LLFI: An Intermediate Code-Level Fault Injection Tool for Hardware Faults

Qining Lu, Mostafa Farahani, Jiesheng Wei, Anna Thomas,and Karthik Pattabiraman
Department of Electrical and Computer Engineering, University of British Columbia
{qining, mostafa, jwei, annat, karthikp}@ece.ubc.ca

***Abstract*—Hardware errors are becoming more prominent with reducing feature sizes - however. tolerating them exclusively in hardware is expensive. Researchers have explored software-based techniques for building error resilient applications for hardware faults. However, software based error resilience techniques need configurable and accurate fault injection techniques to evaluate their effectiveness. In this paper, we present LLFI, a fault injector that works at the LLVM compiler's intermediate representation (IR) level of the application. LLFI is highly configurable, and can be used to inject faults into selected targets in the program in a fine-grained manner. We demonstrate the utility of LLFI by using it to perform fault injection experiments into nine programs, and study the effect of different injection choices on their resilience, namely instruction type, register target and number of bits flipped. We find that these parameters have a marked effect on the evaluation of overall resilience.**

I. Introduction

Hardware faults are increasing due to shrinking feature sizes and manufacturing variations. Simultaneously, diminishing design margins and stringent power constraints are making it harder to provide sufficient redundancy for masking faults from software. Researchers have predicted that computer systems in the future will expose (some) hardware faults to the software layer, and will expect the software to tolerate such faults [1], [2], [3], [4], [5]. Therefore, it is important to design software error resilience mechanisms and evaluate them under hardware faults.

To evaluate software error resilience, Software-Implemented Fault Injection (SWiFI) techniques emulate hardware faults at the software level [6], [7], [8]. SWiFI techniques typically operate at the assembly or machine code levels of the program as it is easier to emulate hardware faults at the low level. However, it is challenging to map the results of the injection back to the program's source code, which is needed for understanding and improving the error resilience of programs. To alleviate the difficulty associated with mapping fault-injection results from the assembly code to the source code, high level fault-injection mechanisms operate at, or close to the source code [9], [5], [10], [2], [11]. These techniques allow faults to be injected directly into program variables or statements. The main advantage of high-level fault-injection mechanisms is that the mapping from the fault injection results to the code is straightforward. However, these techniques are less accurate than low-level techniques for emulating hardware faults, as they do not map one on one to machine code.

In this paper, we present LLFI, a SWiFI tool that allows faults to be injected at the LLVM compiler's intermediate representation (IR) level. LLVM is a widely used compiler infrastructure that has been ported to many different platforms [12]. The LLVM IR is strongly typed, and allows identification of high-level source constructs. At the same time, it is able to represent low-level hardware operations, giving it the ability to accurately represent machine instructions. Based on LLVM, LLFI is able to perform static analysis and inject faults in selected locations in the program. In recent work, we have found that fault injections performed at the LLVM compiler's intermediate code level is accurate compared to assembly-level fault injections [13]. Therefore, LLFI is able to combine the configurability of high-level techniques, with the accuracy of low-level techniques. Furthermore, LLFI helps in understanding the effect of different fault injection parameters on application error resilience at LLVM IR level, which composes the other main contribution of this work.

The rest of this paper is organized as follows. We first provide general background about the area. We then describe the high-level operation of LLFI and its architecture. We demonstrate the configurability of LLFI by using it to investigate the resilience of nine benchmark programs with respect to its different configuration options, namely instruction type, register target and fault type. We find that (1) Instruction types have a marked effect on the *Crash* and Silent Data Corruption (*SDC*) rates of applications, (2) Differences in register targets result in different failure types, and (3) There is no difference between injecting single and double bit flips in *SDC* rates, but there is a difference in *Crash* rates, provided the double bit flips are close in time. Finally, we survey related work and conclude the paper.

II. Fault Model and Background

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

Fault Model: In our fault model, we consider transient and intermittent faults that occur in the processor. These are usually caused by cosmic ray or alpha particle strikes affecting flip flops and logic elements. We consider faults that occur in the processor's computation units, i.e., the ALU and the address computation for loads and stores. However, faults in the memory components such as caches are not considered, since these components are usually protected at the architectural level using ECC or parity. We do not consider faults in the control logic of the processor as this is a small portion of the processor area, nor do we consider faults in the instructions' encoding, as these can be handled through control-flow checking techniques [14]. Other work has made similar assumptions [2], [15], [11], [5], [10].

