# Quantifying the Accuracy of High-Level Fault Injection Techniques for Hardware Faults

Jiesheng Wei, Anna Thomas, Guanpeng Li, and Karthik Pattabiraman
Department of Electrical and Computer Engineering, University of British Columbia
{jwei, annat, gpli, karthikp}@ece.ubc.ca

***Abstract*—Hardware errors are on the rise with reducing feature sizes, however tolerating them in hardware is expensive. Researchers have explored software-based techniques for building error resilient applications. Many of these techniques leverage application-specific resilience characteristics to keep overheads low. Understanding application-specific resilience characteristics requires software fault-injection mechanisms that are both accurate and capable of operating at a high-level of abstraction to allow developers to reason about error resilience.**

**In this paper, we quantify the accuracy of high-level software fault injection mechanisms vis-a-vis those that operate at the assembly or machine code levels. To represent high-level injection mechanisms, we built a fault injector tool based on the LLVM compiler, called LLFI. LLFI performs fault injection at the LLVM intermediate code level of the application, which is close to the source code. We quantitatively evaluate the accuracy of LLFI with respect to assembly level fault injection, and understand the reasons for the differences.**

**Keywords: Fault injection, LLVM, PIN, comparison**

## I. Introduction

Hardware faults are increasing due to shrinking feature sizes and manufacturing variations. Simultaneously, diminishing design margins and stringent power constraints are making it harder to provide sufficient redundancy for masking faults from software. Researchers have predicted that computer systems in the future will expose (some) hardware faults to the software layer, and will expect the software to tolerate such faults [1], [2], [3], [4], [5]. Thus, there is a compelling need to build error resilient software that can tolerate hardware faults. To build error resilient software however, one needs to understand the effects of hardware faults on software and provide customized error detection and recovery support. This is because generic error resilience techniques such as full duplication, which duplicates every instruction in the program, incur very high performance and power overheads [6].

To evaluate the error resilience of a program, one would typically inject faults into the program, and observe the program's behaviour under the fault. Hardware faults can be injected either by modifying the hardware, or by emulating them in software. Because hardware modifications are often costly and impractical, researchers have emulated hardware faults through Software-Implemented Fault Injection (SWiFI) techniques [7], [8], [9]. SWiFI techniques typically operate at the assembly or machine code levels of the program as it is easier to emulate hardware faults at that level. However, it is challenging to map the results of the injection back to the program's source code, which is needed for understanding and improving the error resilience of programs.

To alleviate the difficulty associated with mapping fault-injection results from the assembly code to the source code, researchers have proposed high level fault-injection mechanisms that operate at, or close to the source code [10], [5], [11], [2], [12]. These techniques allow faults to be injected directly into program variables or statements. The main advantage of high-level fault-injection mechanisms is that the mapping from the fault injection results to the code is straightforward. Further, these mechanisms allow programmers fine grained control over where to inject faults in the program.

However, an open question with high-level fault injection techniques is how accurate they are in representing hardware errors. This is because hardware errors can occur anywhere in the program, and in any part of its state. Unfortunately, many elements of the program state are not represented at the source level (e.g., code for stack pointer manipulation), and hence high level techniques will not be able to inject faults into these elements. Furthermore, instructions (or data) at the high levels may correspond to multiple instructions (data) at the low levels, and hence a single fault in a high-level instruction(data), may correspond to multiple faults at the low level. Finally, there may be some operations in the high level that have no counterpart at the low levels (e.g., type cast operations), and hence a high level technique may inject spurious faults that do not occur at the low levels.

In this paper, we quantify the accuracy of high-level SWiFI mechanisms vis-a-vis low level mechanisms for hardware faults. We build a SWiFI mechanism at the LLVM compiler's intermediate code level, LLFI [1] to represent high-level injectors. LLVM is a widely used, open source compiler infrastructure, that supports a wide variety of program languages and features [13]. Most prior work on high-level fault injection also uses the LLVM compiler [5], [11], [2], [12] (see Section VIII), and hence we choose LLVM for building LLFI to represent high-level injectors.

To represent low level injectors, we build PINFI [2], a fault injector at the assembly code level using the PIN tool from Intel [14]. PIN is a dynamic binary instrumentation and analysis framework for tracing and modifying the behaviour of x86 binaries. We then compare these two injectors through fault-injection experiments on a set of six benchmark applications. We also attempt to understand the reasons for the differences, to improve the accuracy of high-level injectors.

Prior work has compared the accuracy of assembly code level fault injection to that of high-level fault injection for *software faults* [15], [16]. Unlike our work which focuses on emulating hardware errors, they focus on emulating software faults at the assembly/machine code levels and quantifying the inaccuracy. Other work has emphasized the importance of modelling hardware faults at the assembly code level to capture corner cases in safety-critical applications [17], [18].

[1]LLFI is available at https://github.com/DependableSystemsLab/LLFI
[2]PINFI is available at https://github.com/DependableSystemsLab/PINFI

However, they do not quantify the inaccuracy in modelling hardware faults at the high-level for non-safety critical applications. *To the best of our knowledge, we are the first to quantitatively compare the accuracy of fault-injection at the assembly code level with that of fault injection at the high level, for hardware faults*.

In summary, this paper makes the following contributions:

- Builds a LLVM-based fault injector, LLFI, that is capable of injecting faults at the LLVM compiler's intermediate code level, to represent high-level injectors,
- Builds a PIN-based fault injector, PINFI, that is capable of injecting faults at the x86 assembly code level, to represent low-level injectors,
- Compares the results of injecting faults with both LLFI and PINFI on a set of standard benchmark programs to quantify the differences between them,
- Identifies the sources of discrepancy between the two injectors, and suggests directions for improving the accuracy of high-level fault injectors.

