Experimental Evaluation of
Software-Implemented Fault Injection
at Different Levels of Abstraction

by

Lucas Palazzi

B.A.Sc., University of Windsor, 2017

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE

in

The Faculty of Graduate and Postdoctoral Studies

(Electrical and Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

October 2019

© Lucas Palazzi 2019
The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled: “Experimental Evaluation of Software-Implemented Fault Injection at Different Levels of Abstraction”, submitted by Lucas Palazzi in partial fulfillment of the requirements for the degree of Master of Applied Science in Electrical and Computer Engineering.

Examing Committee:

Karthik Pattabiraman, Electrical and Computer Engineering
Supervisor

Sathish Gopalakrishnan, Electrical and Computer Engineering
Supervisory Committee Member

Matei Ripeanu, Electrical and Computer Engineering
Supervisory Committee Member
Abstract

Transient hardware faults caused by cosmic ray or alpha particle strikes in hardware components are increasing in frequency due to shrinking feature size and manufacturing variations. Fault injection (FI), where a fault is artificially introduced during a program’s execution to observe its behaviour, 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, including at the compiler’s intermediate representation (IR) of a program or at the assembly code level. IR-level FI has the advantage that it is closer to the source code of the program being evaluated and hence it is easier to derive insights for the design of fault-tolerance mechanisms. Unfortunately, it is not clear how accurate IR-level FI is vis-a-vis assembly-level FI, and prior work has presented contradictory findings. In this thesis, we first perform a thorough comparison study of two contradictory previous studies, and find that the inconsistent findings are due to an implementation detail regarding how candidate injection bits are selected. Further, we perform a comprehensive evaluation of the accuracy of IR-level FI across a range of benchmark programs and compiler optimization levels to supplement our findings. Our results show that IR-level FI can be as accurate as assembly-level FI for silent data corruptions (SDCs) across different benchmarks and optimization levels, but for crashes the accuracy depends on the optimization level (i.e., less accuracy when more optimizations are applied). Finally, we discuss why compiler optimizations may have an effect on the accuracy of IR-level FI for measuring crash probabilities, and present a machine learning-based technique to estimate crash probabilities that are as accurate as those measured using assembly-level FI, using only results from IR-level FI experiments. The proposed technique is shown to be capable of improving the accuracy of crash probabilities from IR-level FI by over 9 times; this allows IR-level FI to be used for accurately measuring both SDC and crash probabilities.
Lay Summary

Soft errors are becoming more prevalent in modern computer systems, often leading to unexpected program behaviour. Fault injection is a testing technique involving intentionally introducing errors into a program to observe how the program reacts to the error. In this thesis, we compare the measurements made by fault injection implemented in two different ways: at a program’s intermediate representation (IR) code level and at its assembly code level. We thus evaluate and quantify the accuracy of IR-level fault injection with respect to assembly-level fault injection, address contradictions found in prior work, and present an approach for improving the accuracy using machine learning techniques.
Preface

This thesis is the result of work carried out by myself, in collaboration with my advisor, Dr. Karthik Pattabiraman, Dr. Guanpeng Li, and Bo Fang. All chapters, with the exception of Chapters 6-7 are based on work published in 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE). I was responsible for designing and conducting the experiments, analyzing the results, and writing the paper. My advisor was responsible for overseeing the project, providing feedback and guidance, and editing and writing parts of the paper. Guanpeng contributed with his prior knowledge and expertise in the area and provided insights and help with experiment design and analyses, along with assistance with the tools used in said experiments. Bo helped with the analysis and provided insights and technical support over the course of the project.

Lucas Palazzi, Guanpeng Li, Bo Fang, and Karthik Pattabiraman. “A Tale of Two Injectors: End-to-End Comparison of IR-level and Assembly-Level Fault Injection”. IEEE 30th International Symposium on Software Reliability Engineering (ISSRE), Oct 2019
# Table of Contents

Abstract ......................................................... iii

Lay Summary ...................................................... iv

Preface .............................................................. v

Table of Contents ................................................ vi

List of Tables ..................................................... ix

List of Figures .................................................... x

List of Abbreviations ........................................... xii

Acknowledgements ............................................... xiii

Dedication ......................................................... xiv

1 Introduction .................................................... 1
  1.1 Transient Hardware Errors ................................. 1
  1.2 Fault Injection and Motivation ............................ 2
  1.3 Approach and Contributions ............................... 4

2 Background ..................................................... 7
  2.1 Definitions ................................................ 7
  2.2 Code Compilation .......................................... 8
  2.3 Fault Injection ............................................ 9
    2.3.1 Instruction sampling ................................ 10
    2.3.2 Bit sampling .......................................... 10
    2.3.3 Assembly-level FI .................................... 11
    2.3.4 IR-level FI ........................................... 11
    2.3.5 FI example ............................................ 11
  2.4 Machine Learning .......................................... 13
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Related work</td>
<td>15</td>
</tr>
<tr>
<td>3.1</td>
<td>Fault Injection for Measuring Error Resilience</td>
<td>15</td>
</tr>
<tr>
<td>3.2</td>
<td>Comparison of IR-level and Assembly-level FI</td>
<td>16</td>
</tr>
<tr>
<td>3.3</td>
<td>Machine Learning for Fault Outcome Estimation</td>
<td>17</td>
</tr>
<tr>
<td>3.4</td>
<td>Summary</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>Analysis of Inconsistencies in Prior Work</td>
<td>19</td>
</tr>
<tr>
<td>4.1</td>
<td>Reproduction</td>
<td>19</td>
</tr>
<tr>
<td>4.2</td>
<td>Hypothesis</td>
<td>22</td>
</tr>
<tr>
<td>4.3</td>
<td>Support for Hypothesis</td>
<td>24</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Case study: EP benchmark</td>
<td>24</td>
</tr>
<tr>
<td>4.4</td>
<td>Discussion: Lessons Learned</td>
<td>25</td>
</tr>
<tr>
<td>4.5</td>
<td>Summary</td>
<td>26</td>
</tr>
<tr>
<td>5</td>
<td>End-to-End Comparison: IR-level vs. Assembly-level FI</td>
<td>28</td>
</tr>
<tr>
<td>5.1</td>
<td>Experimental Setup</td>
<td>28</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Overview</td>
<td>28</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Benchmarks</td>
<td>29</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Fault injection tools and platform</td>
<td>29</td>
</tr>
<tr>
<td>5.1.4</td>
<td>Measurements of accuracy</td>
<td>31</td>
</tr>
<tr>
<td>5.1.5</td>
<td>Research Questions</td>
<td>33</td>
</tr>
<tr>
<td>5.2</td>
<td>Results and Findings</td>
<td>34</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Program SDC probabilities</td>
<td>34</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Program crash probabilities</td>
<td>36</td>
</tr>
<tr>
<td>5.3</td>
<td>Summary and Discussion</td>
<td>37</td>
</tr>
<tr>
<td>6</td>
<td>Effects of Memory Address Instructions</td>
<td>41</td>
</tr>
<tr>
<td>6.1</td>
<td>Overview of Memory Address Instructions</td>
<td>41</td>
</tr>
<tr>
<td>6.2</td>
<td>Measuring Memory Address Instruction Percentages</td>
<td>42</td>
</tr>
<tr>
<td>6.3</td>
<td>Correlation with Crash Probabilities</td>
<td>43</td>
</tr>
<tr>
<td>6.4</td>
<td>Case Study: Runtime memory address instructions and individual optimizations</td>
<td>45</td>
</tr>
<tr>
<td>6.5</td>
<td>Summary</td>
<td>46</td>
</tr>
<tr>
<td>7</td>
<td>Crash Probability Estimations Using Machine Learning</td>
<td>48</td>
</tr>
<tr>
<td>7.1</td>
<td>Experimental Setup</td>
<td>48</td>
</tr>
<tr>
<td>7.2</td>
<td>Results and Discussion</td>
<td>50</td>
</tr>
<tr>
<td>7.3</td>
<td>Summary</td>
<td>54</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Benchmark Details (Chapter 4) ........................................ 21
5.1 Benchmark Details (Chapter 5) ........................................ 30
5.2 Least Squares Regression Analysis Results ...................... 35
5.3 Statistical Test Results .................................................. 35
6.1 Percentages of Memory Address Instructions .................... 44
6.2 Runtime Memory Address Instruction Correlations for Indi-

vidually Optimized Benchmarks ........................................ 46
7.1 Overview of Machine Learning Models ............................ 50
7.2 ML Model Results ...................................................... 51
7.3 Least Squares Regression Analysis Comparison (LLFI vs. ML
Crash Probability Estimates) ........................................... 51
7.4 Statistical Test Result Comparison (LLFI vs. ML Crash Prob-
ability Estimates) ......................................................... 52
List of Figures

1.1 Overview of a system stack, outlining where SWiFI and hardware-level FI can operate ............................................. 3

