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

Vasileios Porpodas
Intel Corporation
vasileios.porpodas@intel.com

Abstract

The experimental evaluation of fault-tolerance studies relies on tools that inject errors while programs are running, and then monitor the execution and the output for faulty execution. In particular, the established methodology in software-based transient-fault reliability studies, involves running each workload hundreds or thousands of times, injecting a random bit-flip in the process. The majority of such studies rely on custom-built fault-injection tools that are based on either a modified processor simulator, or an instrumentation framework. Such tools are non-trivial to develop, and are usually orders of magnitude slower than native execution.

In this paper we present ZOFI, a novel timing-based fault-injection tool that is aimed at being used in fault-coverage studies for transient faults. ZOFI is a zero-overhead tool, meaning that the analyzed workload runs at native speed. This is orders-of-magnitude faster compared to common approaches that are designed around simulators or code instrumentation tools. ZOFI is free software and is available at https://github.com/vporpo/zofi.

1 Introduction and Related Work

Transient faults, also known as soft errors, are faults that occur once and do not persist [40]. They are a major cause of reliability issues, as it has been shown in several studies [9, 26, 37, 41]. These faults can be attributed to a range of factors, including alpha particles striking the silicon circuits, fluctuations in the power supply and others. Such events can cause bit-flips in digital circuits, which can corrupt the state of the logic. A transient fault in a processor can change the outcome of instructions, possibly leading to wrong execution, which can potentially cause a system crash. Transient faults are fairly frequent in large data centers [26], due to the large number of systems, and therefore such systems have to be designed with transient faults in mind.

Software-based error detection techniques, e.g., [6, 12, 27–29, 31, 33, 38, 46, 48, 49] , aim at providing reliability at the software level. Such techniques work by replicating the code at either the instruction, the thread or the process level, in an attempt to detect transient faults. The evaluation of their effectiveness is done with the help of fault-injection tools which apply random register bit-flips and check the result for either: (i) successful detection by the technique, (ii) data corruption, (iii) infinite execution, (iv) exception, or (v) correct execution, also known as masked errors.

1.1 Types of Fault-injection Tools

Fault-coverage evaluation for transient faults implements the single-event upset model. It requires the use of a tool that runs the target workload, pauses it, injects a bit-flip, resumes its execution and then tracks its execution to check for a faulty outcome. In the error detection literature this is commonly done in various ways:

1. **Source Code Editing:** Delete, or modify instructions in the source code or in the compiler intermediate representation. This is a straight-forward approach, but it does not accurately model transient faults. An example of a study that uses this is [30].
2. **Simulator/Emulator:** Modify a processor simulator such that it can inject a fault at a given execution cycle. Depending on the accuracy of the simulation, this can vary from extremely slow (e.g., at the RTL-level), to quite slow (tens/hundreds of MIPS) for the fastest emulators. This is a very popular technique and has been used in several studies, including [5, 11, 13, 18, 20, 22, 23, 27, 28, 36, 39].
3. **Binary, Source, or Compiler-IR Instrumentation:** Build a custom binary-instrumentation tool using a framework like Pin[25], and implement fault-injection with it, or use a tool based on this approach, like LLFI [43], PINFI [47] and SASSIFI [19]. The instrumentation overhead is still quite significant, leading to execution much shower than the non-instrumented binary. A few examples of studies using this approach are [7, 10, 16, 17, 21, 24, 32–35, 38, 46].
4. **Timing-based Tool:** Run the unmodified binary on native hardware, just like with a debugger, pause the execution, inject the fault and resume the execution at native speed. This is by far the fastest of the three, and as we explain in the text, it is also equally accurate for statistical fault coverage analysis as long as micro-architectural accuracy is not required. We are aware of a relatively small number of studies that use this approach [8, 29, 42, 44, 45, 48].

In these studies a custom in-house tool was developed, usually designed around a debugger, like GDB [15].

In this paper we present ZOFI, a complete fault-coverage analysis tool, based on the time-based design, that executes the unmodified binary on native hardware with no extra overheads whatsoever. It executes the test binary at full native speed, then pauses it a given time for the fault injection to occur. 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, or will get interrupted by a signal. At that point ZOFI compares its output against the original run, and determines the type of the execution outcome. In case of an infinite execution, ZOFI will force-stop the workload. After repeating this process for a large number of such test runs and collecting statistics data, the final results are summarized and reported to the user.

The ZOFI tool is free software, distributed under the GPL version 2.0 license. The project is hosted at https://github.com/vporpo/zofi and is implemented in C++. The tool is designed for modern x86_64 Linux-based systems.

2 Background and Motivation

This section presents an overview of the established fault coverage evaluation methodology (Section 2.1), and motivates the need for timing-based injection tools, like ZOFI, in Sections 2.2 and 2.3.

2.1 Overview of the Fault-Coverage Evaluation Methodology for Transient Faults

An overview of a typical fault coverage evaluation of a binary is shown in Figure 1. The input to the process is the binary that we are analyzing, along with its inputs, which are not shown in the figure.

The first step is to run the binary, measure its execution time (or cycles) and collect all its outputs. This includes `stdout`, `stderr`, exit code and all output files. We refer to this initial run as the "Original Run", and it is shown as the top-most run in Figure 1. The execution time (cycles) of the original run is used:

1. as an upper bound to the fault injection time (cycle), and
2. as an estimate of how long the analyzed workload should take to complete, such that we can detect and force-stop infinite executions.

The original output is used to check the output of the test runs, as we will explain shortly.