Our results show that LLFI is accurate for emulating hardware errors that cause Silent Data Corruptions (SDCs), but not crashes. When compared to PINFI which does fault injections at the assembly code level, LLFI has nearly the same SDC percentages for the benchmarks programs considered. This result holds for fault injections across all instructions, and also for specific instruction types. For crashes however, the differences between LLFI and PINFI are as much as 40%, showing that fault-injections at the high level does not accurately emulate crash causing errors.

## II. Fault Model and Background

In this section, we first describe our fault model and the general notion of error resilience. We then briefly describe the two systems, LLVM and PIN, that are used in this paper.

### A. Fault Model

We consider transient hardware faults that occur in the processor. These are usually caused by cosmic ray or alpha particle strikes affecting flip flops and logic elements. We consider faults that occur in the processor's computation units, i.e., the ALU and the address computation for loads and stores. However, faults in the memory components such as caches are not considered, since these components are usually protected at the architectural level using ECC or parity. We do not consider faults in the control logic of the processor as this is a small portion of the processor area, nor do we consider faults in the instructions' encoding, as these can be handled through control-flow checking techniques [19]. Related work has made similar assumptions [2], [20], [12], [5], [11].

### B. Error Resilience

We define the resilience of an application as its ability to withstand hardware faults if they occur, without leading to an incorrect output. Incorrect outputs are also known as Silent Data Corruptions (SDCs) and are among the most insidious of failures to recover from, as there is no external indication that the application has malfunctioned (unlike a crash or a hang, where either an exception is raised or a timeout occurs). We are primarily interested in evaluating the resilience of applications using Software Implemented Fault Injection (SWiFI). Therefore, we only inject faults into the program's data or instructions that are visible at the assembly code or higher levels, rather than into the micro-architectural structures where the faults will occur. Further, we consider only activated faults (i.e., faults that are read by the program), as we are not interested in fault masking at the hardware level.

### C. LLVM

LLVM [13] is a compiler infrastructure for lifelong program analysis and optimization. Like most compilers, LLVM consists of a front-end to translate code from a high-level language such as C/C++ to an intermediate representation (IR), and a backend to translate the IR code to machine code for specific platforms such as x86 processors, ARM etc. The IR code is transformed by multiple optimization passes, including user-written ones, before being converted to the machine code by the backend.

The LLVM IR is a typed language, in which source-level constructs can be easily represented. In particular, it preserves the variable and function names, making source mapping feasible. Further, LLVM has extensive support for program analysis and transformations which makes it easier to study the effect of fault injection at a higher level than the assembly language.

### D. PIN

PIN is a dynamic binary instrumentation and analysis framework from Intel used for tracing and modifying the behaviour of x86 binaries. PIN performs instrumentation at runtime on x86 binaries, and hence requires no recompilation of the program [14]. PIN consists of three parts: (1) a virtual machine to perform dynamic binary translation, (2) code cache to keep translated code and use it for speeding up the analysis, and (3) Rich API that third-party tools (such as ours) can tap into to analyze and instrument the translated binary. The API abstracts away the details of the platform and architecture and allows tool developers to focus on the core logic of their tools. Further, PIN takes care of saving and restoring the register state whenever the third-party tool is invoked.

## III. LLVM Fault Injector: LLFI

LLFI is a fault injection tool that works at the LLVM compiler's IR level, and allows fault-injections to be performed at specific program points, and into specific instructions. LLFI supports various fault injection customizations, and enables tracing the propagation of the fault among instructions in the program.

Figure 1 shows the working of LLFI, which consists of three steps. In Step 1, LLFI takes the program IR as input, and applies custom fault injection instruction and operand(s) selector to determine which instructions/operands are fault injection candidates. In Step 2, LLFI instruments the fault injection instructions/operands with calls to fault injection functions. The fault injection functions are designed to perturb the specific instruction operand according to the specified fault type at runtime (e.g. flip one bit of the operand for bit-flip faults). In Step 3, the compiled program is executed at runtime, and LLFI randomly selects one runtime instance of the instrumented instructions to trigger the fault injection function and inject into the selected instruction operand value.

Because hardware faults occur randomly at runtime, LLFI picks a random instruction from the set of all dynamically executed instructions at runtime to inject into. This is possible because the fault injection function is invoked at runtime, and

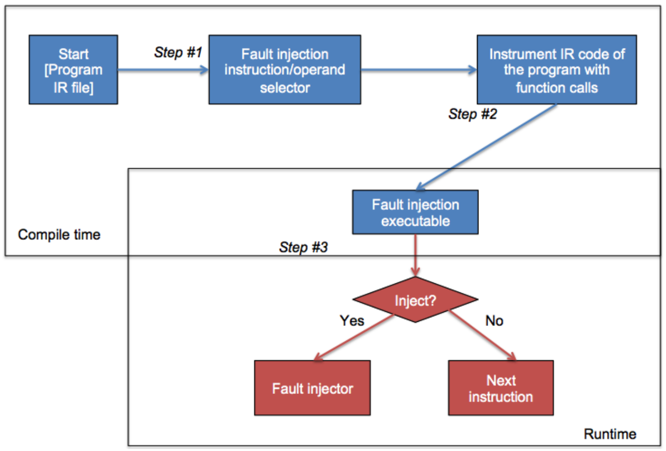

Start [Program IR file]
Step #1
Fault injection instruction/operand selector
Instrument IR code of the program with function calls
Step #2
Compile time
Fault injection executable
Step #3
Inject?
Yes
No
Fault injector
Next instruction
Runtime

*Fig. 1:* LLFI work flow

can hence choose which invocation of an instruction to inject into (this is done by first profiling the program to obtain the total count of executed instructions). Further, by instrumenting the program once with the set of all fault-injection functions, and injecting the fault at runtime, LLFI ensures that the same executable file (with the instrumentation in it) is used in all the fault injection runs. Finally, this method makes it unnecessary recompile the code for each fault injection. Other work on high-level fault injection has followed a similar approach [10], [11], [12].

