REFINE: Realistic Fault Injection via Compiler-based Instrumentation for Accuracy, Portability and Speed

Giorgis Georgakoudis
Queen's University Belfast
Belfast, United Kingdom
g.georgakoudis@qub.ac.uk

Ignacio Laguna
Lawrence Livermore National Laboratory
Livermore, California, USA
ilaguna@llnl.gov

Dimitrios S. Nikolopoulos
Queen's University Belfast
Belfast, United Kingdom
d.nikolopoulos@qub.ac.uk

Martin Schulz
Lawrence Livermore National Laboratory
Livermore, California, USA
Technische Universität München
Munich, Germany
schulzm@in.tum.de

ABSTRACT

Compiler-based fault injection (FI) has become a popular technique for resilience studies to understand the impact of soft errors in supercomputing systems. Compiler-based FI frameworks inject faults at a high intermediate-representation level. However, they are less accurate than machine code, binary-level FI because they lack access to all dynamic instructions, thus they fail to mimic certain fault manifestations. In this paper, we study the limitations of current practices in compiler-based FI and how they impact the interpretation of results in resilience studies.

We propose REFINE, a novel framework that addresses these limitations, performing FI in a compiler backend. Our approach provides the portability and efficiency of compiler-based FI, while keeping accuracy comparable to binary-level FI methods. We demonstrate our approach in 14 HPC programs and show that, due to our unique design, its runtime overhead is significantly smaller than state-of-the-art compiler-based FI frameworks, reducing the time for large FI experiments.

CCS CONCEPTS

• **Computing methodologies → Simulation tools**; *Model verification and validation*; • **Software and its engineering →** *Compilers*; • **Hardware → Analysis and design of emerging devices and systems**;

KEYWORDS

Resilience, Fault Injection, High-Performance Computing, Compiler-based Instrumentation

1 INTRODUCTION

As soft-error rates may increase in future supercomputing systems [7], understanding the effect of soft errors in these systems is becoming increasingly important. Some errors may escape hardware- and system-level detection and correction mechanisms, which could directly impact the results of the scientific applications that run on these systems. A number of resilience techniques have been proposed to cope with these errors at different levels of the HPC software and hardware stack. However, to be able to build efficient and correct resilience techniques, a critical piece of the puzzle is to have an accurate way to quantify the manifestation of errors—from visible failures to silent errors—on different applications and inputs.

Fault injection (FI) is a well-established technique that has been used for years to quantify the effect of soft errors [9, 12, 16, 42]. A wide variety of FI techniques exists, and while most use some kind of software approach to inject faults, some approaches can be as extreme as physically irradiating hardware components [9, 15, 27, 29]. In any of these cases, their purpose is to accelerate the process of observing error manifestations since real soft errors per bit occur rarely in reality [30].

A common method is to inject directly at the binary level [8, 21, 33, 40, 41]. This approach has shown to be both sufficiently accurate as well as efficient. However, it also has a few significant drawbacks: due to its low-level nature it makes it hard both to port to new platforms and to correlate observations with higher-level code structures.

As a counterpart to binary-level FI, a second approach, compiler-based FI, has recently become popular. It instruments high-level code via transformations done by the compiler. Compiler-based FI has several advantages over binary-level FI. First, performing injections in the compiler permits close integration with error-propagation analysis as both classes of analysis (error propagation and FI) operate in the same software layer. Error-propagation analysis is harder to implement at a low level as the original structure of the programs, such as loops, data structures and annotations (e.g., to enable parallelism [10]), may not be available at that level. Second, compiler-based FI provides portability—the generated code with the FI can be transparently compiled for different architectures, whereas binary-level FI usually requires architecture-specific tools, like PIN [28] or Valgrind [24] to transform binary code.

On the downside, though, compiler-based FI has three disadvantages compared to binary-level FI, all caused by the fact that existing methods perform injection at the compiler intermediate representation (IR) [6, 31, 32, 36]: (1) not all low-level dynamic binary instructions are available at the IR level for FI; (2) instrumentation at the IR level interferes with code generation and optimizations—even if FI instrumentation is done after all IR optimizations are applied, the code that is input to the compiler backend can be significantly different from the original non-faulty code, which may generate very different (many times unoptimized) machine binary code; and (3) since code cannot be fully optimized (because of (2)), most frameworks incur significant (unnecessary) overhead, increasing the time to complete FI studies in large applications.

Because of the above limitations, current practices in compiler-based FI are *inaccurate* in quantifying error manifestations in applications. Resilience studies often quantify the proportions of different error manifestations using FI, such as the percentage of aborts, silent data corruption (SDC), and benign cases. It is important to get an accurate picture of these proportions; for example, an application that experiences a large percentage of SDCs may require algorithmic error detection mechanisms, at the expense of runtime overhead. A concern in the HPC community is that a significant number of resilience studies have been based on this FI method [3, 4, 6, 17, 18, 25, 31, 32, 34–36] (including our own work), which can potentially skew FI results and, in some cases, lead to incorrect conclusions. There has been research done in showing these inaccuracies. For example, Wei et al. [41] show that for errors that cause SDC, the accuracy of compiler-based FI can be significantly different from that of binary-level FI; however, no solutions have been proposed to the problem to the best of our knowledge.

In this paper, we identify the main accuracy problems in current compiler-based FI practices and propose `REFINE`[1], a compiler-based FI framework that delivers the advantages of compiler-based FI, while providing the accuracy of binary-level FI. `REFINE` implements FI in the backend code of the LLVM compiler. As the backend code seats at a lower level than the IR code—where most tools today perform FI—the accuracy of FI at this level is comparable to the accuracy of binary-level FI, while still performing FI in a compiler and still enabling the correlation with high-level program abstractions and data structures.

The main contributions of our paper are:

- We identify the main problems that existing compiler-based FI frameworks have, which are potentially shared with the studies that use them.
- We present the design and implementation of `REFINE`, a compiler-based FI that addresses these problems by performing efficient FI in the backend of the LLVM compiler.
- We measure the accuracy of `REFINE` and compare it against state-of-the-art compiler-based FI and binary-level FI. We show that `REFINE`'s accuracy is comparable to that of binary-level FI, and that it is significantly different from existing compiler-based FI tools.

In our experiments, we inject faults into 14 HPC programs (AMG2013, CoMD, HPCCG, Lulesh, XSBench, miniFE and 8 NAS Parallel Benchmarks) and use rigorous statistical methods [20] to determine the number of samples for having an acceptable margin of error ($\leq 3\%$) and confidence level (95%). Moreover, we employ a rigorous statistical inference technique, chi-squared testing with a significance level of 5%, to evaluate the accuracy of different approaches. We found that `REFINE` is more accurate than state-of-the-art compiler-based FI approaches for all benchmark programs. Furthermore, because of our unique backend FI methodology, our approach can be up to 3× faster than existing FI approaches, increasing in this way the speed in which FI experiments can be performed.

2 BACKGROUND AND RELATED WORK

In this section we take a look at the five most common FI techniques, as they apply to high-performance computing (HPC): radiation, hardware simulation, debugger, binary-level, and compiler based FI. A more comprehensive survey of FI methods can be found in the literature [9, 12, 16, 42].

Radiation-based methods irradiate hardware to induce bit flips in a processor or other component [9, 15]. A common approach is to expose a processor's area to a proton beam and then to measure soft error rates (SER) using a verification program that detects incorrect architectural state [27, 29]. While these methods mimic the root-cause of soft errors (hardware bit flips occur due to actual particles strikes) well, their disadvantages are that they are generally non-deterministic [42] and very costly both in terms of time and money [30].

Hardware simulation methods, which are software based, introduce bit flips using a hardware simulator or hardware model. These methods operate at a low level, and can inject faults into the *architectural* state, i.e., machine state that are accessible by software, such as memory and registers, or into the *microarchitectural* state like pipeline logic (such as latches and RAM cells). As examples of architectural-level injectors, Parasyris et al. [26] show a framework to inject soft errors using the Gem5 simulator, and Sanda et al. [29] describe how soft errors are injected into the IBM Mambo architectural simulator to evaluate the resilience of the POWER6 microprocessor chip. Examples of the microarchitectural-level injections are shown in [22, 38, 39]. Although these approaches inject faults in a realistic manner at the hardware level, their disadvantages is that they are generally heavy-weight and targeted to single-node experiments. Microarchitectural-level injectors suffer from the fact that many faults that affect microarchitectural state do not propagate to application state (they are masked) [23], and as a result such FI approaches are not well suited for application-level error propagation studies.