The rest of the process involves a large number of "Test Runs" which is where fault injection takes place. Each of these runs executes the same binary with the same inputs, but their state gets altered by a fault-injection event (depicted as a red lightning in the figure), at some point during their execution. The injection of the fault takes place at a random point in time (cycle) between zero and the execution time (cycle) of the original binary. The type of fault injected depends on our configuration, but a common fault type is a bit-flip into registers to simulate transient faults in the computational logic. After fault injection the binary resumes its execution and is let free to execute to completion.

Now let's take a look at the fault injection runs. In "Test Run 1" of Figure 1, the fault leads to output (the red deformed block) that differs form that of the original run. This is caught by the comparator component, which reports a "Corruption".

Another example of a test run is "Test Run 2", where the state of the execution gets damaged by the fault in such a way that leads to infinite execution. This can be caused, among others, by faults that lead to a faulty evaluation of the condition bit of a loop latch. If the execution of the binary takes far longer than the original execution time, an alarm goes off that forces the workload to stop. This is reported as "Infinite Execution".

A third example is "Test Run 3", where the fault leads to an illegal operation that triggers an exception. Common examples are: memory instructions that attempt to access memory outside their allocated memory, floating point exceptions and others. This is reported as an "Exception".

"Test Run 4" shows an example where the injected fault does not cause any failure whatsoever. The test run executes to completion, its output is identical to the original one, and it did not trigger any exception. This can happen when the bit affected by the fault remains unused by subsequent computation. For example, it may be overwritten or discarded by the rest of the execution. This type of run reports is reported as "Masked" (also known as "Benign").

Finally, in error detection studies, the system itself has a way of detecting errors and reporting them to the fault-injection framework. This is shown in "Test Run 5". The error detection system can notify the injection tool that it has detected an error in several ways: For example, it can return a specific exit code, or it can print specific output to `stderr`. This is reported as "Detected".

2.2 Fault Coverage Analysis is Time Consuming

Fault coverage is a Monte-Carlo statistical analysis that relies on a large number of experiments. In each experiment we run the workload and we inject a random fault

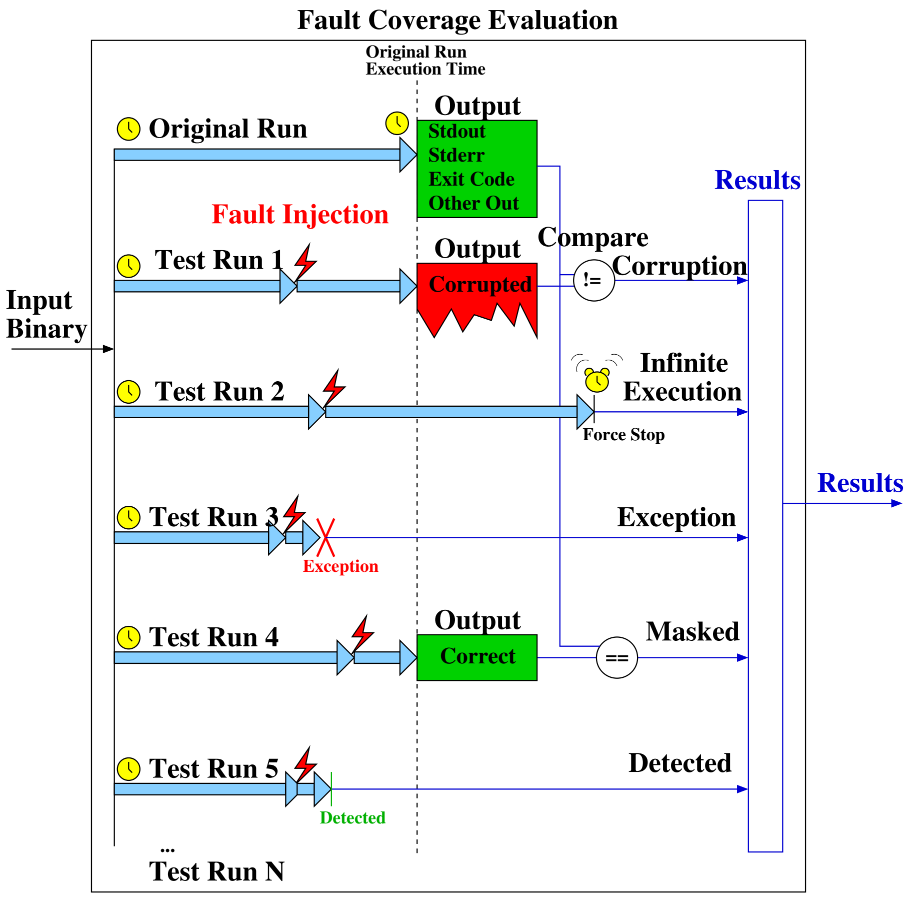


Figure 1. A typical fault coverage evaluation process.

into it at a random time. Thus, the time it takes to run all these experiments is comparable to running all these workloads to completion.

Achieving good accuracy, requires us to run the workload hundreds or thousands of times. Therefore, if our fault-coverage analysis tool introduces a large performance overhead, for example by using a simulator for running the workload, then the analysis becomes impractical even for workloads that would run relatively fast on native hardware. The ZOFI timing-based injection tool is designed to make such analysis practical, even for long-running workloads, by removing the overhead introduced by the fault-injection tool.

2.3 Timing-based Tool: No Need for Cycle Accuracy in Fault-Coverage Analysis

As explained in Section 1.1, there are several types of fault injection tool designs, which are summarized in Table 1. Each of them operates at different speeds, has different fault-injection granularity and accuracy, and may or may not have access to micro-architectural components. In this work, we argue that for most practical use-cases the timing-based design is the best option, because it provides the maximum possible test speed without sacrificing the accuracy of our statistical analysis.