**Customizability and Analysis**: LLFI has features for easy customization and for analyzing error propagation. We do not consider these features further in this paper as we are primarily interested in the accuracy of LLFI, rather than ease of customization or use. However, these features are the main reason one would prefer to use higher-level fault injection techniques [10].

**Accuracy of LLFI**: One of the main reasons for the inaccuracy of LLFI is that it operates at the LLVM IR code level, which does not correspond one-to-one with assembly code. In this section, we qualitatively assess the correspondence between the LLVM IR code and the assembly code for fault-injection purposes. We quantify the effect of these differences in Section VI. The differences are presented in Table I, along with the mapping between them. We partially mitigate the effects of these differences by injecting only into type-cast instructions that correspond to integer and floating point conversion (row 5 of Table I).

## IV. PIN Fault Injector: PINFI

To evaluate the accuracy of LLFI compared with assembly-level fault injection, we develop PINFI. PINFI is built with Intel Pin [14], an assembly-level instrumentation tool for X86-architecture processors. PINFI is built as a PIN tool and uses the API exposed by Pin to inject faults. The work flow of PINFI is similar to LLFI except: (1) LLFI performs instrumentation at compile time, while PINFI does the instrumentation at runtime (when the program is loaded), and (2) LLFI performs the fault injection (instrumentation) at IR code level, while PINFI does the injection at the assembly code level.

To ensure a fair comparison between LLFI and PINFI, we need to ensure that all injected faults are activated. In the case

*TABLE I:* Difference between LLVM IR code and Assembly Language, and the mapping between them

| **LLVM Instruction** | **Assembly Language Instruction** | **Mapping (if possible)** |
|---|---|---|
| The GetElementPtr (GEP) instruction does address computation which is supplied to the load and store instructions for memory access. | A set of add and multiply instructions that computes the address, and offset memory access in computation. | GEP instructions translate multiple add or multiply instructions, but some GEP instructions cannot be mapped to an assembly instruction if they are translated to offset memory access. |
| The PHINode instruction is inserted when choosing between values merging from different basic blocks. | Some value merging operations at assembly level introduces register spilling, there will be register to stack and stack to memory data movement instructions. | Some PHINode instructions can be translated to memory data movement if they introduce register spilling. |
| Function call | PUSH/POP instructions for Caller/Callee saved registers before and after a function call, and Stack pointer stores return address | None, since these instructions do not exist in the LLVM IR code. |
| Conditional branch instructions | Jump instructions where the target is specified in a register | None, since branch targets are basic block labels in LLVM |
| There are many type cast instructions since LLVM IR is strictly typed. | There are also type cast instructions, but far less than LLVM IR. | type-cast instructions for integer/floating point conversion correspond to assembly code level instructions, but not for other types |

of LLFI, the LLVM compiler will automatically identify the def-use chain of an instruction, and so we can avoid injecting faults into instructions whose value is not used. However, this analysis is much more complicated at the assembly language level where PINFI operates. In particular, we did the following to ensure high fault activation in PINFI.

- In X86 assembly, branch condition instructions set the flag register, and different conditional jump instructions read different bits in the flag register to decide their branch target (Figure 2(a)). To ensure fault activation, we first find the dependent flag register bit(s) of the conditional jumps, and only inject faults into the dependent bit(s) before the conditional jump instructions. For example, in Figure 2(a), the $cmp$ instruction sets the flag register, while the $jl$ instruction only reads the $OF$ $(bit\ 11)$ of flag register to decide the branch target. So we only inject into this bit.
- For floating point operations, X86 instructions usually use XMM registers as the destination register, and hence all 128 bits of XMM register are fault injection candidate bits. However, double-precision floating-point operations only use the lower 64 bits for computation (Figure 2(b)), and hence we prune the target injection space to the lower 64 bits for double-precision floating point operations.

```
cmp $0xa4, %eax
jl 4006e0
```

(a) Flag register

```
addsd %xmm2, xmm0
```

(b) Floating point operation

*Fig. 2:* Examples of PINFI heuristics to increase fault activation

## V. Experimental Setup

We perform fault injection experiments to compare the accuracy of LLFI vis-a-vis PINFI for different failure types, and different instruction categories. In this section, we first introduce the benchmarks we use for the evaluation, and then we describe the experimental procedure.

**Benchmarks:** We choose six programs to evaluate the high-level injector, LLFI against the assembly fault injector, PINFI. Four of the benchmarks are from the SPEC CPU 2006 suite [21], and two are from the SPLASH-2 suite [22]. The benchmark characteristics are presented in Table II. We choose these benchmarks to represent a wide range of commodity and scientific applications. We run each benchmark to completion with a test or default input that comes with the benchmark suite [3].

In both cases, we compile the programs with the LLVM compiler, with the same standard optimizations enabled, to enable a fair comparison. We feed the produced IR code to LLFI, and produce an executable file. We then compile the IR file without passing it to LLFI, and feed the produced executable file to PINFI after linking.