Error Resilience: We define the resilience of an application as its ability to withstand hardware faults if they occur, without leading to a crash, hang or incorrect output, also known as Silent Data Corruptions (SDCs). We are primarily interested in evaluating the resilience of applications using Software Implemented Fault Injection (SWiFI). Therefore, we only inject faults into the program's data or instructions that are visible at the assembly code or higher levels, rather than into the micro-architectural structures where the faults will occur. Further, we consider only activated faults (i.e., faults that are read by the program before being overwritten), as we are not interested in fault masking at the hardware level.

LLVM: LLVM [12] is a compiler infrastructure for life-long program analysis and optimization. Like most compilers,

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

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

III. LLFI: Workflow and Design

LLFI is a fault injection tool that works at the LLVM compiler's IR level, and allows fault-injections to be performed at specific program points, and into specific instructions. LLFI is closely integrated with the LLVM compiler, and can hence support a wide variety of programs and programming languages and architectures [12].

Workflow: Figure 1 shows the workflow of LLFI. LLFI first takes the program's source code as input and converts it to the LLVM IR using LLVM compiler. LLFI then instruments the IR code with fault injection functions at every program point where a fault can potentially be injected. The determination of these program points is based on the compile-time options provided by the user. Examples of compile-time options are the types of instructions and the register operands of the instructions. After this phase, the instrumented IR code is passed to the LLVM compiler's backend and the machine code is generated. We link the generated machine code with custom libraries to generate two executables. The first executable is the *profiling executable*, which profiles each fault injection site in the program to find the number of times it is executed. This information is passed to the second *fault injection* executable, which chooses an injection site at random from the total number of fault injection site executions (gathered by the profiling phase). The fault is then injected at this site by our custom fault injection library, and the program is executed to completion. The type of fault injected as well as the number of faults are decided based on a set of runtime options that are specified by the user. The fault injection results are then gathered and aggregate statistics are computed.

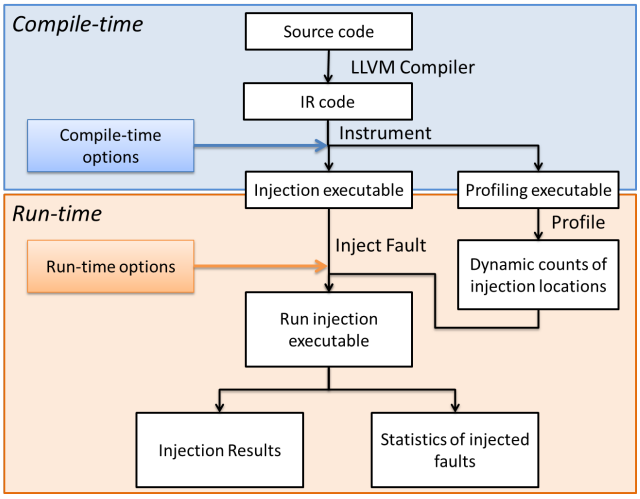

Fig. 1: LLFI workflow from the user perspective

Customizability and Analysis: LLFI has the following features for easy customization.

- *Instruction selection*: LLFI allows choosing a specific instruction or instruction type to inject into. This is done at compile-time.
- *Register selection*: LLFI allows choosing a specific register or operand of the instruction to inject into. This is also done at compile-time.
- *Fault type selection*: LLFI gives the user the ability to specify different kinds of faults to inject at selected points in the program. This is done at runtime.

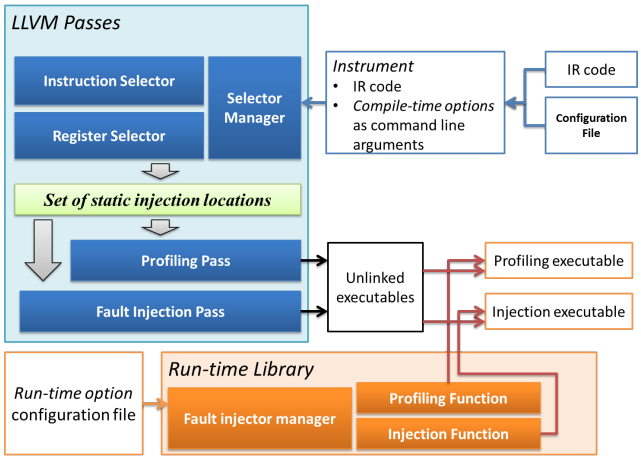

Fig. 2: LLFI architecture and implementation

Architecture: Figure 2 shows the high-level organization of LLFI. It consists of two main components as follows:

