Improving the Accuracy of IR-level Fault Injection

Lucas Palazzi, *Member, IEEE,* Guanpeng Li, *Member, IEEE,* Bo Fang, *Member, IEEE,*
and Karthik Pattabiraman, *Senior Member, IEEE*

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 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 corruption (SDC) probability estimation across different benchmarks and optimization levels. Further, we present a machine-learning-based technique for improving the accuracy of *crash* probability measurements made by IR-level FI, which takes advantage of an observed correlation between program crash probabilities and instructions that operate on memory address values. We find that the machine learning technique provides comparable accuracy for IR-level FI as assembly code level FI for program crashes.

Index Terms—Resilience, fault injection, IR-level fault injection, intermediate representation, machine learning, LLVM, PIN

1 Introduction

Hardware faults are becoming more common in commodity computer systems due to the effects of process scaling and manufacturing variations [1], [2], [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], [5], [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)[1] is the process of systematically introducing errors into the program and observing the outcome. Because the injection space is very large, many FI tools use the Monte Carlo method to sample the space of potential injection targets to obtain a statistical estimate of the techniques' coverage (other space reduction strategies are also possible).

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 often comes 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 higher levels is 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], [10], [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 x86-64 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.

In the conference version of this paper [12], we presented an analysis of the contradictory prior work [13], [14] that has examined the accuracy of IR-level FI with respect to assembly-level FI, showing that IR-level FI is in fact as accurate as assembly-level FI with respect to SDC probability measurements. We further supported this result by conducting a thorough comparison study between IR- and assembly-level FI, which we include in this paper (Section 4). We showed empirically that IR-level FI is as accurate

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

as assembly-level FI when measuring the SDC probability of a program; however, the accuracy of crash probability measurements depends on the amount of optimizations used to compile the program.

This paper then expands on these findings by discussing *memory address instructions*, which we define as assembly or IR instructions that operate on register values that are eventually used as the address operand in a memory load or store instruction. The dominant cause of application crashes is errors propagating to memory operations [15], [16], i.e., a load or store instruction attempting to read or write to a memory location that is "out of bounds". We find that the program crash probabilities measured by IR-level and assembly level FI are correlated with the program's memory address instruction percentage, supporting this intuition. Because back end optimizations typically affect an application's memory operations, a program's IR code can be different from its assembly code with respect to memory address instructions. We present a machine-learning-based technique that takes advantage of the correlations between crash probability measurements and memory address instruction percentages to improve the accuracy of crash probability measurements made using IR-level FI. The technique allows accurate crash probability estimates for a program using only IR-level FI experiments, offering both the benefits of IR-level FI and the accuracy of assembly-level FI.

Our contributions are as follows:

- By conducting an empirical study using rigorous statistical tests, we find that 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 4.2.1).
- We find that 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 4.2.2).
- Based on the observed correlation between the crash probabilities and the amounts of instructions executed that operate on *memory address* values, we present a machine-learning-based approach to obtain crash probability measurements that are as accurate as those when measured using assembly-level FI, while only requiring IR-level FI experiments to be conducted, thus improving the accuracy of IR-level FI crash probability measurements.

2 Background

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

2.1 Definitions

- **Fault injection (FI):** The process of systematically introducing faults/errors[2] into a program to observe program behaviour under fault/error conditions, e.g., to test a program's robustness and error-handling capabilities. Though FI has many uses, we use it to assess error coverage/resilience in this paper.
- **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 the program to throw an exception or execute an exit statement (e.g., due to an incorrect branch), and as a result the program terminates before completing its expected execution.
- **Hang:** A failure causes an infinite loop or a longer-than-usual execution that triggers termination based on a set timeout. *See Section 2.3 for how hangs are handled in our experiments.*
- **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 [13], [18], [19], [20], [21], [22], [23]).
- **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.

2.2 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 [24]. 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.

2.3 Fault Injection (FI) and Fault Model

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

[2] We use the Laprie et al. [17] terminology to distinguish faults from errors. Though we are injecting errors, we refer to the process as *fault injection* for historical reasons, unless otherwise specified.

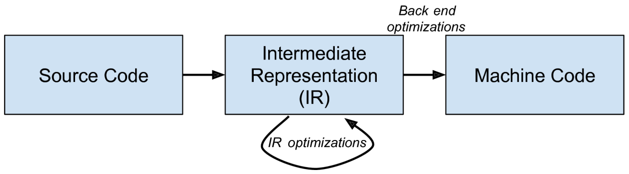


Fig. 1: LLVM/Clang code compilation flow

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, crash, or benign output. In this paper, we consider program *hangs* as part of the *crash* category; we set the execution timeout for each benchmark to be sufficiently long so as to allow the program to complete most longer-than-usual executions. As a result, we observe a negligible percentage of hang outcomes in our experiments (less than 0.5% of all fault injection outcomes). Once the FI runs have completed, we obtain a statistical estimate of the SDC/crash probabilities.

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 experiments, 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 [25]. Related work has made similar assumptions [9], [26], [27], [28], [29].

2.3.1 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 dynamic 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.

2.3.2 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.

2.3.3 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 [14], [30]. Common assembly-level fault injectors include BIFIT [31], PINFI [13], FITgrind [32] and others [33], [34]. 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.

2.3.4 IR-level FI

IR-level FI uses compiler techniques to inject errors into the compiler's intermediate representation (IR) code. Popular IR-level fault injectors are LLFI [8], KULFI [29], VULFI [35], 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 errors 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.

2.4 Machine Learning

In this study, we use *supervised* machine learning (ML) to make predictions on the data collected in our experiments. We provide some definitions of relevant terminology below.

Features: The inputs of the ML model. Each data example in the data set has a feature vector containing the values of each feature for that given data example.

Labels: The outputs of the ML model. When given a set of features, the model will give a prediction/estimate of the label for that set of feature values.

Training: The process of fitting a model to a set of data; this is the stage where the model "learns" the patterns in input-output relationships that exist in the training data set.

Training data set: A set of data examples that are used to *train* the ML model. In supervised machine learning, the training data is labeled with the expected outputs of each data example.

Test data set: A separate set of labeled data examples that are used to test how well the model performs on data it has not seen before. The test data set is never used to train the model.

3 Related Work

3.1 Fault Injection for Measuring Error Resilience

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.

The injection of *software* faults is a common use case for software-implemented fault injection, and has been done at the assembly/binary level in prior research [36], [37], [38], [39]. For example, Cotroneo et al. address the accurate mutation of binary code for injecting software errors when the source code of the target is unavailable [36]. While such papers address fault injections at the binary/assembly level, we are instead concerned with emulating transient *hardware* faults (i.e., soft errors), albeit at the software level.

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 [40]. 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 [41]. 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 [41]. For example, a paper by Cho et al. found that assembly-level FI can only capture a subset of system-level behaviour caused by soft errors [42]. However, our focus 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. 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 [43]. Finally, Li et al. used LLFI to estimate program SDC probabilities [11]. Studies such as these use IR-level FI under the implicit assumption that it is as accurate as assembly-level FI in measuring SDCs.

3.2 Comparison of IR-level and Assembly-level FI

Two prior papers directly compare the accuracy of IR-level FI with that of assembly-level FI [13], [14]. Wei et al. compare the accuracy of IR-level FI with that of assembly-level FI for emulating hardware errors, and find that *"LLFI is accurate for emulating hardware errors that cause Silent Data Corruption (SDCs), but not crashes"* [13]. Georgakoudis et al. also investigates the accuracy of IR-level FI with respect to assembly-level FI for emulating hardware errors, however they find that IR-level FI is significantly less accurate than assembly-level FI, claiming the inaccuracies are due to assembly-level dynamic binary instructions and back end compiler optimizations that are not available at the IR level [14].

Clearly, these two studies have come to contradictory conclusions, and 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. [13] claim that IR-level FI is accurate for emulating SDC-causing hardware errors, while Georgakoudis et al. claim otherwise [14]. This contradiction is especially peculiar considering both papers claim to use the same FI tools (i.e., LLFI and PINFI) and similar experimental setups.