Debugger-based methods use a debugger, which, by controlling the execution of the program, injects faults into application-level state, such as registers [11, 13, 14, 37]. Although they can inject soft errors into any application state, they have two disadvantages. First, they are slow due to the overhead caused by having to frequently handle software interrupts (traps). Second, in parallel multi-process programs, e.g., MPI programs, it is difficult to control in what process or code location to inject errors since this decision must be done by the debugger, which has limited knowledge of the source code and of the runtime systems (e.g., MPI). Nevertheless, this method provides a practical solution—debuggers are available in most platforms—that works in many scenarios.

Binary-level approaches use dynamic binary instrumentation tools, such as PIN [28], DynamoRIO [5], and Valgrind [24], to inject soft errors [8, 21, 33, 40, 41]. The advantages of these methods is that, since they deal with low level hardware instructions, FI is accurate with respect to what is expected in reality from errors that propagate to application (architectural) state. Two PIN-based tools

[1] **RE**alistic **F**ault **IN**j**E**ction (REFINE)

Table 1: Advantages of the fault injection techniques that are relevant to HPC.

	Fault-Injection Techniques for HPC				
	Radiation	Hardware Sim.	Debugger	Binary	Compiler
Accuracy, reliability	✓	✓	✓	✓	
Access to source code abstractions			~		✓
Architecture independence	~				✓
Suitable for multi-process, multi-node experiments				✓	✓
Suitable for application-level error propagation analysis					✓
Low-cost solution in terms of money		✓	✓	✓	✓

✓fully applies; ~ applies to some degree

to inject soft errors are BIFIT [21] and PINFI [41]. The FITgrind [40] tool uses Valgrind for error FI. Although this method is considered to be the most accurate and flexible software-based FI method for soft errors, its main disadvantage is that it is not portable since the underlying instrumentation tools are naturally architecture-dependent. This is a major drawback for resilience studies in HPC due to the variety and heterogeneity that exists in supercomputing hardware. Another disadvantage is that, because they operate at the binary level, high-level source-code abstractions, such as loops, data structures and annotations are lost at that level, thus making it hard to inject errors into particularly selected structures for the purpose of application-level FI.

Compiler-based methods perform FI using a compiler and address the two major problems of binary-level FI methods: they provide a cross-platform solution (instrumented code can be generated for different platforms under the same compiler) and they allow fault injections in particular given source-code abstractions. Recently, a number of compiler-based FI frameworks have been developed using the LLVM compiler [19]—some examples are LLFI [36], KULFI [32], VULFI [31], and FlipIt [6]. These methods are easy to use in large-scale parallel programs, and cooperate well with error propagation analysis frameworks since these frameworks typically operate at the compiler level too. The key disadvantages are that, first, most of them operate at the intermediate representation (IR) level and as a result do not have access to all low level instructions, and that reduces their accuracy; and second, they interfere with the optimization workflow of the compiler, which generates very different, usually poorly optimized, binary code. The latter has a large impact in terms of the time duration of large FI campaigns.

Overall, the latter two methods are the dominant ones, as they are practical as well as efficient, yet each one has a significant drawback. In this paper we aim at creating an FI framework that combines the positive aspects of both. We start with a compiler-based FI, since it provides the ability to correlate the result to the source as well as to drive the FI by source level arguments, and discuss how to augment this approach to achieve the same accuracy as provided by binary level FI (see Table 1).

3 PROBLEM DESCRIPTION

In this section, we describe our fault model and provide a more detailed view of the problems in compiler-based FI.

3.1 Fault Model

We focus on soft errors (also known as transient faults) in the computation nodes of HPC systems. These faults are temporary and they upset the application-visible architectural state of the system. We assume that certain parts of the machine, such as memory and cache, can be protected by parity or ECC (error correcting codes), but that other parts (such as logic and/or latches) are unprotected and can be subject to these errors. We assume a single bit-flip error per application run as previous research has shown that multiple bit flips in a short time (or CPU cycles) is extremely low [30].

We assume that faults are propagated to instructions. For fault injection, we assume a random, uniform distribution when selecting an instruction to inject a fault to—if an application executes N instructions, any instruction has the same probability $1/N$ of having a fault. Note that FI selects an instruction out of the pool of instructions executed *dynamically*. This means that certain instructions emitted statically by the compiler may execute multiple times; each of those instruction instances has an equal probability to be selected. Since an instruction may have multiple output registers, choosing the register to insert the fault is based on another random, uniform distribution. Finally, deciding which bit to flip from a register follows a random, uniform distribution too.

3.2 Overview of LLVM and its IR

We provide a quick overview of the LLVM compiler, and explain only the intermediate representation (IR) aspects that are relevant to this paper. A more detailed explanation can be found in the LLVM documentation [19].

The LLVM IR is a language-independent representation of a program. LLVM uses the IR internally for the various transformation passes, including optimization, from source to machine code. It resembles an assembly language for a LOAD/STORE architecture, akin to RISC-like instruction sets. The LLVM IR assumes an infinite number of available virtual registers and follows the SSA-form to facilitate code analysis and optimization. A program in the IR is functionally equivalent to the source code parsed by the compiler frontend and to the machine code emitted by the backend, after LLVM lowers the IR to machine code. Most LLVM-based FI tools take advantage of this for portability.

```
define internal i32 @eamForce(%struct.SimFlatSt* ↩
    nocapture %s) #0 {
entry:
  %ptime.i.i248 = alloca %struct.timeval
  %ptime.i.i = alloca %struct.timeval
  %nbrBoxes = alloca [27 x i32]
  %dr290 = alloca [3 x double]
  ...
  %8 = load %struct.DomainSt** %domain
  %9 = load %struct.LinkCellSt** %boxes
  %call10 = call %struct.HaloExchangeSt* ↩
      @initForceHaloExchange(%struct.DomainSt* %8, ↩
      %struct.LinkCellSt* %9)
  ...
for.end388:
  %ePotential = getelementptr inbounds %↩
      struct.SimFlatSt* %s, i64 0, i32 7
  store double %etot.7.lcssa, double* %ePotential
  ret i32 0
}
```

(a) LLVM IR

```
_eamForce:
## %entry
  push  rbp
  mov rbp, rsp
  push  r15
  push  r14         Func. prologue
  push  r13
  push  r12
  push  rbx
  sub rsp, 280
  ...               Register spilling
  mov qword ptr [rbp - 224], rcx
  mov qword ptr [r14 + 88], rax
  mov rdi, qword ptr [rbx + 16]
  mov rsi, qword ptr [rbx + 24]
  call  _initForceHaloExchange
  ...
## %for.end388
  xor eax, eax
  add rsp, 280
  pop rbx
  pop r12
  pop r13           Func. epilogue
  pop r14
  pop r15
  pop rbp
  ret
```

(b) x64 assembly

Listing 1: Code excerpt from the CoMD application

3.3 LLVM Intermediate Representation FI

Injecting faults in the IR-level results in loss of accuracy, because the IR abstracts away important aspects of the machine architecture, such as register allocation and machine instruction selection. Moreover, adding FI code in the IR interferes with machine code generation, which impacts both accuracy and speed. Next, we elaborate on these problems.

3.3.1 Unavailability of Machine Instructions. The LLVM IR abstracts away many types of machine instruction emitted during code generation for a specific target. Listing 1 shows an indicative example using a code excerpt from the CoMD proxy application [1] (a molecular dynamics code). Specifically, listing 1a shows the function `eamForce` in its (simplified) IR form and listing 1b shows it in assembly. The IR assumes an infinite number of available registers, hence it misses function prologue and epilogue code sequences or any stack management instructions for register spilling and filling. However, those instructions are an integral part of lowering to machine code for register allocation and conforming to the ABI of function calling. More importantly, those instructions could be subject to soft errors; thus, FI tools must take them into account when injecting.

```
define i32 @compute_residual(...) {
  ...
  %sub = fsub double %0, %1
  fi = call double @injectFault0(i64 3076, double %↩
      sub, i32 11, i32 0, i32 1, %i32 0, i8* ↩
      getelementptr inbounds (...))
  %call = tail call double @fabs(double %fi) #2
  %cmp3 = fcmp ogt double %call, %local_residual.08
  fi1 = call i1 @injectFault1(i64 3078, i1 %cmp3, ↩
      i32 47, i32 0, i32 1, i32 0, i8* ↩
      getelementptr inbounds (...))
  %local_residual.1 = select i1 %fi1, double %call,↩
       double %local_residual.08
  ...
}
```