Since fault coverage is a statistical analysis that does not require cycle accuracy, there is no real need to sacrifice execution speed for cycle accuracy. Therefore, we believe that the proposed timing-based design of ZOFI should be the choice of preference for most studies, similarly to sampling-based profiling tools (e.g., [2, 3]) which are adequate for many profiling scenarios.

Simulator-based tools give us access to the micro-architectural state, which is not accessible by the other techniques. However, this comes at the cost of a huge overhead in execution speed, limiting its practical uses to small execution kernels or traces. We believe that for any fault-coverage study that does not need access to micro-architectural state, it is more preferable to use any of the alternative techniques.

Therefore, once the cycle accuracy is not required, one could use any of the other designs interchangeably, with an equivalent outcome. None of the emulators, the

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

instrumentation-based, or the timing-based tools have access to the micro-architectural components, as they are all limited to the state exposed by the ISA. Moreover, while emulators and instrumentation-based tools do have an instruction-based injection accuracy (as they can stop at precisely a specific number of dynamic instructions), we argue that this is not really needed for statistical analysis, as the faults get injected at random cycles anyway. Therefore, we can safely use a random time instead of a random dynamic instruction, and this should give us equivalent results. As a result, timing-based fault injection tools, like ZOFI, are as effective as emulators and instrumentation-based tools, in their fault injection analysis, but with the additional benefit of being orders of magnitude faster.

3 ZOFI Design

This section describes ZOFI in more detail. It provides a description of the design and implementation, and lists a set of important features.

3.1 Overview

Just like any other fault-coverage analysis tool (as shown in Figure 1), the high-level steps followed by the ZOFI tool are the following:

1. Run the workload with no fault injection to measure its execution time and collect its original output. We refer to this as the "Original Run".
2. Fork and spawn a new process for the test run of the workload. This is where fault injection will take place.
3. In the meantime, pick a random time point between zero and the execution time of the original run. This will be the time point when the fault injection will take place. Sleep for this amount of time.
4. Wake up and stop the test process. Analyze the currently executed instruction and randomly pick a register. The type of register to be picked is controlled by the user options provided.
5. Flip a random bit of that register.
6. Let the process continue to completion.
7. Collect the output and compare it against the original run.

3.2 Design

ZOFI is designed around the `ptrace` Linux system call, that provides the means by which one process (the ZOFI tool) can observe and control the execution of another process (the workload). This is commonly used by debuggers for observing, controlling and updating the state of the debugged binary. The `ptrace` calls give you full access to the register state, the memory and the code. It is therefore a perfect fit for our fault injection tool.

Fault injection works as follows. The execution of the workload is interrupted by a signal emitted by the main process of the tool, and its register state is imported into the tool. So at this point, ZOFI has access to the instruction pointer, but does not know what type of instruction this is, in order to analyze it and access the registers it reads/writes. The analysis of the instruction is done using the capstone [1] library. This provides the list of the accessed registers, the type of each access (read/write), whether this is an explicitly or implicitly accessed register and other information, giving the user a good level of control on the type of registers that are targeted by faults.

An overview of the design of the fault-injection part of ZOFI is shown in Figure 2. The tool process (tracer) forks a new child process which becomes the tracee. Initially the child process sets up an alarm so that it receives a signal after some time if it gets stuck in an infinite execution loop. After an initial synchronization, the tool process sleeps for a random time until it is time for the fault injection. Meanwhile the child process switches to the target binary image with a call to `execve()`. Once the tracer process wakes up, it stops the tracee with a `kill()`, and after a synchronization point it performs the fault injection. Then it lets the tracee continue, and waits for it to either run to completion, or stop with an exception, or never complete due to infinite execution. After the final synchronization point, the tool's main process compares the exit state of the test run with that of the original run and reports the result to the results-collection part of the tool.

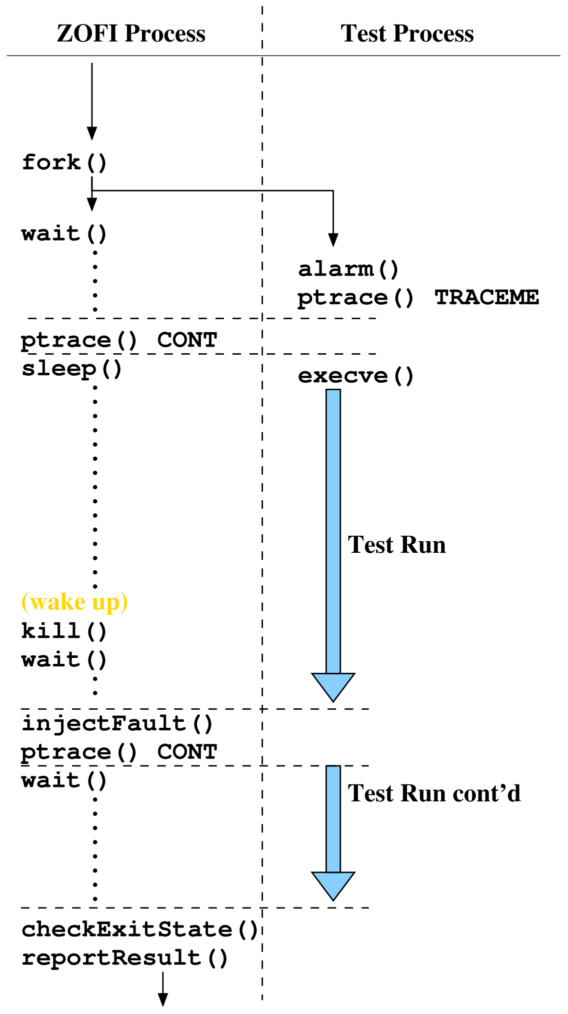


Figure 2. The design of the fault-injection run.