2.1 LLVM/Clang code compilation flow ........................................... 8
2.2 Source code of a C function multiplying two integers ........... 11
2.3 Intermediate representation (IR) code of the C function shown in Figure 2.2 ............................................................... 12
2.4 Visual example of a bit-flip error in a 32-bit data value .......... 12
2.5 x86 assembly code of the C function shown in Figure 2.2 .... 13

4.1 Program SDC probabilities measured by LLFI and various versions of PINFI .......................................................... 21
4.2 The bits of an XMM register that are used in x86 floating-point instructions ............................................................ 23

5.1 Program SDC probabilities measured by LLFI and PINFI .......................................................... 34
5.2 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 ......................................................... 39
5.3 Program crash probabilities measured by LLFI and PINFI ......................................................... 40

6.1 X86 assembly code segment from the backprop benchmark; instructions 1 and 2 are considered memory address instructions 42

7.1 Inputs (features) and output (label) of our ML-based predictor 49
7.2 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 ......................................................... 52
List of Figures

7.3 Program crash probabilities estimated by LLFI improved with the random-forest regression model vs. PINFI . . . . . . . . . . . . . . . . [54]
List of Abbreviations

FI  Fault Injection
ML  Machine Learning
SDC  Silent Data Corruption
Acknowledgements

I would like to express my gratitude to Dr. Karthik Pattabiraman, whose invaluable help and guidance as my supervisor has made my experience in the program extremely rewarding. His enthusiasm and dedication to pursuing new and exciting ideas is one that I find inspiring.

In addition to my supervisor, I would like to thank my thesis examining committee, Dr. Matei Ripeanu and Dr. Sathish Gopalakrishnan, for their valuable feedback.

I would like to thank my lab colleagues Guanpeng Li and Bo Fang, whose technical help throughout my research endeavours I could not have done without. I wish them both all the best as they continue their careers post-graduation.

Finally, I express my thanks to the rest of my colleagues in the Dependable Systems Lab for their thoughtful discussions and interactions throughout my time at UBC.
To my parents, who consistently motivate and support me no matter how far I stray from home.
Chapter 1

Introduction

1.1 Transient Hardware Errors

Processor performance has increased exponentially in recent decades, allowing modern computers to achieve great speed and capabilities for both high-performance computers as well as consumer-grade machines. One of the biggest factors contributing to these improvements is smaller and faster transistor technology that allows hardware designers to pack more transistors and components into smaller packages. This trend is continuing, and as computer hardware designs evolve to improve performance the result is a trend toward reduced size and increased density of individual components. However, while this increase in density brings impressive improvements in performance, it also introduces problems related to reliability as processors are becoming more susceptible to transient hardware errors [2, 11, 21, 43].

Transient hardware errors (or soft errors) are caused by external events, such as high-energy particles passing through transistors that lead to data corruption in logic values. As device feature sizes get smaller and chips become more dense, they become more vulnerable to data corruption due to these high-energy particle strikes [21]. Soft errors, unlike design faults or bugs, do not occur consistently and do not cause permanent defects in the hardware, instead randomly altering the electrical signals of stored or transferring data values which can result in unexpected program behaviour and incorrect execution. After a soft error occurs, there is no implication that the system is any less reliable than before.

These types of errors typically manifest in the form of a bit-flip, i.e., a single bit in a data value that is “flipped” from a 1 value to a 0 value or vice versa, thus corrupting the data value in some way. A single bit-flip error in the hardware can propagate to the software that is running on the underlying hardware in three notable ways:

1) the error can be masked by the software, either intentionally through software protection techniques or inadvertently by the program overwriting or ignoring the corrupt data, resulting in no observable deviation
1.2 Fault Injection and Motivation

from the expected program behaviour (i.e., a benign error),

(2) the error can cause the program to throw an exception resulting in the program crashing or hanging, and

(3) the error can go undetected by the program and lead to incorrect output or program behaviour with no indication thereof (i.e., a silent data corruption or SDC).

Traditionally these errors have been masked using protection techniques in the hardware itself, such as modular redundancy and guard banding. However these techniques are becoming more difficult to deploy as they consume high amounts of energy and incur high cost, especially in commodity systems where limiting energy consumption is of high importance. Thus, researchers have proposed various software techniques to detect and recover from hardware faults that are exposed to the software with low performance and energy overheads [12, 47, 49]. These software-implemented techniques are more flexible and cost-effective, and hardware designers are increasingly relying on such software techniques to handle the errors that are exposed to the software at low cost [28, 34, 39].

1.2 Fault Injection and Motivation

An important consideration for deploying any software protection technique is a quantitative evaluation of its ability to detect (or recover from) hardware faults, i.e., the coverage of the protection technique. When proposing such a technique, researchers typically run fault injection experiments to evaluate its coverage. Fault injection (FI) is the process of intentionally and systematically introducing errors into a program and observing the outcome, thus obtaining a statistical estimate of its coverage. Because the injection space is very large, typical fault injection tools use Monte Carlo simulation (i.e., the use of repeated random sampling to make numerical estimations of unknown probabilities). This method samples the space of potential faults to obtain a statistical estimate of a technique’s coverage. Thus, in the context of this thesis, fault injection is used to answer the question: “How well can my software protection technique protect my system against soft errors?”, or, more specifically, “What is the probability of an SDC or crash occurring given that an error has manifested in my system?”.

A key design consideration in a FI tool is the level of abstraction at which it operates. Figure 1.1 shows the various levels of abstraction within a typical system stack where one can design a FI tool. Initially, most studies for
1.2. Fault Injection and Motivation

![System Stack Diagram]

Figure 1.1: Overview of a system stack, outlining where SWiFI and hardware-level FI can operate.

Evaluating the resilience of fault-tolerant systems relied on hardware-level FI techniques, such as heavy-ion radiation and pin-level injection [1]. These techniques inject faults directly at the hardware level, and as such offer the most accurate modeling of the effects of real-world soft errors. However, these techniques require expensive hardware to deploy, and prove to be especially costly if iterative experiments must be conducted throughout the design process [21]. Further, the exact locations of the introduced faults often cannot be controlled, thus reducing the controllability and repeatability of the techniques.

An alternative to hardware-level FI is to inject faults through the software layer; this method is known as software-implemented fault injection (SWiFI) and can operate anywhere in the software layer, from the program’s assembly/machine code to its source code. This approach overcomes many of the difficulties of hardware-level FI, especially with respect to controllability and repeatability [21]. Further, SWiFI techniques offer high portability among different hardware platforms, as well as lower hardware and development cost [44]. It is also much easier to map results back to the source code using SWiFI than with hardware-level FI, which is an important characteristic for understanding and improving the effects of software fault tolerance mechanisms [29].

In general, the closer to the application level a fault injection technique operates, the easier it is to map the results back to the source code for post-experiment analysis. However, raising the level of abstraction farther from the hardware layer comes at a cost in accuracy: it is difficult to model hardware errors in software as not all locations are accessible to the software [18]. Thus, fault injection that operates at a program’s assembly code level is considered to be the most accurate SWiFI technique when mod-
eling soft errors that propagate to a program’s application (architectural) state, as assembly/machine code deal directly with low-level hardware instructions \cite{16,20}. However, assembly-level FI still has some challenges: (1) it has limited portability due to its platform-specific instruction set, and (2) it is challenging to map results to the source code to obtain insights for the design of software mechanisms, as it does not preserve all source code-level information and program structures (e.g., loops and data structures).

To alleviate this difficulty, researchers have proposed implementing FI tools at the intermediate representation (IR) code level of modern compilers such as LLVM/Clang \cite{5,29}. 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, register file sizes, etc., 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 \cite{4,27,45}.

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 considered by many to be more accurate. 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 level. To the best of our knowledge, the only two studies that have examined this question \cite{16,48} come to conflicting conclusions, even while claiming to use the same FI tools and similar experimental setups.

### 1.3 Approach and Contributions

In this thesis, we refer primarily to SWiFI techniques when we say fault injection (FI); thus for our experiments we inject errors in the software layer instead of directly producing faults in the hardware. We specifically explore the accuracy of IR-level FI accuracy compared with that of assembly-level FI.
We first provide an experimental analysis of contradictory prior work [16, 48] and present the reasons for their conflicting findings. In addition, 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, using LLFI [29], 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 set of 25 benchmark programs, including those used in the aforementioned prior studies [16, 48], as well as several other benchmarks not used in those studies. In addition, we consider a wide range of compiler optimization levels in our study, unlike prior studies which do not compare results across different compiler optimization levels.

The main contribution of this work is to systematically and painstakingly gather benchmarks and experimental configurations used in prior work, and to perform a detailed comparison between IR-level and assembly-level FI 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 regarding the accuracy of IR-level FI and design experiments to validate the accuracy of the FI tools that are in question.

We then expand on our experimental findings by proposing a technique for improving the accuracy of IR-level FI to match that of assembly-level FI. This technique is based on an observed correlation between the IR-level estimated crash probability of a program and the amounts of program instructions that operate on memory address values in the respective IR and assembly code. We use this observation to design a set of experiments that evaluate a variety of machine learning models, trained on data collected from our extensive set of benchmarks and fault injection results, for their crash probability estimation capabilities.