LLVM Passes: These are implemented as passes in the LLVM compiler. A pass is an analysis or transformation module that is invoked whenever a compilation unit (e.g., function) is encountered for the first time by LLVM. The passes are responsible for instrumenting the IR code with the fault injection and profiling functions of LLFI. They take as input the program IR code, and a set of compile-time options specified as a *Yaml* file. A script then parses these options and passes them as command line arguments to LLFI's passes.

LLFI has two passes, namely the *profiling pass* and the *fault injection pass*, which are responsible for producing the profiling executable and the fault injection executable respectively. Both passes invoke the *Instruction Selector*, *Register Selector* and the *Selector Manager* to decide what to instrument. The *Instruction Selector* is responsible for deciding what IR instructions should be chosen for instrumentation based on the command line arguments passed in by the script. Similarly, the *Register Selector* is responsible for deciding which register operands of the IR instructions should be chosen for instrumentation. The *Selector Manager* component ensures that these choices are compatible with each other. For example, not all IR instructions will have a destination register, and if this option is chosen by the register selector, then the corresponding instruction cannot be chosen for injection.

Runtime Library: There are two runtime libraries in LLFI, which implement the profiling and fault injection functions respectively. These libraries are linked to the compiled IR code after the instrumentation phase of LLFI to form the profiling and fault-injection executables. The profiling libraries gather the dynamic execution counts of each location that is

TABLE I: Benchmarks used for evaluating LLFI

Benchmark	Description	Input
bfs	Breadth-First Search	1M
cutcp	Distance-Cutoff Coulombic Potential	small
histo	Saturating Histogram	default
mri-q	Magnetic Resonance Imaging - Q	small
sad	Sum of Absolute Differences	default
sgemm	Dense Matrix-Matrix Multiply	small
spmv	Sparse-Matrix Dense-Vector Multiplication	medium
stencil	3-D Stencil Operation	small
tpacf	Two Point Angular Correlation Function	small

instrumented by the LLVM passes and write this data out to a profile file after the program ends. The fault injection libraries read the profile file to obtain the total execution counts of the instrumented locations, and then choose a location at random from the executed locations. They then inject a fault into the chosen instruction when the corresponding fault injection function is invoked at runtime. The type of fault injected, and the number of injected faults are determined by the runtime parameters passed to the library.

IV. Experimental Setup

In this section, we first introduce the benchmarks we use, and then the experimental procedure to demonstrate LLFI.

Benchmarks: We choose nine programs from the PARBOIL benchmark suite [16] to evaluate the configurability of LLFI. The benchmark characteristics are presented in Table I. We choose these benchmarks to represent a wide range of commodity and scientific applications. We run each benchmark to completion with a test or default input that comes with the suite.

System: The experiments were carried out on a Intel Xeon E5 based machine, with 32 GB of RAM and 400 GB Hard drive. The machine was running Linux 3.2.

Research Questions: The research questions aim to validate the configurability and injection capabilities of LLFI.

- **RQ1** What is the effect of injecting faults into different types of instructions at the LLVM IR level ?
- **RQ2** What is the effect of injecting faults into different kinds of registers or operands at the LLVM IR level ?
- **RQ3** How does the choice of single bit flip or double bit flip fault type affect the overall results of the fault injection ?

Experimental procedure: To answer the three research questions (RQs), we configured LLFI for three scenarios, namely (1) injecting faults in different instruction types, (2) injecting faults in source registers and destination registers, and (3) injecting faults under single bit flip and double bit flip scenarios. For each case, we ran 1000 fault injections per benchmark (more in some cases), to obtain tight error bounds on the injections. The 95% error bars ranged from 0.6167% to 1.58% depending on the benchmark. *In total, we inject over a million faults across all the benchmarks and configurations.*

To study the effect of instruction type (RQ1), we considered 17 different major instruction types in the LLVM IR code, and performed fault injections for each of these types. Table II shows the list of selected instruction types. To study the effect of register types (RQ2), we injected faults into (1) source registers of each instruction, and (2) destination register of each instruction. When the instruction had more than one source register, we chose one at random to inject into. In both the above experiments, we only injected single bit flip faults.