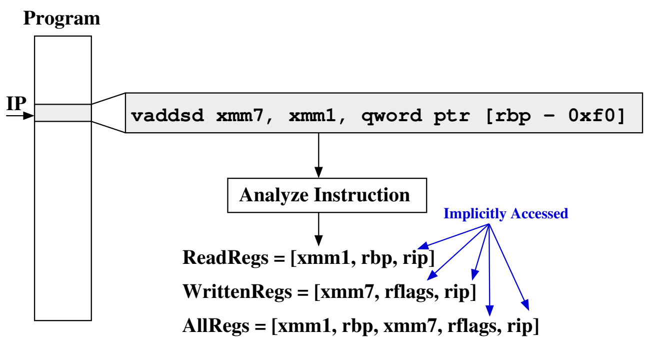

Program
IP
vaddsd xmm7, xmm1, qword ptr [rbp - 0xf0]
Analyze Instruction
Implicitly Accessed
ReadRegs = [xmm1, rbp, rip]
WrittenRegs = [xmm7, rflags, rip]
AllRegs = [xmm1, rbp, xmm7, rflags, rip]

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

3.3 Modeling of a Transient Fault

We model the transient fault in a similar way as in most fault injection studies: with a random bit-flip in a random register from those accessed by the current instruction. Currently, ZOFI implements a register bit-flip injection, since this is the most widely used model in the studies, but this could be easily extended to include bit-flips in memory. The reasoning is that ECC memory is commonly used in server-grade hardware, and therefore such systems can automatically recover from transient faults in memory cells.

The random register bit-flip is implemented as follows. After the child process gets stopped by the parent, the instruction pointer points to the instruction that is about to be executed. ZOFI analyzes the instruction using the capstone library [1], and collects the registers read and written either explicitly or implicitly (for example the instruction pointer, or the condition flags in x86). An example of this is shown in Figure 3, where an instruction is analyzed and the registers accessed are collected into three pools: one for the registers being read, one for those being written, and one for all registers being accessed. Depending on the user's input, ZOFI may filter out some of these registers, for example the user may wish to skip the implicitly accessed registers, or collect the instruction pointer only when the instruction is a control-flow one, or even restrict the errors to registers read or written.

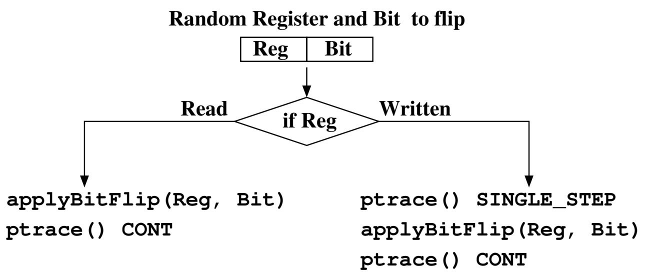


Figure 4. The injection process is different when the register is read from when it is written.

The next step is to randomly select one of the registers from the corresponding pool. Once the register is selected, ZOFI randomly selects a bit where the bit-flip should take place. If we are injecting errors only into registers read by the instruction, we are now ready to apply the bit-flip and continue the execution, as the bit-flip will be processed by the current instruction and will flow through the execution logic. This is shown on the left hand side of Figure 4. If, however, we are injecting errors into the output (written) registers of the instruction, we cannot follow the same process, as a bit-flip performed while the child process is stopped, will be overwritten by the output of the current instruction once it gets executed. Therefore, in this case, ZOFI will first step into the next instruction and will then apply the bit-flip. This is shown on the right hand side of Figure 4. Next, the execution of the tracee process continues.

3.4 Features

In this section we present a list of ZOFI's most important features:

- Execution of test runs at native speed, as there is no emulation or instrumentation involved.
- Support for multiple concurrent test runs (jobs) using the `-j N` flag. In this way the test throughput increases almost linearly to the number of cores.
- Built-in tracking of the execution state, output checking and collection of statistics.
- The user can fully override the comparator function that checks the output. The user-defined comparator runs in a system shell.
- Support for multi-threaded workloads (since v0.9.4).
- Option to avoid injecting faults into the system's library code. This is particularly helpful for accurate measurements of workloads that are protected by fault-tolerance techniques linked against non-protected system libraries.
- Fine control of where the faults will be injected.
- Modular design and implementation in modern C++ and well documented code. Easy to modify and extend.

3.5 Limitations

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.

4 Results

This section provides a quantitative evaluation of ZOFI by presenting a set of results on: (i) its performance overhead compared to a native run (Section 4.2), (ii) a fault-coverage analysis of the NAS NPB-2.3[4] benchmark suite for various types of fault injection (Section 4.3), and (iii) an evaluation of the accuracy of the results as we increase the count of test runs (Section 4.4).

4.1 Experimental Setup

The target platform is a Linux-4.9.0, glibc-2.23 based system with an Intel® Core™ i5-6440HQ quad-core CPU and 8 GB RAM. We evaluated ZOFI using the NAS NPB-2.3[4] benchmarks that we compiled with gcc-4.8.2[14] using `-O3 -mtune=native` flags. We used ZOFI 0.9.1 from https://github.com/vporpo/zofi.

4.2 ZOFI Performance Overhead

As already mentioned in previous sections, ZOFI introduces no performance overhead, because the binary is completely unmodified, and executes on the physical hardware at native speed while being `ptraced`. In order to further back our claim, we measured the execution time of the the NAS benchmarks, class A: (i) by calling them directly from the terminal, and (ii) through the ZOFI tool with the fault injection disabled. The normalized results are shown in Figure 5.

