# A Tale of Two Injectors: End-to-End Comparison of IR-level and Assembly-Level Fault Injection

Lucas Palazzi, Guanpeng Li, Bo Fang, and Karthik Pattabiraman
Department of Electrical & Computer Engineering, The University of British Columbia, Vancouver, Canada
{lpalazzi, gpli, bof, karthikp}@ece.ubc.ca

***Abstract*—Fault injection (FI) is a commonly used experimental technique to evaluate the resilience of software techniques for tolerating hardware faults. Software-implemented FI can be performed at different levels of abstraction in the system stack; FI performed at the compiler's intermediate representation (IR) level has the advantage that it is closer to the program being evaluated and is hence easier to derive insights from for the design of software fault-tolerance mechanisms. Unfortunately, it is not clear how accurate IR-level FI is vis-a-vis FI performed at the assembly code level, and prior work has presented contradictory findings. In this paper, we perform an analysis of said prior work, find an inconsistency in the FI methodology used in one study, and show that it results in a flawed comparison between IR-level and assembly-level FI. We further confirm this finding by performing a comprehensive evaluation of the accuracy of IR-level FI across a range of benchmark programs and compiler optimization levels. Our results show that IR-level FI is as accurate as assembly-level FI for silent data corruptions (SDCs) across different benchmarks and optimization levels.**

***Index Terms*—Resilience, fault injection, LLVM, PIN, comparison**

## I. Introduction

Hardware faults are becoming more common in commodity computer systems due to the effects of process scaling and manufacturing variations [1]–[3]. This has led to a concomitant increase in the rates of hardware faults that are exposed to the software running on these systems. This is because techniques to mask all hardware faults from software, such as full duplication in hardware, consume too much energy, making their use challenging in commodity systems. Therefore, researchers have proposed various software techniques to detect and recover from hardware faults exposed to the software, with low performance and energy overheads [4]–[6].

An important consideration for deploying any software technique is a quantitative evaluation of its coverage, i.e., the technique's ability to detect (or recover from) hardware faults. When proposing such a technique, researchers typically use fault injection tools to evaluate its coverage. Fault injection (FI) is the process of systematically introducing errors[1] into the program and observing the outcome. Because the injection space is very large, typical FI tools use Monte Carlo simulation to sample the space of potential faults, and obtain a statistical estimate of the techniques' coverage.

A key design consideration in a FI tool is the level of abstraction at which it operates. The higher the level of abstraction, the easier it is to draw meaningful insights from the tool, as the findings can be directly translated to the design of software mechanisms. However, raising the level of abstraction may come with a cost in the accuracy of the FI process, as hardware faults occur in the lower levels of the system stack, and modeling them at the higher levels can be challenging. To alleviate this difficulty, researchers have proposed implementing FI tools at the intermediate representation (IR) of modern compilers such as LLVM/Clang [7], [8]. The main advantages of this approach are: (1) many software protection techniques are implemented at the IR level, and it is straightforward to use the results of the evaluation to improve the coverage of these techniques, and (2) IR-level injections typically abstract the effects of the machine architecture such as instruction encodings and register file sizes, thereby making the results applicable to a wide variety of hardware platforms. Further, the IR of LLVM includes IR-level program type information, which is useful in guiding the software techniques towards more vulnerable parts of the program. Consequently, a wide range of software fault-tolerance techniques use IR-level injections to validate their results [9]–[11].

However, there has been little work on validating the results of IR-level FI with respect to FI performed at the assembly code level, which is arguably more accurate as it is closer to the hardware. This is concerning, as many of the insights used in software fault-tolerance techniques are derived from IR-level fault injections, and inaccuracies in the latter call into question the efficacy of these techniques. Further, the dominant platform for IR-level studies, LLVM, has significant differences with x86 assembly language on which many of these studies are based, so it is not clear how well the results of FI performed at the IR-level match those of FI performed at the assembly language level. The only two previous studies that have examined this question [12], [13] (to the best of our knowledge), come to conflicting conclusions, while claiming to use the same FI tools and similar experimental setups.

In this paper, we provide an analysis of prior work [12], [13] and present the reasons for their conflicting findings. Further, we provide an extensive comparison study to re-examine the accuracy of the statistical estimates of coverage derived from FI studies at the IR level with respect to FI performed at the assembly level. Specifically, we compare the results of FI performed at the LLVM IR level with those at the x86 assembly level, as these are the dominant platforms used by prior work in this area. We conduct FI experiments on a

[1]We refer primarily to Software Implemented Fault Injection (SWiFI) techniques when we say FI in this paper.

set of 25 benchmark programs, including those used in the aforementioned prior studies [12], [13], as well as several other benchmarks not used in those studies. Finally, we consider a wide range of compiler optimization levels in our study, unlike prior studies which considered only one level.

The main contribution of our work is to systematically and painstakingly gather benchmarks and experimental configurations used in prior work, and to perform a detailed comparison between them in order to understand the reason for their conflicting findings. This is challenging as many details are sparse in the prior work, and it is unclear whether the conflicting results are due to differences in their experimental setups, benchmarks, or compiler optimization decisions. *To the best of our knowledge, we are the first to reconcile contradictory studies about IR-level FI and assembly-level FI, design experiments to validate the accuracy of FI tools that are in question, under different experimental configurations.*

Our main findings are as follows:

- We find that an inconsistent fault model (specifically the bit-sampling model) used in Georgakoudis et al. [13] is the reason for the contradictory results found in Wei et al. [12] (Section IV).
- IR-level FI is as accurate as assembly-level FI for emulating hardware errors that cause *SDCs*, as well as in measuring the relative ranking of program SDC probabilities, at all optimization levels (Section VI-A).
- For *crash*-causing errors, IR-level FI is only comparable to assembly-level FI at the lowest optimization level, *-O0*, but not at higher optimization levels, *-O1* to *-O3*, suggesting that IR-level FI becomes less accurate with respect to crashes when (more) compiler optimizations are applied (Section VI-B).

Our findings thus confirm the results of prior work [12], but go well beyond it in exploring the limits and implications of the results, across a larger suite of benchmarks and configurations. Further, our study highlights common pitfalls in experimental comparisons of FI tools, beyond the specifics of the tools studied. Finally, our results enable software developers to choose when to use IR-level FI tools for evaluation of error resilience.

## II. Background

In this section, we provide some relevant definitions and describe the general notions of code compilation and fault injection as they pertain to this study.

### A. Definitions

- **Intermediate representation (IR):** A code representation of a program typically used internally by a compiler (e.g., LLVM) between the source code and target language (e.g., assembly), independent of both the source language and target architecture.
- **Compiler optimization:** A code transformation applied by the compiler with the goal of improving the program in some way (e.g., decrease runtime, reduce memory accesses, etc.). Many mainstream compilers will often package multiple individual optimizations together in one pass for convenience (e.g., the *-O#* flags used in LLVM and GCC).
- **Fault:** A defect in the computer system that may or may not end up being read by the program.
- **Error:** A fault that has been activated (i.e., read by the program) and has resulted in some deviation of system behaviour from a fault-free run. This may or may not be observable as the error may only affect inconsequential system states, or be corrected by fault-tolerance mechanisms.
- **Benign error:** An error that does not cause an observable deviation from the expected system behaviour (i.e., the error was either masked or handled by the program).
- **Failure:** An error has resulted in an observable deviation from expected system behaviour (e.g., crash, SDC).
- **Silent data corruption (SDC):** A failure that causes the program to produce an incorrect output, but with no indication that the failure has occurred.
- **Crash:** A failure that causes an exception, and as a result the program terminates before completing its execution.
- **Program SDC/crash probability:** The probability of an error causing an SDC/crash for a given program and input (other work uses a similar definition [12], [14]–[19]).
- **Error Resilience**: The error resilience of an application is its ability to withstand hardware faults if they occur, without leading to an SDC or crash.