To understand the effect of single bit flip versus double bit flip faults (RQ3), we performed another fault injection experiment where we injected double-bit flips in the program. The second bit flip was injected either into the same instruction (i.e., register) or in a different instruction as the first flip. In this experiment, we only considered the source registers of instructions. In cases where a different instruction was chosen for the double bit flip, we select it within a dynamic window size of 'w' from the first instruction. This is to emulate the effect of double-bit flip faults that happen closely clustered in time, but do not affect the same location [17]. We also varied the window size for double-bit flip faults over five values, namely 1, 4, 10, 100 and 1000 instructions, to emulate different ranges of impact for the fault.

TABLE II: Instruction Types that we consider for the IR instructions

Instruction type	Description
fadd	float point addition
fsub	float point subtraction
fmul	float point multiplication
fdiv	float point division
add	integer addition
sub	integer subtraction
mul	integer multiplication
fcmp	float point comparison
icmp	integer comparison
and	bitwise AND
or	bitwise OR
xor	bitwise XOR
shift	left or right shifting
load	load data from memory
store	store data to memory
getelementptr	calculate address with offset values
sext	signed extension
phi	used to implement Static Single Assignment (SSA) form [18]

Failure categorization: As mentioned earlier, we consider only activated faults in the results. For a fault to be activated, the injected location or register must be read by another instruction in the program. This is because we are interested in the behaviour of the program given that a fault has occurred in it, as our goal is to study error resilience (Section II). Our activation rate is close to 100% as we inject into the source register, thus ensuring that the faulty value is used right away, or into destination registers, immediately after the value is written back.

We classify the outcome of activated faults based on the program's behaviour. If the program is terminated by the OS due to an exception, it is classified as a crash. If the program produces the same output as the golden (fault-free run), it is classified as a benign fault. Otherwise, if the output differs from the golden run, we run benchmark-specific acceptance tests on the output to determine if it is an SDC. This is because each program may have a range of acceptable, correct outputs due to small differences in floating point computations. If the program takes substantially longer than the golden run and times out, we classify it as a hang.

V. Results

We organize the results based on the research questions (RQs) introduced in Section IV. We first present the overall results of the injections in Figure 3.

Overall, across applications, the percentages of crashes ranges from 10% to 45% (average is 30%), the percentage of SDCs ranges from 2% to 38% (average is 13%), while Hangs are negligible. All other injections result in *Benign* faults, i.e., faults that do not have any noticeable effect on the application's output despite being activated. These results are for single-bit fault injections in the source registers of the program.

A. Instruction Type Effects (RQ1)

Figure 4 shows the distributions of the fault injection outcomes across benchmarks, categorized based on instruction

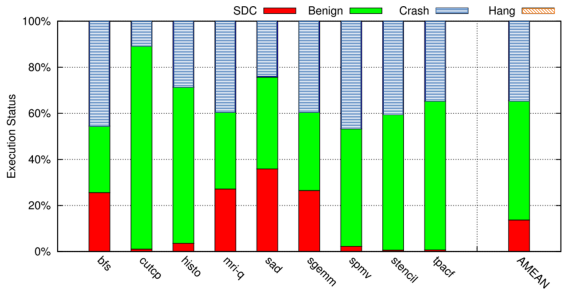

Fig. 3: Results for injection of single bit flip faults in source registers for different benchmarks

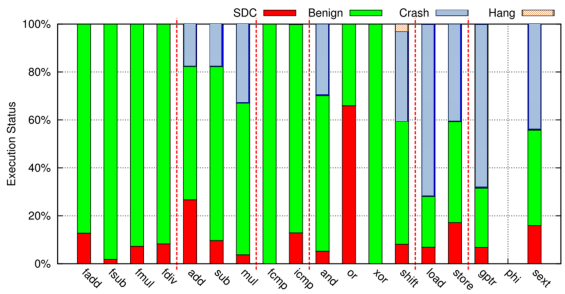

Fig. 4: Results of injecting single bit flips in source registers for different instruction types