In the conference version of this paper, published in [12], we perform an analysis of these two studies to find the root cause of the inconsistent findings. Our findings show that a modification made by Georgakoudis et al. [14] 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. [14] is inconsistent with the bit selection method used by LLFI, the paper's comparison between LLFI and PINFI is invalid. Thus, our prior work in [12] shows that IR-level FI is indeed as accurate as assembly-level FI, with respect to SDC errors.

3.3 Machine Learning for Fault Outcome Estimation

Numerous studies have been published at the intersection of program error resilience and machine learning. In this section, we focus on studies that use machine learning for evaluating program resilience and/or vulnerability.

Many studies use machine learning or statistical models to estimate the resilience and/or vulnerability of software components, identifying those with high-risk properties [44], [45], [46], [47], [48]. These studies are largely focused on identifying high-risk components within *specific* computing systems and do not directly apply to general computing applications. Further these approaches do not estimate resilience or crash probability within the context of IR-level fault injection as is the case in our work.

Farahani et al. propose a learning-based reliability prediction technique to estimate resilience to transient faults, using features at both the architecture and microarchitecture levels [49]. Our work differs from this in that we focus on predicting program-level tolerance to transient faults. A paper by Lu et al. uses machine-learning-based techniques to detect SDC-causing errors in programs [50]. However, this work uses machine learning to quantify the SDC proneness of *individual program variables* while our work attempts to evaluate the overall fault coverage of a program.

More recently, a paper by Kalra et al. [51] investigates the use of machine learning and statistical methods for predicting the resiliency of GPU applications. Their tool, PRISM, extracts features that characterize program resiliency allowing them to predict error outcomes without running fault injection campaigns. While similar to our work in this study, their work is limited to GPU applications and focused on predicting SDC outcomes while our work applies to the crash probabilities of CPU applications.

4 End-to-End Comparison: IR-level vs. Assembly-level FI

In this section, we conduct an extensive set of FI experiments to evaluate the accuracy of IR-level FI with respect to assembly-level FI. We first describe the experimental setup in terms of the benchmarks, FI tools, platforms, and measurement metrics used for our experiments. We then present our results for both SDC and crash outcomes, finishing the section with a discussion of our findings.

4.1 Experimental Setup

4.1.1 Overview

The experiments presented in this section are conducted according to the following process:

- We choose 25 benchmark programs on which to perform fault injection experiments using both IR-level (LLFI) and assembly-level (PINFI) FI respectively.
- Four separate sets of fault injections are performed for each benchmark, each one with the benchmark compiled using a different compiler optimization level (*-O0*, *-O1*, *-O2*, and *-O3*).
- We measure the SDC and crash probabilities for each set of FI experiment outcomes (i.e., each benchmark-optimization pair).
- We apply a variety of statistical tests and analyses to compare the results, and use these to draw conclusions on the accuracy of IR-level FI measurements compared to assembly-level FI.

4.1.2 Benchmarks

In our experiments, we choose a total of 25 different benchmarks from 7 publicly available benchmark suites. Their details are shown in Table 1. We choose these benchmarks because they are (1) from a broad selection of application domains, (2) open source and compatible with both fault injection tools, and (3) used in the two related FI studies [13], [14] discussed in Section 3.2. The benchmarks that were included in [13], [14] are indicated as such in Table 1, as well as whether the program is the CPU or memory-bound (if such information was available) [52], [53], [54], [55], [56].

We include all of the benchmarks used in Wei et al. [13], and all but three of the benchmarks used in Georgakoudis et al. [14]; *AMG2013*, *lulesh*, and *miniFE* are not used because they are either (1) not compatible with the platform used for our experiments, or (2) not compatible with LLFI when compiled using some of the pertinent optimization levels. In addition to those taken from [13], [14], we also include several benchmarks not used in the two prior studies.

TABLE 1: Benchmark Details (Section 4)

Benchmark	Suite	Input
blackscholes^{1}	PARSEC	`1 in_16K.txt output.txt`
fluidanimate^{1}	PARSEC	`1 10 in_5K.fluid out.fluid`
lud^{1}	Rodinia	`-v -i 512.dat`
backprop^{1}	Rodinia	`65536`
kmeans^{2}	Rodinia	`-i 819200.txt -k 1`
bfs	Rodinia	`1 graph1MW_6.txt`
bzip2^{2†}	SPEC	`-1kvv image.jpg`
libquantum^{2†}	SPEC	`33 5`
hmmer1†	SPEC	`--seed 10000000 ig.hmm`
mcf^{2†}	SPEC	`inp.in`
ocean^{1†}	SPLASH-2	`-p1 -o`
raytrace^{1†}	SPLASH-2	`-p1 -m64 inputs/car.env`
CoMD‡	Mantevo	`-x 10 -y 10 -z 10 -N 50`
HPCCG‡	Mantevo	`64 64 64`
XSBench^{1‡}	CESAR	`-s small`
BT^{1‡}	NPB	`S`
CG^{2‡}	NPB	`S`
DC^{2‡}	NPB	`10000000 ADC.par`
EP^{1‡}	NPB	`W`
FT^{2‡}	NPB	`W`
IS^{2}	NPB	`S`
LU^{2‡}	NPB	`W`
MG^{2}	NPB	`S`
SP^{1‡}	NPB	`W`
UA^{2‡}	NPB	`W`

†Benchmark used in Wei et al. [13]
‡Benchmark used in Georgakoudis et al. [14]
^{1}CPU bound program
^{2}Memory bound program

We use the default inputs included in the benchmark suites, or example inputs where the former are not available. Finally, each benchmark is compiled four times, each one using a different compiler optimization level (*-O0*, *-O1*, *-O2*, and *-O3*); separate FI experiments are conducted for each of the four optimization levels.

4.1.3 Fault injection tools and platform

To conduct our FI experiments, we again use LLFI[3] and PINFI[4] as our IR-level and assembly-level FI tools, respectively. This is to be consistent with the studies discussed in Section 3.2. In addition, these FI tools are (1) *flexible*: LLFI and PINFI are fully open source and configurable to conduct FI experiments with customized setups, and (2) *popular*: both tools have wide adoption in both industry and academia.

All experiments are conducted on 64-bit Intel x86-64 machines, and the benchmarks are all executed on single threads. LLVM/Clang 3.4 is used to compile from the benchmarks' C/C++ source code to their respective LLVM IR (.ll) files and executables. PINFI uses Intel PIN 3.5 to access and instrument the compiled machine code of the benchmarks.

For each set of FI experiments (i.e., each benchmark at each optimization level), we randomly perform 1,000 fault injection runs using both LLFI and PINFI respectively. Thus, we perform a total of $100,000$ fault injection runs (= 25 * 4 * 1000) for each tool.

4.1.4 Measurements of accuracy

To evaluate the accuracy of the SDC and crash probabilities measured using IR-level and assembly-level FI, we apply

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

a multitude of visual and statistical tests. We first show a graphical overview of the results for each benchmark to visually compare the outcomes of FI execution for each FI tool, with calculated errors bars representing a 95% confidence interval for 1,000 FI runs. We then apply three different types of statistical analyses 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. We describe the details of these tests below.

Least squares regression analysis: The first statistical analysis we apply is based on a least squares regression model [57]. 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 approach is to measure the *linear* relationship between the respective SDC and crash probabilities obtained using LLFI and those using PINFI. The model plots the PINFI probabilities (y-axis) against those of LLFI (x-axis). We have verified that our datasets meet all the assumptions needed for applying a linear regression model i.e., (1) normality of residuals (verified manually), (2) little to no auto-correlation (using Durbin-Watson test), and (3) homoscedasticity (visual test).

In the ideal situation where LLFI produces the exact same measurements as PINFI (i.e., LLFI is exactly as accurate 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. Note that R^2 values close to 1 indicate a high correlation. We estimate the slope and y-intercept parameters with a 95% confidence interval.

Paired sample *t*-test: We use a paired sample *t*-test to compare the SDC and crash probability measurements made by LLFI and PINFI. The paired sample *t*-test is used to determine whether the mean difference between two sets of measurements is zero. We verified that our dataset meets all the *t*-test assumptions.