### B. Code Compilation

In the context of this paper, we consider the compilation of a program in the structure shown in Figure 1; this is the structure that is pertinent to the LLVM/Clang compiler [20]. The front end processes the program's source code (e.g., C/C++ code) and generates an intermediate representation (IR) of the program, while the middle and back ends perform platform-independent and platform-specific optimizations on the code, respectively.

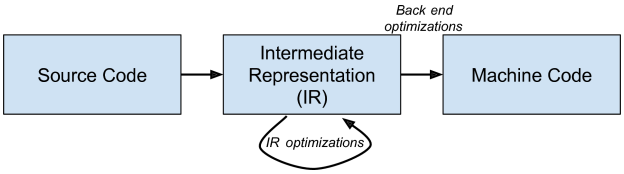


Fig. 1: LLVM/Clang code compilation flow

### C. Fault Injection (FI)

Fault injection (FI) is a software testing technique used to evaluate a program's error coverage. A typical FI experiment will consist of many individual FI *runs* (typically hundreds), each run being a single execution of the program with an error introduced. Once an error is introduced in the program, it can result in a failure, which is either an SDC or a crash[2], or a benign output. Once all FI runs have completed, we can then obtain a statistical estimate of the SDC/crash probabilities.

[2]In this paper, we consider program *hangs* as part of the *crash* category.

In this paper, we are interested in emulating transient hardware errors (i.e., soft errors) caused by cosmic ray or alpha particle strikes affecting flip flops and logic elements. These errors typically manifest in the form of bit-flips, and thus in our FI experiements a single bit-flip is injected per FI run. We consider errors that occur in the processor's computation units, e.g., arithmetic operations and address computations for load and store instructions. However, errors in 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 errors in the control logic of the processor as this is a small portion of the processor area, nor do we consider errors in the instruction encoding, as these can be handled through control-flow checking techniques [21]. Related work has made similar assumptions [9], [22]–[25].

*1) Instruction sampling:* In each FI run, a dynamic instruction needs to be determined as the FI target. Since soft errors occur randomly, we choose a dynamic instruction at random among the total executed sequence of instructions in the program with a uniform distribution. Thus, if the program has $N$ total dynamic instructions in the execution, each instruction has $1/N$ probability to be sampled in each FI run. This sampling methodology makes an implicit assumption that each instruction takes approximately the same amount of time to execute. This is because we are performing the injection at the program level, where we do not have detailed information about the microarchitectural or cache state of the instruction. This is a common assumption in program-level FI techniques.

*2) Bit sampling:* Once a dynamic instruction is chosen as the FI target, a single bit within the destination register of that instruction needs to be selected as the target of injection. As in the instruction sampling, a register bit is randomly selected to be the target. Since we are interested in the program behaviour given that an error has occurred (as our goal is to measure error resilience), we only consider activated faults (i.e., errors). Thus, we only sample from bits in the destination register that are used by the program. For example, if an instruction writes a 64-bit value to a 128-bit destination register, only the 64 bits corresponding to the written value are sampled from, with each bit having a probability of $1/64$ to be sampled.

*3) Assembly-level FI:* FI can be conducted at different levels of abstraction, including at the IR and assembly code levels. Assembly-level FI tools utilize dynamic binary instrumentation (e.g., PIN, DynamoRIO and Valgrind) to access the assembly code for FI. They are considered to be accurate for studying hardware faults, such as soft errors, since assembly code is close to hardware [13], [26]. Common assembly-level fault injectors include BIFIT [27], PINFI [12], FITgrind [28] and others [29], [30]. The main drawbacks of assembly-level FI are that (1) it has limited portability because it operates at the platform-specific assembly code level, and (2) it is difficult to obtain insights for software design, since IR-level code abstractions (e.g., loops and data structures) are not available at the assembly level. Therefore, it is difficult to map FI locations back to the source code for further investigation. In this paper, we use **PINFI** to implement assembly-level FI experiments.

*4) IR-level FI:* IR-level FI uses compiler techniques to inject faults into the compiler's intermediate representation (IR) code. Popular IR-level fault injectors are LLFI [8], KULFI [25], VULFI [31], and FlipIt [7]. In addition to its high platform portability, the IR level preserves the information of the program source code. Hence, it is easier to map the FI locations back to the source code. It also allows the injection of faults into specific code structures (e.g., loops and data structures). Moreover, the IR level is where significant program analysis tools are available. Therefore, IR-level FI makes the post-analysis much easier compared to assembly-level FI. However, the main concern is accuracy, as there are various back end optimizations performed on the code that are not available to the IR. For example, since the IR is platform-independent and assumes an infinite number of available registers, register allocation is not performed until the back end compilation stage, and hence there can be a mismatch between the number of memory operations in the IR and assembly code. In this paper, we use **LLFI** to implement IR-level FI experiments.

## III. Related Work

There is a large body of work on using fault injection to measure the error resilience of computer programs, using both hardware and software techniques. Initially, most studies that investigated error resilience to transient hardware errors relied on hardware FI, which involves injecting faults through the hardware layer either with or without physical contact [32].

On the other hand, the use of software techniques to emulate transient hardware errors has seen increased interest over the last decade, as it does not require expensive hardware and is more flexible [33]. It is important to note however that, while software techniques offer improvements in cost, flexibility, and portability, it is often difficult or impossible to inject faults into locations that are inaccessible to software [33]. For example, a paper by Cho et al. [34] found that assembly-level FI can only capture a subset of system-level behaviour caused by soft errors. However, our focus (and specifically the focus of the two prior studies compared in Section IV) is on the subset of errors that make their way to the application, and can therefore be modeled using higher-level FI techniques.

IR-level FI techniques that operate at the compiler level have become especially popular in recent years, as they are portable and allow injections into IR-level source code abstractions. Many studies have adopted such techniques to study transient hardware faults that cause SDCs. For example, Thomas and Pattabiraman used LLFI to evaluate their technique for detecting SDC-causing errors [9]. Calhoun et al. used FlipIt [7], an LLVM-based FI tool, to investigate how SDCs propagate through a specific HPC computation kernel [10]. Chen et al. introduced LADR, an application-level SDC detector that was evaluated using IR-level FI experiments [35]. Finally, Li et al. used LLFI to estimate program SDC probabilities [11]. Studies such as these use IR-level FI under the assumption that it is almost as accurate as assembly-level FI in measuring SDCs.

To the best of our knowledge, only two prior studies directly compare the accuracy of IR-level FI with that of assembly-level FI. Wei et al. compare the accuracy of IR-level FI with that of assembly-level FI for emulating hardware errors. To represent IR-level FI, they introduce LLFI, which performs FI at the LLVM compiler's IR level. To represent assembly-level FI, they introduce PINFI, which performs FI at the x86 assembly level. They conduct FI experiments on a set of standard benchmarks using both LLFI and PINFI, and compare the results. Based on the results of the experiments, the authors claim that *"LLFI is accurate for emulating hardware errors that cause Silent Data Corruption (SDCs), but not crashes"*.

