Quantifying the Accuracy of High-Level Fault Injection Techniques for Hardware Faults

Jiesheng Wei, Anna Thomas, Guanpeng Li, Karthik Pattabiraman
Dependable Systems Lab
University of British Columbia (UBC)
Hardware Errors: Traditional “Solutions”

- **Guard-banding**
  
  Guard-banding wastes power as gap between average and worst-case widens due to variations

- **Duplication**
  
  Hardware duplication (DMR) can result in 2X slowdown and/or energy consumption
Our Research: Application-level Selective Fault-Tolerance

• Add detectors to applications to selectively detect errors causing Silent Data Corruption (SDCs) i.e., incorrect outputs
Application-level Fault Injection

- To obtain coverage estimates for applications
- Iteratively improve coverage based on the errors missed by fault tolerance mechanisms
  - Analyze the errors that are missed by the FTM's

Is obtained coverage sufficient?

- YES
- NO

Insert fault tolerance mechanisms in the application's source code

Inject Faults into application protected with mechanisms
Low-level Fault Injection

• Inject faults into programs at the assembly code level e.g., NFTAPE, FERRARI, GOOFI, Xception

• **Pros:**
  • Accurate at emulating hardware faults in registers, instructions and computation units (e.g., ALUs)

• **Cons:**
  • Difficult to map injection results back to source code
  • Difficult to inject faults into selected source data
High-Level Fault Injection

• Inject faults directly at the source code or similar levels e.g., PROPANE, Relax, Kulfi

• Pros:
  • Easy to map back injection results to source code
  • Ability to inject faults into specific data-types

• Cons:
  • Difficult to emulate hardware faults accurately
High-Level Fault Injection: Reasons for Potential Inaccuracies

• Lack of one-to-one mapping
  • A single source code statement may map to multiple assembly code statements (e.g., pointers)
  • Some source statements have no analogue in the assembly code (e.g., type-cast statements)

• Hidden States
  • Many elements in assembly code cannot be seen in the source code (e.g., stack manipulation code)
High-Level Vs. Low-Level Injectors: Accuracy Comparison

Accuracy

Low-level Injectors

High-level Injectors

How big is this gap in practice?
Related Work

• **Software Faults** [Madeira00][Natella13]
  • Emulate software faults at the assembly code level
  • Inverse of our problem, as software faults occur in the source code level and are more accurate at that level

• **Safety-critical systems error consequences**
  [Skarin-EDCC08][Pattabiraman-DSN08]
  • Examine consequences of not considering faults at the assembly language level in design of FT mechanisms
  • Do not quantitatively measure how much the gap is
This Paper: Research Question

• How **accurate** is fault injection at the **high-level** (i.e., source code or similar levels) compared to fault injection at the **low-level** (i.e., assembly code or similar levels)?

  • For different kinds of failures (e.g., crashes, SDCs)

  • For different kinds of instructions (e.g., loads)
Our Approach

• Build a high-level fault injector to inject faults at the LLVM compiler’s IR level: LLFI

• Build a low-level fault injector to inject faults using Intel’s PIN tool: PINFI

• Compare the outcomes of LLFI and PINFI by injecting similar faults into benchmarks
Fault Model

• Single bit-flip in the destination registers of a single dynamic instruction in the program

• Models transient faults in the computational parts of the processor (e.g., ALU, registers)

• Does not model memory/cache faults – assumes that these are ECC-protected

• Does not model faults in the instruction encoding
Outline

• Motivation and Approach

• LLFI Architecture and Operation

• PINFI Architecture and Operation

• Experimental Evaluation

• Conclusions
LLVM Fault Injector: LLFI

Works at LLVM compiler’s intermediate (IR) code level [Lattner’05] – LLVM widely used in industry
How does LLFI work?

- Start
- Fault injection instruction/register selector
- Instrument IR code of the program with function calls

Compile time

Fault injection executable

Profiling executable

Inject?
- Yes: Custom fault injector
- No: Next instruction

Runtime
Outline

• Motivation and Approach

• LLFI Architecture and Operation

• PINFI Architecture and Operation

• Experimental Evaluation

• Conclusions
PINFI Architecture

• Built using Intel’s PIN tool for dynamic binary analysis [Luk-2005]
• Modifies executable to inject faults at runtime
Corner Cases in x86 Assembly