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 sample *t*-test for each optimization level using the whole set of 25 benchmarks, so that we can compare the significance of the results at each optimization. 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% 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.*

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 same application protected with different fault tolerance techniques). Therefore, a fault injection technique needs to be sensitive to the *relative* rankings of the SDC probabilities.

To examine the sensitivity of the measurements made using 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 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 non-normal variations.

4.1.5 Research Questions

In this study, we are interested in comparing the accuracies of IR-level FI and assembly-level FI, and further measuring how this accuracy differs when different compiler optimizations are applied. We focus on measuring both SDC and crash probabilities. By injecting faults into a set of benchmarks compiled with different optimization levels, we get a complete analysis of the accuracies of IR-level and assembly-level FI, and can thereby determine if any inaccuracies in the SDC or crash measurements can be attributed to the applied compiler optimizations.

We therefore ask the following two research questions:

RQ1 Does IR-level FI provide significantly different measurements of program SDC probability as assembly-level FI, across compiler optimizations?

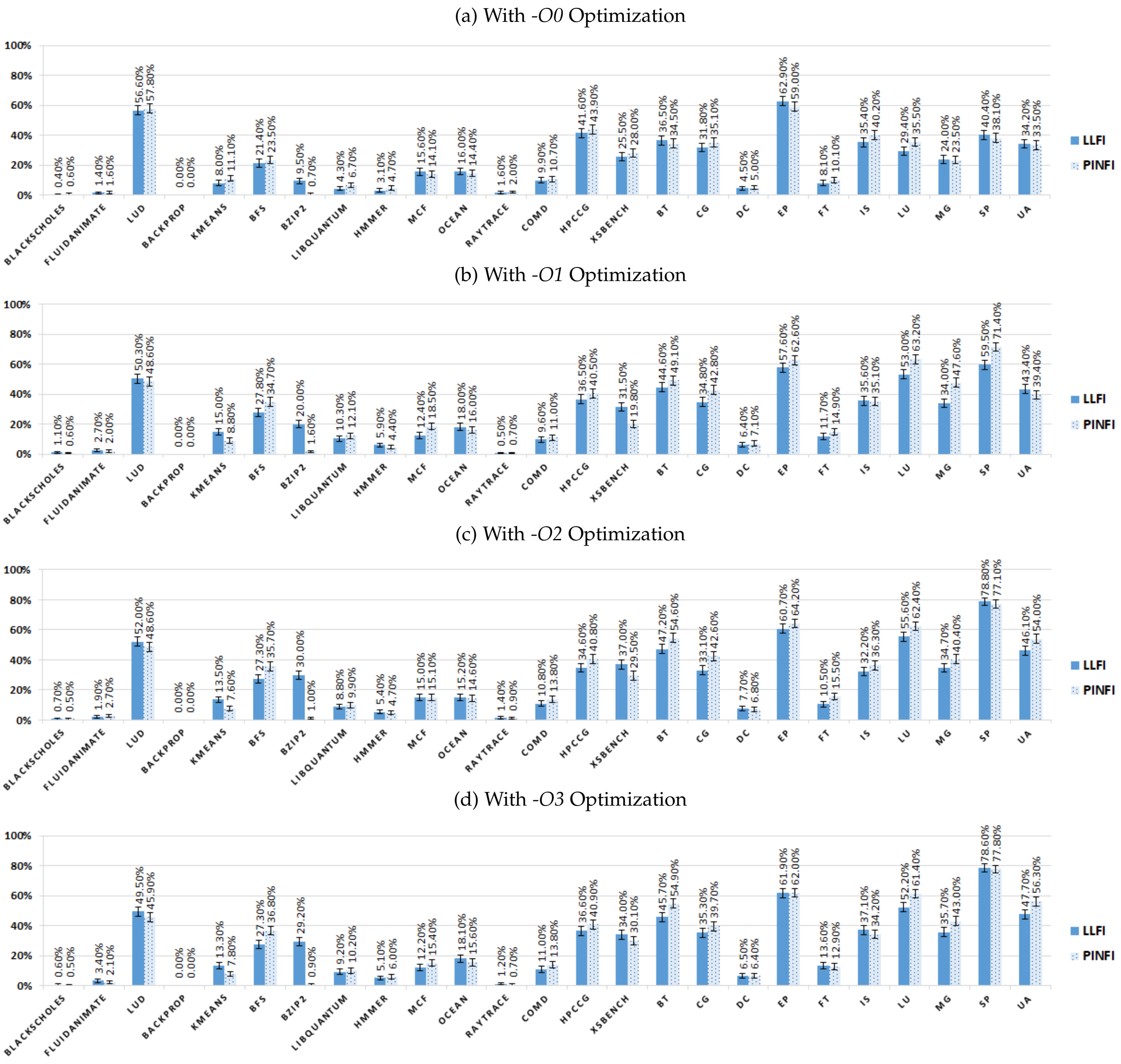
RQ2 Does IR-level FI provide significantly different measurements of program crash probability as assembly-level FI, across compiler optimizations?

4.2 Results and Findings

In this section, we present our experimental results based on fault injection experiments conducted on the 25 benchmarks listed in Table 1, separating the results based on optimization level.

4.2.1 Program SDC probabilities

Figure 2 shows the SDC probabilities obtained using LLFI and PINFI for each benchmark. We present the numerical results using bar graphs with error bars (95% confidence interval for 1,000 runs) for visual comparison. The least squares regression analysis results are shown in Table 2,



(a) With *-O0* Optimization

(b) With *-O1* Optimization

(c) With *-O2* Optimization

(d) With *-O3* Optimization

Fig. 2: Program SDC probabilities measured by LLFI and PINFI

TABLE 2: Least Squares Regression Analysis Results

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

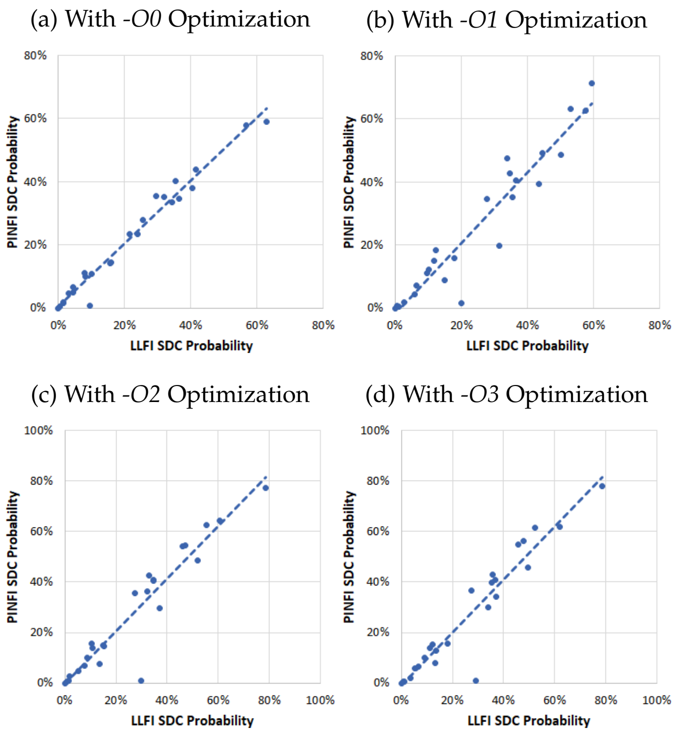
TABLE 3: Statistical Test Results

		-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 4.1.4)
[‡] Measured using Spearman's rank test (Section 4.1.4)

while the t-test and Spearman's rank test results are shown in Table 3.

Visual comparison: Figure 2 shows that the SDC probabilities measured by PINFI and LLFI are close to each other, 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



(a) With *-O0* Optimization

(b) With *-O1* Optimization

(c) With *-O2* Optimization

(d) With *-O3* Optimization

Fig. 3: Program SDC probabilities measured by PINFI plotted against those measured by LLFI 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.