*TABLE II:* Characteristics of Benchmark Programs

| **Benchmark** | **Benchmark Suite** | **Description** | **Lines of Code** | **Input** |
|---|---|---|---|---|
| bzip2 | SPEC | File compression and decompression program | 8293 | test |
| libquantum | SPEC | A library for the simulation of a quantum computer | 4358 | test |
| ocean | SPLASH-2 | Large-scale ocean movements simulation based on eddy and boundary currents | 5329 | default |
| hmmer | SPEC | Uses statistical description of a sequence family's consensus to do sensitive database searching | 35992 | test |
| mcf | SPEC | Solves single-depot vehicle scheduling problems planning transportation | 2685 | test |
| raytrace | SPLASH-2 | Renders a three-dimensional scene using ray tracing | 10861 | default |

**System**: The experiments were carried out on a Intel core i7 based machine, with 8 GB of RAM and 400 GB Hard drive. The machine was running Debian Linux Version 6.0.

**Research Questions:** To compare the accuracies of LLFI and PINFI, we are interested both in injecting faults in the aggregate (i.e., across all instructions), and in specific instruction categories (e.g., arithmetic operations). By injecting faults into specific instruction categories, we can obtain insights into which classes of instructions contribute most to the inaccuracy (if any), and how to mitigate the inaccuracies. Therefore, it is important to calibrate the accuracy of LLFI both in the aggregate and for specific instruction categories.

We attempt to answer the following research questions in comparing LLFI and PINFI.

- **RQ1** How many instructions of each category do LLFI and PINFI consider as injection targets at the LLVM IR code and assembly code levels respectively ?
- **RQ2** How accurate is LLFI compared to PINFI for measuring the SDC rate of applications, both in the aggregate and for specific instruction categories ?
- **RQ3** How accurate is LLFI compared to PINFI in measuring the crash rate of applications, both in the aggregate and for specific instruction categories ?

[3]We cannot use the SPEC ref inputs as we need to run each benchmark to completion thousands of times, and the ref inputs take a long time to complete.

We do not consider hangs as the percentage of hangs observed in our experiments was negligible.

**Experimental procedure:** First, we run LLFI or PINFI on the program, and select specific instructions as fault injection targets. However, the instructions in the LLVM IR (which is used by LLFI) do not correspond one on one to instructions in the assembly code used by PIN. To enable a fair comparison between LLFI and PINFI, we divide both the IR instructions and assembly language instructions into five broad categories based on the types of operation they perform. The five categories are described in Table III. We do not consider store instruction here because we compare LLFI and PINFI through fault injection into destination registers of instructions, and store instructions do not have destination registers.

Second, for each category in Table III, LLFI (or PINFI) profiles the number of dynamic instances of the selected instruction category, say $N$. For LLFI, $N$ represents the number of LLVM IR instructions executed under the chosen category, while for PINFI, it represents the number of assembly instructions executed under the category.

Third, for each program, we perform 1000 fault injections, into the instruction category chosen for the experiment using both LLFI and PINFI. As we have a total of five categories and two tools, this represents a total of $10,000$ fault injections per benchmark program.

For each fault injection run, LLFI and PINFI randomly choose one of the $N$ instructions belonging to a specific category, and each inject a single *bit-flip* into the target register or memory location of the chosen instruction. We choose target registers or memory locations as our injection targets as our fault model considers transient errors in the processor's computational elements (Section II-A). In other words, we assume that any error in the computation/data paths of the processor shows up in the result of the executed instruction, and hence we corrupt its target. This is similar to what prior work has done [12], [23], [2], [11], [5].

*TABLE III:* Fault injection instruction categories

| **Instruction category** | **Description** | **LLFI selection criteria** | **PINFI selection criteria** |
|---|---|---|---|
| arithmetic | arithmetic and logic operations | instructions that perform arithmetic or logical operations | instructions that perform arithmetic or logical operations |
| cast | type cast operations | instructions with 'cast' opcode | instructions with 'convert' category |
| cmp | branch condition instructions | 'cmp' instructions | instructions whose next instruction is conditional branch |
| load | memory load operations | 'load' instructions | 'mov' instructions with memory as the source and register as the destination |
| all | all instructions | 'all' in the configuration | 'all' in the configuration |

**Failure categorization:** As mentioned earlier, we consider only activated faults in the results. For a fault to be activated, the injected location or register must be read by another instruction in the program. This is because we are interested in the behaviour of the program given that a fault has occurred in it, as our goal is to study error resilience (Section II-B).

We classify the outcome of activated faults based on the program's behaviour. If the program is terminated by the OS due to an exception, it is classified as a crash. We also obtain the golden run of the program when no fault is injected to compare the output with the program's output after injecting the fault. Any deviation is classified as an SDC. Hangs are detected through a timeout mechanism if the program takes substantially longer than the golden run.

## VI. Results

In this section, we present the fault injection results of LLFI and PINFI across six benchmarks. We first present the aggregate fault injection results in Section VI-A. Then Section VI-B to Section VI-D presents results to answer the three research question in Section V.

### A. *Aggregate Fault Injection Results*

Figure 3 shows the breakdown of the fault injection outcome (i.e. crash, SDC and benign) for each benchmark, with both LLFI and PINFI injecting faults to 'all' instructions. $x$ axis represents the benchmark and average value, and $y$ axis represents the percentages observed in each fault injection outcome category.