Our main findings are as follows:

- We find that an inconsistent fault model (specifically the bit-sampling model) used in [16] is the root cause for the contradictory results found in [16] and [48] (Chapter 4).

- By conducting a thorough comparison 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
1.3. Approach and Contributions

relative ranking of program SDC probabilities, at all optimization levels (Section 5.2.1).

• Using the same comparison study, 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 5.2.2).

• Based on an observed correlation between the measured crash probabilities and the amounts of instructions executed that operate on memory address values (Chapter 6), we present a machine learning-based technique 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 (Chapter 7). Our best-performing model is able to estimate program crash probabilities with significant improvements in accuracy (i.e., with a 9x improvement in mean squared error) when compared to standard IR-level FI measurements.

Our findings thus confirm the results of prior work [48], 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, enabling software developers to choose when to use IR-level FI tools for evaluations. Finally, we show that machine learning is a useful tool that has the potential to provide a boost in accuracy when measuring program crash probabilities using IR-level FI.
Chapter 2

Background

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

2.1 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 assembly language. The IR is independent of both the source language and target architecture. Compilers will typically apply many non-target-specific code optimizations at the IR level. See Section 2.2.

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 injection (FI): The process of systematically introducing errors into a program in order to test its robustness and error-handling capabilities. It can be done in both software and hardware.

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).
2.2. Code Compilation

**Failure:** An error that 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 probability:** The probability of an error causing an SDC for a given program and input.

**Program crash probability:** The probability of an error causing a crash for a given program and input.

### 2.2 Code Compilation

A program must be compiled from its source code to machine code before it can be executed. The structure of a typical compiler can be described in three phases: (1) front end, (2) middle end, and (3) back end. The front end processes the program’s source code (e.g., C/C++ code) and generates an intermediate representation (IR) of the program. The middle end then performs various platform-independent optimizations on the IR. The back end then takes the optimized IR and generates the target-specific machine code for the program while also performing some platform-specific optimizations, such as register allocation. The LLVM/Clang compiler [23] uses this approach, and we can therefore describe its program compilation flow in three parts: (1) source code, (2) intermediate representation (IR), and (3) machine code, as illustrated in Figure 2.1.

![Figure 2.1: LLVM/Clang code compilation flow](image)

A program can be compiled with different compiler optimizations before it is executed. Common compiler optimization levels in the LLVM compiler
2.3 Fault Injection

Fault injection (FI) is a software testing technique used to evaluate a program’s error coverage. In this thesis, we refer primarily to Software Implemented Fault Injection (SWiFI) techniques when we say FI, meaning the injections are performed in the software layer as opposed to directly at the hardware layer. A typical set of FI experiments will consist of many individual FI runs, each run being a single execution of the program with an error introduced. Once an error is introduced, the program can result in either an SDC, a crash\(^1\) or a benign output. Once all FI runs have completed, we can then obtain a statistical estimate of the SDC/crash probabilities. In the experiments conducted for this study, we use 1,000 FI runs to obtain our probabilities, as 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.

In this thesis, 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. Prior research has shown that the single bit-flip model is a valid approximation of soft error manifestation with respect to resilience studies [7, 40], and thus in our FI experiments a single bit-flip error 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 [35]. Related work has made similar assumptions [10, 13, 17, 42, 45].

\(^1\)In this thesis, we consider program hangs as part of the crash category.
2.3. Fault Injection

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 therefore choose a dynamic instruction at random among the total executed sequence of instructions 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.

2.3.2 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, 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 [16, 20]. Common assembly-level fault injectors include BIFIT [24], PINFI [48], and FITgrind [41]. The main drawbacks of assembly-level FI are twofold: (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 thesis, we use PINFI to implement assembly-level FI experiments.
2.3.4 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 [29], KULFI [42], VULFI [46], and FlipIt [5]. 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. In this thesis, we use LLFI to implement IR-level FI experiments.

2.3.5 FI example

We now briefly demonstrate the fault injection process through use of an example. Figure 2.2 shows a sample C code function that multiplies two integers. We now look at how IR-level and assembly-level FI techniques would perform fault injection.

```
int mult() {
    int a = 5;
    int b = 3;
    int c = a * b;
    return c;
}
```

Figure 2.2: Source code of a C function multiplying two integers
2.3. Fault Injection

When compiled to LLVM IR code, the resulting code is as shown in Figure 2.3. Using the instruction sampling methodology described in Section 2.3.1 above, let’s assume the randomly selected instruction is \%3 = \text{\texttt{mul nsw i32} \%1, \%2} as indicated (*) in the figure, i.e., the instruction carrying out the multiplication operation. In this case, we would inject an error into the destination register \%3. Under an error-free run, the \%3 register would hold a value of 5 * 3 = 15 at this point in the program’s execution.

Next, using the bit sampling methodology described in Section 2.3.2, we assume the bit position chosen within the 32-bit data value is randomly selected to be the 10th least significant bit (assuming zero indexing). The resulting bit-flip is shown visually in Figure 2.4, with the value stored in the register changing from 15 to 1039. This error would therefore result in an SDC as the function would return an incorrect result (5 * 3 \neq 1039).

Figure 2.4: Visual example of a bit-flip error in a 32-bit data value.

```
define i32 @mult() #0 {
    %a = alloca i32, align 4
    %b = alloca i32, align 4
    %c = alloca i32, align 4
    store i32 5, i32* %a, align 4
    store i32 3, i32* %b, align 4
    %1 = load i32* %a, align 4
    %2 = load i32* %b, align 4
    * %3 = mul nsw i32 %1, %2
    store i32 %3, i32* %c, align 4
    %4 = load i32* %c, align 4
    ret i32 %4
}"
```
2.4 Machine Learning

We now look at the equivalent code compiled to assembly, shown in Figure 2.5. As is the case in the IR example, we assume the randomly selected instruction is the instruction performing the multiplication operation, i.e., \texttt{imull -8(%rbp), %eax}. We assume the same bit position is chosen within the destination register (in this case, the destination register is \%eax). Note that the \%eax register is a 32-bit register, and thus the entire register width is considered as potential FI locations for bit selection. The injected bit-flip can therefore result in the same SDC outcome as shown in Figure 2.4.

2.4 Machine Learning

Machine learning (ML) is the process of making predictions based on inferred patterns that are learned from data. In \textit{supervised machine learning}, algorithms build mathematical models based on existing data with already-labeled outputs (i.e., a \textit{training data set}); this learning process is known as \textit{training} the model. The trained model can then be used to make predictions on unlabeled data sets based on the patterns that are learned from the training data. In this thesis (Chapter 7), we use \textit{supervised} machine learning to make predictions on the data collected in our FI experiments. We provide some definitions of relevant terminology below.

\textbf{Supervised machine learning:} A subset of ML techniques that is used to train a model on data with existing labels (i.e., expected outputs);
2.4. Machine Learning

the model can then be used to make predictions on unlabeled data (i.e., data where the output is unknown).

**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 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.
Chapter 3

Related work

In this chapter we identify three areas of research related to the work presented in this thesis. The first category includes work related to using different fault injection techniques to measure the error resilience of programs; this presents an overview of the body of work involving FI performed at various levels of abstraction. Next, we review the body of work comparing IR-level and assembly-level FI, specifically; this section is important in providing the proper context in which the bulk of the work presented in this thesis falls. Finally, we present prior work in the area of machine learning with respect to fault injection and error resilience evaluation; this

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. 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 [1].

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 [18, 21]. 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 [18]. For example, a paper by Cho et al. [9] found that assembly-level FI under a simple bit-flip model can only capture a subset of system-level behaviour caused by soft errors. However, when compared with high-fidelity logic-level injection, a simple bit-flip model produces equivalent results and insights as shown in prior work [7]. Further, our focus (and specifically the focus of the two prior studies compared in Chapter 4) is on the subset of errors that do make their way to the application layer and can therefore be modeled using higher-level
3.2 Comparison of IR-level and Assembly-level FI

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 [45]. Calhoun et al. used FlipIt [5], an LLVM-based FI tool, to investigate how SDCs propagate through a specific HPC computation kernel [3]. Chen et al. introduced LADR, an application-level SDC detector that was evaluated using IR-level FI experiments [8]. Finally, Li et al. used LLFI to estimate program SDC probabilities [27]. 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

To the best of our knowledge, only two prior studies directly compare the measurements made using IR-level FI with those of assembly-level FI. Wei et al. compare IR-level FI with 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. [16] 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, concluding 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. [48] claim that LLFI is accurate for emulating SDC-causing hardware errors, while Georgakoudis et al. [16] claim otherwise. This contradiction is peculiar considering that both papers claim to use the same FI tools (i.e., LLFI and PINFI) and similar experimental setups.

In this thesis, 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 works by using a much larger set of benchmarks (larger than both studies combined), testing across four different levels of optimization, and employing a more rigorous statistical analysis of the results.