It is fairly clear from the figure that there is practically no performance difference between the two measurements. ZOFI does indeed execute the binaries at native speed.

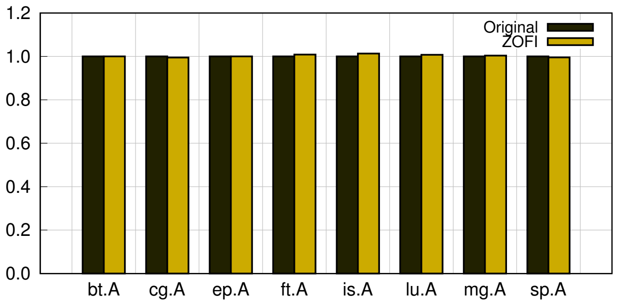


Figure 5. Execution time normalized to the original (native) execution. The ZOFI tool introduces no performance overhead.

We also performed a second set of experiments, where we compared the execution time of the original code versus the ZOFI run with an injection that lead to a masked fault. The time taken by the ZOFI run, including the injection was practically identical to the original run.

4.3 Case Study: NAS Benchmarks

We compiled the full NPB2.3[4] suite with the class W problem size, and then used ZOFI to run it, using the configuration listed in Table 2. Each binary was executed 1000 times (`-test-runs 1000`), and we used all 4 cores of the target processor (`-j 4`). We also used the custom diff shown in Listing 1 which filters out the output, removing those strings that vary on each run (like the throughput Mop/s, the exact execution time and the compilation date). ZOFI gives us the option to choose the register types that the faults will be injected to. For these measurements we tested five of them, as listed in Table 2, ranging from explicitly accessed written registers ("we"), to explicitly or implicitly read or written, and with faults to the program counter enabled ("rweico").

Table 2. ZOFI Configuration.

Flag	Value
-j	4
-test-runs	1000
-diff-cmd	See Listing 1
-inject-to	"we"(Figure 6(a)), "rwe"(Figure 6(b)), "rwei" (Figure 6(c)), "rweic" (Figure 6(d)), and "rweico" (Figure 6(e))

For the first set of results[1], we injected fautls only into the explicitly accessed written registers ("we"), as shown in Figure 6(a). For the second run of the benchmarks, we injected faults into both explicitly read and written registers ("rwe"), and we report the results in Figure 6(b). Similarly, we enabled fault injection into implicit registers too ("rwei"), and report the results in Figure 6(c). Next, we allowed control-flow instructions to cause faults to the instruction pointer (Figure 6(d)), and finally, we allowed all instructions to cause faults to the instruction pointer (Figure 6(e)).

Analyzing the fault coverage results is beyond the scope of this paper. However, we will mention some very obvious points. First of all, enabling faults in the input registers (Figure 6(b)), increases the number of exceptions. This is rather intuitive, as these registers are commonly used memory address pointers. A bit-flip in a register that holds a memory address has a high probability of triggering a `SIGSEGV` signal because of the attempt to access illegal memory. The second point is that errors in implicitly accessed registers (Figure 6(c)) does not seem to have a large effect. Given that we randomly modify a bit in the flag register, the chances of it being the one that is being used in the subsequent instruction seems rather low. The final point is that errors in the program counter (Figure 6(e)) cause a dramatic increase in the number of exceptions. This is expected, as faults in the program counter will likely cause a jump to an illegal address, which will trigger an exception.

[1]When checking the output of the benchmarks, we observed that is.W does not to print the "Verification Successful", like the rest of the benchmarks, even in the original run. Therefore we are not confident about the results for this specific benchmark.

4.4 Accuracy of Results as we Increase the Number of Test Runs

It is a well known fact that the higher number of the test runs in a Monte Carlo approach, the better the accuracy of the results. Most studies in the software-based error-detection literature use a few hundreds (e.g., [22, 28]) to a few thousands (e.g., [48]) of runs for the fault coverage evaluation. The number of runs is usually selected in an arbitrary manner. In this section we are looking into the error that one can expect across independent runs of ZOFI, for a couple of benchmarks of the NPB2.3 suite. We selected cg.W, ft.W and mg.W because these run the fastest among all benchmarks in the suite. To this end, we varied the number of program runs from 50 to 6400, running each configuration 10 times, and calculating the arithmetic mean and standard deviation of the results across these 10 repetitions. This is a total of 382500 runs, which took about a day to run on our i5-6440HQ quad-core system. The configurations that we tested are summarized in Table 3.

Table 3. ZOFI Configuration for results of Section 4.4.

Flag	Value
-j	4
-test-runs	50, 100, 200, 400, 800, 1600, 3200, 6400
-inject-to	"rwe"
-diff-cmd	See Listing 1

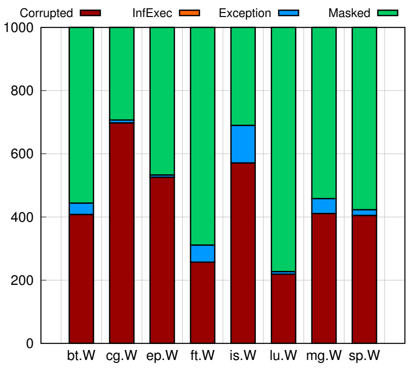
Figures 7(a), 7(b) and 7(c) show the arithmetic mean of the fault coverage results for cg.W, ft.W and mg.W, as we increase the test runs from 50 to 6400. In other words, each run of the ZOFI tool reports the percentages of each outcome. These results will vary across runs, so we collect this data across all 10 runs and we report the arithmetic mean in Figures 7(a), 7(b) and 7(c) and the standard deviation in Figures 8(a), 8(b) and 8(c). Each line in the standard-deviation plots represents the outcome (i.e., Corruption, Infinite Execution, Exception and Masked).