-O3, respectively, indicating that for most benchmarks, the SDC probabilities measured using LLFI are almost indistinguishable from those measured by PINFI.

Least squares regression analysis: The results from the least squares linear regression analysis (shown in Table 2) 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, respectively, at each optimization level. The R^2 values are also high, with three out of four values measuring above 0.9. This shows that the data fits very closely with the line given by the slope and y-intercept values. This regression analysis is visually illustrated in Figure 3 using the plotted data points; we can see here 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.

Paired sample *t*-test: Table 3 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 tools are significantly different from each other.

Spearkman's rank correlation test: Finally, the results from the Spearman's rank correlation test (Table 3) 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.

Conclusion: Based on the above analyses, we address RQ1 by concluding that there is no evidence to suggest that LLFI and PINFI give significantly different measurements of the SDC probability of a program.

4.2.2 *Program crash probabilities*

We now conduct the same analysis on the crash probability measurements as we did for SDCs. Figure 4 shows the crash probabilities obtained using LLFI and PINFI for each benchmark. As in SDCs, the least squares regression analysis is shown in Table 2, and the t-test and Spearman's rank test results are shown in Table 3.

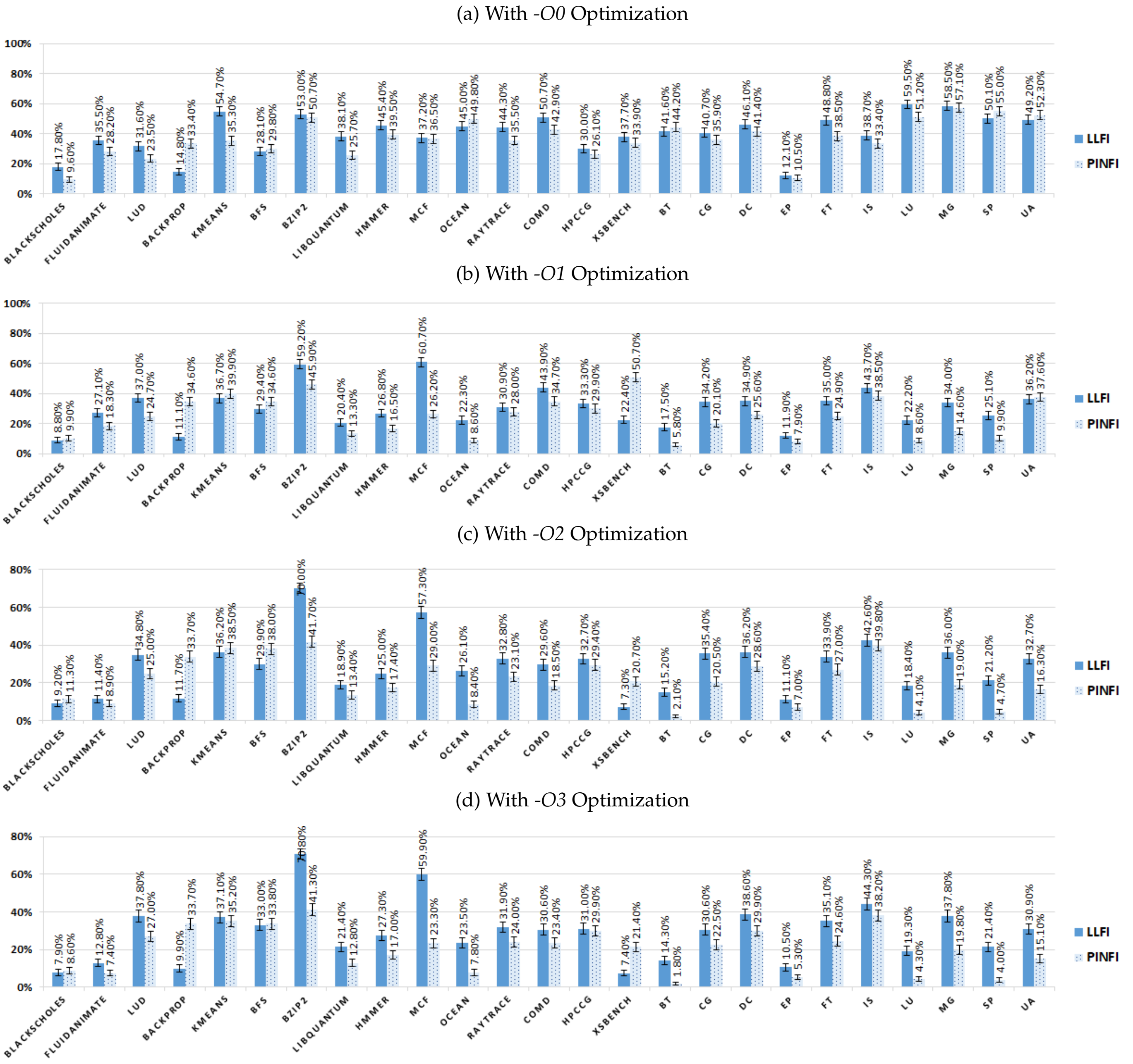
Visual comparison: Figure 4 shows that unlike SDC probabilities, the crash probabilities do not consistently match between LLFI and PINFI for all optimizations. Further examination reveals that at *-O0* the crash probabilities tend to be similar with overlapping error bars for most of the benchmarks, 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 the lowest optimization level *-O0*, but not at the other optimizations *-O1* to *-O3*.

Least squares regression analysis: The results from the least squares linear regression analysis (Table 2) 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 the optimization level increases, the less accurate the crash probabilities measured by LLFI become compared to PINFI.

Paired sample *t*-test: Table 3 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 using LLFI and PINFI.

Spearman's rank correlation test: As shown in Table 3, the results from the Spearman's rank test indicate a moderate-to-strong monotonic relationship between the crash probabilities measured using LLFI and 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*.

Conclusion: Based on the above analysis, for RQ2 we conclude that the crash probabilities do not consistently



(a) With *-O0* Optimization

(b) With *-O1* Optimization

(c) With *-O2* Optimization

(d) With *-O3* Optimization

Fig. 4: Program crash probabilities measured by LLFI and PINFI

match between LLFI and PINFI for all optimization levels. Further, the accuracy of the crash probability measurements is influenced by the compiler optimizations applied to the program, especially going from *-O0* to *-O1*.

4.3 Discussion

When a program is compiled using a specified optimization level, optimization passes are applied to the code at both the IR level and in the compiler back end. As a result, the compiled IR of a program only has *some* of the optimizations applied, while the corresponding executable of the program will have the rest of the optimizations as well (those applied in the compiler back end). While IR-level optimization passes typically apply platform-independent code transformations that affect the data flow of the program, back end optimizations often target platform-specific transformations such as those involving register allocation and memory operations.

We observed that the SDC probabilities are measured accurately by IR-level FI when compared with assembly-level FI. This observation makes sense, considering SDCs (i.e., incorrect outputs) can be mostly attributed to errors in a program's data flow, which is relatively unaffected by back end optimizations. On the other hand, we found that the accuracy of crash probability measurements noticeably suffers when more optimizations are applied to the program. A program's IR and assembly code share common front and middle end compilations, but have differing back end compilations due to optimizations not visible at the IR level. Thus, the accuracy of crash probability measurements seem to be affected by back end optimizations (Section 5).

5 Effects of Memory Address Instructions

Section 4 presents us with some useful insights into the accuracy of IR-level FI with respect to assembly-level FI. We observed that while IR-level FI can provide SDC probability measurements as accurately as assembly-level FI even with all of the standard optimizations applied, it does not provide the same level of accuracy in its crash probability measurements when those optimizations are applied.

In this section, we expand on our insights as to why this might be the case by introducing the concept of *memory address instructions*. We then illustrate how the patterns observed in the crash probability measurements in Section 4 are correlated with a program's percentage of memory address instructions. This section provides the basis for our machine-learning-based prediction technique in Section 6.

5.1 Overview of Memory Address Instructions

We begin this section with an introduction to what we call *memory address instructions*. In short, a memory address instruction is an instruction whose output (i.e., the value in its destination register) is a memory address value that is eventually used by a load or store instruction that reads or writes to memory.