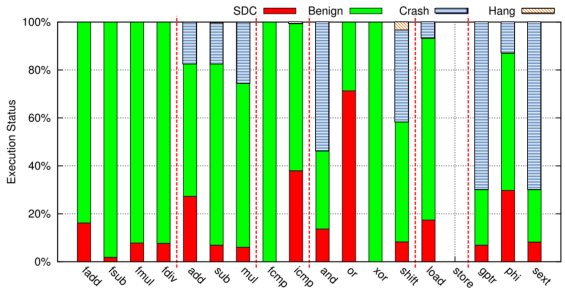

Fig. 5: Results of injecting single bit flips in destination registers for different instruction types

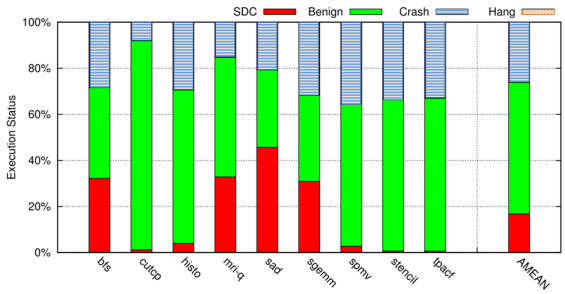

Fig. 6: Results of injecting single bit flip in destination registers for different benchmark programs

types listed in Table II. These results were obtained through single-bit flip injections in the source registers of the benchmark programs. The failure modes for each instruction type are calculated as the mean across all benchmarks. Note that 'phi' instructions have no source registers, and are hence not considered in this experiment. We make the following observations from the figure:

(1) *Crash* rate is negligible (or zero) in floating point instructions (*fadd, fsub*, *fmul*, and *fdiv*), comparison instructions (*fcmp*, *icmp*), and logical operation instructions, (*or* and *xor*). This is because these instructions are almost never used in memory operand or pointer computations, which are likely to cause crashes when in error.

(2) For *and* and *shift* instructions *Crash* rate increases to near 30% and 37%, respectively. One possible explanation for increase in *Crash* rate for these two instructions can be that these instructions are used for memory address calculation or alignment. Hence, faults in *and* and *shift* instructions can propagate to memory addresses and result in a crash.

(3) Comparison instructions (*icmp* and *fcmp*) are highly fault-tolerant, as evidenced by their high *Benign* rates. As most of comparisons in the program check if two variables are *Lesser Than/Greater Than* each other, it is very unlikely that injected faults change the outcome of comparison, especially for floating point comparisons. On the other hand, for arithmetic instructions (*add*, *sub*, *mul* and *div*), we observed that *Crash* rate is higher by about 17%, 17%, and 23%, respectively. From *SDC* rate perspective, these instructions have relatively similar behavior as their floating point counterparts.

(4) Injecting faults in memory instructions has a strong impact on the application failure mode. For *load* instructions, injecting single bit flip faults in source registers (shown in Figure 4) results in more than 70% *Crash* rate and 7% *SDC* rate. A similar effect is observed for *store* instructions, where the *Crash* rate is 40%, and the *SDC* rate is 17%.

Figure 5 shows the results by instruction type when single-bit flips are injected into the destination registers of the instructions. Note that store instructions have no destination register and are hence not considered in this experiment. We observe that there are relatively few differences between injections into the source and destination registers for the instructions in Figure 4, 5. An exception is that *load* instructions have much lower *Crash* rates for destination register fault injections than source register injections. More precisely, the *Crash* rate of load instructions is only 7%, when faults are injected into their destination addresses. This is because a fault in the destination register of a load address affects the loaded value, whereas a fault in the source register affects the address being loaded from. The latter is more likely to cause a crash than the former. For all other instruction types, the difference between source and destination register injections is negligible.

B. Register Selection Effects (RQ2)

Figure 6 shows the results of injecting single bit flips in the destination register of instructions, for each benchmark program. Figure 3 shows the same result when injecting into the source registers of instructions. As mentioned before, injecting single bit flip faults in source registers result in 13% *SDC* rate, and 34% *Crash* rate, on average across benchmarks. The same results for destination register fault injection are 17% and 26%, for *SDC*, and *Crash* rates, respectively.