(a) FI LLVM IR

```
_compute_residual:
  ...
  vsubsd  xmm2, xmm2, qword ptr [rdx]
  vandpd  xmm2, xmm2, xmm1
  vmaxsd  xmm0, xmm2, xmm0
  ...
```

(b) x64 assembly without FI instrumentation

```
_compute_residual:
  ...
  vsubsd  xmm0, xmm0, qword ptr [r15]
  {mov  edi, 3076}[mov  esi, 11}
  {xor  edx, edx}{mov ecx, 1}
  {xor  r8d, r8d}{mov r9, r13}
  call  _injectFault0
  vandpd  xmm0, xmm0, xmmword ptr [rip + LCPI0_0]
  vmovsd  qword ptr [rbp - 56], xmm0
  vucomisd  xmm0, qword ptr [rbp - 48]
  seta  al
  mov qword ptr [rsp], r14
  movzx esi, al
  {mov  edi, 3078}{mov  edx, 47}
  {xor  ecx, ecx}{mov r8d, 1}
  {xor  r9d, r9d}
  call  _injectFault1
  vmovsd  xmm0, qword ptr [rbp - 56]

  ...
```

(c) x64 assembly including FI instrumentation

Listing 2: Code excerpt from the HPCCG application

3.3.2 Code Generation Interference. The way in which most LLVM IR-based FI tools inject faults into IR instructions is by adding a function call to the target instruction [6, 31, 32, 36]. This function call performs large changes to the value of the result of the instruction or its arguments, and it may get inlined after optimization passes. Instrumenting the IR in this way interferes with code generation in the backend of the compiler. In particular, the backend lowers the instrumented IR, which is functionally equivalent but structurally different to the uninstrumented one. This significantly changes the output of instruction selection, register allocation and optimization of the machine code generation passes. Ultimately, it results in a binary which is significantly different from the binary that FI targets to emulate.

Listing 2 shows an indicative example using a code excerpt from the HPCCG application [2]. Here, we use LLFI, a state-of-the-art open-source LLVM-based FI tool, to instrument floating point instructions in the IR. Listing 2a shows a reduced output from the tool LLFI after instrumentation has taken place. The function is identical to the original IR, barring the *injectFault* `call` instructions that instrument the IR instructions `fsub` and `fcmp` respectively. Listing 2b shows the assembly generated without any FI instrumentation, whereas listing 2c shows the assembly produced when including FI at the IR, in the x64 architecture.

As it can be seen, code generation from the FI IR induces a significant number of register spills and reloads. Further, it cannot benefit from memory-to-register optimizations due to the fault injection routines. Hence, it performs all operations mostly on memory operands, using only one XMM register (`xmm0`) and fails to use the `vmaxpd`, which takes only register operands.

4 DESIGN AND IMPLEMENTATION

Here we describe the key conceptual aspects of our approach, `REFINE`, and of our implementation in LLVM.

4.1 Overview of REFINE

Figure 1 gives an overview of `REFINE`. `REFINE` follows a split approach for fault injection using compiler-based instrumentation and an FI library for runtime control of the fault injection process.

The compiler takes a list of functions, instruction types and register type operands to be instrument as input. The compiler-based instrumentation of `REFINE` is carried out by a backend pass during code generation that analyzes the machine instruction representation of the program and augments it with instrumentation code for the fault injection. The generated instrumented binary invokes calls from a user-provided library whenever an instrumented instruction executes. The library controls fault injection at runtime by deciding whether to inject a fault on that specific instruction, which operand to alter and the bit to flip.

4.2 Addressing the Problems of IR Based FI

4.2.1 Access to Machine Instructions. By contrast to existing IR fault injection methods, `REFINE` is part of the backend of the compiler, and as such, it instruments the actual machine instructions generated after the IR has been lowered and any optimization have been applied. This way, `REFINE` has access to the full range of instructions generated by the compiler, including function setup and stack management instructions.

4.2.2 Elimination of Code Generation Interference. `REFINE` does not interfere with code generation for the application under compilation. This is because it does not inject code until after code generation has finished transformations and optimizations on application code. In particular, `REFINE` injects code at the final machine code representation of the application, right before code emission outputs code in assembly or in a binary object format.

Figure 2 shows a diagram of the implementation. Most of its implementation uses the target-agnostic, machine instruction representation (MIR) of LLVM to analyze and instrument code. The MIR of LLVM is a generic representation of the code that includes the control flow of code organized in functions broken down into basic blocks. Although being target agnostic, MIR conveys semantic information common across architectures to describe machine instructions. This includes the types of operands (source or destination) as well as instructions (e.g., memory, branch, arithmetic).

4.2.3 Basic Block Instrumentation Approach. Instead of using function-call-based instrumentation, as IR FI tools do (see Section 3.3.2), `REFINE` uses a *basic block augmentation* approach to add instrumentation code. This approach saves the runtime overhead associated with function calling. `REFINE` analyzes basic block code in the MIR to find potential fault injection targets and intervenes in the control flow to insert extra basic blocks for instrumentation and fault injection. `REFINE` injects three basic blocks for setup and control and a variable number of FI blocks depending on the number of instrumented operands. Those basic blocks and their functionality are:

1. `PreFI` saves any register state that may be clobbered during instrumentation and calls the injection library to return *true* or *false* to trigger, or not, fault injection.
2. `SetupFI` calls the injection library to return the operand and bit to flip, then, it jumps to the FI block for this operand.
3. $\mathrm{FI}_{1\ldots n}$ implements bit flipping for an operand, typically using an XOR instruction.
4. `PostFI` restores clobbered register state and resumes execution in application code.

The target-agnostic part of instrumentation modifies the control flow to include the extra basic blocks and invokes a target-specific module to emit assembly code in them. Code emission is target-specific because it depends on the specific machine architecture on which state registers to save, typically the stack and base pointers and any flag register, which ABI convention to conform to and how to implement bit flipping. For example, on the x64 architecture, there is a different XOR instruction for general (64-bit) and vector (≥ 128-bit) registers.

4.2.4 Control Runtime Library. The instrumented code interfaces at runtime with an external library to control fault injection. The library needs to implement two functions: `selInstr` and `setupFI`. After an instrumented instruction executes, the code injected in `PreFI` invokes the function `selInstr`. The function returns true to trigger fault injection or false to skip it. Since `REFINE` does all the static analysis during compilation, `selInstr` implements only runtime analysis, typically by performing dynamic instruction counting to inject in the right dynamic instruction.

If fault injection does trigger, the code in `SetupFI` calls the function `setupFI` in the library. Note that an instruction may have multiple output registers, each with its own bit size. For example, in x64 architectures, most arithmetic instructions modify the flags register besides the destination register of the operation itself. The function `setupFI` takes as input the number of FI target operands

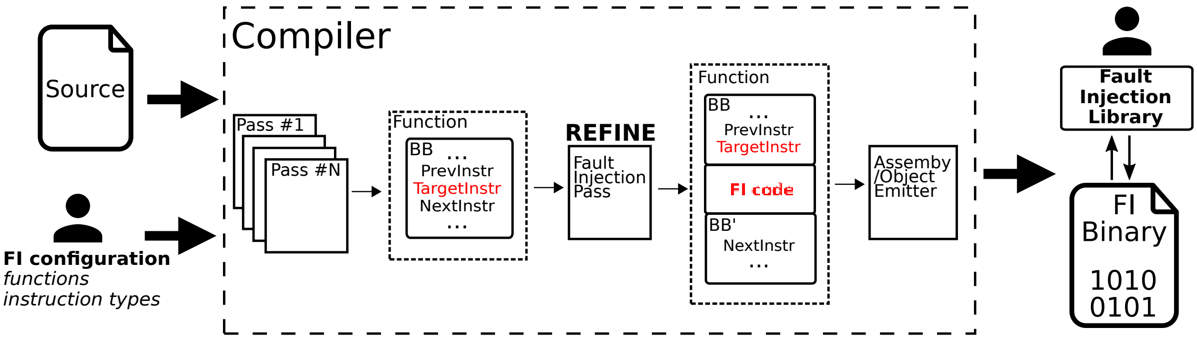


Figure 1: Overview of REFINE

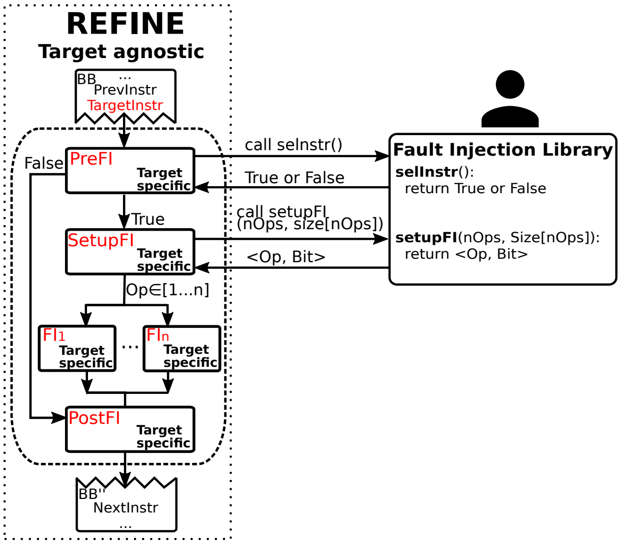


Figure 2: REFINE compiler-based instrumentation

and their respective size in bits. It returns the operand to inject to and which bit to flip. The injected code in the SetupFI basic block jumps to the specific FI block for that operand to perform the actual bit flipping.