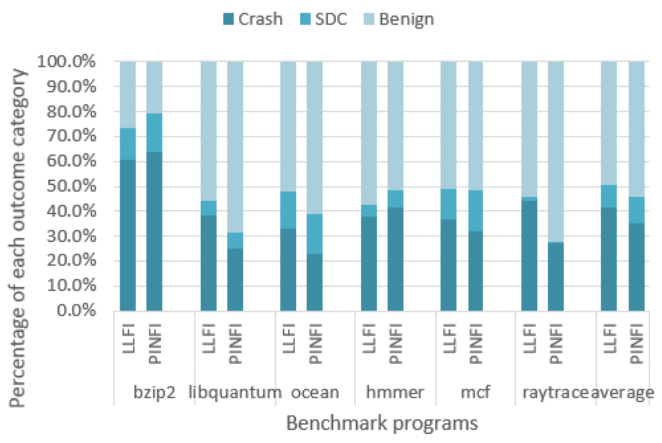


*Fig. 3:* Aggregated fault injection results with LLFI and PINFI

From Figure 3, we find that on average for both tools, the crash percentage is around 30%, and the SDC percentage is around 10%, and the remaining are benign faults (i.e. hang results are negligible). This is consistent with previous fault injection studies [11], [12]. We also find that the difference in the fault injection results between LLFI and PINFI for SDCs is very small. This will be analyzed in detail in the following sections.

### B. *RQ 1: Number of dynamic instructions*

Table IV shows the numbers of executed instructions in each benchmark, for each category in Table III, that are encountered by LLFI and PINFI. From Table IV, we find that:

- LLFI encounters more runtime instructions for the 'all' category than PINFI. This is because assembly code is often more packed than LLVM IR code. For example, a memory load from an array usually consists of two instructions in LLVM IR ($getelementptr$ instruction for getting the address, and $load$ instruction for load operation), while it consists of a single instruction at the x86 assembly level ($mov$ instruction with offset memory access). Thus, LLFI has more instructions to inject than PINFI.
- For arithmetic operation ('arithmetic'), LLFI has fewer instructions to inject than PINFI for most programs. The reason is that arithmetic operations are used for data and address computation at the assembly level, while the $getelementptr$ instruction is used for address computation at the LLVM IR level. This instruction is not considered as an arithmetic operation in LLVM's IR code, and hence LLFI does not inject into it when considering arithmetic operations.
- The number of type cast instructions ('cast') is negligible for both LLFI and PINFI. LLFI and PINFI have similar number of compare instructions ('cmp') for all benchmarks.

### C. *RQ2: SDC results of* LLFI *and PINFI*

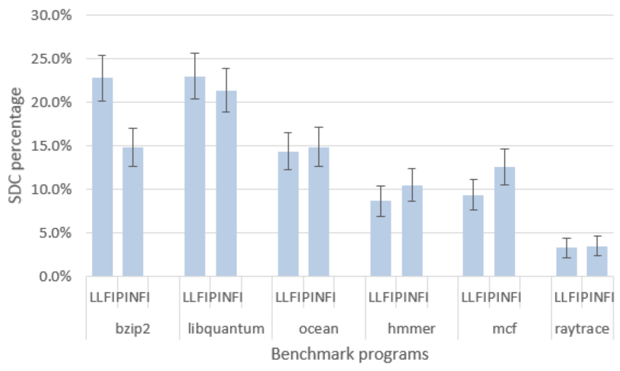
Figure 4 shows the percentage of SDCs (among the activated faults) incurred by the six programs after fault injection using LLFI and PINFI. Sub-figures 4(a) to 4(d) correspond to the results of injecting into each of the five instruction categories in Table III. In each graph, the $x$ axis represents the benchmark, and $y$ axis represents the percentage of SDCs incurred among all activated faults. The error bars represent the 95% confidence interval of SDCs for 1000 injections for either PINFI or LLFI.

From Figure 4 it can be inferred that the difference between LLFI and PINFI is within the measurement error threshold for most programs, regardless of whether we consider all instructions together or only instructions from a particular category. *This means that for injecting errors that cause SDCs,* LLFI *is at least as accurate as assembly level fault injection for most of the programs considered, across all instructions and instruction categories.* We examine individual deviations from this norm below:

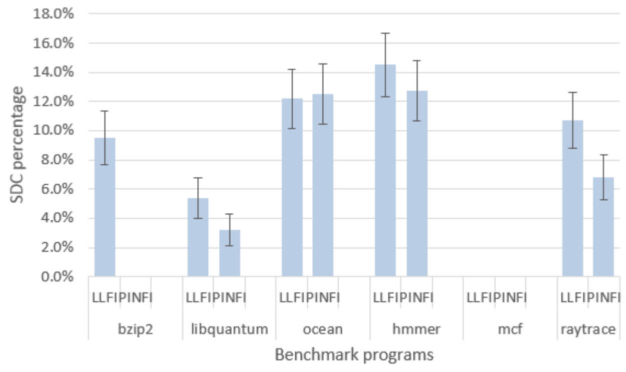
- For arithmetic operation instructions ('arithmetic'), LLFI has slightly higher SDCs for *bzip2*. This is because bzip2 has a lot of memory address computation, and as described in Section V, address computation operations are not classified as arithmetic operations in LLVM IR. However, at the assembly code level, address computation is performed using arithmetic operations, which are classified as arithmetic operations by PINFI, and a fault in these operations is likely to crash the program. Therefore, PINFI experiences a higher percentage or crashes than LLFI (Table V), which lowers its SDC rate.
- For type cast instructions ('cast'), LLFI exhibits a higher percentage of SDCs for *bzip2*. This is because bzip2 has only six type cast instructions, and all six instructions operate on pointer values. Therefore, any fault in these instructions has a high probability of crashing the program (the crash rate for this program is 96% as is shown in Table V), which in turn lowers its probability of resulting in an SDC.
- For compare instructions ('cmp'), both LLFI and PINFI exhibit nearly the same SDC rate. This is because both LLFI and PINFI have similar number of compare instructions.
- For load instructions. LLFI exhibits a much higher SDC rate than PINFI for the *libquantum* program. This is because libquantum consists of many data movement