A major cause for crashes is *segmentation faults* [15], [16], which occur when a memory load or store instruction tries to read from or write to an "illegal" memory location. This can happen when an error propagates to the memory instruction operand, such as the address of load or store instructions. The fault that may cause such an error can occur in the memory instruction itself, or any previous instruction that operates on the stored value of the memory address used.

Many back end optimizations operate on a program's interactions with memory; for example, a register allocation optimization pass (typically applied in the compiler back end) determines how the compiler assigns program variables to the target architecture's limited number of registers, and when it is necessary to spill the excess data to memory. Thus, it is likely that the inaccuracies in crash probability measurements at higher optimization levels are correlated with how the optimizations influence the amount of these memory address instructions.

To illustrate this concept, consider the x86-64 assembly code segment in Figure 5, which is taken from the *backprop* benchmark compiled at *-O3* optimization. The instructions in lines (1) and (2) add `0x4` to the values in registers `rsi` and `rdx`, respectively. Several instructions later, in lines (8) and (9), the values in each of the respective registers are used as memory addresses for load operations. We therefore consider the instructions in lines (1) and (2) to be memory address instructions; if an error were injected into the destination register of one of these instructions, the error will propagate to the address of a memory instruction. Note that we only consider errors that occur in an instruction's destination register, as per our fault model. Thus, instructions (8) and (9) are *not* considered memory address instructions as an injection into one of these instructions would not affect any memory address values.

```
1:    add      $0x4,%rsi
2:    add      $0x4,%rdx
3:    vxorps   %xmm0,%xmm0,%xmm0
4:    vmovsd   0x1350(%rip),%xmm1
5:    vmovss   0x1368(%rip),%xmm3
6:    vxorps   %xmm2,%xmm2,%xmm2
7:    nopl     0x0(%rax)
8:    vmovss   (%rdx),%xmm4
9:    vmovss   (%rsi),%xmm5
```

Fig. 5: x86-64 assembly code segment from the *backprop* benchmark; instructions `1` and `2` are considered *memory address instructions*

5.2 Measuring Memory Address Instruction Percentages

We develop a tool that, given a set of programs compiled to both IR and assembly, measures the amount of memory address instructions executed dynamically at both the IR and assembly level. Our tool profiles memory address instructions from the dynamic execution of the program, but only counts instructions whose return values are used as addresses in subsequent load or store instructions *within the same basic block as the instruction in question*.

This is because tracing memory dependencies across basic blocks through the entire execution of the program incurs very high overhead, and for large programs (such as the types of applications typically tested using fault injection) the time required to perform this analysis would be much too high. That being said, while limiting the profiling to only memory dependencies within basic blocks exclude instructions that propagate memory dependencies between basic blocks, it does capture most of the crash-causing errors; most crashes occur within the same basic block as the faulty instruction [20], [58].

We use the standard definition of a basic block: a set of consecutive instructions with no branches/jumps in or out of the middle of the block. At the IR level, identifying the boundaries of the basic blocks is trivial as LLVM IR code is already structured in basic blocks [24]. For assembly code, we determine this programmatically as part of our profiling tool; we parse the compiled assembly code and identify the boundaries of the blocks based on branch/jump instructions. Note that there may be some corner cases that are not covered by our tool at the assembly code level (e.g., indirect branches that jump to the middle of a basic block). However, these cases are rare in practice, and can be addressed by using a more sophisticated disassembler [59].

Our tool follows the following process at both the IR and assembly levels:

1) We first obtain a sampling of the instructions executed. Sampling, as opposed to using the entire set of executed instructions, provides us with a low-overhead method of obtaining a statistical estimate of the frequency of instruction execution. For example, sampling took at most a few hours on each of the benchmarks, while analyzing the entire set of instructions took several days even for our smallest benchmark. Furthemore, the sampling methodology is consistent with that used for selecting fault injection targets.
   - **IR-level**: We use the FI logs produced by LLFI to pro-

vide us with a sampling of the dynamically executed instructions.

- **Assembly-level**: We use a PIN-based instruction sampling tool to sample instructions as they are executed dynamically. Further, our tool borrows code from PINFI to limit the selection of the instruction sampling to only those instructions that are relevant to the chosen fault model (i.e., the chosen PINFI configuration). This is to keep our tool's analysis consistent with the IR-level analysis and with the fault model in question.

2) Using the static IR and disassembled x86-64 assembly code respectively, we parse the sets of static instructions and record which instructions are memory address instructions with respect to their basic blocks.
3) We obtain the total amounts of memory address instructions executed dynamically by counting the frequency of each recorded instruction in the sampling.

We peform this analysis across the set of benchmarks and optimization levels, obtaining the amounts of both IR- and assembly-level memory address instructions as percentages of the total number of instructions sampled. For each benchmark and optimization level, we show the corresponding percentages measured by our tool in Table 4.

5.3 Correlation with Crash Probabilities

Table 4 shows the measured percentages of memory address instructions for each benchmark, at both the IR and assembly level. At *-O0*, we observe that at the IR and assembly level the values change little compared to the other optimization levels; however, at *-O0* the average ratio between the two values ($x86\text{-}64/IR$) is 0.7914 compared to 0.3479, 0.2806, 0.2573 at *-O1*, *-O2*, and *-O3* respectively. In Section 4.2.2, we observe a similar pattern in the measured crash probabilities. Thus, in this section we investigate how these ratios correlate with those of the crash probability measurements.

To investigate this, we first calculate (1) the ratio of crash probabilities measured by PINFI over that of LLFI, and (2) the ratio of percentage of memory address instructions for assembly code over that of IR. In effect, these ratios are the "gain" applied by the compiler back end optimizations for each respective metric, i.e., the factor by which the compiler back end increases (or decreases) either the crash probability or the percentage of memory-depenent instructions when back end optimizations are applied.

Next, we ask the question: *is the gain in the memory address instruction percentage correlated with the gain in the crash probability measurement?* To answer this question, we calculate the correlation coefficient of the set of ratios at each optimization level. A strong correlation (i.e., close to ± 1) would indicate that any change we see from IR to assembly in the measured crash probability is correlated with the change in percentage of memory address instructions.

We find that at *-O0* the correlation coefficient is -0.01296 while at *-O1*, *-O2*, and *-O3* the coefficients are 0.41205, 0.71066, and 0.64522 respectively. A correlation coefficient above 0.5 typically indicates a moderately strong correlation. The dramatic change in the correlation from *-O0* to the higher levels of optimization suggests that the inaccuracies in the crash rate can be explained by the relative percentages of memory address instructions at *-O1* to *-O3*, but not at *-O0* ()as there is no optimization applied to the program).

6 Crash Probability Estimation Using ML

In Section 4, we observed that IR-level FI is not as accurate as assembly-level FI for measuring the crash probability of programs due to transient hardware faults. Given the significant advantages of IR-level FI over assembly-level FI highlighted in Section 2.3, e.g., easier analysis with respect to the source code and portability, a more accurate measurement using IR-level FI would prove useful.

In Section 5, we found that when optimizations are applied, there is a correlation between the change in the amount of *memory address instructions*, and the change in measured crash probabilities from a program's IR to assembly level code. In other words, there appears to be a correlation between how compiler optimizations affect measured crash probabilities (at the respective levels), and how they affect the percentages of memory address instructions. Further, it is clear from Table 4 that the application of back end optimizations (i.e., not at *-O0*) have a much more significant effect on memory address instructions than when no optimization (i.e., *-O0*) is applied. This is in line with our intuition since we have observed in Section 4 that crash probability measurements are also affected by back end optimizations.

These are useful insights as measuring a program's percentage of memory address instructions at both the IR-level and assembly-level is fairly straightforward. Even more importantly however, is that the process is much quicker than performing thousands of fault injections in order to obtain both the IR- and assembly-level crash probabilities. Therefore, it is worthwhile to pursue the implementation of a technique that uses memory address instruction percentages to supplement IR-level fault injection experiments to provide a program's crash probability estimations that are closer to that of assembly-level FI (without performing any assembly-level FI experiments).