4.2.5 Portability Considerations. The target-agnostic part of REFINE's compile-based instrumentation is readily portable, because it operates on the generic machine instruction representation in the compiler. The target-specific code emission module that generates assembly code needs porting. However, the effort is much less compared to binary-level frameworks, which need to port their tracing API and instruction analysis for each different machine architecture.

The fault injection library is portable as it can be implemented in any high-level language, such as C, as long as it conforms to the ABI calling conventions of the target.

4.3 User-level Workflow

We describe the workflow of using REFINE to perform a FI campaign in an HPC program.

4.3.1 Profiling Phase. REFINE performs a profiling step to obtain a dynamic instruction count. Note that this is also the case for all other fault injection tools, including LLFI and PINFI. Profiling needs to be run once for each application and input. Due to the design of REFINE, the FI binary produced by compile-time instrumentation is used unmodified during profiling, since the fault injection library implements the runtime analysis. Figure 3a shows the workflow diagram for profiling, including the library implementation in pseudo-code.

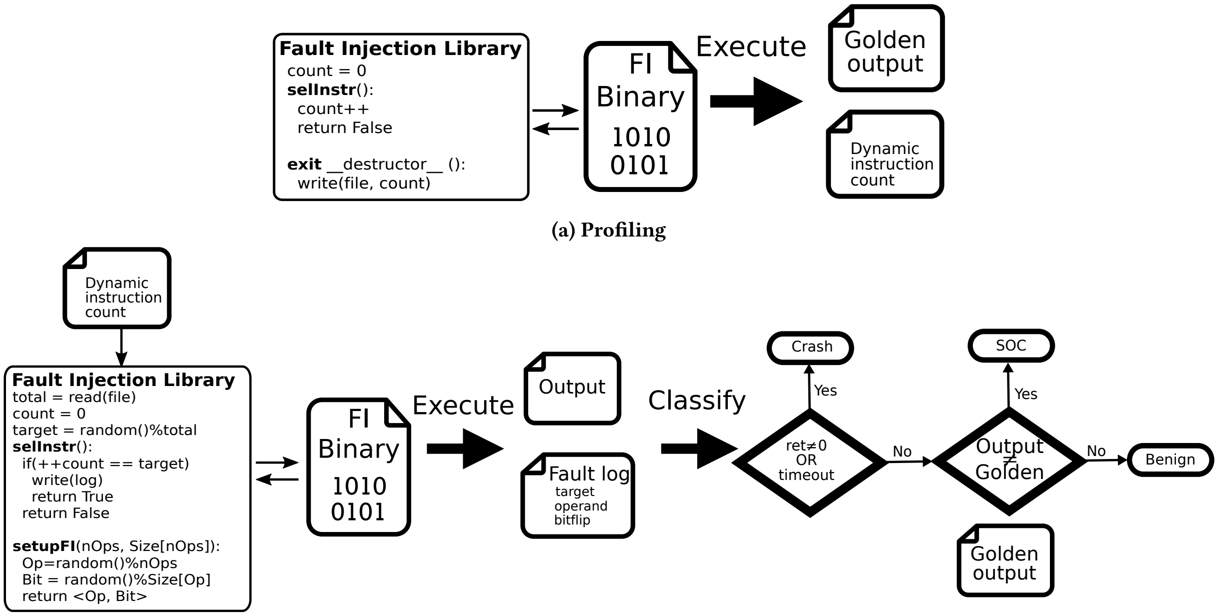
The fault injection library counts the number of dynamic instructions in selInstr and always returns False to skip fault injection. A destructor function writes the dynamic instruction count to a file for persistence. Profiling produces a golden output for the application, that is an error-free output, to be used during fault injection campaigns for determining the occurrence of Silent Output Corruption (SOC) errors.

Figure 3b shows the workflow for the fault injection and classification of the outcome of execution. The dynamic instruction count file from the profiling step is the input to the injection library which implements the single bit-flip fault model. The injection library also outputs a log file which records the target instruction, operand and bit flipped for reference and repeatability.

4.3.2 Output Classification. In our approach, we classify the outcome of a fault injection as either a *crash*, a *Silent Output Corruption* (SOC), or as *benign*. A crash occurs either when the application returns a non-zero exit code or execution times out after a lengthy period: we set this period to be 10× the execution time of the profiling step. A SOC happens when the application's output does not match the error-free, golden output—we use a deterministic application input to obtain the golden output. When comparing application outputs, classification uses only the final results and filters out any intermediate data produced during computation. A benign outcome is when fault injection has no effect on the results of the execution.

4.4 Compiler Flags to Steer REFINE

The backend pass in the compiler exposes a set of flags to control fault injection. Those flags can be used whenever the LLVM compilation driver is invoked. Table 2 summarizes the user interface of REFINE. For example, invoking REFINE on the Clang driver for FI on all functions and instruction types, one would need to include



```
count = 0
selInstr():
  count++
  return False

exit __destructor__ ():
  write(file, count)
```

(a) Profiling

```
total = read(file)
count = 0
target = random()%total
selInstr():
  if(++count == target)
    write(log)
    return True
  return False

setupFI(nOps, Size[nOps]):
  Op=random()%nOps
  Bit = random()%Size[Op]
  return <Op, Bit>
```

(b) Fault injection and classification

Figure 3: Workflow of REFINE for profiling and fault injection

Option	Arguments	Description
-fi	true or false	Enables/disables FI (default: false)
-fi-funcs	comma-separated list of function names or regex	Performs FI only in the provided functions
-fi-instrs	stack, arithm, mem, all	Performs FI only on the selected instr. types

Table 2: Compiler flags interface of REFINE

the following options to the compiler flags: `-mllvm -fi=true -mllvm -fi-funcs=* -mllvm -fi-instrs=all`. This is the actual string of options we use in our experimentation.

4.5 Discussion

Lastly, we discuss the restrictions of our implementation and study. Real-world faults that affect the OP codes of instructions can yield either *incorrect valid OP codes* or *invalid OP codes*. In this implementation, REFINE can produce only correct valid OP codes when injecting a fault in the OP code. This is because the final, assembly emitting stage of the compiler aborts on an invalid OP code to avoid generating an incompatible binary. Although, this restriction exists in the current implementation of REFINE, it can be addressed either by extending the runtime injection library to corrupt the memory addresses of OP codes or by relaxing the verification requirements for instruction validity in the compiler. This is an enhancement we plan to perform in future versions of REFINE. Also, although most of REFINE's functionality is implemented as a target agnostic pass, the code emission module still needs to be ported on different architectures. Nevertheless, based on our experience implementing code emission for the x64 architecture, this only requires modest programming effort.

5 EVALUATION

In this section, we describe our evaluation of REFINE. We evaluate two key aspects of REFINE: (1) its accuracy in injecting faults with respect to existing compiler-based and binary-based FI frameworks; (2) its runtime and compile time overheads. The former is critical in producing accurate results in resilience studies, whereas the latter is important for large-scale FI studies, which under existing approaches can take a enormous amount of time to complete.

5.1 Hardware and Software Platform

We perform our experiments on nodes with the octo-core Intel Xeon E5-2670 processor and 32GB RAM. Nodes run Chaos Linux 5.5 with kernel version 2.6.32. The REFINE implementation extends LLVM version 3.9. Also, we use Intel PIN v3.0-76991 needed by the binary-level FI tools.

5.2 Comparison Tools

We use LLFI [2] and PINFI [3] as comparison FI frameworks, as both represent state-of-the-art techniques for compiler-based and binary-based FI, respectively. LLFI in particular has been used in several recent resilience studies [3, 4, 25, 34, 35].