*TABLE IV:* Runtime instructions of the benchmark programs for LLFI and PINFI

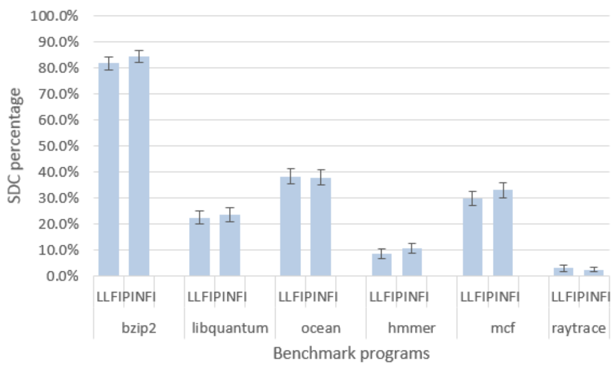
| Programs | All | | Arithmetic | | Cast | | Cmp | | Load | |
|---|---|---|---|---|---|---|---|---|---|---|
| | LLFI | PINFI | LLFI | PINFI | LLFI | PINFI | LLFI | PINFI | LLFI | PINFI |
| bzip2 | 487081311 | 345535913 | 18530760 (4%) | 50433646 (14%) | 30606431 (6%) | 6 (0%) | 38540680 (8%) | 38227320 (11%) | 335748373 (69%) | 243088790 (70%) |
| mcf | 7162446297 | 3800867922 | 482659382 (7%) | 532203970 (14%) | 6 (0%) | 6 (0%) | 836141657 (12%) | 827164028 (22%) | 3833040057 (54%) | 2155207386 (57%) |
| hmmer | 4077115017 | 2292170072 | 482968327 (12%) | 369334397 (16%) | 10506166 (0%) | 17426657 (1%) | 268007691 (7%) | 268007694 (12%) | 2489538548 (61%) | 1495918948 (65%) |
| libquantum | 716159246 | 445866958 | 37728075 (5%) | 38531240 (9%) | 110944 (0%) | 110616 (0%) | 56928497 (8%) | 57166980 (13%) | 357370593 (50%) | 242788525 (54%) |
| ocean | 1056629348 | 566050809 | 215580829 (20%) | 187358712 (33%) | 1236605 (0%) | 1238928 (0%) | 31542955 (3%) | 31542560 (6%) | 638292229 (60%) | 328446760 (58%) |
| raytrace | 13370543488 | 6229897840 | 1660765146 (12%) | 1706697298 (27%) | 2327664 (0%) | 2870179 (0%) | 539958621 (4%) | 539804535 (9%) | 5686126390 (43%) | 3409330274 (55%) |



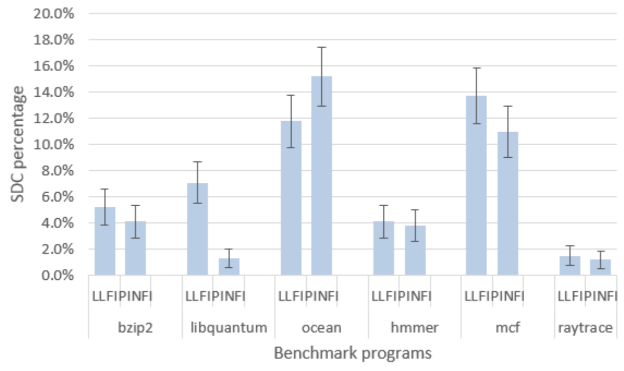
(a) Arithmetic operation instructions



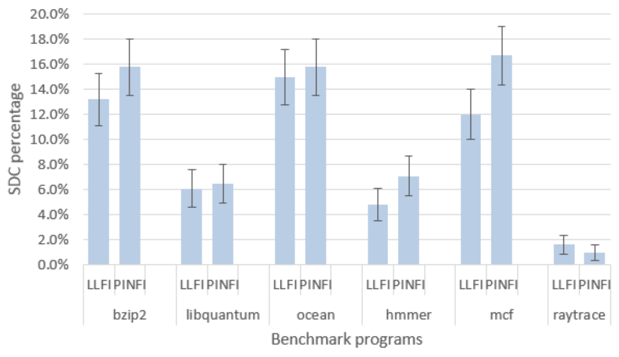
(b) Cast instructions



(c) Compare instructions



(d) Load instructions



(e) All instructions

*Fig. 4:* SDC Results for LLFI and PINFI. Subfigures (a) to (d) represent the SDC results for different instruction categories; and subfigure (e) represented all instructions.

operations, which are translated to *load* instructions in LLVM IR, and are considered as injection targets by LLFI. However, assembly code moves the data directly from one location to another using a mov instruction, and does not have a corresponding *load* instruction. Therefore, LLFI injects into more data operations and hence has a higher SDC rate compared to PINFI.

### D. *RQ3: Crash results of* LLFI *and PINFI*

Table V shows the percentage of crashes incurred by the benchmark programs for LLFI and PINFI. From Table V, we find that LLFI and PINFI have similar crash percentages for compare instructions. However, for the other categories of instructions, there are considerable differences in the percentage of crashes. The maximum differences between the two tools are as follows: 17% in the 'all' category (ocean), 40% in the 'arithmetic' category (bzip2), 32% in the 'cast' category (hmmer), and 21% in the 'load' category (hmmer). In Section VII, we explain the reasons for the difference in crash rates.

## VII. Discussion

The results of the fault injection experiments in Section VI illustrate that the SDC rates obtained with LLFI closely match those obtained with PINFI for the benchmarks. This shows that LLFI is a good choice if one's interest is in SDC causing errors, as ours is (recall that our goal is to study error resilience, which is the ability of a program to prevent an error from becoming an SDC). However, as far as crashes are concerned, there are substantial differences between the two injectors. We examine the reasons for the differences, and how to resolve them (future work).