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 [3, 6, 22, 36, 38]. 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 [15]. 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 [30]. 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. [19] 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 thesis, their work is limited to GPU applications and focused on predicting SDC outcomes while our work applies to the crash probabilities of CPU applications.

3.4 Summary

Fault injection has traditionally been a popular technique to evaluate fault coverage, and there is a large body of research whose goal is to make the FI process easier, more useful, and more efficient. Further, there is a large body of research that relies on fault injection as a tool for evaluating various software protection techniques proposed in said research. The work in this thesis takes a closer look at IR-level FI in particular through an analysis of the contradictory prior work presented in this chapter, and present an application of machine learning to more accurately estimate crash probabilities using IR-level FI.

To the best of our knowledge, we are the first study to systematically address contradictory findings regarding the accuracy of IR-level FI. Further, we are the first to evaluate the use of machine learning to provide more accurate crash probability estimations to supplement IR-level fault injection.
Chapter 4

Analysis of Inconsistencies in Prior Work

Our overview of related work in Section 3.2 outlines two studies \cite{16, 48} that come to contradictory conclusions in their work regarding the accuracy of IR-level FI. Wei et al. \cite{48} conduct a comparison study and find that IR-level FI is as accurate as assembly-level FI for emulating SDC-causing hardware errors. On the other hand, Georgakoudis et al. \cite{16} conduct a similar study, using the same tools as in \cite{48}, but come to find strong inaccuracies in the measurements produced by IR-level FI when compared to assembly-level FI. As a result, these two studies directly contradict each other and have caused confusion in the community; many fault tolerance studies rely on IR-level FI and analysis \cite{4, 8, 27, 45}, and these inconsistent findings 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 results. Our investigation consists of the following three steps:

1. We first attempt to reproduce the results in Georgakoudis et al. \cite{16}, 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.

4.1 Reproduction

In this step, we attempt to replicate the FI experiments presented in Georgakoudis et al. \cite{16}. We use the same set of benchmarks and program inputs used in their experiments, and the FI experiments are performed using a similar setup.
IR-level FI tool

In their study, LLFI\(^2\) was used to represent IR-level FI. The tool was used without any modification, hence we do the same. LLFI operates at the LLVM compiler’s IR level, taking the compiled LLVM IR of the program as input and performing both the profiling run and the FI runs at the IR level.

Assembly-level FI tool

PINFI\(^3\) operates at the x86 assembly code level. It is built as a tool for Intel PIN [31], 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, and (2) taking a program’s executable file as the input.

To represent assembly-level FI, Georgakoudis et al. 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” [16]. 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. [48], which uses the version hosted on the official PINFI repository.

Because the two studies use different versions of PINFI, we perform our experiments using both versions. This is to observe any effects the modifications may have on the results, hence either eliminating the modifications as the culprit or providing evidence to the contrary. To differentiate between the two versions, we label the official version of PINFI used by Wei et al. as PINFI-v1, and the modified version used by Georgakoudis et al. as PINFI-v2.

FI methodology

In our experiments, we adopt an FI methodology based on that used by Georgakoudis et al. [16]. We conduct the FI experiments on 12 of the benchmarks that are used in [16], using the same program inputs used in that study (outlined in Table 4.1). We conduct a total of 1000 FI runs for each benchmark, with each benchmark compiled using standard optimization (-O3); this is consistent with the methodology in [16].

\(^2\)https://github.com/DependableSystemsLab/LLFI
\(^3\)https://github.com/DependableSystemsLab/pinfi
\(^4\)https://github.com/ggeorgakoudis/REFINE/tree/master/pinfi
4.1. Reproduction

Table 4.1: Benchmark Details (Chapter 4)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT</td>
<td>A</td>
</tr>
<tr>
<td>CG</td>
<td>B</td>
</tr>
<tr>
<td>DC</td>
<td>W</td>
</tr>
<tr>
<td>EP</td>
<td>A</td>
</tr>
<tr>
<td>FT</td>
<td>B</td>
</tr>
<tr>
<td>LU</td>
<td>A</td>
</tr>
<tr>
<td>SP</td>
<td>A</td>
</tr>
<tr>
<td>UA</td>
<td>B</td>
</tr>
<tr>
<td>CoMD</td>
<td>-d ./pots/ -e -i 1 -j 1 -k 1 -x 32 -y 32 -z 32</td>
</tr>
<tr>
<td>HPCCG</td>
<td>128 128 128</td>
</tr>
<tr>
<td>lulesh</td>
<td>(default)</td>
</tr>
<tr>
<td>XSbench</td>
<td>-s small</td>
</tr>
</tbody>
</table>

Figure 4.1: Program SDC probabilities measured by LLFI and various versions of PINFI

Results

Figure 4.1 shows the SDC probabilities obtained from our experiments using LLFI, PINFI-v1, and PINFI-v2. Results obtained using PINFI-v3 are also included for sake of brevity; PINFI-v3 will be introduced in the final step of the analysis (Section 4.3).

From these results, we observe that the SDC probabilities measured by LLFI are similar to those measured by PINFI-v1, which is in fact consistent with the results presented by Wei et al. [48]. On the other hand, the SDC probabilities measured by LLFI are clearly very different from those of PINFI-v2, which is consistent with the results presented by Georgakoudis et al. [16]. More specifically, the probabilities obtained using PINFI-v1 and PINFI-v2 do not match! Clearly, the conclusions of each of the two prior studies are dependent on the implementation of PINFI they used (i.e., the
original or modified version). We discuss this in more detail in the next step of the analysis.

4.2 Hypothesis

In this step, we construct a hypothesis based on the results found in Section 4.1 and some further analysis. The main takeaway from the results in the previous section is that experiments conducted with PINFI-v1 and those conducted with PINFI-v2 produce very different SDC probability measurements, and one’s conclusions regarding the accuracy of IR-level FI depend on which version is used. In other words, experiments conducted using PINFI-v1 support the findings in [48], while those conducted using PINFI-v2 support the findings in [16].

It seems from these results that understanding the modifications made in PINFI-v2 are the key to getting to the bottom of this issue. To begin, we compare the source code for both PINFI-v1 and PINFI-v2, both of which are publicly available in their respective repositories, and isolate any differences that could potentially lead to different results. We find that most changes are largely inconsequential, such as general changes to the code structure and different random number generator implementations.

However, the most glaring change is one that alters how PINFI determines the range of bits in the sampled instructions that are considered as FI sites. In PINFI-v1, the bits of a destination register that are considered as possible FI targets are limited to those that are actually used by the instruction or program (i.e., only the bits that are used to represent the data stored in that register), as opposed to simply considering the entire range of bits in the register. This is to ensure that all injected faults are activated, as not all bits in a register will be used for certain instruction types. This approach is in fact consistent with the methodology used in LLFI; since at the IR-level we do not have the same low-level information such as register size, we only inject into the data itself. On the other hand, this modification to PINFI-v2 alters its bit-selection methodology such that it considers the entire range of bits in the instruction’s 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.

To illustrate how this concept manifests in a FI experiment, we consider x86 scalar double-precision floating-point instructions (e.g., addsd, mulsd). x86 floating-point instructions typically use XMM registers as the destination
4.2. Hypothesis

register, which are 128-bit registers. When designing a fault injection tool to inject into these instructions, one could consider all 128-bits in the destination register as potential fault injection sites. However, one must also consider that 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 in fact unused and therefore irrelevant to the program\(^5\). This is illustrated in Figure 4.2.

Thus, if a fault is injected into the upper 64 bits it is very likely that it will simply result in a benign error, regardless of the actual error resilience of the program under test. 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. PINFI-v1 specifically takes cases like this into account to ensure that all faults are activated \[^{[48]}\], however PINFI-v2 applies a more general bit-selection methodology and does not consider cases such as this.

We therefore hypothesize that the modification that alters how the range of potential injected bits is selected is the reason for the inconsistencies between Wei et al. \[^{[48]}\] and Georgakoudis et al. \[^{[16]}\]. In the next step of this analysis, we conduct additional experiments to validate our hypothesis.

\[^{5}\text{This is the case for scalar floating-point instructions; packed floating-point instructions (e.g., addpd, mulpd) will often store two aligned values thus utilizing the upper 64 bits as well.}\]
4.3 Support for Hypothesis

In this step, we conduct additional FI experiments to validate our hypothesis: that a modification to PINFI’s bit-selection mechanism is the root cause of inconsistent findings in [48] and [16].

To test our hypothesis, we create our own modified version of PINFI, which we label as PINFI-v3. PINFI-v3 is a fork of the PINFI-v1 code, with only a single modification to the part of the code that is responsible for bit selection, mirroring that of PINFI-v2. We leave all other parts of the code unchanged. We then run the exact same FI experiments as in Step 1 (Section 4.1) with PINFI-v3, the results of which are shown in Figure 4.1. The results show clearly that the SDC probabilities obtained using PINFI-v3 very closely match those obtained using PINFI-v2, indicating that our hypothesis is valid; the modification to how the injected bit is selected is the reason for PINFI-v1 and PINFI-v2 producing very different measurements, and thus it is also the root cause for the inconsistencies between Wei et al. [48] and Georgakoudis et al. [16].