What is particularly interesting, is that the arithmetic mean across 10 runs of ZOFI seems to be quite accurate even for just 50 test runs of the workload. However, the error across these 10 runs of the tool is quite high, meaning that we could not trust the fault coverage reported by a single 50-test run of the tool.

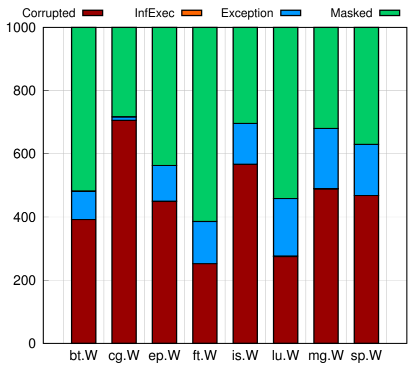
As expected, the standard deviation decreases as we increase the number of runs, as can be observed in Figures 8(a), 8(b) and 8(c). There is, however, some noise in these results, particularly towards smaller values of the number of test-runs, which would probably improve if we ran the tool more than 10 times. We can also observe that the decrease of the error is rather small compared to

Listing 1. -diff-cmd for the experiment of Section 4.3.

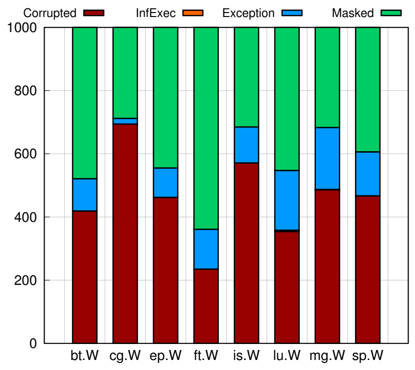
```
-diff-cmd /usr/bin/diff <(/bin/grep -v ''\([Tt]ime\)\|\(Mop/s total\)\|\(Compile date\)'' %
      ORIG_STDOUT) <(/bin/grep -v ''\([Tt]ime\)\|\(Mop/s total\)\|\(Compile date\)'' %TEST_STDOUT)
```



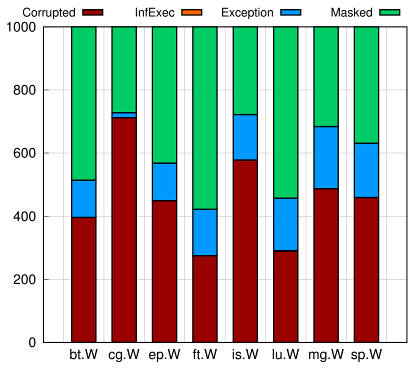
(a) Fault coverage with faults into explicit output registers only ("we").



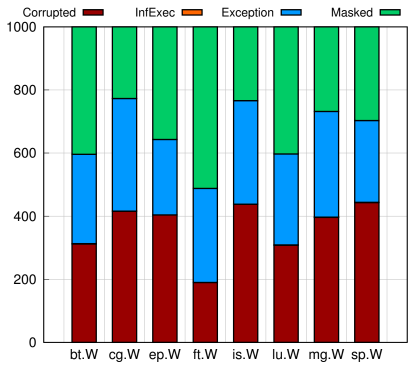
(b) Fault coverage with faults into explicit input and output registers ("rwe").



(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").



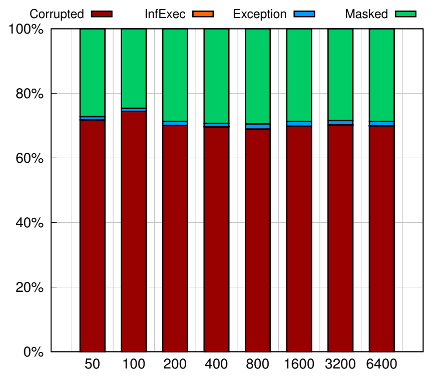
(e) Fault coverage with faults into all input and output registers, including the instruction pointer for all instructions.("rweico").

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

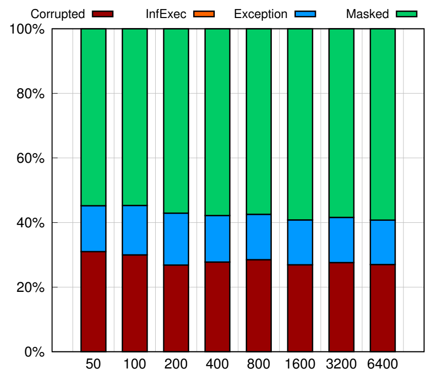
the increase of the number of runs. Please note that the horizontal axis of these plots is in logarithmic scale, and even so the error deviation decreases at a diminishing rate.

5 Conclusion

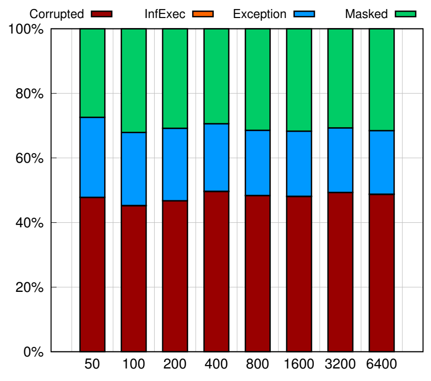
This work presented ZOFI, a timing-based Zero-Overhead Fault Injection tool for very fast fault coverage evaluation at native speed. Unlike the frameworks used in the majority of software-based error detection studies, ZOFI does not instrument the binary nor does it run it on a simulator. Instead, it runs the unmodified workload directly on native hardware, with no performance overhead whatsoever. It is a timing-based tool, meaning that it operates at time-points, not at cycles. It collects fault-coverage statistics across multiple test runs and reports their summary upon completion. ZOFI is distributed as free software and is available at https://github.com/vporpo/zofi.