1) *GetElementPtr instructions*: As mentioned earlier, LLVM IR uses the *getelementptr* instruction to perform pointer address computations. However, at the assembly code level, pointer computations are performed with regular arithmetic add/subtract/multiply instructions. On the face of it, it seems like this problem can be solved by treating all *getelementptr* instructions as equivalent to arithmetic operations at the LLVM IR level. However, not all *getelementptr* instructions are translated to arithmetic operations - some address computations are compressed in the memory offset computation part of the assembly language instruction. To remove this discrepancy, we will need a heuristic to decide when to treat a *getelementptr* instruction as an arithmetic instruction and inject faults into only such instructions.
2) *Cast instructions*: These contribute to inaccuracies when they deal with pointer conversion as in the *bzip2* benchmark. To remove this discrepancy, we will need to identify such cases, and not inject faults into them at the LLVM IR level.
3) *Mov instructions*: In assembly code, *mov* instructions are used to move data both between registers and between registers and memory. In LLVM IR however, there are separate instructions for these two operations, and hence there are many more instructions corresponding to *mov* instructions in the assembly code. To remove this discrepancy, we need to inject into only those instructions that have a corresponding analogue at the assembly code level.

## VIII. Related Work

We classify related work on fault injection into three broad categories: (1) Program-level fault injection for hardware faults, (2) Assembly code level fault injection for hardware faults, and (3) Fault injection for software faults.

**Program-level fault injection for hardware faults**: There have been many attempts to build a fault injector for hardware faults at the program level. Propane [10] is perhaps the first tool that injects faults at the program level and traces their propagation in the program. Propane allows injection of both hardware faults (data errors) and software faults. However, to the best of our knowledge, the accuracy of Propane has not been measured with regard to hardware fault injection.

Pattabiraman et al. [24] present an approach to selectively protect critical data in a program by duplicating its backward slice. Relax [2] is a code transformation technique to tolerate soft errors in programs through structured blocks and exception handling. Cong et al [5] use static analysis to identify instructions that must be duplicated for protecting soft-computing applications, or applications with relaxed correctness properties. Similar to LLFI, the authors of the above papers develop fault injectors based on the LLVM compiler to validate their technique. However, none of them validate the fault injector itself with regard to its accuracy in injecting hardware faults.

Thomas et al. [12] also build a static analysis technique for identifying critical data in soft-computing applications to protect against significant deviations in the correct output, or what they call Egregious Data Corruption (EDC). They also perform fault injection at the LLVM compiler level. Unlike the above papers, however, they provide a limited validation of their injector with regard to EDC causing errors. However, EDCs are only a (small) subset of Silent Data Corruptions (SDCs), and their evaluation is confined to soft-computing applications. In contrast, we evaluate the accuracy of LLFI for general-purpose applications, and for the full set of SDC and crash causing errors.

Finally, in recent work, Sharma et al. present KULFI [11], which stands for "Configurable Injector". Like LLFI, KULFI is built using the LLVM compiler infrastructure, and operates on the IR code. To the best of our knowledge, KULFI has not been validated with regard to assembly code level fault injection. Further, the authors of KULFI use it to compare the error resilience of algorithms for both SDC and crash causing errors. However, as we have seen in this paper, performing fault injections at the LLVM level may not be accurate for crash causing errors, though we have not directly validated KULFI's accuracy for such faults.

Note that five of the six papers above use the LLVM compiler and its infrastructure for performing their experiments. This is also why we use LLVM for building LLFI.

**Assembly code level fault injection**: There has been substantial amount of work in fault-injection at the assembly language level for emulating hardware faults. Examples of this approach are NFTAPE [7], GOOFI-2 [8] and Xception [9]. NFTAPE uses break-point based injection at the machine code level. GOOFI-2 supports three methods of fault injection, namely instrumentation-based, exception-based and Nexus-based. All three methods operate at the assembly code (or lower levels). Xception uses debug registers and features found in many modern processors to inject faults at runtime. While

*TABLE V:* Crash percentage of the benchmark programs for LLFI and PINFI

| Programs | All | | arithmetic | | Cast | | Cmp | | Load | |
|---|---|---|---|---|---|---|---|---|---|---|
| | LLFI | PINFI | LLFI | PINFI | LLFI | PINFI | LLFI | PINFI | LLFI | PINFI |
| bzip2 | 60% | 64% | 23% | 63% | 66% | 96% | 3% | 2% | 64% | 74% |
| mcf | 37% | 32% | 22% | 19% | 0% | 0% | 3% | 2% | 33% | 47% |
| hmmer | 38% | 41% | 20% | 13% | 12% | 44% | 2% | 2% | 36% | 57% |
| libquantum | 38% | 25% | 2% | 4% | 0% | 1% | 1% | 0% | 36% | 50% |
| ocean | 33% | 23% | 11% | 2% | 0% | 0% | 0% | 0% | 37% | 43% |
| raytrace | 44% | 27% | 1% | 1% | 22% | 39% | 3% | 4% | 37% | 44% |

Xception allows a high degree of configurability for the fault injector, it also operates at the assembly language level.

A recent paper by Cho et al [25] evaluates the accuracy of assembly code level fault injection versus injections at the Register Transfer Language (RTL) level. They find that single bit flips at the RTL level may manifest as multiple bit flips at the assembly code level. Unlike our work which attempts to calibrate the accuracy of higher levels of fault injection with respect to assembly language level injection, they are interested in benchmarking the accuracy of the assembly level injectors. Thus, their study is complementary to ours.