In this section, we present a ML-based crash probability prediction technique that effectively "closes the gap" between IR- and assembly-level FI with respect to crash-causing errors. We evaluate several different ML algorithms on the data gathered in prior sections (i.e., FI experiment results and memory address instruction percentages) and compare the resulting estimations against the original IR-level crash probability measurements.

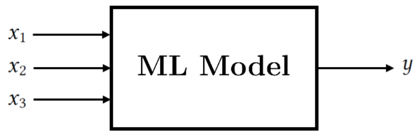
6.1 Experimental Setup

We evaluate a variety of supervised ML models to determine which would best fit the data. The types of ML models we evaluate are limited to *regression*, as we want to map our inputs to a continuous-valued output. Our goal is to find a model, or a handful of models, that can predict a given program's crash probability as accurately as with assembly-level FI within a reasonable degree of confidence.

Our desired model takes three inputs, with the output being the crash probability measured using assembly-level FI experiments as outlined in Figure 6. In practice, such

TABLE 4: Percentages of Memory Address Instructions

	Memory Address Instructions (% of total instructions)							
	-O0		*-O1*		*-O2*		*-O3*	
Benchmark	IR	x86-64	IR	x86-64	IR	x86-64	IR	x86-64
backprop	54.2 %	41.3 %	32.8 %	16.4 %	28.5 %	14.5 %	25.5 %	13.6 %
bfs	45.2 %	25.8 %	20.0 %	7.7%	18.9 %	3.3%	18.4 %	3.0%
blackscholes	16.0 %	9.9%	9.9%	5.6%	10.9 %	6.8%	10.0 %	5.0%
BT	68.2 %	67.9 %	18.5 %	0.8%	22.2 %	0.0%	20.0 %	0.1%
bzip2	33.6 %	11.1 %	30.7 %	10.7 %	29.5 %	2.9%	30.0 %	2.7%
CG	59.3 %	0.6%	34.3 %	0.1%	33.8 %	0.7%	30.4 %	0.7%
comd	61.6 %	45.4 %	28.2 %	13.6 %	26.9 %	8.4%	28.4 %	13.9 %
DC	51.4 %	35.9 %	21.0 %	1.4%	19.5 %	2.5%	23.7 %	2.1%
EP	16.4 %	12.5 %	8.4%	0.0%	9.9%	0.0%	7.0%	0.0%
fluidanimate	42.2 %	15.2 %	32.7 %	1.8%	11.3 %	5.1%	11.8 %	3.0%
FT	29.2 %	64.9 %	20.4 %	4.9%	21.5 %	6.2%	18.7 %	3.4%
hmmer	68.4 %	45.8 %	29.6 %	14.0 %	28.3 %	13.3 %	31.5 %	2.9%
hpccg	54.4 %	37.3 %	34.8 %	0.9%	33.6 %	2.1%	35.2 %	0.8%
IS	38.2 %	33.6 %	39.0 %	31.6 %	39.2 %	30.1 %	40.7 %	38.6 %
kmeans	53.6 %	36.3 %	16.2 %	0.9%	16.6 %	1.9%	18.6 %	2.8%
libquantum	47.1 %	32.2 %	9.8%	0.9%	10.7 %	0.7%	11.6 %	0.4%
LU	71.6 %	86.2 %	24.0 %	1.6%	22.8 %	0.8%	23.3 %	1.2%
lud	58.0 %	28.7 %	48.8 %	0.3%	44.7 %	0.3%	46.8 %	0.0%
mcf	41.7 %	39.2 %	33.9 %	57.7 %	31.7 %	16.9 %	34.1 %	14.6 %
MG	80.8 %	68.8 %	24.1 %	5.7%	36.3 %	5.2%	36.1 %	5.3%
ocean	67.0 %	74.6 %	26.9 %	0.0%	29.6 %	0.1%	30.7 %	0.2%
raytrace	46.6 %	27.5 %	27.4 %	3.3%	30.6 %	4.4%	31.5 %	4.4%
SP	76.6 %	75.0 %	24.7 %	1.8%	27.3 %	0.0%	28.0 %	0.3%
UA	72.1 %	65.6 %	25.8 %	9.5%	40.0 %	7.3%	38.0 %	5.7%
xsbench	41.9 %	43.5 %	31.1 %	61.4 %	30.9 %	58.1 %	30.5 %	58.5 %



x_1 = estimate of crash probability as measured by IR-level FI

x_2 = percentage of memory address instructions in IR code

x_3 = percentage of memory address instructions in assembly code

y = estimate of crash probability as measured by assembly-level FI

Fig. 6: Inputs (features) and output (label) of our ML-based predictor

a predictor could be integrated in a IR-level FI tool (such as LLFI), allowing users to measure a program's crash probability at the IR level with comparable accuracy to that of assembly-level FI, thus closing the gap between IR-level and assembly-level FI as far as crashes are concerned.

As this is a relatively small data set, we are not concerned with the computational cost to train the models. We evaluate many different models, each with different sets of available hyper-parameters, to find the 'best' model. Table 5 provides an overview of the models selected for training in our experiments.

Our data set has a total of 100 examples (25 benchmarks × 4 optimization levels) that we split randomly into a training set of size 80 and a test set of size 20 (i.e., an 80-20 split). Each data sample has three features and one label (Figure 6). We use 5-fold cross-validation to tune the hyper-parameters for each model. We use mean squared error (MSE) as the score function, and thus we choose the set of hyper-parameters that result in the best MSE from cross-validation. We then compare the MSE of each of the models based on the test set predictions, after re-training them on the entire training set.

TABLE 5: Overview of Machine Leaning Models

Model type	Description
Linear regression	Ordinary least squares linear regression
Huber regression	Linear regression model that is robust to outliers
Lasso regression	Linear model with L1 regularization
Ridge regression	Linear model with L2 regularization
Support vector regression	Support vector machine algorithm for regression
Kernel ridge regression	combines ridge regression with the kernel trick
Bayesian ridge regression	ridge regression model defined in probabilistic terms, with explicit priors on the parameters
Random forest regression	meta estimator that fits a number of decision trees on various sub-samples of the dataset
Multi-layer perceptron	Artificial neural network

6.2 Results

Table 6 shows an overview of the results. For each model, we report the MSE of the predictions on the training set, the coefficient of determination (R^2) between the predictions and true labels of the test set, and the MSE of the predictions on the test set. The table is sorted in order of test set error (MSE); the model with the lowest test set error (i.e., the best performing model) is a **random forest regression** model, and is listed first in the table. In the absence of a ML model, "current practice" would be to simply use the crash probability obtained using IR-level FI as the estimate of the crash probability. Thus, we also compare the MSE values obtained using our ML predictions against those from a "current practice" model that predicts $\hat{y} = x_1$, shown in Table 6.

In addition to reporting the MSE and R^2 values for each trained model, we provide the predicted crash probabilities and perform the same statistical analyses on the predictions, as was done in Section 4, for the "best" model (i.e., the random forest regression model). The crash probability predictions are plotted for each benchmark and optimization level in Figure 8, alongside the measurements made by PINFI for comparison.

The least squares regression analysis results are shown in Table 7, and the t-test p-values and Spearman's rank correlation coefficients are shown in Table 8. Plots of the measurements with least squares regression lines are shown in Figure 7.