Thus injecting into destination registers results in higher *SDC* rate but lower *Crash* rate. There are two possible reasons for this phenomenon. First, when injecting into destination register, one is emulating a fault in the instruction being executed, but not in other instructions (unless the fault propagates). When injecting into the source registers however, the fault propagates to every downstream instruction that uses the source register, thereby resulting in a higher *Crash* rate (and correspondingly smaller *SDC* rate). Secondly, when injecting into destination registers, we exclude stores as they do not have a destination register. As was observed in Figure 4, store instructions have

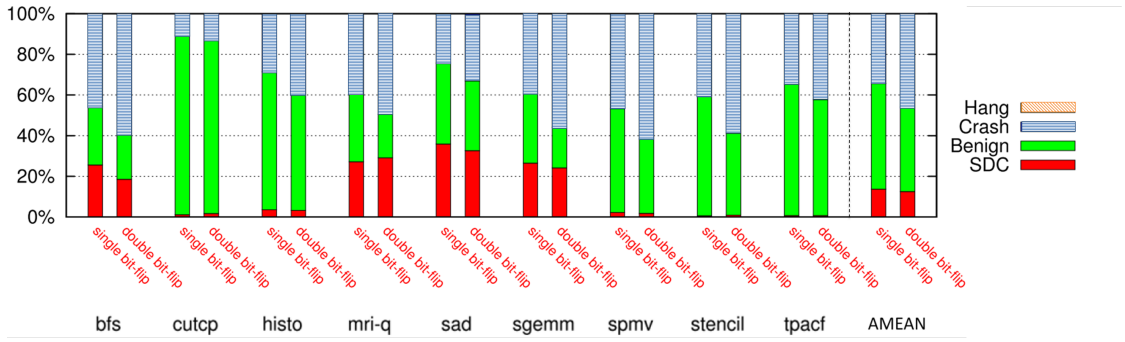

Fig. 7: Results of injecting single bit-flip compared with double bit-flip in the source registers for different benchmark programs

a large contribution to the crash rate, and hence injecting faults in destination registers corresponds to lower crash rates. Likewise, we observed in the previous section that injecting into the destination register of loads results in lower *Crash* rate than the corresponding injections into the source registers. Therefore the crash rate is higher for source registers.

C. Single Versus Double Bit Flips (RQ3)

We first consider the effect of double bit flips into the same register for source register injections, i.e., both bit flips happen in the same source register of the same instruction in Figure 7. From the figure, we can see that double bit flips lead to an increase in the *Crash* rate, consistently across all benchmarks. The increase in the *Crash* rate is offset by the decrease in the *Benign* fault rate. Surprisingly perhaps, double bit flip faults have no significant impact on the *SDC* rate compared to single bit flip faults. One possible reason is that it is more important which values (or instruction types) are corrupted, rather than the exact nature of the corruption, for determining SDCs. We observed a similar behaviour for injections in the destination registers, but do not present the results due to space constraints.

We now consider the effects of injecting double bit flips into different instructions rather than the same instruction. In this experiment, we assume that the registers are chosen from instructions that are separated by almost 'w' dynamic instructions in the program, where 'w' is the window size in instructions. Figure 8 shows the effect of varying 'w' from 1 to 1000 instructions on the different failure modes of the program across all benchmarks. In the figure, *Single* shows the failure rates for single bit flip type, and *Double* shows the failure rates when double bit flips are injected into the same register of the same instruction (as described above). The other points in the figure correspond to different window sizes.

From the figure, we can observe that the *SDC* rate is nearly constant for all values of the window size, and does not differ from the *SDC* rate for double bit faults in the same register. However, the *Crash* rate increases for double bit flips up to a window size of 4, after which it stabilizes. The *Benign* rate correspondingly decreases and saturates at a window size of 4. This shows that there is no significant effect of double bit flip faults that affect instructions that are more than a distance of 4 instructions apart. In other words, the effect of double bit flip faults in different instructions wanes considerably for instructions that are farther than 4 instructions apart.

VI. Implications of the Results

From our results, the key findings that emerge are: (1) Differences exist among different types of instructions in a program in terms of their failure rates, (2) Injecting faults into source registers leads to higher crash rates compared to destination registers, and (3) There is no difference in the SDC rates of programs whether single bit flips or double bit flips (in the same or different instructions) are injected.