A more recent paper by Georgakoudis et al. [13] investigates the accuracy of IR-level FI with respect to assembly-level FI for emulating hardware errors. The paper claims that IR-level FI is significantly less accurate than assembly-level FI, and that the inaccuracies are due to assembly-level dynamic binary instructions and back end compiler optimizations that are not available at the IR level. Further, they find that such inaccuracies manifest for both SDCs and crashes. To address the limitations of IR-level FI, they present REFINE, a fault injector that performs IR-level FI at the compiler back end, as opposed to at the IR level. The paper evaluates this tool by comparing FI experiment results using REFINE, LLFI, and PINFI, and comes to the conclusion that while REFINE is accurate when compared to PINFI, LLFI is not.

By looking at the previous two papers, it is unclear to a reader whether FI performed at the IR level is as accurate as assembly-level injection for evaluating SDC-causing hardware errors. Wei et al. [12] claim that LLFI is accurate for emulating SDC-causing hardware errors, while Georgakoudis et al. [13] claim otherwise. This contradiction is peculiar considering that both papers claim to use the same FI tools (i.e., LLFI and PINFI) and similar hardware platforms (i.e., x86 processors). In this paper, we perform an analysis of these two studies to find the root cause of the inconsistent findings. To confirm our findings, we present an 'end-to-end' comparison between IR-level and assembly-level FI, expanding on those done in prior work by (1) using a much larger set of benchmarks (larger than both studies combined), (2) testing across four different levels of optimization, and (3) employing a more rigorous statistical analysis of the results.

## IV. Analysis of Inconsistencies in Prior Work

As mentioned in Section III, there are two studies that come to contradictory conclusions in their work regarding the accuracy of IR-level FI. While Wei et al. [12] claim that IR-level FI is as accurate as assembly-level FI for emulating SDC-causing hardware errors, Georgakoudis et al. [13] claim otherwise. Both studies claim to use LLFI and PINFI to conduct their respective FI experiments. The inconsistency in their results calls into question the accuracy of IR-level FI. To address this discrepancy, we conduct our own investigation on the two studies by attempting to replicate their experiments.

Our investigation consists of the following three steps: (1) We first attempt to reproduce the results in Georgakoudis et al. [13], using the same experimental setup that they used. (2) We then examine any differences between the two studies to isolate potential causes for the contradictory results, and develop a hypothesis. (3) Finally, we design our own experiments to test the hypothesis and draw our conclusions.

**Step 1: Reproduction:** In this step, we conduct the same FI experiments presented in Georgakoudis et al. [13], using the same set of benchmarks and program inputs used in their experiments. In their study, LLFI[3] was used to represent IR-level FI, without any modification, and hence we do the same. To represent assembly-level FI, they use PINFI. However, the paper mentions that PINFI is slightly modified to *"render it compatible with the recent version of the Intel PIN framework and for faithfully implementing the fault model"* [13]. This modified version of PINFI was made publicly available by the research group[4]. The use of a modified version of PINFI is a notable difference from Wei et al. [12], which uses the version hosted on the official PINFI repository[5].

Because the two studies use different versions of PINFI, we perform our experiments using both versions to observe any effects the modifications may have on the results. To differentiate between the two versions, we label the official version of PINFI used by Wei et al. as **PINFI-v1**. We label the modified version used by Georgakoudis et al. as **PINFI-v2**.

In our experiments, we adopt the original FI methodologies that are used in both studies. We conduct a total of 1000 FI runs for each benchmark in the experiment, and only one optimization level (*-O3*) is used to compile the benchmarks, as is done in the studies. We conduct the FI experiments on 12 benchmarks that are used by Georgakoudis et al., using the same program inputs they used [13]. Figure 2 shows the SDC probabilities obtained from our experiments using LLFI, PINFI-v1, and PINFI-v2. Results obtained using PINFI-v3 are also included, which we will introduce in Step 3.

**Step 2: Hypothesis:** From these results, we observe that the SDC probabilities measured by LLFI are similar to those measured by PINFI-v1. On the other hand, the SDC probabilities measured by LLFI are significantly different from those of PINFI-v2. Further, the results obtained using PINFI-v1 and PINFI-v2 do not match (the latter are almost half the value of the former), suggesting that modifications in the latter significantly affect the results of the experiments; we postulate that these modifications are the cause for the inconsistencies.

Next, we compare the source code for both PINFI-v1 and PINFI-v2, and isolate any differences that could potentially lead to different results. We find that one of the code modifications in PINFI-v2 alters how PINFI determines the range of bits in the sampled instructions that are considered as FI sites. In PINFI-v1, only the range of bits in the chosen instruction that are used by the program are considered. This is consistent with the methodology used in LLFI. On the other hand, PINFI-v2 considers the entire range of bits in the instruction's

[3]https://github.com/DependableSystemsLab/LLFI

[4]https://github.com/ggeorgakoudis/REFINE/tree/master/pinfi

[5]https://github.com/DependableSystemsLab/pinfi

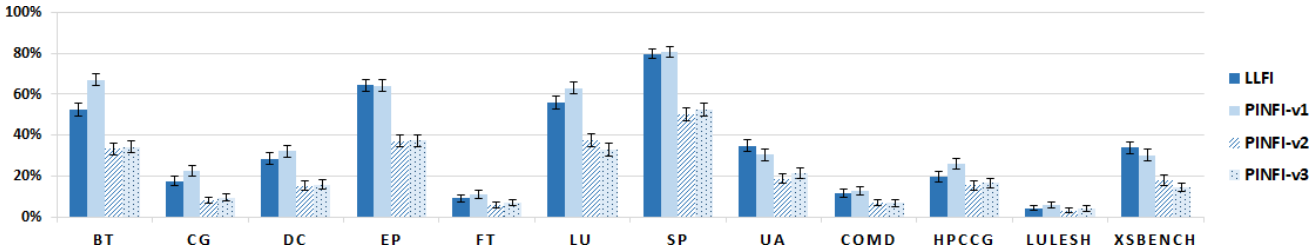

Fig. 2: Program SDC probabilities ($y$-axis) measured by LLFI and PINFI versions plotted for each benchmark ($x$-axis).

destination register, regardless of whether the entire range of bits is actually used by the program. In other words, PINFI-v1 limits its bit selection to only activated faults, while PINFI-v2 considers both activated and unactivated faults.

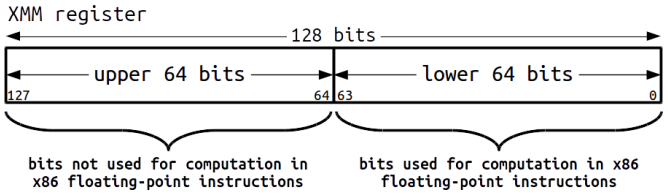

Fig. 3: The bits of an `XMM` register that are used in x86 floating-point instructions

To illustrate this concept, we consider x86 scalar double-precision floating-point instructions (e.g., `addsd`, `mulsd`). x86 floating-point instructions typically use `XMM` registers as the destination register, which are 128-bit registers. As such, one could consider all 128-bits in the destination register as potential fault injection sites. However, these instructions perform operations using double-precision floating-point values, and hence only the *lower 64 bits* are used for computation, while the upper 64 bits are unused and irrelevant to the program[6]. This is illustrated in Figure 3. Thus, if a fault is injected into the upper 64 bits it is very likely that it will result in a benign error, regardless of the inherent error-resilience of the program. While PINFI-v1 considers such cases to ensure that all faults are activated [12], PINFI-v2 does not.

In the case of floating-point instructions, this difference in bit-sampling models would likely lead to a difference in the measured SDC probabilities (as floating-point instructions are typically only used in the program's data flow). Other cases where there is a difference in the bit-sampling model could potentially lead to differences in measured crash probabilities, such as cases involving memory access instructions.