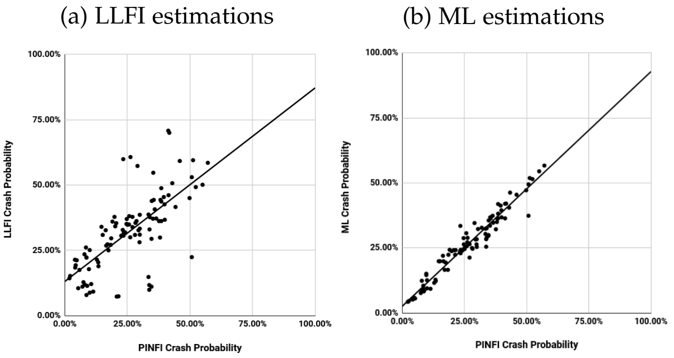
TABLE 6: ML Model Results

	Training MSE	Test MSE	Test R^2
Random Forest Model[1]	0.000876	0.001824	0.8754
Neural Net Model	0.001522	0.003803	0.7597
SVR Model	0.009554	0.007247	0.5421
Lasso Model	0.008206	0.007309	0.5381
Bayesian Ridge Model	0.008168	0.007404	0.5322
Huber Model	0.008468	0.007486	0.5270
Ridge Model	0.008153	0.007644	0.5170
LR Model	0.008152	0.007682	0.5146
Kernel Model	0.002118	0.022943	-0.4497
Current practice[2]	0.01595	0.01651	0.41144

Models are sorted in order of increasing test set MSE (i.e., the best performing models are listed first)

[1] Model with lowest test MSE

[2] Using model that simply predicts $\hat{y} = x_1$


(a) LLFI estimations

(b) ML estimations

Fig. 7: Program Crash probabilities measured by PINFI plotted against those measured by LLFI (a) and those predicted using a random forest regression model (b), with least squares regression line. The ideal line of best fit is a line with slope of 1 and y-intercept of 0.

TABLE 7: Least Squares Regression Analysis Comparison (LLFI vs. ML Crash Probability Estimates)

	slope, m	y-intercept, b	R^2
LLFI estimates	0.7405 ± 0.1536	0.1313 ± 0.0447	0.4868
ML estimates	0.9022 ± 0.0442	0.0265 ± 0.0129	0.9446

TABLE 8: Statistical Test Result Comparison (LLFI vs. ML Crash Probability Estimates)

	LLFI estimates	ML estimates
p-value†	6.036e−8	0.70127
Correlation coeff.‡	0.73321	0.97171

†Measured using paired sample t-test (Section 4.1.4)

‡Measured using Spearman's rank test (Section 4.1.4)

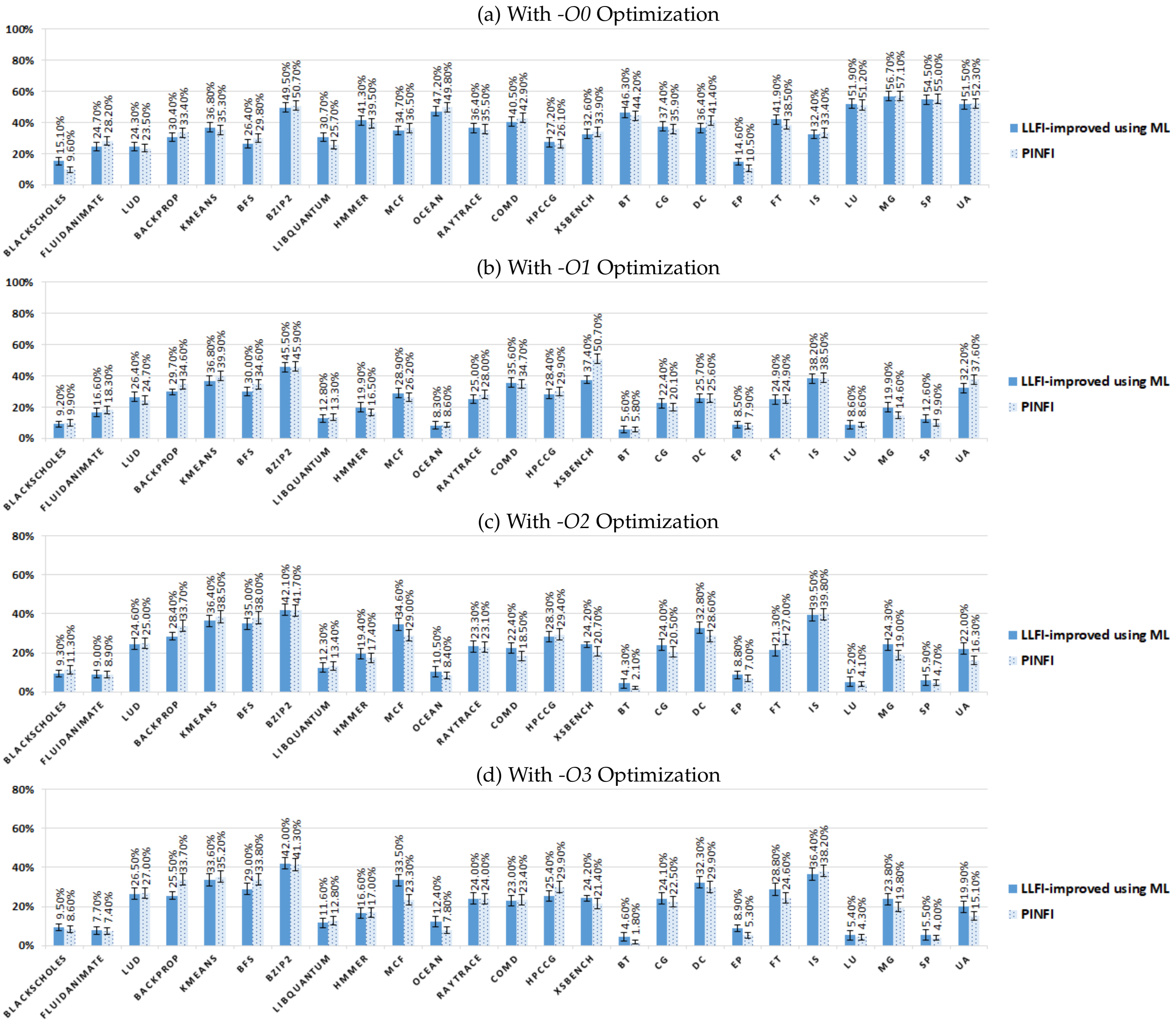
We find that for all models apart from the kernel model, both the training and test MSE of the supervised ML models are much lower than that of IR-level FI (i.e., current practice). The coefficient of determination R^2 (taken between the actual and predicted labels of the test set) ranges from 0.4497 to 0.8754, indicating a wide range of explanatory power depending on the model; a R^2 close to 1 indicates a strong explanation of variability in the model, while a value close to 0 indicates a weak explanation.

A better indicator of a good ML model than R^2 is its test error. The model that results in the best test error is a *random forest regression model with 16 decision trees and 3 maximum features per tree*. This model offers a test MSE that is much smaller than most other models at 0.001824, and over 9 times smaller than IR-level FI on its own (this is roughly an order of magnitude better). Random forest regression is an algorithm that produces a number of decision trees on random subsets of the dataset, a very different process from the other models that are mostly different versions of linear regression models. Our results indicate that a decision tree-based model is best suited to making predictions on this type of dataset, with a multi-layer perceptron model (i.e., a neural network) coming a close second.

Next, we compare the predictions made using the random forest regression model with the LLFI crash probability measurements using the least squares regression analysis and paired sample t-test (Section 4). We notice a sizeable improvement in the least squares regression analysis from LLFI to the ML predictions (Figure 7). With respect to the paired sample t-test, the p-value for the LLFI measurements is notably smaller than the cutoff of 0.05 while the p-value for the ML predictions is much higher at 0.70127. This tells us that while we can reject the null hypothesis for LLFI crash probability measurements (i.e., LLFI and PINFI crash probability measurements are significantly different), we cannot reject the null hypothesis for the ML predictions (i.e., the ML predictions are not significantly different from PINFI crash probability measurements).

The Spearman's rank correlation test gives us similar findings: the ranked correlation coefficient for the ML predictions is much closer to 1 than that of the LLFI measurements. In fact, we see that the correlation coefficient for the ML predictions is very nearly 1 with a value of 0.97171. We therefore conclude that the predictions made using the random forest regression model are extremely sensitive to the relative rankings of program crash probabilities.