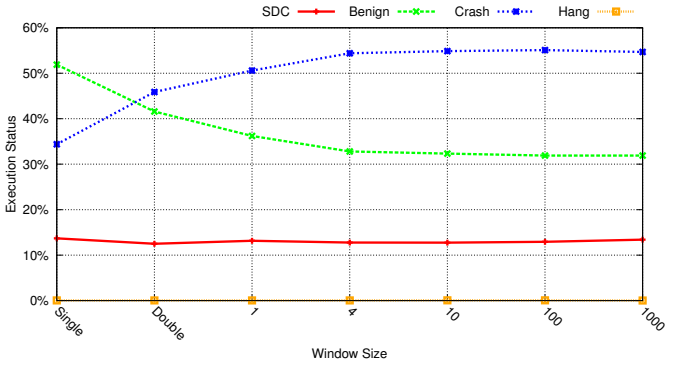

Fig. 8: Effect of varying the window size for double bit flip injections across benchmark programs

Our results indicate that some instruction types are more resilient to errors than others, e.g., comparison instructions. Other work [19] has attempted to investigate similar correlations at the assembly language level, but has not found evidence of such correlations. This suggests that the resilience characteristics of individual instructions is more apparent at the intermediate code level, possibly because the low-level hardware features are abstracted away at the intermediate level, and hence the resilience features of the application dominate. We will further test this hypothesis in future work.

Another implication of our work is that it does not matter for the SDC rate of the program whether single-bit flip or double bit-flip faults are injected. A recent study by Cho et. al. [17] has found that many hardware faults manifest as multiple bit flips in the program, and hence traditional single-bit flip injection may not be sufficient to model these faults. Our results show that if the primary focus is on SDCs, then single bit flip fault models may be sufficient for analyzing the resilience of the program, compared to double bit flips. A similar result was obtained by Ayatollahi et a. [20] when performing injections at the assembly language level.

VII. Related Work

Program-level fault injection for hardware faults: Propane [9] injects faults at the program level and traces their propagation in the program. However, Propane operates on the program's source code level and may not be accurate in emulating hardware faults. A number of papers [21], [2],

[11], [5] present software techniques to protect programs from hardware faults. Similar to LLFI, the authors of the above papers develop fault injectors based on the LLVM compiler to validate their technique. However, none of them describe the fault injector in detail, nor do they attempt to make it configurable.

KULFI [10], which stands for "Configurable Injector", is also built using the LLVM compiler infrastructure, and operates on the IR code. KULFI shares many of the goals of LLFI, including configurability and usability. XRay [22] is a tool built on top of KULFI to aid in visualizing the fault injection results and mapping it back to the program's code. Another fault injector similar to KULFI is *FlipIt* [23], which aims to inject faults into large-scale parallel applications. None of the above papers however consider the effects of the fault injection parameters on application resilience.

In our prior work [24], we have described the design of LLFI, and have evaluated it in the context of soft-computing applications, or applications in which the output can differ from the correct output up to an acceptable margin. Our earlier work focused on validating the *accuracy* of LLFI with regard to assembly level fault injections. We have subsequently quantified the accuracy of LLFI with regard to assembly-level injection for general-purpose applications [13]. However, neither paper considered the architecture of LLFI, nor did they evaluate the effect of the different fault injection parameters.

Assembly code level fault injection: There has been substantial amount of work in fault-injection at the assembly language level for emulating hardware faults. Examples of this approach are NFTAPE [6], GOOFI-2 [7] and Xception [8]. NFTAPE uses break-point based injection at the machine code level. While it allows the user to define their own injectors that can operate at the source code level, no support is provided for statically analyzing the code and customizing the injector. GOOFI-2 supports three methods of fault injection, namely instrumentation-based, exception-based and Nexus-based. Xception uses debug registers and features found in many modern processors to inject faults at runtime. All three methods however operate at the assembly code, compared to our approach which works at the intermediate code level.

Fault injection for software faults: Techniques for injecting software faults in programs typically operate at the source-code level, or at levels close to the source code (e.g., on the abstract syntax tree). G-SWiFT is a technique that attempts to emulate software faults at the machine code level [25], by identifying patterns of assembly code instructions corresponding to high-level software constructs and injecting faults in them to emulate software bugs. In recent work, Giuffrida et al. present EDFI [26], a fault-injection tool for injecting software faults at the LLVM IR level. Similar to LLFI, they transform the code at compile-time with fault-injection instrumentation, and defer the decision of when and where to inject faults to runtime. However, their constraints are different from ours as they aim to accurately emulate the effects of software faults.