We therefore hypothesize that the modification of how the injected bit is selected is the reason for the inconsistencies between Wei et al. [12] and Georgakoudis et al. [13].

**Step 3: Support for the hypothesis:** To test our hypothesis, we create our own modified version of PINFI, which we label **PINFI-v3**. PINFI-v3 is a fork of the PINFI-v1 code, with a modification to the part of the code that is responsible for bit selection, to match that of PINFI-v2. We then run the same FI experiments with PINFI-v3, the results of which are shown in Figure 2. The results show that the SDC probabilities obtained using PINFI-v3 closely match those obtained using PINFI-v2, indicating that our hypothesis is valid.

**Discussion of findings:** In summary, the modification made by Georgakoudis et al. [13] to the bit-sampling model used in PINFI caused a disparity in the types of faults that were injected by the two tools (i.e., only activated faults vs. unactivated faults), which we found significantly alters the SDC probability measurements made by PINFI. Since the modified PINFI bit selection method used in Georgakoudis et al. [13] is inconsistent with the bit selection method used by LLFI, the paper's comparison between LLFI and PINFI is invalid.

Note that the low-level mechanisms that allowed for this change to the bit sampling model are fundamentally not available at the IR level; the intricacies related to register size and bit usage are machine-specific. This is a key advantage of IR-level FI, i.e., one does not need to worry about these low-level issues when designing FI experiments at the IR level.

As was shown in our investigation, using a different bit-sampling model can affect the results of a FI experiment very significantly. We are not claiming one model is superior to the other, as it depends on the goal of the study. For example, for error resilience measurements, activated faults are what matter [11], while for raw FIT rate measurements, all faults may matter regardless of activation. However, if the goal is for an apples-to-apples comparison, e.g., comparing the SDC measurements between LLFI and PINFI, one must keep the FI configurations consistent in both FI experiments. Unfortunately, Georgakoudis et al. [13] investigated the accuracy of LLFI by applying a significantly different bit-sampling model in PINFI, and hence their results are not valid in this respect.

## V. End-to-End Comparison: Experimental Setup

In Section IV, we showed that the conclusions drawn by Georgakoudis et al. [13] are based on a flawed FI comparison study. We now expand on this finding by conducting an extensive set of FI experiments that confirm the conclusions drawn by Wei et al. [12]. In this section, we first describe the benchmarks, FI tools, and metrics for our experiments.

### A. Benchmarks

In our experiments, we choose a total of 25 different benchmarks from 8 publicly available benchmark suites. Their details are shown in Table I. We choose these benchmarks because they are (1) from a broad selection of application

[6]This is the case for *scalar* floating-point instructions; *packed* floating-point instructions (e.g., `addpd`, `mulpd`) will often store two aligned values thus utilizing the upper 64 bits as well.

TABLE I: Details of Benchmarks

| Benchmark | Suite | Input |
|---|---|---|
| blackscholes | PARSEC | `1 in_16K.txt output.txt` |
| fluidanimate | PARSEC | `1 10 in_5K.fluid out.fluid` |
| lud | Rodinia | `-v -i 512.dat` |
| backprop | Rodinia | `65536` |
| kmeans | Rodinia | `-i 819200.txt -k 1` |
| bfs | Rodinia | `1 graph1MW_6.txt` |
| bzip2† | SPEC | `-1kvv image.jpg` |
| libquantum† | SPEC | `33 5` |
| hmmer† | SPEC | `--seed 10000000 ig.hmm` |
| mcf† | SPEC | `inp.in` |
| ocean† | SPLASH-2 | `-p1 -o` |
| raytrace† | SPLASH-2 | `-p1 -m64 inputs/car.env` |
| CoMD‡ | Mantevo | `-x 10 -y 10 -z 10 -N 50` |
| HPCCG‡ | Mantevo | `64 64 64` |
| XSBench‡ | CESAR | `-s small` |
| BT‡ | NPB | `S` |
| CG‡ | NPB | `S` |
| DC‡ | NPB | `10000000 ADC.par` |
| EP‡ | NPB | `W` |
| FT‡ | NPB | `W` |
| IS | NPB | `S` |
| LU‡ | NPB | `W` |
| MG | NPB | `S` |
| SP‡ | NPB | `W` |
| UA‡ | NPB | `W` |

†Benchmark used in Wei et al. [12]
‡Benchmark used in Georgakoudis et al. [13]

domains, (2) open source and compatible with both fault injectors, and (3) used in the two related FI studies as indicated in Table I [12], [13]. The benchmarks represent a large range of computational complexity and execution times, however we keep the size of the inputs small when possible to avoid overly long FI experimental times.

We include all of the benchmarks used in Wei et al. [12], and all but three of the benchmarks used in Georgakoudis et al. [13]; *AMG2013*, *lulesh*[7], and *miniFE* are not used because they are either (1) incompatible with the platform used for our experiments, or (2) incompatible with LLFI when compiled using some of the pertinent optimization levels. We also include several benchmarks not used in the two prior studies. We use the default inputs included in the benchmark suites, or example inputs where the former are not available.

### B. Fault Injection Tools

To be consistent with the tools used in prior studies [12], [13], we choose LLFI and PINFI for representing IR-level and assembly-level FI respectively. LLFI operates at the LLVM compiler's IR level. It takes the compiled LLVM IR of the program as input and performs both the profiling pass and the FI runs at the IR level. PINFI operates at the x86 assembly code level. It is built as a tool for Intel PIN [36], an assembly-level instrumentation tool for x86 processors. PINFI operates in a similar fashion to LLFI, with 2 differences, (1) instrumenting the program at runtime (rather than compile-time), and (2) taking a program's executable file as the input (rather than its source code).

[7]*lulesh* was in fact used as one of the benchmarks for the analysis of prior work in Section IV; this is because the one optimization level that was used to compile the benchmark for those FI experiments (-O3) does not cause the same LLFI compatibility issues as the other optimizations.

### C. Platforms and Compilations

All experiments are conducted on 64-bit Intel x86 machines. We use LLVM/Clang 3.4 to compile from the benchmarks' C/C++ source code to their LLVM IR files and executables. PINFI uses Intel PIN 3.5 to access the compiled machine code of the benchmarks.

### D. Measurement of Accuracy

We first show a graphical overview of the results for each benchmark to visually compare the outcomes of FI execution for each fault injector. We then apply three types of statistical tests to quantify the difference between IR-level and assembly-level FI: (1) least squares regression analysis; (2) paired sample *t*-test; and (3) Spearman's rank correlation test.

*1) Least squares regression:* The first statistical analysis we apply is a least squares regression model [37]. The analysis is performed for each optimization level across the set of benchmarks. The method of least squares is a standard approach in regression analysis to obtain the line of best fit for a set of data points. The reason for using this analysis is to measure the linear relationship between the SDC/crash probabilities obtained using LLFI and those using PINFI. The model plots the PINFI probabilities against those of LLFI.

In the ideal situation where LLFI produces the exact same measurements as PINFI, these data points would form a straight line with a slope of 1 and *y*-intercept of 0 (i.e., having a linear equation of $y = x$). For example, if for a given benchmark and optimization level the SDC probability measurements obtained from LLFI and PINFI fall on the line $y = x$, it indicates that LLFI and PINFI measure the same SDC probability (for that benchmark and optimization level). Thus, the linear equation and the corresponding $R^2$ value obtained from this analysis provide an indication of how close the data points are to the ideal situation (i.e., higher $R^2$ values are better). We estimate the slope and *y*-intercept parameters with the 95% confidence intervals.