4.3.1 Case study: EP benchmark

In this section, we present a short case study comparing the injection sites chosen by the different versions of PINFI for one benchmark, EP, taken from our experiments presented earlier in this chapter. From the fault injection logs of both PINFI-v1 and PINFI-v2, we record the following for each injected fault:

- the instruction that was chosen for injection
- the size of the register that was injected into
- the number of bits within that register that are considered as potential FI targets by that version of PINFI
- the actual bit position that was chosen for injection

Through some analysis on the recorded data, we make the following observations:

---

Some additional code was added to both PINFI-v1 and PINFI-v2 to log some of the needed information. The measured SDC probabilities were nearly identical, within the confidence intervals, to those before the code was added.
4.4 Discussion: Lessons Learned

- Roughly 38% of all fault injections performed by PINFI-v2 were in a bit position that would be considered “out of range” by PINFI-v1. In other words, 38% of injections performed by PINFI-v2 were not activated. This clearly has the potential to skew the final results.

- Of the 38% that were injected “out of range”, 91.6% of those injections resulted in a benign output. This follows our presumption that a fault injected outside the range of bits used by the program is likely to result in a benign output (as discussed in Section 4.2).

- Of the remaining 62% of PINFI-v2 injections that were “within range”, roughly 59% resulted in SDC errors. This is much closer to the 64% that was originally measured using PINFI-v1, suggesting that the FI runs that are injected “out of range” significantly skew the results of the experiment in favour of higher benign rates and lower SDC rates.

4.4 Discussion: Lessons Learned

As was shown in our investigation, using a different bit-sampling model can affect the results of a FI experiment very significantly. While we do not know the exact reasoning for the specific modification to PINFI in the study [16] that implemented this change, we assume that the authors were unaware of the full implications surrounding the change as uncovered in this chapter.

One might make the observation that injecting faults into unused register bits is more aligned with the effects of bit-flips caused by cosmic rays, as cosmic rays can affect any bit and not just the ones used by the program under test. However, we emphasize that this still depends on the fault model of choice: one may choose to inject into all available register bits (i.e., to more closely quantify the fault sensitivity of the program) or specifically into bits that are read by the program (i.e., to measure the error sensitivity). When measuring fault sensitivity, the focus is to measure a system’s response with respect to the raw error rate, i.e., modeling the soft errors at the lowest hardware level. The fault model used in [16] does not include the entire fault space that is pertinent to measuring the actual fault sensitivity (as these mechanisms are not available at the software layer), but does include a slightly larger fault space than the fault model used to measure error sensitivity. As such, the fault model used in [16] falls somewhat ambiguously in between measuring fault and error sensitivity.

Ultimately, it is up to the researchers to decide which FI configurations are best suited to reflect the desired fault model. 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, as this makes a significant difference in the results as shown in Figure 4.1. Unfortunately, since the modified PINFI bit selection method used by Georgakoudis et al. [16] is inconsistent with the bit selection method used by LLFI, the paper’s comparison between LLFI and PINFI is invalid.

This finding emphasizes that careful consideration should be given when defining the parameters of one’s FI study. Clearly, these considerations can have a major impact on the results and ultimately the design decisions that are made based on interpretation of the results. This becomes especially important when conducting any sort of comparison between two tools. While there will certainly be fundamental differences between how the tools are implemented (hence, the need for a comparison), all other factors should be controlled for equivalency in order to provide a fair and ultimately 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 implementation details must become common practice in future FI studies.

It should be noted 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 machinespecific. This is a key advantage of IR-level FI, i.e., one does not need to worry about these low-level intricacies when designing FI experiments at the IR level.

4.5 Summary

In summary, the modification made by Georgakoudis et al. [16] 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. By including bits that go unused by the program as FI candidate bits, the probability of the program execution resulting in a benign output is increased, which can skew the overall results (i.e., lower SDC probability measurements). This is illustrated in Figure 4.1 which
4.5. Summary

shows that PINFI-v2 measures a lower SDC probability than PINFI-v1 for all of the benchmarks.
Chapter 5

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

In Chapter 4 we presented an analysis of previously published contradictory studies, and showed how a modification to how bits are selected for injection skewed the results presented by Georgakoudis et al. [16]. Our conclusions seem to confirm the findings presented by Wei et al. [48]. In this chapter, we expand on this finding by conducting an extensive set of FI experiments to further confirm [48]. 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 chapter with a discussion of our findings.

5.1 Experimental Setup

5.1.1 Overview

The experiments presented in this chapter 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).
5.1. Experimental Setup

- 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.

5.1.2 Benchmarks

In our experiments, we choose a total of 25 different benchmarks from 7 publicly available benchmark suites. Their details are shown in Table 5.1. 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 [16, 48] discussed in Chapter 4. The benchmarks that were included in [16, 48] are indicated as such in Table 5.1.

We include all of the benchmarks used in Wei et al. [48], and all but three of the benchmarks used in Georgakoudis et al. [16]; AMG2013, lulesh\(^{7}\), 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 [16, 48], 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. 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.

5.1.3 Fault injection tools and platform

To conduct our FI experiments, we again use LLFI\(^{8}\) and PINFI\(^{9}\) as our IR-level and assembly-level FI tools, respectively. This is to be consistent with the studies discussed in Chapter 4. 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 machines, and the benchmarks are all executed on single threads. LLVM/Clang 3.4 is used.

\(^{7}\)Notice that lulesh was in fact used as a benchmark for the analysis in Chapter 4; this is because the one optimization level that was used for the Chapter 4 experiments does not cause any compatibility issues with LLFI, however other optimizations do.

\(^{8}\)https://github.com/DependableSystemsLab/LLFI

\(^{9}\)https://github.com/DependableSystemsLab/pinfi
5.1. Experimental Setup

Table 5.1: Benchmark Details (Chapter 5)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Suite</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>blackscholes</td>
<td>PARSEC</td>
<td>1 in_16K.txt output.txt</td>
</tr>
<tr>
<td>fluidanimate</td>
<td>PARSEC</td>
<td>1 10 in_5K.fluid out.fluid</td>
</tr>
<tr>
<td>lud</td>
<td>Rodinia</td>
<td>-v -i 512.dat</td>
</tr>
<tr>
<td>backprop</td>
<td>Rodinia</td>
<td>65536</td>
</tr>
<tr>
<td>kmeans</td>
<td>Rodinia</td>
<td>-i 819200.txt -k 1</td>
</tr>
<tr>
<td>bfs</td>
<td>Rodinia</td>
<td>1 graph1MW_6.txt</td>
</tr>
<tr>
<td>bzip2†</td>
<td>SPEC</td>
<td>-lkvv image.jpg</td>
</tr>
<tr>
<td>libquantum†</td>
<td>SPEC</td>
<td>33 5</td>
</tr>
<tr>
<td>hmmer†</td>
<td>SPEC</td>
<td>--seed 10000000 ig.hmm</td>
</tr>
<tr>
<td>mcf†</td>
<td>SPEC</td>
<td>inp.in</td>
</tr>
<tr>
<td>ocean†</td>
<td>SPLASH-2</td>
<td>-p1 -o</td>
</tr>
<tr>
<td>raytrace†</td>
<td>SPLASH-2</td>
<td>-p1 -m64 inputs/car.env</td>
</tr>
<tr>
<td>CoMD‡</td>
<td>Mantevo</td>
<td>-x 10 -y 10 -z 10 -N 50</td>
</tr>
<tr>
<td>HPCCG‡</td>
<td>Mantevo</td>
<td>64 64 64</td>
</tr>
<tr>
<td>XSBench‡</td>
<td>CESAR</td>
<td>-s small</td>
</tr>
<tr>
<td>BT‡</td>
<td>NPB</td>
<td>S</td>
</tr>
<tr>
<td>CG‡</td>
<td>NPB</td>
<td>S</td>
</tr>
<tr>
<td>DC‡</td>
<td>NPB</td>
<td>100000000 ADC.par</td>
</tr>
<tr>
<td>EP‡</td>
<td>NPB</td>
<td>W</td>
</tr>
<tr>
<td>FT‡</td>
<td>NPB</td>
<td>W</td>
</tr>
<tr>
<td>IS‡</td>
<td>NPB</td>
<td>S</td>
</tr>
<tr>
<td>LU‡</td>
<td>NPB</td>
<td>W</td>
</tr>
<tr>
<td>MG</td>
<td>NPB</td>
<td>S</td>
</tr>
<tr>
<td>SP‡</td>
<td>NPB</td>
<td>W</td>
</tr>
<tr>
<td>UA‡</td>
<td>NPB</td>
<td>W</td>
</tr>
</tbody>
</table>

†Benchmark used in Wei et al. [48]
‡Benchmark used in Georgakoudis et al. [16]
5.1. Experimental Setup

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 \((= 25 \times 4 \times 1000)\) for each tool.