Regarding PINFI, we make modifications to the available source code to render it compatible with the recent version of the Intel PIN framework and for faithfully implementing the fault model.

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

Program	Input
AMG2013	-in sstruct.in.MG.FD -r 24 24 24
CoMD	-d ./pots/ -e -i 1 -j 1 -k 1 -x 32 -y 32 -z 32
HPCCG-1.0	128 128 128
lulesh	(default)
XSBench	-s small
miniFE	-nx 18 -ny 16 -nz 16
BT	A
CG	B
DC	W
EP	A
FT	B
LU	A
SP	A
UA	B

Table 3: Benchmark programs and their input

In addition, for a fair comparison, we implement an important performance optimization on the original implementation of PINFI, which dramatically increases the speed of execution: PINFI now removes any instrumentation and detaches from the application once the single fault has been injected.

5.3 Experiments Setting

In each experiment, we use one fault injection tool—PINFI, LLFI or REFINE— injecting a single fault during execution, selecting randomly in a uniform fashion the target instruction to inject the fault, the destination register, and the bit to flip.

Table 3 shows the applications used for experimentation and their input. All applications execute sequentially on a single thread.

We perform 1,068 experiments for each application and tool configuration. In total, considering the 14 HPC benchmark programs and the three tools we compare, there are $1,068 \times 14 \times 3 = 44,856$ FI experiments. We use the method shown by Leveugle et al. [20] to calculate the number of samples for each experiment, which ensures a margin of error $\leq 3\%$ for a confidence level of 95%. The confidence intervals presented in plots later reflect this setting.

5.4 Accuracy Results

First, we show a graphical overview of the outcomes of execution under fault injection to contrast visually the results of different tools. Then, we present a rigorous statistical analysis using PINFI as the measure of accuracy by performing chi-squared tests to infer similarities with other tools.

5.4.1 Confidence Intervals Comparison. Figure 4 shows the sampled probabilities of each outcome (crash, SOC or benign) for all tools and applications. Each application results into its own distribution of outcomes, depending on the impact of faults to its execution.

As a rule of thumb, sampled probabilities that fall within the confidence interval of the baseline PINFI are considered similar. Notably, this is the case for all applications when contrasting PINFI and REFINE. However, that is not true for LLFI. A concise way of visualizing diversions and similarities is by plotting the probability mass function (PMF) of each tool as stacked bars corresponding to the different outcomes. By visual inspection, REFINE and PINFI have similar PMFs whereas LLFI diverge for most applications.

Table 4: Contingency table for LLFI vs. PINFI (AMG2013)

	Data categories			
Tool	Crash	SOC	Benign	Total
LLFI	395	168	505	1068
PINFI	269	70	729	1068
Total	664	238	1234	

5.4.2 Chi-squared Tests. Next we present our evaluation using chi-squared tests on contingency tables for each pair of tools. As a reference, the chi-squared test is widely used to determine whether there is a significant difference between the expected frequencies and the observed frequencies in one or more categories, between two approaches.

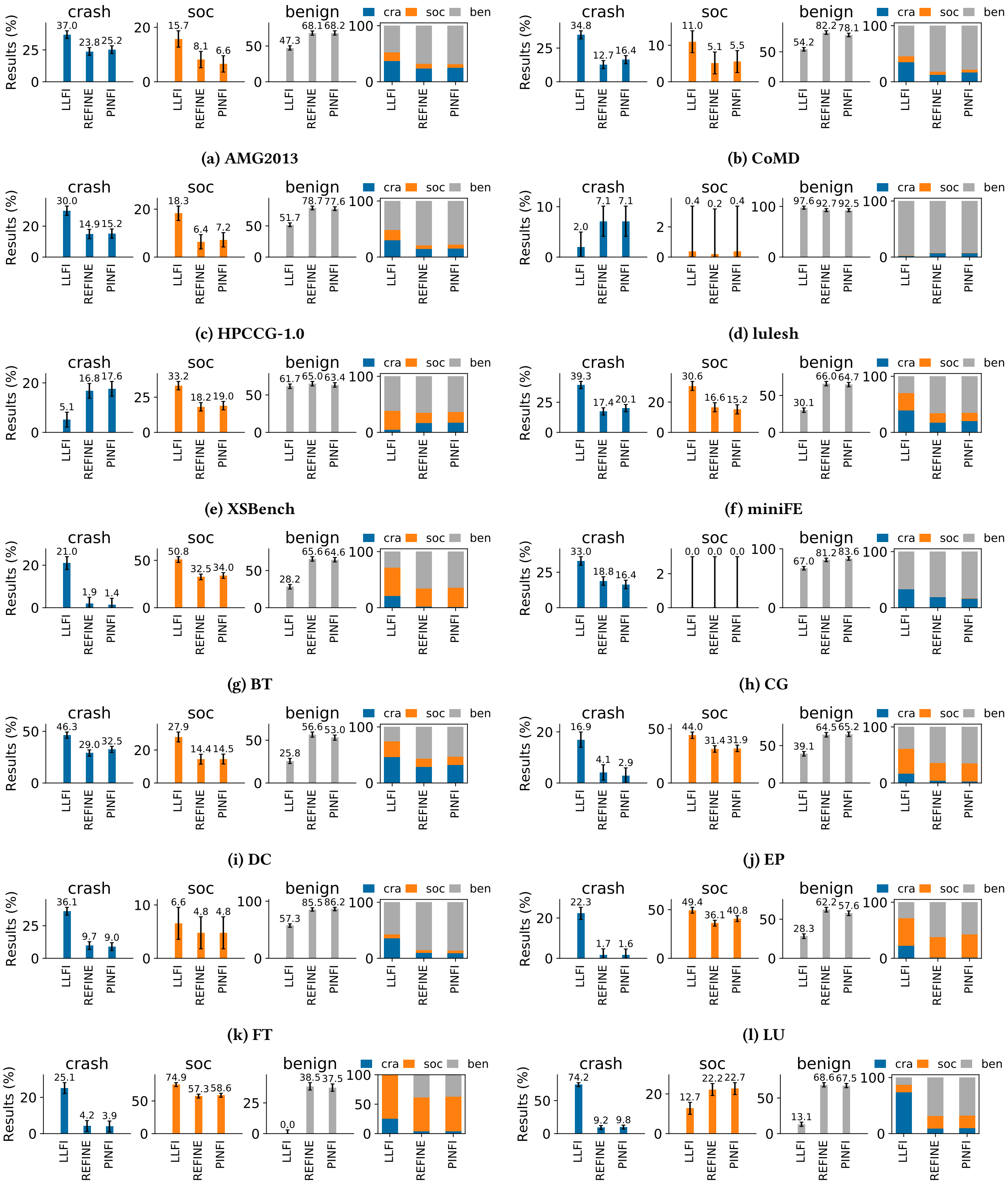
For each program and pair of tools, the chi-squared test uses a contingency table of the frequencies of fault outcomes as input. For each application, we compare the outcomes of approach A versus approach B, and test it for (dis)similarity. Table 4 shows an example of a contingency table to test LLFI (approach A) and PINFI (approach B) for similarity.

Our null hypothesis H_0 is ***using approach A or B has no effect in the outcome frequencies (Crash, SOC, Benign)***; our alternative hypothesis H_a is ***using approach B has significant effect in the outcome frequencies with respect to approach A***. Thus, rejecting the null hypothesis H_0 means that the choice of the fault injection tool is important for determining the fault outcomes. Since PINFI is the most accurate tool, we use it as the baseline to compare with LLFI and REFINE and evaluate their accuracy. If the null hypothesis is rejected, then the tool under test is *significantly different* from the baseline PINFI, thus it is less accurate. We choose the significance level to be $\alpha = 0.05$, therefore, if the calculated p-value of the chi-squared test is less α, the tool is deemed significantly different to PINFI.

Table 5 shows the results of the chi-squared tests across applications. Notably, LLFI is significantly different from PINFI for all applications. The computed p-values are close to 0, which is far below the significance level. The fact that LLFI is different to PINFI is a strong result of high certainty. By contrast, REFINE is never significantly different from PINFI, with only two cases in which the p-value is close to the significance level (CG and CoMD). This means REFINE and PINFI are similar, effectively sampling the same population of outcomes from fault injection.

5.5 Speed Results

Next, we compare the execution time that each tool takes to carry out the experimentation campaign. Here, we measure the runtime of all application fault injection experiments—we do not measure compilation time since this is done once and the execution time of experiments represent the bulk of any FI campaign. A fast FI tool has multiple advantages: (1) it makes it feasible to experiment with larger program inputs; and (2) it enables the collection of more samples to increase accuracy in the same timeframe of a slower tool.