*2) Paired sample t-test:* We use a paired sample *t*-test[8] to compare the SDC and crash probability measurements made by LLFI and PINFI. Our null hypothesis states that the mean difference between the probabilities measured using LLFI and those measured using PINFI is zero. In other words, if the null hypothesis were to hold true, all observable differences would be explained by random variation, thus implying that the measurements made by LLFI and PINFI are not significantly different. We use a two-tailed alternative hypothesis that assumes the mean difference is not equal to zero, which would imply that there is non-random variation in the measurements.

We perform the paired *t*-test on the set of benchmarks at each individual optimization level, so that we can compare the significance of the results at each level. The *p*-values calculated using the test give us the probability of observing the experiment results under the null hypothesis (i.e., a high *p*-value indicates increased support for the null hypothesis). We use a significance level of $0.05$, which corresponds to a 95%

[8]Our dataset meets all *t*-test assumptions as we examined.

confidence level. If the *p*-value is less than $0.05$, we reject the null hypothesis and conclude that the measurements made using LLFI and PINFI are (statistically) significantly different. Otherwise, we do not reject the null hypothesis.

*3) Spearman's rank correlation test:* Program SDC and crash probabilities are application-specific. This is because different programs have different characteristics of propagating SDC- and crash-causing errors. Often developers need to use FI to find which applications produce higher SDC probabilities than others to make design choices among them (these include different versions of the application protected with different techniques). Therefore, a fault injection technique needs to be sensitive to the relative rankings of the SDC probabilities.

To examine the sensitivity of the measurement by both injectors, we conduct a Spearman's rank correlation test. This test is used to assess whether the relationship between two variables is monotonic, i.e., if one value increases or decreases, the other does the same. A Spearman's rank correlation coefficient close to 1 indicates a strong monotonic relationship. In our case, this would mean that LLFI and PINFI are both equally sensitive in distinguishing the ranking of program SDC/crash probabilities. Note that the Spearman's rank correlation test does not assume normality of the measurement errors unlike the above two tests, and is hence more robust to variations from the normal distribution.

## VI. End-to-End Comparison: Results

In this section, we present our experimental results based on fault injection experiments conducted on the 25 benchmarks listed in Table I. We compile each benchmark with the four optimization levels respectively (*-O0*, *-O1*, *-O2*, and *-O3*). For each benchmark at each optimization level, we perform 1000 fault injections using both LLFI and PINFI respectively to obtain our SDC, crash, and benign probabilities; this gives us a sufficiently large sample size to estimate the program SDC and crash probabilities with tight error bars calculated with a 95% confidence interval. Thus, we perform a total of $100,000$ fault injection runs (= 25 * 4 * 1000) for each tool.

### A. Program SDC Probabilities

Figure 4 shows the SDC probabilities obtained using LLFI and PINFI for each benchmark. We present the numerical results using bar graphs with error bars for visual comparison. The error bars represent the 95% confidence interval for 1000 runs. The least squares regression analysis results are shown in Table II, while the *t*-test and Spearman's rank test results are shown in Table III.

Figure 4 shows that the SDC probabilities measured by PINFI and LLFI are close, with the error bars overlapping between the two for the majority of the benchmarks. This observation is consistent across all four optimization levels. The mean absolute errors between the SDC probability measurements from LLFI and PINFI are $2.192\%$, $4.988\%$, $4.796\%$, and $4.428\%$ for *-O0* to *-O3*, respectively, indicating that for most benchmarks, the SDC probabilities measured using LLFI are almost indistinguishable from those measured by PINFI.

Further, the results from the least squares linear regression analysis (Table II) show that the data follows a strong linear relationship. At every optimization level, the slope, $m$, is close to 1 and the $y$-intercept, $b$, is almost 0, with little variance. Furthermore, the values of 1 and 0 are within the confidence ranges for the slope and intercept values at each optimization level. The $R^2$ values are also high at each optimization level, with three out of four values above $0.9$. This shows that the data fits closely with the line given by the slope and $y$-intercept values. This regression analysis is illustrated in Figure 5, which shows how closely the data points fit the regression line for the SDC probabilities for each optimization. We can therefore conclude that, at all four optimization levels, the SDC probabilities obtained using LLFI and those obtained using PINFI follow a strong linear relationship.

Table III shows the *p*-values obtained from the paired sample *t*-test performed on the SDC probabilities measured using LLFI and PINFI. As all of the *p*-values are well above $0.05$, the results from this test are not sufficient to reject the null hypothesis. Therefore, there is no evidence that suggests the SDC probabilities measured using the two tools are significantly different from each other.

Finally, the results from the Spearman's rank test (Table III) indicate a strong monotonic relationship between the SDC probabilities measured using LLFI and those from PINFI. The correlation coefficients measured are all above $0.9$ and close to 1. We therefore conclude that LLFI is as sensitive to distinguishing the ranking of individual program SDC probabilities as PINFI.

TABLE II: Least Squares Regression Analysis Results

| | | slope, $m$ | $y$-intercept, $b$ | $R^2$ |
|---|---|---|---|---|
| SDC | *-O0* | $0.9948 \pm 0.0689$ | $0.0060 \pm 0.0188$ | 0.9732 |
| | *-O1* | $1.1197 \pm 0.1426$ | $-0.0177 \pm 0.0445$ | 0.9147 |
| | *-O2* | $1.0381 \pm 0.1463$ | $-0.0024 \pm 0.0495$ | 0.8975 |
| | *-O3* | $1.0472 \pm 0.1431$ | $-0.0084 \pm 0.0485$ | 0.9030 |
| Crash | *-O0* | $0.8129 \pm 0.2264$ | $0.0398 \pm 0.0956$ | 0.6915 |
| | *-O1* | $0.5216 \pm 0.3562$ | $0.0842 \pm 0.1179$ | 0.2716 |
| | *-O2* | $0.5191 \pm 0.2565$ | $0.0619 \pm 0.0823$ | 0.4160 |
| | *-O3* | $0.4867 \pm 0.2436$ | $0.0637 \pm 0.0796$ | 0.4099 |

TABLE III: Statistical Test Results

| | | *-O0* | *-O1* | *-O2* | *-O3* |
|---|---|---|---|---|---|
| *p*-value† | SDC | 0.4210 | 0.3920 | 0.6208 | 0.7834 |
| | Crash | 0.0217 | 0.0215 | 0.0031 | 0.0016 |
| Correlation coeff.‡ | SDC | 0.9636 | 0.9400 | 0.9285 | 0.9354 |
| | Crash | 0.8398 | 0.6154 | 0.6672 | 0.6659 |

†Measured using paired sample *t*-test (Section V-D2)
‡Measured using Spearman's rank test (Section V-D3)

### B. Program Crash Probabilities

Figure 6 shows the crash probabilities obtained using LLFI and PINFI for each benchmark. As in SDCs, the least squares regression analysis is shown in Table II, and the *t*-test and Spearman's rank test results are shown in Table III.