VIII. Conclusion

We presented LLFI, a fault injection tool that operates at the intermediate code level of the LLVM compiler infrastructure. LLFI is highly configurable, and allows users to select instructions, registers and fault types to inject into programs. We demonstrated the capabilities of LLFI on nine benchmark programs, and studied their error resilience behaviour. Our experiments found that (1) instruction type is highly correlated with failure outcome, (2) register targets for injection influences the failure rate of applications, and (3) injecting double bit faults changes the crash rate but not the SDC rate of applications, and only within a short window of 4 instructions.

Note: *LLFI is publicly available for download at* https://github.com/DependableSystemsLab/LLFI.

Acknowledgments: This research was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) through the Discovery Grants program, and a research gift from Lockheed Martin Corporation.

References

[1] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: saving dram refresh-power through critical data partitioning," in *ASPLOS*, 2011, pp. 213–224.
[2] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: an architectural framework for software recovery of hardware faults," in *ISCA*, 2010, pp. 497–508.
[3] S. Narayanan, J. Sartori, R. Kumar, and D. L. Jones, "Scalable stochastic processors," in *DATE*, 2010, pp. 335–338.
[4] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate data types for safe and general low-power computation," in *PLDI*, 2011, pp. 164–174.
[5] J. Cong and K. Gururaj, "Assuring application-level correctness against soft errors," in *ICCAD*, 2011, pp. 150–157.
[6] D. T. Stott, B. Floering, D. Burke, Z. Kalbarczpk, and R. K. Iyer, "NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors," in *DSN*, 2000, pp. 91–100.
[7] D. Skarin, R. Barbosa, and J. Karlsson, "GOOFI-2: A tool for experimental dependability assessment," in *DSN*, 2010, pp. 557–562.
[8] R. Maia, L. Henriques, D. Costa, and H. Madeira, "XceptionTM - enhanced automated fault-injection environment," in *DSN*, 2002, pp. 547–550.
[9] M. Hiller, A. Jhumka, and N. Suri, "PROPANE: an environment for examining the propagation of errors in software," in *International Symposium on Software Testing and Analysis*, 2002, pp. 81–85.
[10] V. C. Sharma, A. Haran, Z. Rakamaric, and G. Gopalakrishnan, "Towards Formal Approaches to System Resilience," in *PRDC*, 2013.
[11] A. Thomas and K. Pattabiraman, "Error detector placement for soft computation," in *DSN*, 2013, pp. 1–12.
[12] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004, pp. 75–86.
[13] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the accuracy of high-level fault injection techniques for hardware faults," in *DSN*, 2014.
[14] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability*, vol. 51, pp. 63–75, 2002.
[15] S. K. S. Hari, S. V. Adve, and H. Naeimi, "Low-cost program-level detectors for reducing silent data corruptions," in *DSN*, 2012, p. 181.
[16] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, 2012.
[17] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," in *DAC*, 2013.
[18] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, pp. 451–490, 1991.
[19] B. Sangchoolie, F. Ayatolahi, R. Barbosa, R. Johansson, and J. Karlsson, "Benchmarking the hardware error sensitivity of machine instructions," in *Proceedings of the 2013 IEEE Workshop on Silicon Errors in Logic-System Effects (SELSE Workshop)*, 2013.
[20] F. Ayatolahi, B. Sangchoolie, R. Johansson, and J. Karlsson, "A study of the impact of single bit-flip and double bit-flip errors on program execution," in *SAFECOMP*. Springer, 2013, pp. 265–276.
[21] K. Pattabiraman, Z. Kalbarczyk, and R. K. Iyer, "Automated derivation of application-aware error detectors using static analysis: The trusted illiac approach," in *TDSC*, vol. 8, 2011, pp. 44–57.
[22] S. Chen, G. Bronevetsky, B. Li, M. C. Guix, and L. Peng, "Evaluating application resilience with xray," 2015.
[23] J. Calhoun, L. Olson, and M. Snir, "Flipit: An llvm based fault injector for hpc," in *Euro-Par 2014: Parallel Processing Workshops*. Springer, 2014, pp. 547–558.
[24] A. Thomas and K. Pattabiraman, "Llfi: An intermediate code-level fault injector for soft-computing applications," in *SELSE Workshop*, 2013.
[25] H. Madeira, D. Costa, and M. Vieira, "On the emulation of software faults by software fault injection," in *DSN*, 2000, pp. 417–426.
[26] A. K. Cristiano Giuffrida and A. S. Tanenbaum, "EDFI: A dependable fault injection tool for dependability benchmarking experiments," in *PRDC*, 2013.