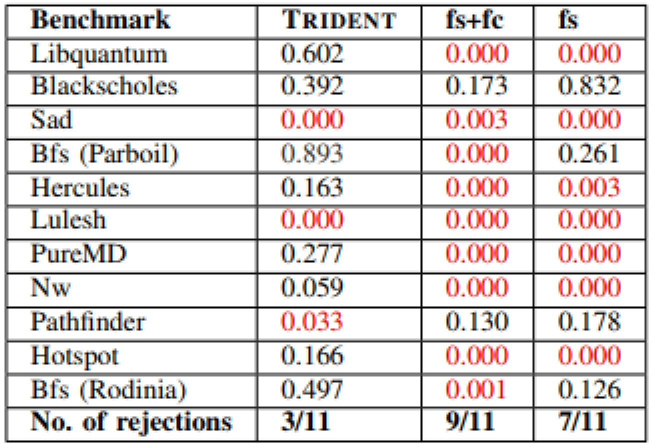


Fig. 5: Overall SDC Probabilities Measured by FI and Predicted by the Three Models (Margin of Error for FI: ±0.07% to ±1.76% at 95% Confidence)

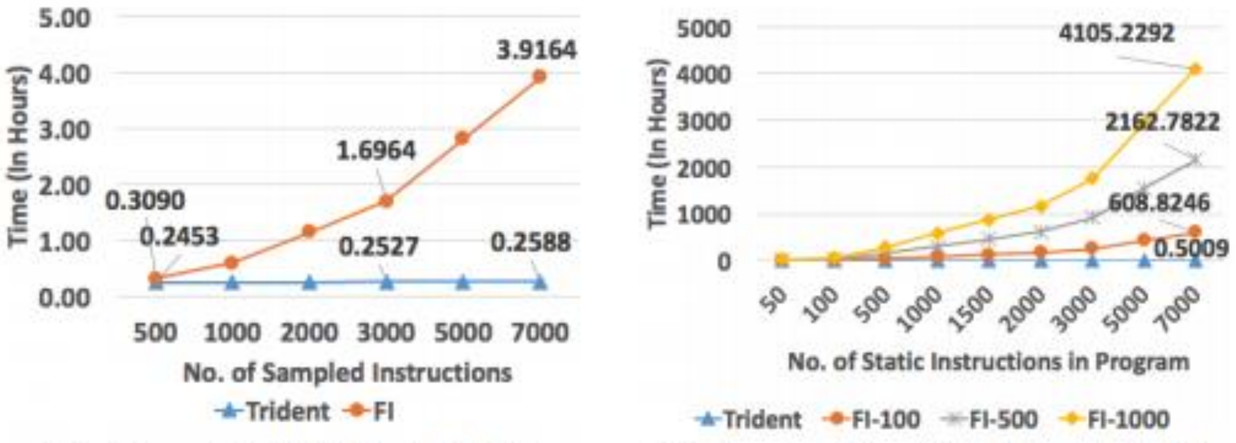
- SDC Probability of Individual Instructions:



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**

Scalability

- Overall SDC probability
- Instruction SDC probability



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

Fig. 6: Computation Spent to Predict SDC Probability

Time taken

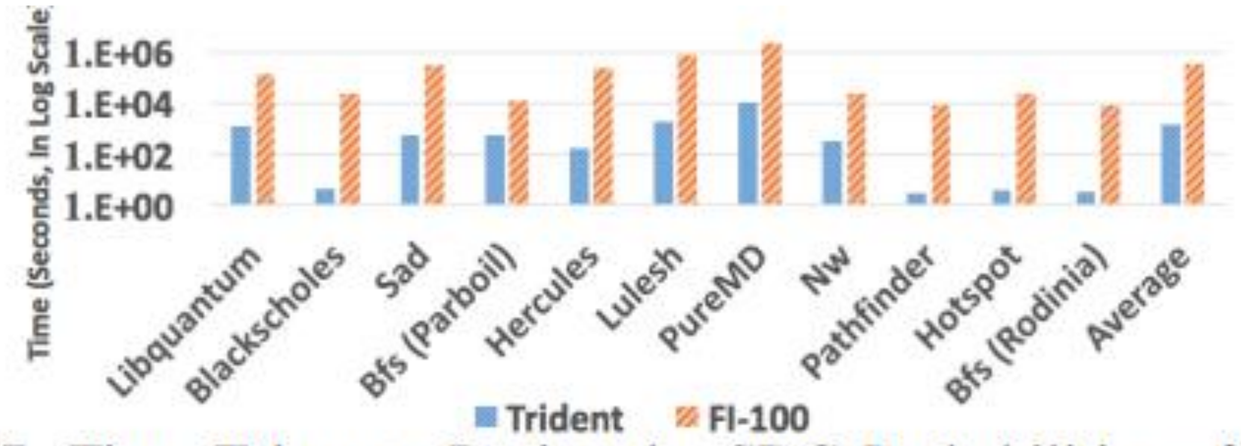


Fig. 7: Time Taken to Derive the SDC Probabilities of Individual Instructions in Each Benchmark

Discussion

- Sources of inaccuracy
  - Errors in Store Address
  - Memory Copy
  - Manipulation of Corrupted Bits
  - Conservatism in Determining Memory Corruption
- Threats to validity
  - Benchmarks
  - Platforms
  - Program Input
  - Fault Injection Methodology

Conclusion

- proposed TRIDENT, a three-level model for soft error propagation in programs.
- abstracts error propagation at static instruction level, control-flow level and memory level, and does not need any fault injection (FI).
- implemented TRIDENT in the LLVM compiler, and evaluated it on 11 programs.
- 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.
- plan to extend TRIDENT to consider (1) Multiple inputs of a program, and (2) Platforms other than CPUs, such as GPUs or special-purpose accelerators.