5.1.4 Measurements of accuracy

To evaluate the relative accuracy of the SDC and crash probabilities measured using IR-level compared with assembly-level FI, we first apply a visual comparison between the measurements. 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 analyses below.

**Visual comparison**

We first show a graphical overview of the results for each benchmark to visually compare the outcomes of FI execution for each FI tool. Assuming a random sampling (from an approximately normal distribution) of the fault space of size \(n = 1000\) FI runs and a normal score of \(z = 1.96\) corresponding to a 95% confidence interval, we present the measured SDC and crash probabilities \(p\) with standard error using the equation

\[
p \pm z\sqrt{\frac{p(1-p)}{n}}.
\]

We then plot the measurements using bar graphs, along with error bars corresponding to the measured margins of error for visual comparison.

**Least squares regression analysis**

The next analysis we apply is based on a least squares regression model [32]. 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
5.1. Experimental Setup

PINFI. The model plots the PINFI probabilities (y-axis) against those of LLFI (x-axis).

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\(^{10}\) 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.

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.

\(^{10}\)Our dataset meets all $t$-test assumptions as we examined.
5.1. Experimental Setup

**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.

5.1.5 Research Questions

In this study, we are interested in measuring the accuracy of IR-level FI with respect to 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 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?
5.2 Results and Findings

In this section, we present our experimental results based on fault injection experiments conducted on the 25 benchmarks listed in Table 5.1 separating the results based on optimization level.

(a) With -O0 Optimization

(b) With -O1 Optimization

(c) With -O2 Optimization

(d) With -O3 Optimization

Figure 5.1: Program SDC probabilities measured by LLFI and PINFI

5.2.1 Program SDC probabilities

Figure 5.1 shows the SDC probabilities obtained using LLFI and PINFI for each benchmark. We present the numerical results using bar graphs with
5.2. Results and Findings

Table 5.2: Least Squares Regression Analysis Results

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

Table 5.3: Statistical Test Results

<table>
<thead>
<tr>
<th></th>
<th>-O0</th>
<th>-O1</th>
<th>-O2</th>
<th>-O3</th>
</tr>
</thead>
<tbody>
<tr>
<td>p-value†</td>
<td>SDC</td>
<td>0.4210</td>
<td>0.3920</td>
<td>0.6208</td>
</tr>
<tr>
<td></td>
<td>Crash</td>
<td>0.0217</td>
<td>0.0215</td>
<td>0.0031</td>
</tr>
<tr>
<td>Correlation coeff.‡</td>
<td>SDC</td>
<td>0.9636</td>
<td>0.9400</td>
<td>0.9285</td>
</tr>
<tr>
<td></td>
<td>Crash</td>
<td>0.8398</td>
<td>0.6154</td>
<td>0.6672</td>
</tr>
</tbody>
</table>

†Measured using paired sample $t$-test (Section 5.1.4)
‡Measured using Spearman’s rank test (Section 5.1.4)

The least squares regression analysis results are shown in Table 5.2 while the $t$-test and Spearman’s rank test results are shown in Table 5.3.

Visual comparison: Figure 5.1 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.

Least squares regression analysis: The results from the least squares linear regression analysis (shown in Table 5.2) 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 error bars (95% confidence interval for 1,000 runs) for visual comparison.
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 5.2 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 5.3 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 two tools are significantly different from each other.

**Spearman’s rank correlation test:** Finally, the results from the Spearman’s rank correlation test (Table 5.3) 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.

### 5.2.2 Program crash probabilities

We now conduct the same analysis on the crash probability measurements as we did for SDCs. Figure 5.3 shows the crash probabilities obtained using LLFI and PINFI for each benchmark. As in SDCs, the least squares regression analysis is shown in Table 5.2, and the $t$-test and Spearman’s rank test results are shown in Table 5.3.

**Visual comparison:** Figure 5.3 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
5.3 Summary and Discussion

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 5.2) 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 more optimizations are applied to the program, the less accurate the crash probabilities measured by LLFI become compared to PINFI.

Paired sample $t$-test: Table 5.3 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 made using LLFI and PINFI.

Spearman's rank correlation test: As shown in Table 5.3, 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, 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 influenced by the compiler optimizations applied to the program, especially going from -O0 to -O1.

5.3 Summary and 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
5.3. Summary and Discussion

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. This knowledge provides us with some insights into why we observe some of the trends in the above results.

Based on the results presented earlier in this chapter, we conclude 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 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 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, we observe that the accuracy of crash probability measurements seem to be affected specifically by back end optimizations. We discuss this in more detail in Chapter 6.

Finally, with respect to the analysis of prior work presented in Chapter 4, we conclude that the results of this study are consistent with the findings in the study by Wei et al. [48], thus also confirming the results of our analysis in Chapter 4.
5.3. Summary and Discussion

Figure 5.2: 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.
5.3. Summary and Discussion

(a) With -O0 Optimization

(b) With -O1 Optimization

(c) With -O2 Optimization

(d) With -O3 Optimization

Figure 5.3: Program crash probabilities measured by LLFI and PINFI
Chapter 6

Effects of Memory Address Instructions

Chapter 5 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 accuracy in its crash probability measurements when those optimizations are applied.

In this chapter, 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 Chapter 5 are correlated with a program’s percentage of memory address instructions. This chapter provides the basis for our machine learning-based prediction technique in Chapter 7.

6.1 Overview of Memory Address Instructions

We begin this chapter with an introduction to what we call *memory address instructions*. In short, a memory address instruction is an instruction that operates on a register value that is eventually used as a memory address when the program reads or writes to memory. We explain this concept in some more detail in this section.

A major cause for crashes is *segmentation faults* [14], 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, and so it is likely that the inaccuracies in crash probability measure-
6.2. Measuring Memory Address Instruction Percentages

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 assembly code segment in Figure 6.1 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 a fault were injected into the destination register of one of these instructions, the fault will propagate to the address of a memory instruction. Note that we do not consider instructions (8) and (9) to be memory address instructions, as the addresses for these instructions are held in the source register and we only inject into destination registers in our FI experiments. If a mov instruction instead performs a store operation instead of a load (i.e., the destination register holds the memory address), then that instruction would be considered memory address instructions.

Figure 6.1: X86 assembly code segment from the backprop benchmark; instructions 1 and 2 are considered memory address instructions

6.2 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 does not include instructions that propagate memory dependencies between basic blocks, it does capture most of the crash-causing faults; most crashes occur within the same basic block as the faulty instruction [25, 26].

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

1. We first obtain a sampling of the instructions executed. This provides us with a statistical estimate of the frequency of execution for the instructions in question.

   - **IR-level**: We use the FI logs produced by LLFI to provide 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 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 perform 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 6.1.

### 6.3 Correlation with Crash Probabilities

In this thesis, we are interested in how a program’s percentage of memory address instructions correlates with the probability of the program crashing.
under a given fault model. Further, we are interested in how back end optimizations affect this property of a program.

To investigate this, we 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-dependent 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 address this, we calculate the correlation coefficient of the set of ratios

