ZOFI: Zero-overhead Fault Injection Tool for Fast Transient Fault Coverage Analysis

Vasileios Porpodas

Intel Corporation

HAILONG JIANG

Sept. 20th, 2019

BACKGROUND AND MOTIVATION

Overview of the Fault-Coverage Evaluation Methodology for transient Faults

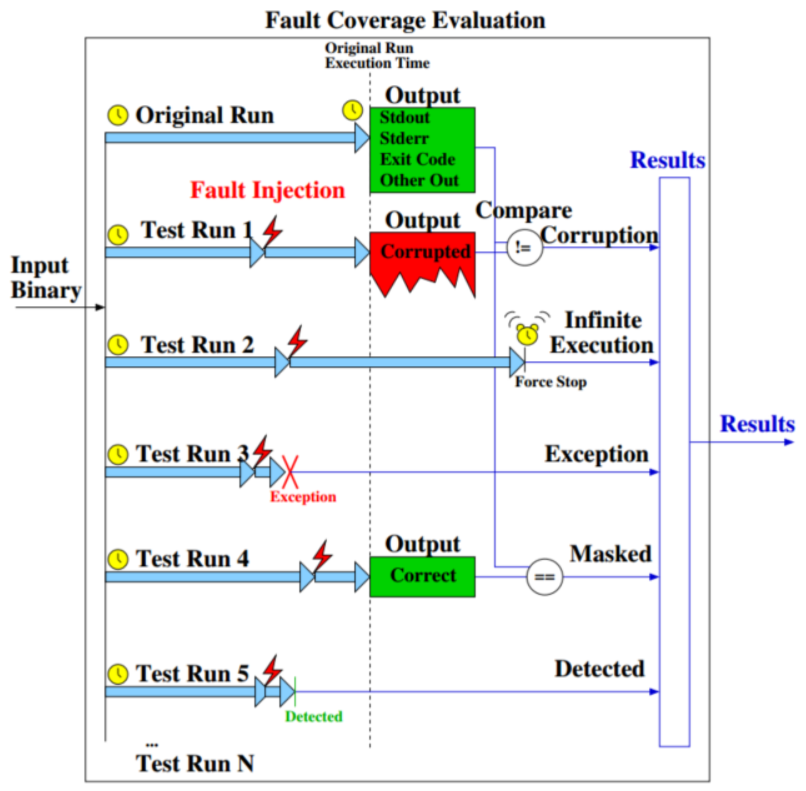


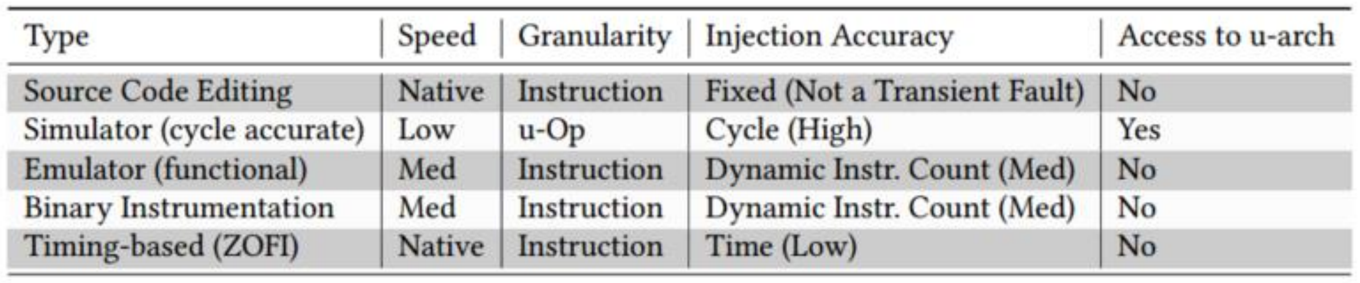
Figure 1. A typical fault coverage evaluation process.

Question: Achieving good accuracy, we need to run application hundreds and thousands of time, so overhead is a big concern.

RELATED WORK

- Type of Fault-injection Tools
  - **Source Code Editing:** modify instructions in the source code or in the compiler intermediate representation
  - **Simulator and Emulator:** modify a processor simulator: F-SEFI
  - **Binary, Source or Compiler-IR Instrumentation:** build a custom binary-instrumentation tool using a framework like Pin.
  - Timing based tool: use a random time instead of a random dynamic instruction.

Table 1. High-level comparison of fault injection tool designs.