Figure 6 shows that unlike SDC probabilities, the crash probabilities do not consistently match between LLFI and

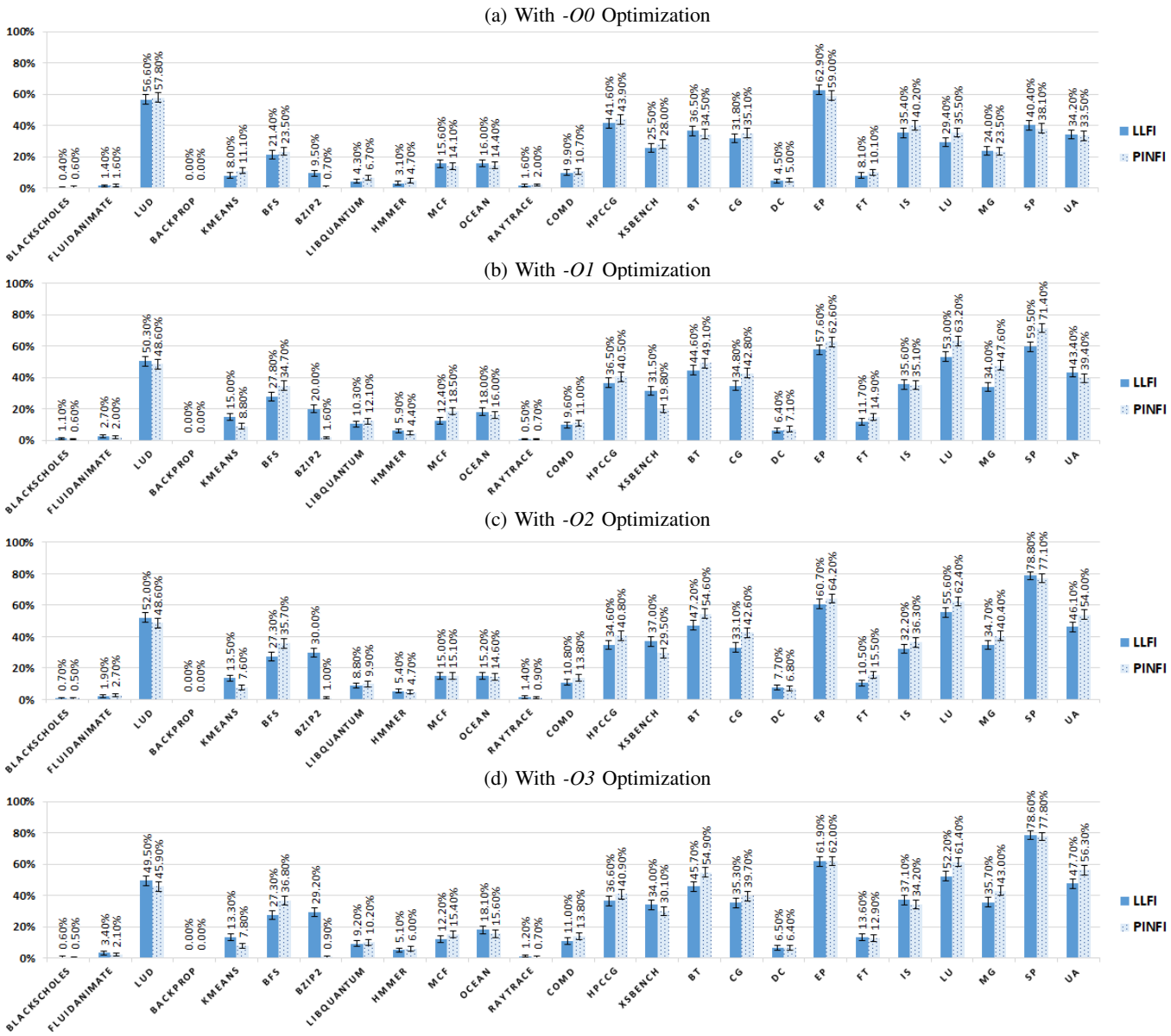


Fig. 4: Program SDC probabilities ($y$-axis) measured by LLFI and PINFI plotted for each benchmark ($x$-axis).

PINFI for all optimizations. Further examination reveals that at *-O0* the crash probabilities tend to be similar with overlapping errors for most of the benchmarks, while at *-O1*, *-O2*, and *-O3* this is not the case. In addition, the mean absolute error between the crash probability measurements from LLFI and PINFI are $6.428\%$, $11.232\%$, $11.412\%$, and $11.664\%$ for *-O0* to *-O3*, respectively. This indicates that the crash probabilities of LLFI and PINFI are similar at optimization levels *-O0*, but not at the other optimization levels, namely *-O1* to *-O3*.

The results from the least squares linear regression analysis (Table II) follow the same pattern. At *-O0*, the slope of the line of best fit is $0.8129$, with a slope of 1 falling within the confidence interval. However, at *-O1*, *-O2*, and *-O3* the slopes are only $0.5216$, $0.5191$, and $0.4867$ respectively. The $R^2$ values also follow this trend, dropping from $0.6915$ at *-O0* to $0.2716$ at *-O1*. At all optimization levels, the $y$-intercept is close to 0, with 0 falling within the confidence interval. While the linear relationship at *-O0* is not as strong as those for the SDC probabilities, we find that as more optimizations are applied to the program, the less accurate the crash probabilities measured by LLFI become compared to PINFI.

Table III shows the $p$-values obtained from the paired sample $t$-test performed on the crash probabilities measured using LLFI and PINFI. As all of the $p$-values are below $0.05$, we reject the null hypothesis that the mean difference between the probabilities measured using LLFI and those measured using PINFI is zero. This suggests that there is a statistically significant variation in the crash probability measurements made using LLFI and PINFI.

As shown in Table III, the results from the Spearman's rank test indicate a moderate-to-strong monotonic relationship between the crash probabilities measured using LLFI and

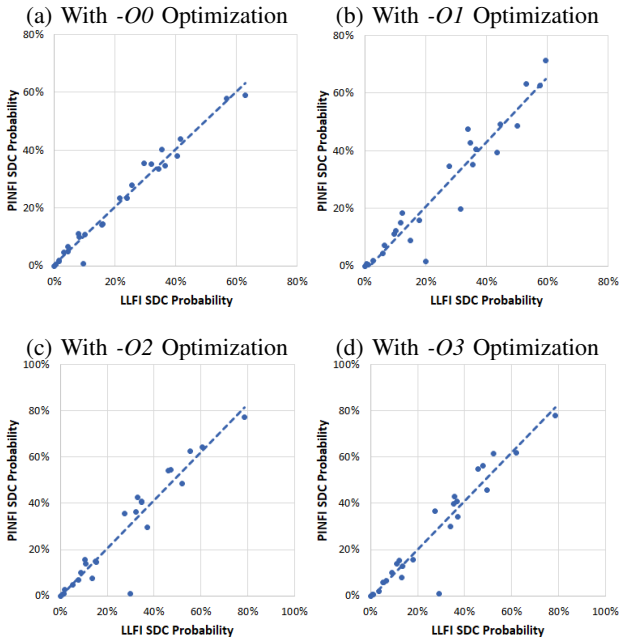


Fig. 5: Program SDC probabilities ($y$-axis) measured by PINFI plotted against those measured by LLFI ($x$-axis) at each optimization level, with least squares regression line. The ideal line of best fit is a line with slope of 1 and $y$-intercept of 0.

that of PINFI. At *-O0*, the correlation coefficient is 0.8398, indicating a strong monotonic relationship. At *-O1*, *-O2*, and *-O3* however, this number drops to between 0.6 and 0.7. Thus, LLFI is sensitive to distinguishing the ranking of program crash probabilities at *-O0*, but not at *-O1*, *-O2*, and *-O3*.

Therefore, we conclude that the crash probabilities do not consistently match between LLFI and PINFI for all optimizations. Further, the accuracy of the crash probability measurements is affected by the compiler optimizations applied to the program, especially going from *-O0* to *-O1*.

## VII. Discussion

### A. Discussion of FI Results

Based on the results presented in Section VI, we conclude that the SDC probabilities are measured accurately by IR-level FI when compared with assembly-level FI. This observation makes sense, considering that SDCs (i.e., incorrect outputs) can be mostly attributed to faults in a program's data flow, which is relatively unaffected by back end optimizations.