(a) cg.W

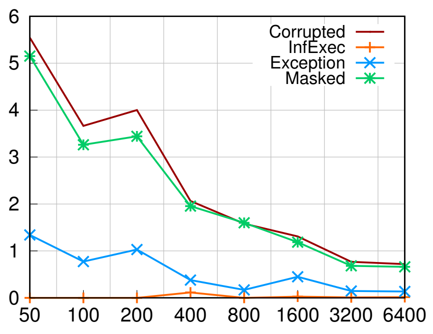


(b) ft.W

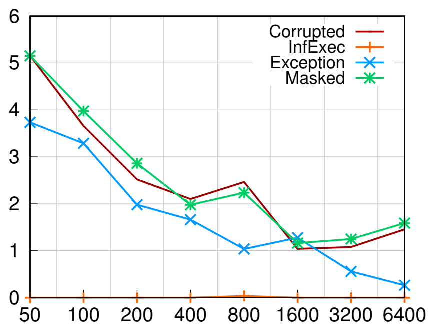


(c) mg.W

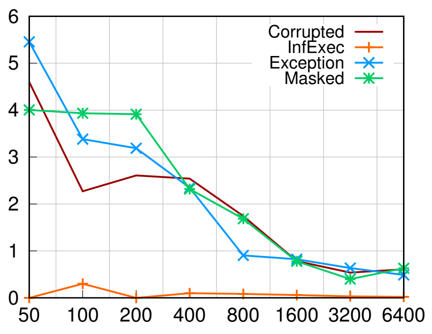
Figure 7. Mean fault coverage (%) across 10 repetitions of ZOFI, as we vary the number of runs from 50 to 6400, injecting into explicitly written registers ("rwe").



(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.

References

[1] [n. d.]. Capstone. https://www.capstone-engine.org.

[2] [n. d.]. Intel VTune Amplifier. https://software.intel.com/en-us/vtune.

[3] [n. d.]. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org.

[4] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Russell L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. 1991. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications* (1991).

[5] Chun-Kai Chang, Sangkug Lym, Nicholas Kelly, Michael B Sullivan, and Mattan Erez. 2018. Hamartia: A fast and accurate error injection framework. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 101–108.

[6] J. Chang, G.A Reis, and D.I August. 2006. Automatic Instruction-Level Software-Only Recovery. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on.* 83–92.

[7] Jonathan Chang, George A Reis, and David I August. 2006. Automatic instruction-level software-only recovery. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on.* IEEE, 83–92.

[8] Chao Chen, Greg Eisenhauer, Matthew Wolf, and Santosh Pande. 2018. LADR: low-cost application-level detector for reducing silent output corruptions. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 156–167.

[9] C. Constantinescu. 2003. Trends and Challenges in VLSI Circuit Reliability. *Micro, IEEE* 23, 4 (July 2003), 14–19.

[10] Rebecca L Davidson and Christopher P Bridges. 2018. Error resilient GPU accelerated image processing for space applications. *IEEE Transactions on Parallel and Distributed Systems* 29, 9 (2018), 1990–2003.

[11] Moslem Didehban and Aviral Shrivastava. 2016. nZDC: A compiler technique for near Zero Silent Data Corruption. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[12] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. 2010. Shoestring: Probabilistic Soft Error Reliability

on the Cheap. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, USA, 385–396.

[13] Shuguang Feng, Shantanu Gupta, Amin Ansari, Scott A Mahlke, and David I August. 2011. Encore: low-cost, fine-grained transient fault recovery. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 398–409.

[14] Free Software Foundation. [n. d.]. GCC: GNU Compiler Collection. http://gcc.gnu.org.

[15] Free Software Foundation. [n. d.]. GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb.

[16] Giorgis Georgakoudis, Ignacio Laguna, Dimitrios S Nikolopoulos, and Martin Schulz. 2017. Refine: Realistic fault injection via compiler-based instrumentation for accuracy, portability and speed. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 29.

[17] G Georgakoudis, I Laguna, H Vandierendonck, DS Nikolopoulos, and M Schulz. 2019. *SAFIRE: Scalable and Accurate Fault Injection ForParallel Multithreaded Applications*. Technical Report.

[18] Qiang Guan, Nathan Debardeleben, Sean Blanchard, and Song Fu. 2014. F-sefi: A fine-grained soft error fault injection tool for profiling application vulnerability. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 1245–1254.

[19] Siva Kumar Sastry Hari, Timothy Tsai, Mark Stephenson, Stephen W Keckler, and Joel Emer. 2017. SASSIFI: An architecture-level fault injection tool for gpu application resilience evaluation. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 249–258.

[20] Manolis Kaliorakis, Sotiris Tselonis, Athanasios Chatzidimitriou, Nikos Foutris, and Dimitris Gizopoulos. 2015. Differential fault injection on microarchitectural simulators. In *2015 IEEE International Symposium on Workload Characterization*. IEEE, 172–182.

[21] Gokcen Kestor, Ivy Bo Peng, Roberto Gioiosa, and Sriram Krishnamoorthy. 2018. Understanding scale-dependent soft-error behavior of scientific applications. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 482–491.