(a) AMG2013

(b) CoMD

(c) HPCCG-1.0

(d) lulesh

(e) XSBench

(f) miniFE

(g) BT

(h) CG

(i) DC

(j) EP

(k) FT

(l) LU

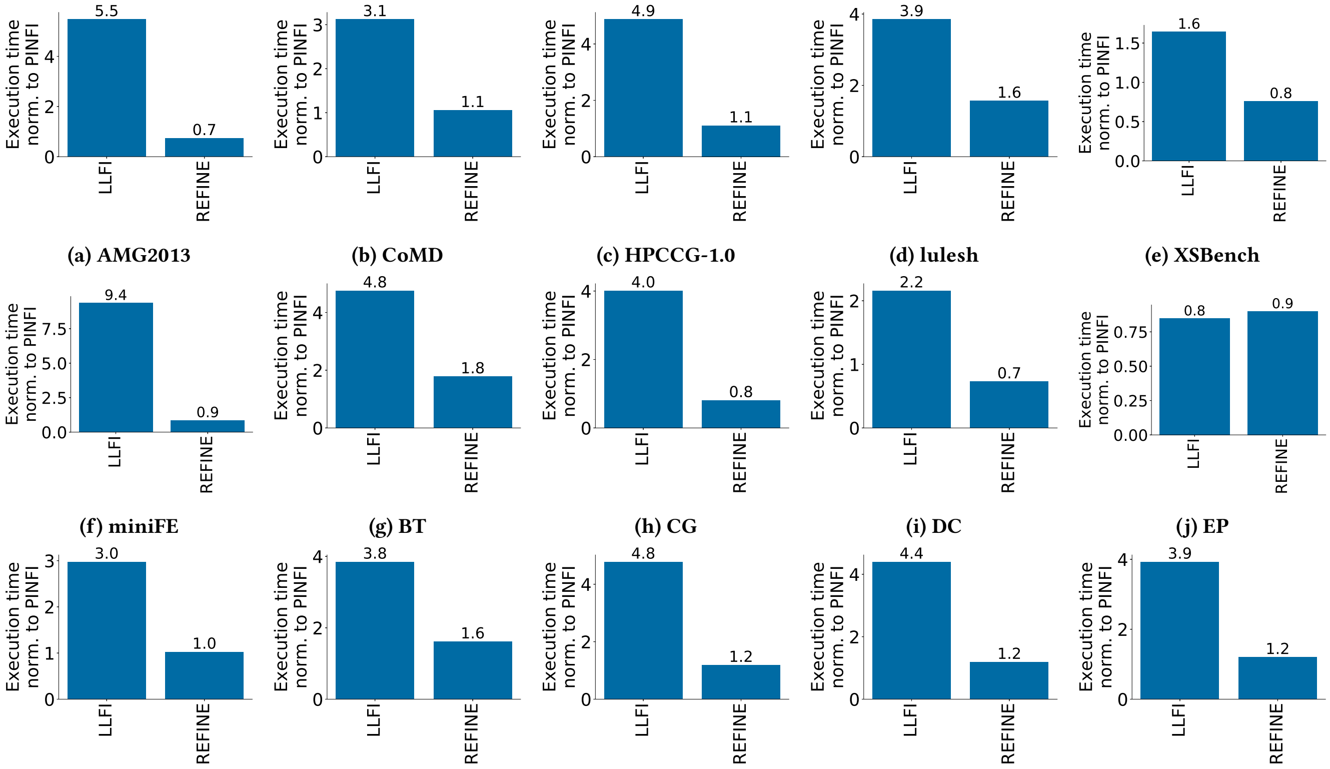
(m) SP

(n) UA

Figure 4: Results of fault injection outcomes

Table 5: Chi-squared test results ($\alpha = 0.05$)

Base	Compassion	p-value	Signif. diff.?	p-value	Signif. diff.?	p-value	Signif. diff.?	p-value	Signif. diff.?
LLFI	**PINFI**	AMG2013		CoMD		HPCCG-1.0		XSBench	
		≈ 0.00	yes	≈ 0.00	yes	≈ 0.00	yes	≈ 0.00	yes
		miniFE		lulesh		BT		CG	
		≈ 0.00	yes	≈ 0.00	yes	≈ 0.00	yes	≈ 0.00	yes
		DC		EP		FT		LU	
		≈ 0.00	yes	≈ 0.00	yes	≈ 0.00	yes	≈ 0.00	yes
		SP		UA					
		≈ 0.00	yes	≈ 0.00	yes				
REFINE	**PINFI**	AMG2013		CoMD		HPCCG-1.0		XSBench	
		0.40	no	0.08	no	0.81	no	0.69	no
		miniFE		lulesh		BT		CG	
		0.14	no	0.60	no	0.26	no	0.06	no
		DC		EP		FT		LU	
		0.13	no	0.55	no	0.92	no	0.21	no
		SP		UA					
		0.92	no	0.83	no				



(a) AMG2013 **(b) CoMD** **(c) HPCCG-1.0** **(d) lulesh** **(e) XSBench**

(f) miniFE **(g) BT** **(h) CG** **(i) DC** **(j) EP**

(k) FT **(l) LU** **(m) SP** **(n) UA** **(o) Total**

Figure 5: Experimentation time

Figure 5 shows total execution time per application for LLFI and `REFINE`, normalized with respect to PINFI. Figure 5o shows the aggregated total, or overall, execution time of all applications. PINFI is most of the time the fastest due to its dynamic binary instrumentation approach, at the expense of portability. `REFINE`'s speed is comparable to that of PINFI; it is 20% slower than PINFI overall, and 30% faster than PINFI in the best case. `REFINE` is always faster than LLFI—3× in total as shown in Figure 5o—except for EP, in which LLFI fault injection results in a large number of crashes that terminate early execution.

6 CONCLUSIONS

Compiler-based fault injection frameworks are gaining popularity in HPC to perform resilience studies due to its platform independence and close relationship to the source code; however, they lack the high accuracy that their counterpart binary-level frameworks have. In this work, we identify the key drawbacks that current practices in compiler-based FI have, and address these issues in the `REFINE` framework, using the backend layer of the LLVM compiler to select target low-level instructions, and via careful, non intrusive instrumentation to improve speed. The impact of this work is important since many recent resilience studies have made conclusions that rely on the current practices of compiler-based FI—these practices can potentially skew the outcomes of FI results, which in turn can lead to incorrect conclusions. In particular, using statistical inference techniques, we show that our compiler-based FI framework is more accurate than state-of-the-art compiler-based FI approaches for all the tested benchmark programs. We also show that by selecting the right method for fault injection code instrumentation, our approach can be up to 3× faster than existing FI approaches, significantly increasing the speed in which FI experiments can be performed.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their valuable comments and helpful suggestions. This work is partially supported by the Engineering and Physical Sciences Research Council (UK) under grant: EP/M01147X/1 and by the European Commission (H2020-EU) under grants: 688540 and 732631. This work was partially performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DEAC52-07NA27344 (LLNL-CONF-735589).

REFERENCES

[1] [n. d.]. CoMD Proxy App. http://www.exmatex.org/comd.html. ([n. d.]).

[2] [n. d.]. HPCCG Mini Application. https://mantevo.org/packages.php. ([n. d.]).

[3] Fatimah Adamu-Fika and Arshad Jhumka. 2015. An Investigation of the Impact of Double Bit-Flip Error Variants on Program Execution. In *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 799–813.

[4] Rizwan A Ashraf, Roberto Gioiosa, Gokcen Kestor, Ronald F DeMara, Chen-Yong Cher, and Pradip Bose. 2015. Understanding the propagation of transient errors in HPC applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, 72.

[5] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization (CGO 2003)*. IEEE, 265–275.

[6] Jon Calhoun, Luke Olson, and Marc Snir. 2014. *FlipIt: An LLVM Based Fault Injector for HPC.* Springer International Publishing, Cham, 547–558. https://doi.org/10.1007/978-3-319-14325-5_47

[7] Franck Cappello, Al Geist, Bill Gropp, Laxmikant Kale, Bill Kramer, and Marc Snir. 2009. Toward exascale resilience. *The International Journal of High Performance Computing Applications* 23, 4 (2009), 374–388.

[8] Vinay K Chippa, Srimat T Chakradhar, Kaushik Roy, and Anand Raghunathan. 2013. Analysis and characterization of inherent application resilience for approximate computing. In *Proceedings of the 50th Annual Design Automation Conference*. ACM, 113.