On the other hand, we find that the accuracy of crash probability measurements are affected by different optimizations applied to the program. Since a program's IR and assembly share common front and middle end compilations, but have differing back end compilations, the accuracy of crash probability measurements seem to be affected by back end optimizations (which are not visible at the IR level). A major cause for crashes is segmentation faults, which are commonly known as illegal memory accesses [38]. They are due to errors propagating into memory operators, such as the address operators of load or store instructions. Since many back end optimizations operate on a program's memory operations, it is likely that the inaccuracies in crash probability measurements at higher optimization levels are due to how the optimizations affect the amount of these memory instructions.

The results of this study are consistent with the findings in the study by Wei et al. [12], thus also confirming the results of our analysis in Section IV.

### B. When should we use IR-level FI?

In light of the findings discussed in the previous section, we now address the following question: *when is it okay to choose IR-level FI techniques over assembly-level FI?*

For the case of SDCs, IR-level FI is as accurate as assembly-level FI even in the presence of aggressive compiler optimizations. Thus, we conclude that IR-level FI can be used when SDCs are the program outcome of interest, e.g., when (1) characterizing program-level SDC error propagation or (2) quantifying the SDC probability of a program.

In the case of crashes however, IR-level FI does not offer the same level of accuracy as assembly-level FI, especially when compiler optimizations are applied. Therefore, the use of IR-level FI should be avoided when quantifying the crash probability of a program. We will explore methods to improve the accuracy of IR-level FI in future work.

### C. Lessons Learned

In Section IV, we find that a small change in how the injected bit is selected can significantly affect the results of a FI experiment. While we do not know the exact reasoning for the specific modification to PINFI in the study [13] that implemented this change, we assume that the authors were unaware of the full implications surrounding the change as uncovered in this paper.

While it may be argued that injecting faults into unused register bits are more representative of how hardware faults occur in the real world, most program-level FI studies (including ours) aim to measure error resilience - this is the probability of the program not failing given that a fault has occurred. This is because it is very difficult, if not impossible, to accurately capture the effect of all hardware faults at the program level as there may be significant masking in the lower layers [34]. With that said, we do not take a position on the "right" way to perform fault injection experiments in this paper. One may choose to inject into all register bits (i.e., to measure "fault sensitivity") or specifically into bits that are read by the program (i.e., to measure "error sensitivity"). It is, however, important to be consistent as these differences can make a significant difference in the results (Section IV).

The finding that a seemingly insignificant change in a FI methodology can alter results so drastically means that careful consideration should be given when defining the parameters of one's FI study. This becomes especially important when conducting any sort of comparison between two tools. While there will certainly be differences between how the tools are implemented, all other factors should be controlled for

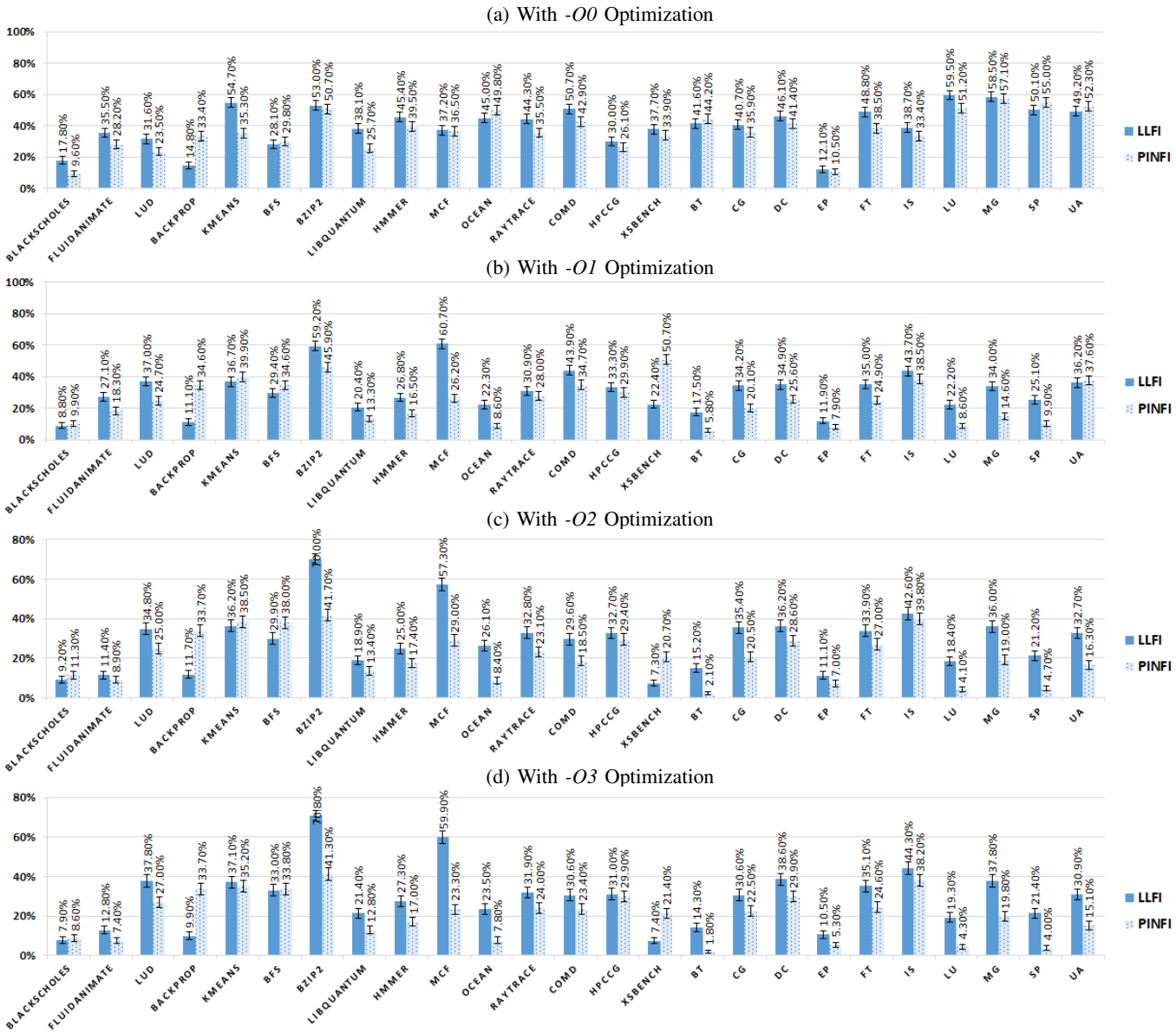


Fig. 6: Program crash probabilities ($y$-axis) measured by LLFI and PINFI plotted for each benchmark ($x$-axis).

equivalency to provide a fair and useful comparison. This is not only limited to the bit sampling methodology as is the case in our findings, but should also be considered when deciding on other factors such as instruction selection or fault types. While it often may be easy to miss small variations between different tool implementations, we believe that the careful consideration of seemingly negligible implementation details must become common practice in future FI studies.

## VIII. Conclusion

Fault Injection (FI) at the intermediate representation (IR) level is a promising alternative to assembly code level FI for evaluating software techniques for protecting programs from hardware faults. In this paper, we conduct a thorough investigation into the results of two papers, Wei et al. [12] and Georgakoudis et al. [13], which draw contradictory conclusions regarding the accuracy of IR-level FI for measuring SDC probability. We find that the root cause of the contradiction is in the modification made to PINFI by Georgakoudis et al. [13] in how the bit to be injected is selected. We then conduct a set of FI experiments using 25 benchmark programs at four different compiler optimization levels, and find that IR-level FI is as accurate as assembly-level FI in measuring the SDC probability of a program, regardless of the optimization level.