**Fault injection for software faults**: Techniques for injecting software faults in programs typically operate at the source-code level, or at levels close to the source code (e.g., on the abstract syntax tree). G-SWiFT is a technique that attempts to emulate software faults at the machine code level [15], by identifying patterns of assembly code instructions corresponding to high-level software constructs and injecting faults in them to emulate software bugs. Because software bugs occur primarily at the source code level, it is important to calibrate the accuracy of assembly-code level injection techniques with respect to the source code level. Cotroneo [16] perform one such characterization and find that injecting software faults at the machine code level may not be representative of residual software faults. Unlike software faults, hardware faults occur within the microprocessor or memory and affect the program's execution. Because the executable file is in assembly/machine language, hardware faults are easier to emulate at that level. Thus, when injecting hardware faults at high level, it is important to calibrate their accuracy with assembly code level injections. This is the inverse of the problem that the above software-fault injection papers face.

## IX. Conclusion

In this paper, we quantitatively compare the accuracy of high-level fault injection techniques with assembly code level fault injection techniques for hardware faults. We develop two fault injectors, LLFI to represent a high-level fault injector, and PINFI, to represent a low-level fault injector. We compare the accuracy of LLFI with PINFI with regard to crashes and SDCs through fault-injection experiments on six benchmark applications. Our results show that LLFI is highly accurate for injecting SDC-causing errors, but not for crash causing errors, compared to PINFI. Therefore, higher-level fault injection techniques are suitable for studying SDC-causing errors, but not for studying crash-causing errors in programs.

## Acknowledgements

This work was supported in part by a Discovery grant from the Natural Science and Engineering Research Council (NSERC), Canada, and a research gift from Lockheed Martin Company. We thank the anonymous reviewers of DSN for their comments that have helped to improve this paper.

## References

[1] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: saving dram refresh-power through critical data partitioning," in *ASPLOS*, 2011, pp. 213–224.

[2] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: an architectural framework for software recovery of hardware faults," in *International Symposium on Computer Architecture*, 2010, pp. 497–508.

[3] S. Narayanan, J. Sartori, R. Kumar, and D. L. Jones, "Scalable stochastic processors," in *DATE*, 2010, pp. 335–338.

[4] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate data types for safe and general low-power computation," in *PLDI*, 2011, pp. 164–174.

[5] J. Cong and K. Gururaj, "Assuring application-level correctness against soft errors," in *IEEE International Conference on Computer-Aided Design*, 2011, pp. 150–157.

[6] N. Nakka, K. Pattabiraman, and R. K. Iyer, "Processor-level selective replication," in *DSN*, 2007, pp. 544–553.

[7] D. T. Stott, B. Floering, D. Burke, Z. Kalbarczpk, and R. K. Iyer, "NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors," in *International Computer Performance and Dependability Symposium*, 2000, pp. 91–100.

[8] D. Skarin, R. Barbosa, and J. Karlsson, "GOOFI-2: A tool for experimental dependability assessment," in *DSN*, 2010, pp. 557–562.

[9] R. Maia, L. Henriques, D. Costa, and H. Madeira, "XceptionTM - enhanced automated fault-injection environment," in *DSN*, 2002, pp. 547–550.

[10] M. Hiller, A. Jhumka, and N. Suri, "PROPANE: an environment for examining the propagation of errors in software," in *International Symposium on Software Testing and Analysis*, 2002, pp. 81–85.

[11] V. C. Sharma, A. Haran, Z. Rakamaric, and G. Gopalakrishnan, "Towards Formal Approaches to System Resilience," in *PRDC*, 2013.

[12] A. Thomas and K. Pattabiraman, "Error detector placement for soft computation," in *DSN*, 2013, pp. 1–12.

[13] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004, pp. 75–86.

[14] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005, pp. 190–200.

[15] H. Madeira, D. Costa, and M. Vieira, "On the emulation of software faults by software fault injection," in *DSN*, 2000, pp. 417–426.

[16] R. Natella, D. Cotroneo, J. A. Duraes, and H. S. Madeira, "On fault representativeness of software fault injection," *IEEE Transactions on Software Engineering*, vol. 39, pp. 80–96, 2013.

[17] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. Iyer, "SymPLFIED: Symbolic program-level fault injection and error detection framework," in *DSN*, 2008, pp. 472–481.

[18] D. Skarin and J. Karlsson, "Software implemented detection and recovery of soft errors in a brake-by-wire system," in *EDCC*, 2008, pp. 145–154.

[19] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability*, vol. 51, pp. 63–75, 2002.

[20] S. K. S. Hari, S. V. Adve, and H. Naeimi, "Low-cost program-level detectors for reducing silent data corruptions," in *DSN*, pp. 181–188.

[21] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM Sigarch Computer Architecture News*, vol. 34, pp. 1–17, 2006.

[22] S. C. Woof, M. Ohara, E. Torriet, J. P. Singhi, and A. Guptat, "The SPLASH2 programs: characterization and methodological considerations," in *ISCA*, 1995, pp. 24–36.

[23] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: exploiting application-level fault equivalence to analyze application resiliency to transient faults," in *ASPLOS*, 2012, pp. 123–134.

[24] K. Pattabiraman, Z. Kalbarczyk, and R. K. Iyer, "Automated derivation of application-aware error detectors using static analysis: The trusted illiac approach," in *TDSC*, vol. 8, 2011, pp. 44–57.

[25] H. Cho, S. Mirkhani, C.-Y. Cher, J. Abraham, and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," in *Design Automation Conference (DAC)*, 2013, pp. 1–10.