Based on these results, we conclude that supervised ML is a reasonable approach to estimating a more accurate program crash probability measurement with only IR-level FI experiments. ML is able to take advantage of the correlations between the percentage of memory address instructions in a program and its crash probability, giving us a more accurate estimate of the latter using only IR-level FI. This allows


(a) With *-O0* Optimization

(b) With *-O1* Optimization

(c) With *-O2* Optimization

(d) With *-O3* Optimization

Fig. 8: Program crash probabilities estimated by LLFI improved with the random-forest regression model vs. PINFI

practitioners to reap the benefits of performing FI at the IR level while still obtaining comparable measurements of a program's crash probability as assembly-level FI.

7 Conclusion and Future Work

7.1 Conclusion

As random bit-flips caused by transient hardware errors are becoming more common, researchers and software designers are relying more on fault injection (FI) to evaluate the resilience of software techniques in vulnerable systems. In this study, we investigated the accuracy of IR-level FI, i.e., fault injection that is performed at the IR level of a program. IR-level FI has benefits over assembly-level FI, including easier portability and mapping to the original source code.

In this paper, we first performed 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 compared the results of FI performed at the LLVM IR level with those at the x86-64 assembly level, as these are the dominant platforms used by prior work in this area. Our findings showed that IR-level FI provides measurements of program SDC probabilities that are as accurate as those measured using assembly-level FI, however the same is not true for crash probability measurements.

Further, based on an observed correlation between measured crash probabilities and the amount of executed instructions that operate on memory address values, we proposed a machine-learning-based approach to improving the accuracy of crash probability measurements using IR-level FI. We trained a variety of different algorithms, and find that the "best" model (i.e., the one with the best test set error) resulted in a test set error that is over 9 times smaller than that of the raw IR-level FI measurements.

7.2 Future Work

There are two potential directions for future work.

1. Extending the comparison to other platforms: We focused on evaluating the accuracy of LLVM IR-level FI with respect to x86-64 assembly-level FI. While these two platforms are popular and are commonly used in both research and industry, it would be useful to conduct the same comparisons on other common platforms.

2. Improving the machine learning study Our machine learning study can be improved by (1) adding additional features to improve prediction capabilities, (2) implementing feature selection techniques to identify the most and least important features, and (3) improving the size and quality of the training data set.

All the data and tools in this paper are available at: *https://github.com/DependableSystemsLab/ISSRE19*
We have also released the crash rate estimator tool at: *https://github.com/DependableSystemsLab/LLFI-CrashRateEstimator*

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 TDSC for their insightful comments.

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.

[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] L. Palazzi, G. Li, B. Fang, and K. Pattabiraman, "A tale of two injectors: End-to-end comparison of IR-level and assembly-level fault injection," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, Oct 2019.

[13] 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.

[14] 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

[15] 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.

[16] Weining Gu, Z. Kalbarczyk, and R. K. Iyer, "Error sensitivity of the linux kernel executing on powerpc g4 and pentium 4 processors," in *International Conference on Dependable Systems and Networks, 2004*, June 2004, pp. 887–896.

[17] J.-C. Laprie, "Dependable computing and fault tolerance: Concepts and terminology," in *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995,*. IEEE, 1985, p. 2.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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

[23] 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

[24] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

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

[33] 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

[34] 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.

[35] 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.

[36] D. Cotroneo, A. Lanzaro, and R. Natella, "Faultprog: Testing the accuracy of binary-level software fault injection," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 1, pp. 40–53, Jan 2018.

[37] J. A. Duraes and H. S. Madeira, "Emulation of software faults: A field data study and a practical approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 11, pp. 849–867, Nov 2006.

[38] H. Madeira, D. Costa, and M. Vieira, "On the emulation of software faults by software fault injection," in *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*, June 2000, pp. 417–426.

[39] D. Cotroneo, A. Lanzaro, R. Natella, and R. Barbosa, "Experimental analysis of binary-level software fault injection in complex software," in *2012 Ninth European Dependable Computing Conference*, May 2012, pp. 162–172.

[40] 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.

[41] 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.

[42] 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*. New York, NY, USA: ACM, 2013, pp. 101:1–101:10. [Online]. Available: http://doi.acm.org/10.1145/2463209.2488859

[43] 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.

[44] A. A. Porter and R. W. Selby, "Empirically guided software development using metric-based classification trees," *IEEE Software*, vol. 7, no. 2, pp. 46–54, March 1990.

[45] L. C. Briand, V. R. Brasili, and C. J. Hetmanski, "Developing interpretable models with optimized set reduction for identifying high-risk software components," *IEEE Transactions on Software Engineering*, vol. 19, no. 11, pp. 1028–1044, Nov 1993.

[46] F. Lanubile, A. Lonigro, and G. Vissagio, "Comparing models for identifying fault-prone software components." in *SEKE*, 1995, pp. 312–319.

[47] N. Ohlsson, M. Zhao, and M. Helander, "Application of multivariate analysis for software fault prediction," *Software Quality Journal*, vol. 7, no. 1, pp. 51–66, Mar 1998.

[48] C. Catal and B. Diri, "A systematic review of software fault prediction studies," *Expert systems with applications*, vol. 36, no. 4, pp. 7346–7354, 2009.

[49] B. Farahani and S. Safari, "A cross-layer approach to online adaptive reliability prediction of transient faults," in *2015 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*. IEEE, 2015, pp. 215–220.

[50] Q. Lu, G. Li, K. Pattabiraman, M. S. Gupta, and J. A. Rivers, "Configurable detection of sdc-causing errors in programs," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 3, pp. 88:1–88:25, Mar. 2017.

[51] C. Kalra, F. Previlon, X. Li, N. Rubin, and D. Kaeli, "Prism: Predicting resilience of gpu applications using statistical methods," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC'18. Piscataway, NJ, USA: IEEE, 2018, pp. 69:1–69:14.

[52] A. Merkel and F. Bellosa, "Memory-aware scheduling for energy efficiency on multicore processors." *HotPower*, vol. 8, pp. 123–130, 2008.

[53] A. Jaleel, "Memory characterization of workloads using instrumentation-driven simulation," *Web Copy: http://www. glue. umd. edu/ajaleel/workload*, 2010.

[54] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber *et al.*, "The nas parallel benchmarks summary and preliminary results," in *Supercomputing'91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. IEEE, 1991, pp. 158–165.

[55] N. Gholkar, F. Mueller, and B. Rountree, "Uncore power scavenger: A runtime for uncore power conservation on hpc systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: ACM, 2019, pp. 27:1–27:23. [Online]. Available: http://doi.acm.org/10.1145/3295500.3356150

[56] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ser. ISCA '95. New York, NY, USA: ACM, 1995, pp. 24–36. [Online]. Available: http://doi.acm.org/10.1145/223982.223990

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

[58] G. Li, K. Pattabiraman, C. Cher, and P. Bose, "Experience report: An application-specific checkpointing technique for minimizing checkpoint corruption," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, Nov 2015, pp. 141–152.

[59] D. Andriesse, X. Chen, V. Van Der Veen, A. Slowinska, and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 583–600.

Lucas Palazzi is a former MASc student at the University of British Columbia (UBC), having graduated the program in 2019. He received his BASc in Electrical Engineering at the University of Windsor in 2017. Lucas's research interests include fault tolerance and fault injection techniques and methodologies.

Bo Fang received his bachelor's from Wuhan University, China in 2006 and MASc and PhD from the University of British Columbia (UBC) in 2014 and 2020 respectively. He is now a research associate at Pacific Northwest National Laboratory. Bo's research interests include fault tolerance applications and systems, compilers, machine learning accelerators, and quantum computing.

Guanpeng Li received his BASc (2014) and PhD (2019) degrees from the University of British Columbia, and has recently joined the Department of Computer Science at the University of Iowa as an assistant professor. Guanpeng's research interests include error resilient systems, compilers, and software testing.

Karthik Pattabiraman received his MS and PhD degrees from the University of Illinois at Urbana-Champaign (UIUC) in 2004 and 2009 respectively. After a post-doctoral stint at Microsoft Research (Redmond), Karthik joined the University of British Columbia (UBC) as an assistant professor of electrical and computer engineering. Karthik's research interests include building error resilient software systems, software engineering, and security.