Type	Speed	Granularity	Injection Accuracy	Access to u-arch
Source Code Editing	Native	Instruction	Fixed (Not a Transient Fault)	No
Simulator (cycle accurate)	Low	u-Op	Cycle (High)	Yes
Emulator (functional)	Med	Instruction	Dynamic Instr. Count (Med)	No
Binary Instrumentation	Med	Instruction	Dynamic Instr. Count (Med)	No
Timing-based (ZOFI)	Native	Instruction	Time (Low)	No

ZOFI

- Based on the time-based design, that executes the unmodified binary on the native hardware with no extra overheads whatsoever.
- It executes the test binary, then pause it a given time for the fault injection.
- Once the binary is stopped, the ZOFI tool can modify the state of the target workload by injecting a fault, implemented as a register bit-flip, and then resumes its execution.
- The workload will either execute to completion/ interrupted/ forced stop
- Comparing the result with golden run
- Repeating for a large number of test, and collecting, summarizing and reporting results

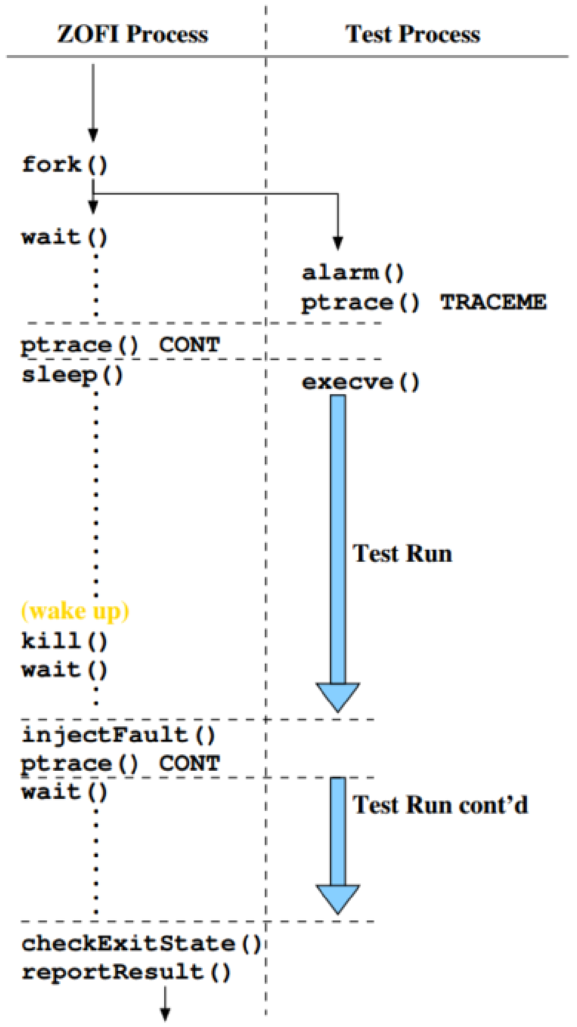
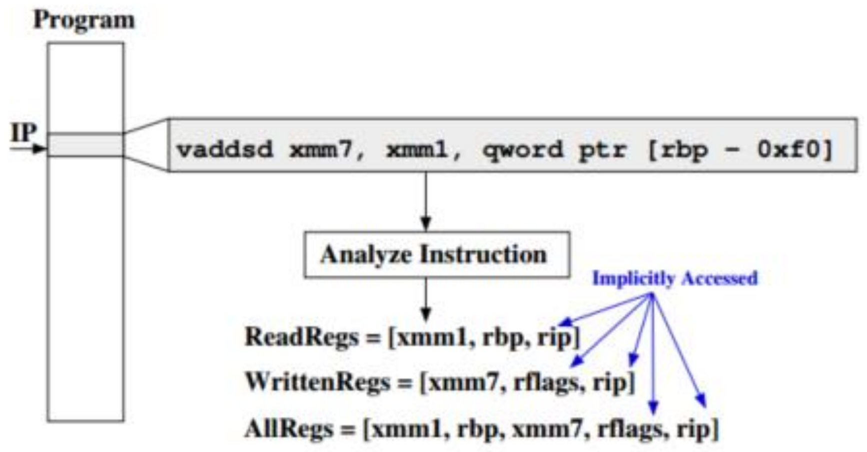


Figure 2. The design of the fault-injection run.

MODELING OF A TRANSIENT FAULT



```
vaddsd xmm7, xmm1, qword ptr [rbp - 0xf0]
```

ReadRegs = [xmm1, rbp, rip]
WrittenRegs = [xmm7, rflags, rip]
AllRegs = [xmm1, rbp, xmm7, rflags, rip]

Figure 3. Analyzing the instruction at the current Instruction Pointer (IP), to collect the accessed registers for each register type.