[9] Jeffrey A Clark and Dhiraj K Pradhan. 1995. Fault injection: A method for validating computer-system dependability. *Computer* 28, 6 (1995), 47–56.

[10] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5, 1 (1998), 46–55.

[11] Bo Fang, Karthik Pattabiraman, Matei Ripeanu, and Sudhanva Gurumurthi. 2014. Gpu-qin: A methodology for evaluating the error resilience of gpgpu applications. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 221–230.

[12] Mei-Chen Hsueh, Timothy K Tsai, and Ravishankar K Iyer. 1997. Fault injection techniques and tools. *Computer* 30, 4 (1997), 75–82.

[13] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. 1995. FERRARI: A flexible software-based fault and error injection system. *IEEE Transactions on computers* 44, 2 (1995), 248–260.

[14] W-I Kao, Ravishankar K. Iyer, and Dong Tang. 1993. FINE: A fault injection and monitoring environment for tracing the UNIX system behavior under faults. *IEEE Transactions on Software Engineering* 19, 11 (1993), 1105–1118.

[15] Johan Karlsson, Peter Liden, Peter Dahlgren, Rolf Johansson, and Ulf Gunneflo. 1994. Using heavy-ion radiation to validate fault-handling mechanisms. *IEEE micro* 14, 1 (1994), 8–23.

[16] Maha Kooli and Giorgio Di Natale. 2014. A survey on simulation-based fault injection tools for complex systems. In *2014 9th IEEE International Conference On Design & Technology of Integrated Systems In Nanoscale Era (DTIS)*. IEEE, 1–6.

[17] Maha Kooli, Giorgio Di Natale, and Alberto Bosio. 2016. Cache-aware reliability evaluation through LLVM-based analysis and fault injection. In *2016 IEEE 22nd International Symposium on On-Line Testing and Robust System Design (IOLTS),*. IEEE, 19–22.

[18] Ignacio Laguna, Martin Schulz, David F Richards, Jon Calhoun, and Luke Olson. 2016. IPAS: Intelligent protection against silent output corruption in scientific applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. ACM, 227–238.

[19] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 75.

[20] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. 2009. Statistical fault injection: Quantified error and confidence. In *2009 Design, Automation Test in Europe Conference Exhibition*. 502–506. https://doi.org/10.1109/DATE.2009.5090716

[21] Dong Li, Jeffrey S Vetter, and Weikuan Yu. 2012. Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 57.

[22] Man-Lap Li, Pradeep Ramachandran, Swarup Kumar Sahoo, Sarita V Adve, Vikram S Adve, and Yuanyuan Zhou. 2008. Understanding the propagation of hard errors to software and implications for resilient system design. In *ACM SIGARCH Computer Architecture News*, Vol. 36. ACM, 265–276.

[23] Shubhendu S Mukherjee, Christopher Weaver, Joel Emer, Steven K Reinhardt, and Todd Austin. 2003. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 29–40.

[24] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (PLDI '07)*. ACM, New York, NY, USA, 89–100. https://doi.org/10.1145/1250734.1250746

[25] Xiang Ni and Laxmikant V Kale. 2016. FlipBack: automatic targeted protection against silent data corruption. *RTS* 1, m3 (2016), m4.

[26] K. Parasyris, G. Tziantzoulis, C. D. Antonopoulos, and N. Bellas. 2014. GemFI: A Fault Injection Tool for Studying the Behavior of Applications on Unreliable Substrates. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 622–629.

[27] P. Ramachandran, P. Kudva, J. Kellington, J. Schumann, and P. Sanda. 2008. Statistical Fault Injection. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. 122–127.

[28] Vijay Janapa Reddi, Alex Settle, Daniel A. Connors, and Robert S. Cohn. 2004. PIN: A Binary Instrumentation Tool for Computer Architecture Research and Education. In *Proceedings of the 2004 Workshop on Computer Architecture Education: Held in Conjunction with the 31st International Symposium on Computer Architecture (WCAE '04)*. ACM, New York, NY, USA, Article 22. https://doi.org/10.1145/1275571.1275600

[29] Pia N Sanda, Jeffrey W Kellington, Prabhakar Kudva, Ronald Kalla, Ryan B McBeth, Jerry Ackaret, Ryan Lockwood, John Schumann, and Christopher R Jones. 2008. Soft-error resilience of the IBM POWER6 processor. *IBM Journal of Research and Development* 52, 3 (2008), 275–284.

[30] Horst Schirmeier, Christoph Borchert, and Olaf Spinczyk. 2015. Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors. In *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*. IEEE, 319–330.

[31] V. C. Sharma, G. Gopalakrishnan, and S. Krishnamoorthy. 2016. Towards Resiliency Evaluation of Vector Programs. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1319–1328. https://doi.org/10.1109/IPDPSW.2016.187

[32] Vishal Chandra Sharma, Arvind Haran, Zvonimir Rakamari'c, and Ganesh Gopalakrishnan. 2013. Towards Formal Approaches to System Resilience. In *Proceedings of the 19th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*. to appear.

[33] A. Shye, T. Moseley, V. J. Reddi, J. Blomstedt, and D. A. Connors. 2007. Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. 297–306. https://doi.org/10.1109/DSN.2007.98

[34] Anna Thomas, Jacques Clapauch, and Karthik Pattabiraman. 2013. Effect of compiler optimizations on the error resilience of soft computing applications. In *Workshop on Application and Algorithmic Error Resilience*.

[35] Anna Thomas and Karthik Pattabiraman. 2013. Error detector placement for soft computation. In *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*. IEEE, 1–12.

[36] Anna Thomas and Karthik Pattabiraman. 2013. LLFI: An intermediate code level fault injector for soft computing applications. In *Workshop on Silicon Errors in Logic System Effects (SELSE)*.

[37] Rajesh Venkatasubramanian, John P Hayes, and Brian T Murray. 2003. Low-cost on-line fault detection using control flow assertions. In *On-Line Testing Symposium, 2003. IOLTS 2003. 9th IEEE*. IEEE, 137–143.

[38] Nicholas J Wang and Sanjay J Patel. 2006. ReStore: Symptom-based soft error detection in microprocessors. *IEEE Transactions on Dependable and Secure Computing* 3, 3 (2006), 188–201.

[39] Nicholas J Wang, Justin Quek, Todd M Rafacz, and Sanjay J Patel. 2004. Characterizing the effects of transient faults on a high-performance processor pipeline. In *2004 International Conference on Dependable Systems and Networks,*. IEEE, 61–70.

[40] Ute Wappler and Christof Fetzer. 2006. Hardware fault injection using dynamic binary instrumentation: FITgrind. *Proceedings Supplemental Volume of EDCC-6* (2006).

[41] J. Wei, A. Thomas, G. Li, and K. Pattabiraman. 2014. Quantifying the Accuracy of High-Level Fault Injection Techniques for Hardware Faults. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 375–382. https://doi.org/10.1109/DSN.2014.2

[42] Haissam Ziade, Rafic A Ayoubi, Raoul Velazco, et al. 2004. A survey on fault injection techniques. *Int. Arab J. Inf. Technol.* 1, 2 (2004), 171–186.

A ARTIFACT DESCRIPTION: [REFINE: REALISTIC FAULT INJECTION VIA COMPILER-BASED INSTRUMENTATION FOR ACCURACY, PORTABILITY AND SPEED]

A.1 Abstract

The description in this appendix details the software and hardware requirements for deploying our fault injector REFINE presented in the paper. Also, we provide extended information on the experimentation process that compares REFINE to the two alternative, state-of-the-art fault injection tools: LLFI for IR-level fault injection and PINFI which is a binary-level fault injector.

A.2 Description

A.2.1 Check-list (artifact meta information).

- **Program: Several publicly available benchmark programs for evaluation of the fault injection tools: AMG2013, CoMD, HPCCG, Lulesh, XSBench, miniFE, NAS Parallel Benchmarks in C**
- **Compilation: GCC 4.9.2 with optimization level O3 for producing compiler-based tools (REFINE, LLFI) and the Pin-tool PINFI. Benchmark programs are compiled with the compiler implementation used for compiler-based tools REFINE (Clang/LLVM3.9), LLFI (Clang/LLVM3.4). Benchmark programs fed to PINFI are compiled with Clang/LLVM3.9**
- **Data set: Standard datasets provided for each benchmark program and any input arguments are documented**
- **Run-time environment: Chaos Linux 5.5, kernel version 2.6.32**
- **Hardware: Intel Xeon E5-2670, tools support any x64 architecture**
- **Execution: Benchmark programs execute sequentially in a single thread**
- **Output: We use the same methodology for all benchmark programs and tools to classify the output of a program under fault injection as a crash, SOC or benign**
- **Experiment workflow: Install LLFI, install Intel Pin, install REFINE, build benchmark programs from source following the compilation process of each tool, run the profiling step for each tool, execute benchmark programs to obtain the results of FI outcomes (crash, SOC, benign) for each tool**
- **Publicly available?: Yes**

A.2.2 How software can be obtained (if available). The REFINE software is released as open source, accessible in the following URL: https://github.com/ggeorgakoudis/REFINE. We also make public the updated, optimized PINFI Pin-based tool. We use the latest version of LLFI [4], which is publicly available.