## Acknowledgement

This research was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) through the Discovery Grants and Strategic Project Grants (SPG) Programmes. We thank the anonymous reviewers of ISSRE'19 for their insightful comments and suggestions.

**All the data and tools in this paper are available at: *https://github.com/DependableSystemsLab/ISSRE19***

## REFERENCES


[1] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, A. A. Chien, P. Coteus, N. A. Debardeleben, P. C. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. Munson, R. Schreiber, J. Stearley, and E. V. Hensbergen, "Addressing failures in exascale computing," *Int. J. High Perform. Comput. Appl.*, vol. 28, no. 2, pp. 129–173, May 2014. [Online]. Available: http://dx.doi.org/10.1177/1094342014522573

[2] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10–16, Nov 2005.

[3] C. Constantinescu, "Intermittent faults and effects on reliability of integrated circuits," in *2008 Annual Reliability and Maintainability Symposium*, Jan 2008, pp. 370–374.

[4] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco, "A flexible software-based framework for online detection of hardware defects," *IEEE Transactions on Computers*, vol. 58, no. 8, pp. 1063–1079, Aug 2009.

[5] Y. Zhang, S. Ghosh, J. Huang, J. W. Lee, S. A. Mahlke, and D. I. August, "Runtime asynchronous fault tolerance via speculation," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12. New York, NY, USA: ACM, 2012, pp. 145–154. [Online]. Available: http://doi.acm.org/10.1145/2259016.2259035

[6] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt, "Techniques to reduce the soft error rate of a high-performance microprocessor," *SIGARCH Comput. Archit. News*, vol. 32, no. 2, pp. 264–, Mar. 2004. [Online]. Available: http://doi.acm.org/10.1145/1028176.1006723

[7] J. Calhoun, L. Olson, and M. Snir, "Flipit: An LLVM based fault injector for HPC," in *Euro-Par 2014: Parallel Processing Workshops - Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part I*, 2014, pp. 547–558. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-14325-5_47

[8] Q. Lu, M. Farahani, J. Wei, A. Thomas, and K. Pattabiraman, "Llfi: An intermediate code-level fault injection tool for hardware faults," in *2015 IEEE International Conference on Software Quality, Reliability and Security*, Aug 2015, pp. 11–16.

[9] A. Thomas and K. Pattabiraman, "Error detector placement for soft computation," in *Proceedings of the 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, ser. DSN '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1–12. [Online]. Available: http://dx.doi.org/10.1109/DSN.2013.6575353

[10] J. Calhoun, M. Snir, L. Olson, and M. Garzaran, "Understanding the propagation of error due to a silent data corruption in a sparse matrix vector multiply," in *2015 IEEE International Conference on Cluster Computing*, Sept 2015, pp. 541–542.

[11] G. Li, K. Pattabiraman, S. K. S. Hari, M. Sullivan, and T. Tsai, "Modeling soft-error propagation in programs," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2018, pp. 27–38.

[12] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "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 (DSN)*, vol. 00, June 2014, pp. 375–382. [Online]. Available: doi.ieeecomputersociety.org/10.1109/DSN.2014.2

[13] G. Georgakoudis, I. Laguna, D. S. Nikolopoulos, and M. Schulz, "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*, ser. SC '17. New York, NY, USA: ACM, 2017, pp. 29:1–29:14. [Online]. Available: http://doi.acm.org/10.1145/3126908.3126972

[14] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "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*, ser. ASPLOS XV. New York, NY, USA: ACM, 2010, pp. 385–396. [Online]. Available: http://doi.acm.org/10.1145/1736020.1736063

[15] S. K. S. Hari, R. Venkatagiri, S. V. Adve, and H. Naeimi, "Ganges: Gang error simulation for hardware resiliency evaluation," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, June 2014, pp. 61–72.

[16] G. Li, Q. Lu, and K. Pattabiraman, "Fine-grained characterization of faults causing long latency crashes in programs," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2015, pp. 450–461.

[17] B. Fang, P. Wu, Q. Guan, N. DeBardeleben, L. Monroe, S. Blanchard, Z. Chen, K. Pattabiraman, and M. Ripeanu, "Sdc is in the eye of the beholder: A survey and preliminary study," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*, June 2016, pp. 72–76.

[18] B. Fang, Q. Guan, N. Debardeleben, K. Pattabiraman, and M. Ripeanu, "Letgo: A lightweight continuous framework for hpc applications under failures," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '17. New York, NY, USA: ACM, 2017, pp. 117–130. [Online]. Available: http://doi.acm.org/10.1145/3078597.3078609

[19] B. Fang, H. Halawa, K. Pattabiraman, M. Ripeanu, and S. Krishnamoorthy, "Bonvoision: Leveraging spatial data smoothness for recovery from memory soft errors," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '19. New York, NY, USA: ACM, 2019, pp. 484–496. [Online]. Available: http://doi.acm.org/10.1145/3330345.3330388

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

[21] 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.

[22] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An architectural framework for software recovery of hardware faults," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 497–508. [Online]. Available: http://doi.acm.org/10.1145/1815961.1816026

[23] 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.

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

[25] V. C. Sharma, A. Haran, Z. Rakamaric, and G. Gopalakrishnan, "Towards formal approaches to system resilience," in *2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing*, Dec 2013, pp. 41–50.

[26] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "Ferrari: a flexible software-based fault and error injection system," *IEEE Transactions on Computers*, vol. 44, no. 2, pp. 248–260, Feb 1995.

[27] D. Li, J. S. Vetter, and W. Yu, "Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov 2012, pp. 1–11.

[28] U. Schiffel and C. Fetzer, "Hardware fault injection using dynamic binary instrumentation: Fitgrind," in *Proceedings Supplemental Volume of EDCC-6*, 01 2006.

[29] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "A systematic methodology for evaluating the error resilience of gpgpu applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 12, pp. 3397–3411, Dec. 2016. [Online]. Available: https://doi.org/10.1109/TPDS.2016.2517633

[30] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "Gpu-qin: A methodology for evaluating the error resilience of gpgpu applications," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2014, pp. 221–230.

[31] S. K. Vishal Chandra Sharma, Ganesh Gopalakrishnan, "Towards reseiliency evaluation of vector programs," in *21st IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems (DPDNS)*, 2016.

[32] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G. H. Leber, "Comparison of physical and software-implemented fault injection techniques," *IEEE Transactions on Computers*, vol. 52, no. 9, pp. 1115–1133, Sept 2003.

[33] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, April 1997.

[34] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," in *Proceedings of the 50th Annual Design Automation Conference*, ser. DAC '13. New York, NY, USA: ACM, 2013, pp.

101:1–101:10. [Online]. Available: http://doi.acm.org/10.1145/2463209.2488859

[35] C. Chen, G. Eisenhauer, M. Wolf, and S. Pande, "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*, ser. HPDC '18. New York, NY, USA: ACM, 2018, pp. 156–167. [Online]. Available: http://doi.acm.org/10.1145/3208040.3208043

[36] 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.

[37] F. A. Morrison, "Obtaining uncertainty measures on slope and intercept of a least squares fit with excel's linest," 2014, online. [Online]. Available: http://pages.mtu.edu/~fmorriso/cm3215/UncertaintySlopeInterceptOfLeastSquaresFit.pdf

[38] B. Fang, Q. Lu, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "epvf: An enhanced program vulnerability factor methodology for cross-layer resilience analysis," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2016, pp. 168–179.