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>-O0</th>
<th>-O1</th>
<th>-O2</th>
<th>-O3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86 IR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>backprop</td>
<td>54.2</td>
<td>41.3</td>
<td>32.8</td>
<td>16.4</td>
</tr>
<tr>
<td>bfs</td>
<td>45.2</td>
<td>25.8</td>
<td>20.0</td>
<td>7.7</td>
</tr>
<tr>
<td>blackscholes</td>
<td>16.0</td>
<td>9.9</td>
<td>9.9</td>
<td>5.6</td>
</tr>
<tr>
<td>BT</td>
<td>68.2</td>
<td>67.9</td>
<td>18.5</td>
<td>0.8</td>
</tr>
<tr>
<td>bzip2</td>
<td>33.6</td>
<td>11.1</td>
<td>30.7</td>
<td>10.7</td>
</tr>
<tr>
<td>CG</td>
<td>59.3</td>
<td>0.6</td>
<td>34.3</td>
<td>0.1</td>
</tr>
<tr>
<td>cmd</td>
<td>61.6</td>
<td>45.4</td>
<td>28.2</td>
<td>13.6</td>
</tr>
<tr>
<td>DC</td>
<td>51.4</td>
<td>35.9</td>
<td>21.0</td>
<td>1.4</td>
</tr>
<tr>
<td>EP</td>
<td>16.4</td>
<td>12.5</td>
<td>8.4</td>
<td>0.0</td>
</tr>
<tr>
<td>fluidanimate</td>
<td>42.2</td>
<td>15.2</td>
<td>32.7</td>
<td>1.8</td>
</tr>
<tr>
<td>FT</td>
<td>29.2</td>
<td>64.9</td>
<td>20.4</td>
<td>4.9</td>
</tr>
<tr>
<td>hammmer</td>
<td>68.4</td>
<td>45.8</td>
<td>29.6</td>
<td>14.0</td>
</tr>
<tr>
<td>hpccg</td>
<td>54.4</td>
<td>37.3</td>
<td>34.8</td>
<td>0.9</td>
</tr>
<tr>
<td>IS</td>
<td>38.2</td>
<td>33.6</td>
<td>39.0</td>
<td>31.6</td>
</tr>
<tr>
<td>kmeans</td>
<td>53.6</td>
<td>36.3</td>
<td>16.2</td>
<td>0.9</td>
</tr>
<tr>
<td>libquantum</td>
<td>47.1</td>
<td>32.2</td>
<td>9.8</td>
<td>0.9</td>
</tr>
<tr>
<td>LU</td>
<td>71.6</td>
<td>86.2</td>
<td>24.0</td>
<td>1.6</td>
</tr>
<tr>
<td>lud</td>
<td>58.0</td>
<td>28.7</td>
<td>48.8</td>
<td>0.3</td>
</tr>
<tr>
<td>mcf</td>
<td>41.7</td>
<td>39.2</td>
<td>33.9</td>
<td>57.7</td>
</tr>
<tr>
<td>MG</td>
<td>80.8</td>
<td>68.8</td>
<td>24.1</td>
<td>5.7</td>
</tr>
<tr>
<td>ocean</td>
<td>67.0</td>
<td>74.6</td>
<td>26.9</td>
<td>0.0</td>
</tr>
<tr>
<td>raytrace</td>
<td>46.6</td>
<td>27.5</td>
<td>27.4</td>
<td>3.3</td>
</tr>
<tr>
<td>SP</td>
<td>76.6</td>
<td>75.0</td>
<td>24.7</td>
<td>1.8</td>
</tr>
<tr>
<td>UA</td>
<td>72.1</td>
<td>65.6</td>
<td>25.8</td>
<td>9.5</td>
</tr>
<tr>
<td>xbench</td>
<td>41.9</td>
<td>43.5</td>
<td>31.1</td>
<td>61.4</td>
</tr>
</tbody>
</table>

6.3. Correlation with Crash Probabilities
6.4 Case Study: Runtime memory address instructions and individual optimizations

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 where there is virtually no optimization applied to the program.

6.4 Case Study: Runtime memory address instructions and individual optimizations

In this section, we present a short case study on correlating runtime memory address instructions and crash probabilities at the x86 level with different individual optimizations applied. Recall that the optimization levels that are considered throughout this thesis (i.e., -O0 to -O3) are in fact multiple individual optimization passes applied in series. In this analysis, we consider the optimization passes individually.

We deal with runtime memory address instructions in this case study, i.e., we count instructions whose return values are used as addresses in subsequent load or store instructions across the entire runtime execution of the program, and not only within the same basic block. We do this only for a handful of select programs: those of which do not have exceedingly long execution times as it is a time-consuming process as explained in Section 6.2.

The goal of this case study is to provide further support for the claim that optimizations that affect the amount of memory address instructions also affect the crash probability. To do this, we select a handful of benchmarks with short runtimes and perform the following:

- We compile each benchmark many times, each time with one of 18 individual optimization passes chosen from optimization level -O3, along with an unoptimized baseline.

- Using a similar approach as in Section 6.2, we measure the amount of runtime memory address instructions as a percentage of the total number of instructions executed, all at the x86 assembly code level.
We use PINFI to measure the crash probability of each individually optimized program.

Using the unoptimized program measurements as our baseline, we calculate how much these measurements increase or decrease from the baseline as a difference (e.g., optimized crash probability − unoptimized crash probability).

We calculate the correlation coefficient between the sets of differences (i.e., crash probability and memory address instruction percentage) for each benchmark. A coefficient close to 1 indicates that any change in the amount of memory address instructions is strongly correlated with a change in crash probability.

We perform this process for 7 benchmarks, however 4 of the benchmarks did not exhibit any significant change in crash probability or memory address instruction percentage when the optimizations were applied and thus there is no meaningful correlation to be found for these benchmarks. We therefore only include results for the remaining 3 benchmarks.

The coefficients are shown in Table 6.2. As we can see, the coefficients are all high enough (i.e., > 0.8) to indicate very strong correlations. This supports our claim that a program’s crash probability is influenced by the amount of memory address instructions in the program’s execution.

Table 6.2: Runtime Memory Address Instruction Correlations for Individually Optimized Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Correlation Coefficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT</td>
<td>0.9735</td>
</tr>
<tr>
<td>MG</td>
<td>0.9288</td>
</tr>
<tr>
<td>ocean</td>
<td>0.8137</td>
</tr>
</tbody>
</table>

### 6.5 Summary

In this chapter, we provide some insights into the correlations between measured crash probabilities by IR-level and assembly-level FI and memory address instructions, i.e., instructions in which faults can propagate to memory address operands of load and store instructions. We show that there is a moderate correlation between the IR-to-assembly “gain” in memory address
instructions and measured crash probabilities, and that this correlation only is significant at optimization levels other than \texttt{-O0}.

This finding supports our intuition: when back end optimizations are applied, both the memory address instructions and measured crash probabilities are affected in similar ways. This is a useful insight 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.

In the following chapter, we introduce a machine learning-based prediction technique that uses memory address instruction percentage measurements to provide crash probability estimations that are much closer to that of assembly-level FI, but with only performing injections at the IR-level.
Chapter 7

Crash Probability Estimations Using Machine Learning

In Chapter 6 we show that, when optimizations are applied, there is a correlation between the amount of memory address instructions in a program and its crash probabilities as measured by IR-level and assembly-level FI. This finding supports our intuition: when back end optimizations are applied, both the memory address instructions and measured crash probabilities are affected in similar ways. This is a useful insight 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. Because of this, 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 chapter, we present a machine learning-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 machine learning algorithms on the data gathered in prior chapters (i.e., FI experiment results and memory address instruction percentages) and compare the resulting estimations against the original IR-level crash probability measurements.

7.1 Experimental Setup

In this chapter, we explore the use of supervised machine learning techniques to predict the crash probability of a program to match the results
7.1. Experimental Setup

of assembly-level FI. Our approach uses features obtained only through IR-level FI experiments and through profiling both a program’s IR and assembly code for memory address instructions.

We evaluate a variety of supervised ML techniques to determine which would train a model that best fits 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 7.1. In practice, such 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.

\[
x_1 = \text{estimate of crash probability as measured by IR-level FI} \\
x_2 = \text{percentage of memory address instructions in IR code} \\
x_3 = \text{percentage of memory address instructions in assembly code} \\
y = \text{estimate of crash probability as measured by assembly-level FI}
\]

Figure 7.1: Inputs (features) and output (label) of our ML-based predictor

To implement ML on our data set, we use the scikit-learn library for the Python programming language [37]. 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 hyperparameters, to find the ‘best’ model. Table 7.1 provides an overview of the chosen machine learning models for our evaluation.

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 7.1). We use 5-fold cross-validation to tune the hyper-
7.2 Results and Discussion

Table 7.1: Overview of Machine Learning Models

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

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.

7.2 Results and Discussion

Table 7.2 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 using a model that simply predicts $\hat{y} = x_1$, shown in Table 7.2.
7.2. Results and Discussion

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 Chapter 5 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 7.3, alongside the measurements made by PINFI for comparison.

The least squares regression analysis results are shown in Table 7.3, and the $t$-test $p$-values and Spearman’s rank correlation coefficients are shown in Table 7.4. Plots of the measurements with least squares regression lines are shown in Figure 7.2.

Table 7.2: ML Model Results

<table>
<thead>
<tr>
<th>Model</th>
<th>Training MSE</th>
<th>Test MSE</th>
<th>Test $R^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Forest Model</td>
<td>0.000876</td>
<td>0.001824</td>
<td>0.8754</td>
</tr>
<tr>
<td>Neural Net Model</td>
<td>0.001522</td>
<td>0.003803</td>
<td>0.7597</td>
</tr>
<tr>
<td>SVR Model</td>
<td>0.009554</td>
<td>0.007247</td>
<td>0.5421</td>
</tr>
<tr>
<td>Lasso Model</td>
<td>0.008206</td>
<td>0.007309</td>
<td>0.5381</td>
</tr>
<tr>
<td>Bayesian Ridge Model</td>
<td>0.008168</td>
<td>0.007404</td>
<td>0.5322</td>
</tr>
<tr>
<td>Huber Model</td>
<td>0.008468</td>
<td>0.007486</td>
<td>0.5270</td>
</tr>
<tr>
<td>Ridge Model</td>
<td>0.008153</td>
<td>0.007644</td>
<td>0.5170</td>
</tr>
<tr>
<td>LR Model</td>
<td>0.008152</td>
<td>0.007682</td>
<td>0.5146</td>
</tr>
<tr>
<td>Kernel Model</td>
<td>0.002118</td>
<td>0.022943</td>
<td>-0.4497</td>
</tr>
<tr>
<td>Current practice</td>
<td>0.01595</td>
<td>0.01651</td>
<td>0.41144</td>
</tr>
</tbody>
</table>