[22] Daya Shanker Khudia and Scott Mahlke. 2013. Low cost control flow protection using abstract control signatures. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 3–12.

[23] Daya Shanker Khudia and Scott Mahlke. 2014. Harnessing soft computations for low-budget fault tolerance. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 319–330.

[24] Guanpeng Li, Karthik Pattabiraman, Siva Kumar Sastry Hari, Michael Sullivan, and Timothy Tsai. 2018. Modeling soft-error propagation in programs. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 27–38.

[25] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, Vol. 40. ACM, 190–200.

[26] S.E. Michalak, K.W. Harris, N.W. Hengartner, B.E. Takala, and S.A Wender. 2005. Predicting the Number of Fatal Soft Errors in Los Alamos National Laboratory's ASC Q Supercomputer. *Device and Materials Reliability, IEEE Transactions on* 5, 3 (Sept 2005), 329–335.

[27] Konstantina Mitropoulou, Vasileios Porpodas, and Marcelo Cintra. 2013. CASTED: Core-adaptive software transient error detection for tightly coupled cores. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE, 513–524.

[28] Konstantina Mitropoulou, Vasileios Porpodas, and Marcelo Cintra. 2013. DRIFT: Decoupled compiler-based instruction-level fault-tolerance. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 217–233.

[29] Konstantina Mitropoulou, Vasileios Porpodas, and Timothy M. Jones. 2016. COMET: Communication-optimised Multi-threaded Error-detection Technique. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES '16)*. ACM, New York, NY, USA, Article 7, 10 pages.

[30] N. Oh, P.P. Shirvani, and E.J. McCluskey. 2002. Control-flow checking by software signatures. *Reliability, IEEE Transactions on* 51, 1 (Mar 2002), 111–122.

[31] N. Oh, P.P. Shirvani, and E.J. McCluskey. 2002. Error Detection by Duplicated Instructions in Super-scalar Processors. *Reliability, IEEE Transactions on* 51, 1 (Mar 2002), 63–75.

[32] Fritz G Previlon, Charu Kalra, Devesh Tiwari, and David R Kaeli. 2019. PCFI: Program Counter Guided Fault Injection for Accelerating GPU Reliability Assessment. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 308–311.

[33] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D.I. August. 2005. SWIFT: software implemented fault tolerance. In *Code Generation and Optimization, 2005. CGO 2005. International Symposium on*. 243–254.

[34] Norman Rink and Jeronimo Castrillon. 2017. Extending a compiler backend for complete memory error detection. *Automotive-Safety & Security 2017-Sicherheit und Zuverlässigkeit für automobile Informationstechnik* (2017).

[35] Norman A Rink and Jeronimo Castrillon. 2017. flexMEDiC: flexible Memory Error Detection by Combined data encoding and duplication. (2017).

[36] Anderson L Sartor, Arthur F Lorenzon, Luigi Carro, Fernanda Kastensmidt, Stephan Wong, and Antonio Beck. 2017. Exploiting idle hardware to provide low overhead fault tolerance for VLIW processors. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 13, 2 (2017), 13.

[37] P. Shivakumar, M. Kistler, S.W. Keckler, D. Burger, and L. Alvisi. 2002. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*. 389–398.

[38] A Shye, T. Moseley, V.J. Reddi, J. Blomstedt, and D.A Connors. 2007. Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance. In *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*. 297–306.

[39] Hwisoo So, Moslem Didehban, Yohan Ko, Aviral Shrivastava, and Kyoungwoo Lee. 2018. Expert: Effective and flexible error protection by redundant multithreading. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 533–538.

[40] Daniel J Sorin. 2009. Fault Tolerant Computer Architecture. *Synthesis Lectures on Computer Architecture* 4, 1 (2009), 1–104.

[41] J. Srinivasan, S.V. Adve, P. Bose, and J.A Rivers. 2004. The Impact of Technology Scaling on Lifetime Reliability. In *Dependable Systems and Networks, 2004 International Conference on*. 177–186.

[42] Venu Babu Thati, Jens Vankeirsbilck, Davy Pissoort, and Jeroen Boydens. 2018. Instruction Level Duplication and Comparison for Data Error Detection: a First Experiment. In *2018 IEEE XXVII International Scientific Conference Electronics-ET*. IEEE, 1–4.

[43] Anna Thomas and Karthik Pattabiraman. 2013. LLFI: An intermediate code level fault injector for soft computing applications. In *Workshop on Silicon Errors in Logic System Effects (SELSE)*.

[44] Ninghan Tian, Daniel Saab, and Jacob A Abraham. 2018. ESIFT: Efficient System for Error Injection. In *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS)*. IEEE, 201–206.

[45] Jens Vankeirsbilck, Niels Penneman, Hans Hallez, and Jeroen Boydens. 2017. Random additive signature monitoring for control flow error detection. *IEEE transactions on Reliability* 66, 4 (2017), 1178–1192.

[46] Cheng Wang, Ho-seop Kim, Youfeng Wu, and Victor Ying. 2007. Compiler-managed software-based redundant multithreading for transient fault detection. In *Proceedings of CGO*.

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

[48] Yun Zhang, Soumyadeep Ghosh, Jialu Huang, Jae W. Lee, Scott A. Mahlke, and David I. August. 2012. Runtime Asynchronous Fault Tolerance via Speculation. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO '12)*. ACM, New York, NY, USA, 145–154.

[49] Yun Zhang, Jae W. Lee, Nick P. Johnson, and David I. August. 2010. DAFT: Decoupled Acyclic Fault Tolerance. In *Proceedings of PACT*.