RESULTS: OVERHEAD

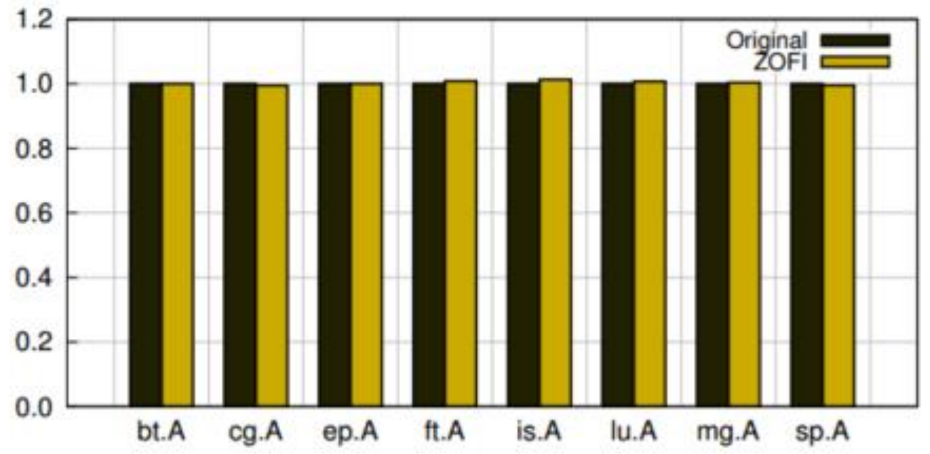
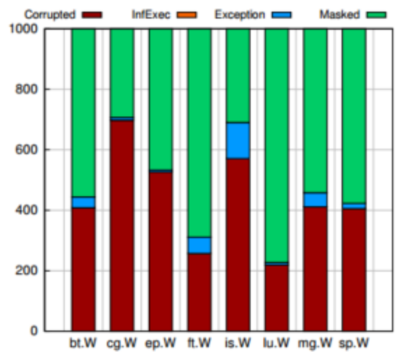
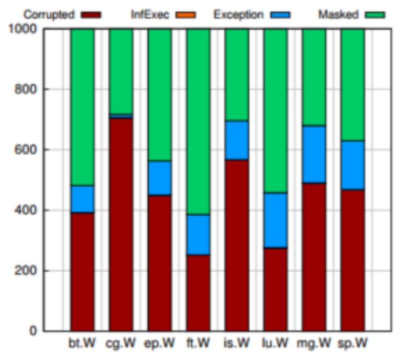


Figure 5. Execution time normalized to the original (native) execution. The ZOFI tool introduces no performance overhead.

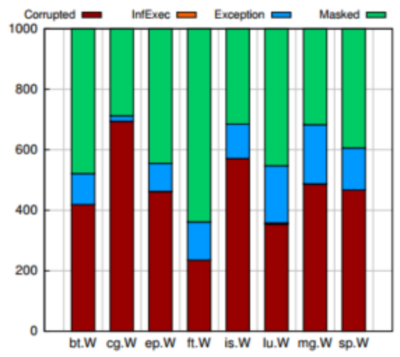
CASE STUDY: NAS BENCHMARKS



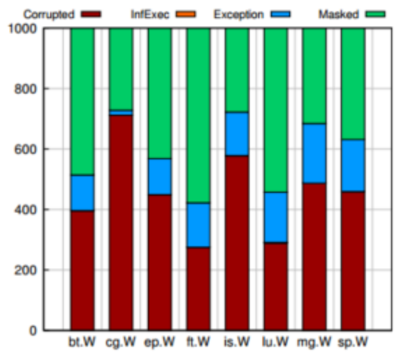
(a) Fault coverage with faults into explicit output registers only ("we").



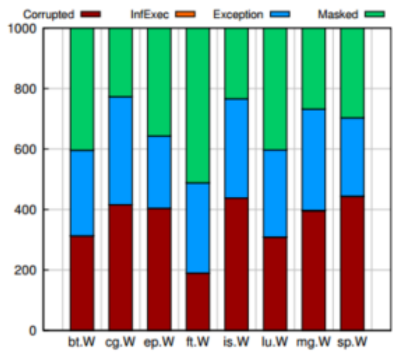
(b) Fault coverage with faults into explicit input and output registers ("rwe").



(c) Fault coverage with faults into all input and output registers, including the implicitly accessed ones ("rwei").