1 Model with lowest test MSE
2 Using model that simply predicts $\hat{y} = x_1$

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

Table 7.3: Least Squares Regression Analysis Comparison (LLFI vs. ML Crash Probability Estimates)

<table>
<thead>
<tr>
<th></th>
<th>slope, $m$</th>
<th>$y$-intercept, $b$</th>
<th>$R^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLFI estimates</td>
<td>0.7405 ± 0.1536</td>
<td>0.1313 ± 0.0447</td>
<td>0.4868</td>
</tr>
<tr>
<td>ML estimates</td>
<td>0.9022 ± 0.0442</td>
<td>0.0265 ± 0.0129</td>
<td>0.9446</td>
</tr>
</tbody>
</table>
7.2. Results and Discussion

(a) LLFI estimations

(b) ML estimations

Figure 7.2: 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 7.4: Statistical Test Result Comparison (LLFI vs. ML Crash Probability Estimates)

<table>
<thead>
<tr>
<th></th>
<th>LLFI estimates</th>
<th>ML estimates</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$-value$^\dagger$</td>
<td>6.036e−8</td>
<td>0.70127</td>
</tr>
<tr>
<td>Correlation coeff.$^\ddagger$</td>
<td>0.73321</td>
<td>0.97171</td>
</tr>
</tbody>
</table>

$^\dagger$Measured using paired sample $t$-test (Section 5.1.4)

$^\ddagger$Measured using Spearman’s rank test (Section 5.1.4)

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 orders of magnitude smaller than IR-level FI on its own. 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 which 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) being 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 5.1.4). We notice a sizeable improvement in the least squares regression analysis from LLFI to the ML predictions (Figure 7.2). 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 for the LLFI crash probability predictions we can reject the null hypothesis (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 opens the door for 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 (at all optimization levels).
7.3 Summary

(a) With \(-O0\) Optimization

(b) With \(-O1\) Optimization

(c) With \(-O2\) Optimization

(d) With \(-O3\) Optimization

Figure 7.3: Program crash probabilities estimated by LLFI improved with the random-forest regression model vs. PINFI

7.3 Summary

In this chapter, we present the use of machine learning techniques to use data gathered from a program’s IR- and assembly-level memory address dependencies as well as IR-level fault injection experiments to obtain better estimates of a program’s crash probability. This approach provides crash probability estimates that are as accurate as assembly-level FI measurements, but eliminates the need to perform assembly-level FI experiments; one only needs to perform FI experiments at the IR level.
Chapter 8

Conclusions and Future Work

8.1 Summary

As random bit-flips caused by transient hardware errors are becoming more common, researchers and software designers are relying more on fault injection to evaluate the resilience of software techniques in vulnerable systems. In this thesis, we investigate 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 as a software-implemented fault injection technique, including easier portability and more direct mapping to the original source code. However, the accuracy of IR-level FI has been recently up for debate [16, 48].

We first conduct an analysis of prior work that has come to contradictory conclusions regarding the accuracy of IR-level FI. Our key finding here is that IR-level FI is in fact as accurate as assembly-level FI with respect to SDC probability measurements, and the reason some prior work has come to opposing conclusions is due to an inconsistency in the bit-sampling models used in the respective IR-level and assembly-level FI tools. We show that injecting faults into all bits of an instruction’s destination register results in significantly different SDC probability measurements than limiting the injection space to only the register bits that are used by the program.

We then conduct a thorough set of fault injection experiments to quantify the accuracy of IR-level FI with respect to assembly-level FI; we do this across a large set of 25 benchmarks and consider four optimization levels for each benchmark. Using various rigorous statistical analyses, we find that IR-level FI does not provide significantly different measurements of SDC probability measurements compared to assembly-level FI, irrespective of the optimization level the program is compiled with. For crash probability measurements, IR-level FI provides similar measurements to assembly-level FI when the lowest level of optimization is applied, but when more opti-
8.2. Threats to Validity

mizations are applied the difference becomes more significant.

We then discuss 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. A dominating cause of application crashes is due to errors propagating to memory operations, i.e., a load or store instruction attempting to read or write to a memory location that is “out of bounds”. As 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 find that the correlation of program crash probabilities measured by IR-level and assembly-level FI with program memory address instruction percentages follows the trend of being influenced by the applied compiler optimizations, supporting this intuition. This insight follows our findings in Chapter 5 which tell us that the crash probability measured by IR-level FI is less accurate compared to assembly-level FI when optimizations are applied.

While IR-level FI does provide measurements of a program’s SDC probability with similar accuracy to assembly-level FI, it is still not accurate enough with respect to crashes to rely completely on IR-level FI studies, especially if crashes are of interest. Thus, in this thesis we finally present a machine learning-based technique that takes advantage of the correlations between crash probability measurements and memory address instruction percentages to train a model for predicting crash probability measurements as accurate as assembly-level FI. Using such a technique in practice could potentially allow accurate crash probability estimates for a program under test using only IR-level FI experiments, offering both the benefits of IR-level FI and the accuracy of assembly-level FI.

The most important aspect of this work is providing insights regarding the accuracy of IR-level fault injection, as prior work has not been entirely clear on this matter. Our analysis of contradictory prior work, extensive set of FI experiment findings, and analysis and discussion of crash probability inaccuracy mitigation together provide valuable information and context for researchers and software designers who plan to use software-implemented fault injection in their work.

8.2 Threats to Validity

An external threat to validity is that we limit our experimental setup to use only the LLVM and x86 platforms for IR-level and assembly-level FI,
8.3 Future Work

respectively. We acknowledge that these platforms do not represent all platforms one may encounter when doing IR- or assembly-level FI experiments, however, we choose these platforms as they are the dominant platforms for these types of applications.

Another external threat to validity with respect to our end-to-end analysis in Chapter 5 is the selection of benchmarks for this analysis. While our set of benchmarks fairly large and covers a wide range of scientific applications, all benchmarks are written in either C or C++ and all are executed serially on the CPU. Thus, we acknowledge that it is not clear whether the results from the FI experiments conducted using these benchmarks will generalize to all programs.

An internal threat to validity with respect to the machine learning analysis in Chapter 7 is the size of our data set. When performing machine learning studies, more data is almost always desirable; a large data set reduces the risk of overfitting to the training data set and gives an increased probability of training a model that fully represents all of the pertinent real-world data. As such, our data set of only 100 examples is small by machine learning standards and thus it is possible that our results are affected by overfitting. We attempt to mitigate this by using cross-validation and only drawing conclusions based on the model’s performance on the test set.

8.3 Future Work

There are three potential directions in which this thesis can be extended for future work.

8.3.1 Extending the comparison to other platforms

In this thesis, we focus on evaluating the accuracy of LLVM IR-level FI with respect to x86 assembly-level FI. While these two platforms are popular and commonly used in both research and industry, it would be useful to conduct the same comparisons on other common platforms. The LLVM/Clang compiler infrastructure provides convenient tools and mechanisms to develop fault injection tools at the IR level, and the same can be said for Intel’s PIN tool with respect to the assembly level. This makes LLVM and x86 convenient platforms to conduct fault injection comparison studies. As this might not be the case for all platforms, extending the comparison to platforms other than LLVM and x86 may incur additional challenges with respect to implementing fault injection tools and analyzing memory address instructions within the code.
8.3. Future Work

8.3.2 Comparing FI results across individual optimizations

In this thesis, we focus on comparing IR-level and assembly-level FI across the four most common compiler optimization passes, i.e., -O0, -O1, -O2, and -O3. However, these optimization passes are in fact many individual optimization passes bundled together. Thus, comparing IR-level and assembly-level FI results across individual optimizations could potentially offer valuable insights, e.g., insights into how each individual optimization pass influences the accuracy of IR-level FI measurements. A paper by Narayanamurthy et al. used meta-heuristic search techniques to find individual optimization passes that improved error resilience [33]; a similar approach, i.e., one that uses search techniques to find individual optimizations that improve accuracy of IR-level FI measurements, is another potential avenue for future work.

8.3.3 Conducting a more extensive machine learning study

In this thesis, we train machine learning models to predict the crash probability as measured using assembly-level FI, using IR- and assembly-level memory address instruction percentages and IR-level crash probability measurements as the features. Our machine learning study is relatively straightforward; a more extensive study could be useful as a direction for future work. Other elements that could provide valuable insights include adding additional features that may improve prediction capabilities, implementing feature selection techniques to identify the most and least important features, and improving the size and quality of the training data set. A significant challenge is obtaining good, sufficient training data, as is the case for many potential machine learning applications. In our case, the training data set is fairly small by machine learning standards, and obtaining more would be time-consuming as both IR-level and assembly-level fault injection campaigns would be necessary for each data example. Investigating ways to potentially avoid this bottleneck is another possible direction for future work.
Bibliography


Bibliography