A.2.3 Hardware dependencies. Implementation and experimentation is based on the x64 architecture, specifically the processor Intel Xeon E5-2670. Although the tools used support any x64 processor, the results we present are based on the evaluation of this particular target machine.

A.2.4 Software dependencies. REFINE implements a backend pass in the LLVM compiler. Therefore, it needs the source tree of LLVM 3.9 to build. Besides LLVM, the Clang 3.9 frontend is needed for C/C++ benchmark program compilation. LLFI bundles together the Clang/LLVM version 3.4, which is the latest version that LLFI is compatible with. For the Pin-based PINFI tool, we deploy the Intel Pin framework v3.0 (rev. 76991). The rest of software dependencies are the benchmark programs detailed in the evaluation section and the meta information check-list.

A.2.5 Datasets. Table 3 in the evaluation section of the paper details the datasets and input arguments used for all the programs.

A.3 Installation

Section A.2.4 discusses the what software needs to be installed for the deployment of FI tools. The building process of benchmark programs was modified and we present the modifications needed for each tool.

A.3.1 LLFI. LLFI needs to operate on the IR of the compiler as produced by LLVM. We follow the approach proposed by LLFI creators [5] for modifying the build system of each program, usually based on GNU make. Application sources are first compiled to the LLVM IR, then IR optimizations are applied, LLFI instruments the resulting IR and the last step invokes the native machine code compiler that implements any machine-level optimizations. The following Makefile is the one used for compiling the program HPCCG with LLFI (comments are removed for clarity) :

```
CXX = $(LLFI_BUILD_ROOT)/llvm/bin/clang++
CXXFLAGS_LL = -emit-llvm -S -w -fno-use-cxa-atexit
LINKER_LL = $(LLFI_BUILD_ROOT)/llvm/bin/llvm-link
USE_MPI =
CPP_OPT_FLAGS = -O3 -ftree-vectorize
USE_OMP =
SYS_LIB = -lm
TARGET = test_HPCCG
CXXFLAGS = $(CPP_OPT_FLAGS) $(OMP_FLAGS) $(USE_OMP) $(↩
     USE_MPI) $(MPI_INC)
LIB_PATHS = $(SYS_LIB)

TEST_CPP = main.cpp generate_matrix.cpp read_HPC_row.cpp \
compute_residual.cpp mytimer.cpp dump_matlab_matrix.cpp \
HPC_sparsemv.cpp HPCCG.cpp waxpby.cpp ddot.cpp \
 make_local_matrix.cpp exchange_externals.cpp \
YAML_Element.cpp YAML_Doc.cpp
TEST_OBJ = $(TEST_CPP:.cpp=.ll)

$(TARGET): $(TARGET).ll
  ${LLFI_BUILD_ROOT}/llfi/bin/instrument --dir 'llfi' --↩
       cflags "$(CXXFLAGS)" --readable ${SYS_LIB} $<

$(TARGET).ll : $(TEST_OBJ)
  $(LINKER_LL) -S $(TEST_OBJ) -o $(TARGET).all.ll
  ${LLFI_BUILD_ROOT}/llvm/bin/opt -O3 $(TARGET).all.ll -o $@

test:
  @echo "Not implemented yet..."

%.ll : %.cpp
  $(CXX) $(CXXFLAGS_LL) $< -o $@

clean:
  @rm -f *.o  *~ $(TARGET) $(TARGET).exe test_HPCPCG
  rm -rf *.ll llfi*
```

The Makefiles for the rest of the benchmark programs are modified in a similar way.

A.3.2 REFINE. To compile benchmark programs for the REFINE tool, we need to use the Clang/LLVM of REFINE and set the appropriate option flags to the compiler driver (clang or clang++) to enable fault injection. The modification needed for compiling a program include extending the compiler flags for enabling FI and

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

[5] https://github.com/DependableSystemsLab/LLFI/wiki/Get-Started-with-LLFI-Using-Command-Line

the fault injection library in the compilation. We show only the changes needed, again for the HPCCG benchmark, the rest of the Makefile is identical to the original one:

```
CC=$(REFINE_BUILD_ROOT)/bin/clang
CXX=$(REFINE_BUILD_ROOT)/bin/clang++
FI_FLAGS = -mllvm -fi="true" -mllvm -fi-funcs="*" -fi-instrs↩
     ="all"
...
CXXFLAGS = $(CPP_OPT_FLAGS) $(OMP_FLAGS) $(USE_OMP) $(↩
     USE_MPI) $(MPI_INC) $(FI_FLAGS)
...
TEST_OBJ = $(TEST_CPP:.cpp=.o) injectlib.o

$(TARGET): $(TEST_OBJ)
  $(LINKER) $(CPP_OPT_FLAGS) $(OMP_FLAGS) $(TEST_OBJ) $(↩
       LIB_PATHS) -o $(TARGET)

injectlib.o: $(REFINE_BUILD_ROOT)/injectlib/injectlib.c
  $(CC) -O3 -c $< -o $@
...
```

Modifications for the rest of the benchmarks are similar.

A.3.3 PINFI. For PINFI, we use Clang/LLVM3.9 when compiling benchmarks to use the Clang/LLVM suite across tools. Modification in Makefiles are the same with REFINE, omitting the FI flags and the fault injection library.

A.4 Experiment workflow

For experimentation, we collect 1068 samples for each fault injector. This means that considering the 14 different benchmarks and the 3 fault injection tools, there is a total of $1,068 \times 14 \times 3 = 44,856$ experiments. Due to the large experimentation space, we perform the experimentation on a cluster of machines, each node being two 8-core Intel Xeon E5-2670 processors in a dual socket configuration and sharing 32GB of RAM. We fully subscribe a node since benchmarks run sequentially to perform the experiments simultaneously. Moreover, we make sure that concurrent execution of benchmarks does not deplete the memory of the node. Further, we set the timeout to a large value, that is 10× of the profiled execution for each tool, to ensure that any slowdown from concurrency does not cause spurious timeouts, wrongfully classified as crashes. This is verified because we observed only 3 timeout outcomes out of all the experiments.

A.5 Evaluation and Expected Result

Our evaluation is based on rigorous statistical methodologies. The choice of 1,068 samples results in a margin of error of ≤3% for a confidence level of 95%. Also, we compare across tools using statistical inference in the form of Chi-squared testing with a significance level $\alpha = 0.05$.

Table 6 presents the complete results across benchmark applications and fault injection tools obtained through experimentation, used for plotting the bar charts and performing chi-squared testing.

Application	Crash	SOC	Benign
AMG2013			
LLFI	395	168	505
REFINE	254	87	727
PINFI	269	70	729
CoMD			
LLFI	372	117	579
REFINE	136	55	877
PINFI	175	59	834
HPCCG-1.0			
LLFI	320	195	553
REFINE	159	68	841
PINFI	162	77	829
XSBench			
LLFI	55	355	658
REFINE	179	194	695
PINFI	188	203	677
miniFE			
LLFI	420	327	321
REFINE	186	177	705
PINFI	215	162	691
Lulesh			
LLFI	21	4	1043
REFINE	76	2	990
PINFI	76	4	988
BT			
LLFI	224	543	301
REFINE	20	347	701
PINFI	15	363	690
CG			
LLFI	352	0	716
REFINE	201	0	867
PINFI	175	0	893
DC			
LLFI	495	298	275
REFINE	310	154	604
PINFI	347	155	566
EP			
LLFI	181	470	417
REFINE	44	335	689
PINFI	31	341	696
FT			
LLFI	386	70	612
REFINE	104	51	913
PINFI	96	51	921
LU			
LLFI	238	528	302
REFINE	18	386	664
PINFI	17	436	615
SP			
LLFI	268	800	0
REFINE	45	612	411
PINFI	42	626	400
UA			
LLFI	792	136	140
REFINE	98	237	733
PINFI	105	242	721

Table 6: Complete results of outcome frequencies