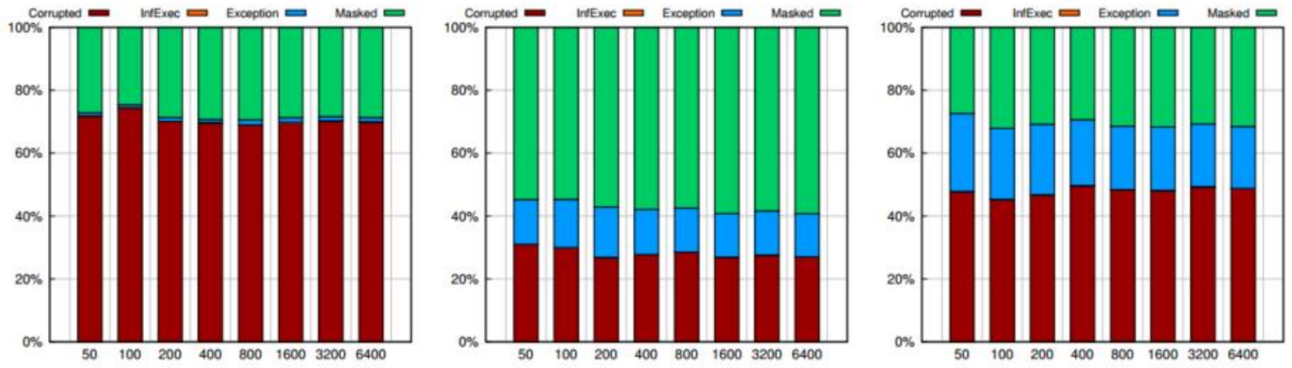
(d) Fault coverage with faults into all input and output registers, including the instruction pointer for control-flow changing instructions.("rweic").



(e) Fault coverage with faults into all input and output registers, including the instruction pointer for all instructions.("rweico").

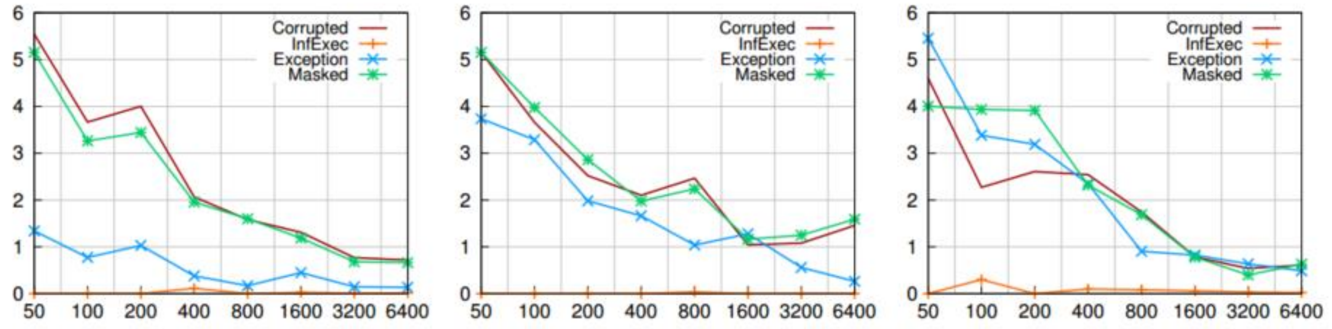
Figure 6. Fault coverage of NPB2.3, for various types of injected registers. The configuration for these tests is shown in Table 2

ACCURACY OF RESULTS



(a) cg.W (b) ft.W (c) mg.W

Figure 7. Mean fault coverage (%) across 10 repetitions of ZOFI, as we vary the number of runs from 50 to 6400, injecting into explicitly written registers ("rwe").



(a) cg.W (b) ft.W (c) mg.W

Figure 8. Standard deviation of the fault coverage results across 10 repetitions of ZOFI, as we vary the number of runs from 50 to 6400. The mean values of these experiments are shown in Figure 7.

LIMITATION

The main limitation of ZOFI is not due to its implementation, but rather due to its timing-based design. If the target workload runs for a very small amount of time, then ZOFI won't be able to effectively inject faults, often attempting to inject the fault after the workload has finished executing. This is a common problem to all timing-based tasks on real systems.

For example, even measuring the absolute execution time of a workload is not reliable when the execution lasts for a very short time. When we get close to the time accuracy of the system, any timing-based functionality becomes unreliable.

We observed that it is best if the workload runs for at least tens or hundreds of milliseconds.

Another limitation is the evaluation of workloads that misbehave when being stopped with a signal. ZOFI relies on signals to pause the applications, therefore it cannot properly analyze such types of workloads