• Branch conditions: Flags Register

<table>
<thead>
<tr>
<th>LLVM IR</th>
<th>X86 Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>%11 = icmp sle i32 %9, %10</td>
<td>cmp $0xa4, %eax</td>
</tr>
<tr>
<td>br i1 %11, label %bb, label %bb2</td>
<td>jl 4006e0 //sets %rflags</td>
</tr>
</tbody>
</table>

• Floating point operations: Data Width

<table>
<thead>
<tr>
<th>LLVM IR</th>
<th>X86 Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>%3 = fadd double %1, %2</td>
<td>addsd %xmm2, %xmm0</td>
</tr>
</tbody>
</table>
Outline

• Motivation and Approach
• LLFI Architecture and Operation
• PINFI Architecture and Operation
• Experimental Evaluation
• Conclusions
Experimental Setup

• **Fault Injection**
  - Single bit-flip in the result of a dynamic instruction
  - 1000 injections per benchmark, per instruction category

• **Benchmarks**
  - Four SPEC2006: bzip2, libquantum, hmmer, mcf
  - Two SPLASH-2: ocean, raytrace

• **Outcomes**
  - Crash, Hang, Benign and Silent data corruption (SDC)
  - SDCs measured by comparing with golden output
# Fault Injection: Insn. Categories

<table>
<thead>
<tr>
<th>Instruction category</th>
<th>LLFI selection criteria</th>
<th>PINFI selection criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>arithmetic</strong></td>
<td>Instructions that perform arithmetic or logical operations</td>
<td>Instructions that perform arithmetic or logical operations</td>
</tr>
<tr>
<td><strong>cast</strong></td>
<td>Instructions with ‘cast’ opcode</td>
<td>Instructions with ‘convert’ category</td>
</tr>
<tr>
<td><strong>cmp</strong></td>
<td>‘cmp’ instructions</td>
<td>Instructions whose next instruction is conditional branch</td>
</tr>
<tr>
<td><strong>load</strong></td>
<td>‘load’ instructions</td>
<td>‘mov’ instructions with memory as the source and register as the destination</td>
</tr>
<tr>
<td><strong>all</strong></td>
<td>All instructions</td>
<td>All instructions</td>
</tr>
</tbody>
</table>
Results: Overall Failure Distribution
Results: SDCs for all instructions

SDC rates are comparable between LLFI and PINFI for “all instructions”

Error bars are computed at the 95% confidence level
Results: SDCs for ‘cmp’ instructions

SDC rates are comparable between LLFI and PINFI for selected insn categories.

Error bars are computed at the 95% confidence level.
Results: Crashes for all instructions

Crash rates differ widely between LLFI and PINFI for “all instructions”

Error bars are computed at the 95% confidence level
Why do crashes have poor accuracy in LLFI?

• **Pointer computations in LLVM IR**
  - Abstracted away by GetElementPtr Instruction
  - Some pointer computations are a part of the instructions’ encoding in assembly code

• **Mov instructions in x86 assembly code can move data between memory and registers**
  - Represented by loads and stores in LLVM IR
  - Some mov instructions are due to register spills
Outline

• Motivation and Approach
• LLFI Architecture and Operation
• PINFI Architecture and Operation
• Experimental Evaluation

• Conclusions
Conclusion

• Evaluate accuracy of high-level fault injection
  • LLFI\(^1\) as the high-level fault injector
  • PINFI\(^2\) as the low-level fault injector

• Results for accuracy of high-Level injection
  • Accurate for SDC causing errors
  • Inaccurate for crash causing errors

int main() {
    int fact, i, n;
    n = atoi (argv[1]);
    fact = 1;

    for( i = 1 ; i <= n; i++ )
        fact = fact * i;

    print fact;
}

int main() {
    entry:
        n₁ = atoi (argv[1]);
        br BB₁

    BB:
        fact₁ = mul fact₀, i₀
        i₁ = add i₀, 1
        br BB₁

    BB₁:
        i₀ = phi [l, entry] , [i₁, BB]
        fact₀ = phi [l, entry], [fact₁, BB]
        cond = sle i₀ , n₁
        br cond, label BB, label Return

    Return:
        print fact₀ }

insert fault injection function
int main() {
    int fact, i, n;
    n = atoi (argv[1]);
    fact = 1;

    for( i = 1 ; i <= n; i++ )
        fact = fact * i;

    print fact;
}

int main() {
    entry:
        n = atoi (argv[1]);
        br BB1

    BB:
        fact1 = mul fact0, i0
        fi10 = call inject(10, fact1)
        i1 = add i0, 1
        br BB1

    BB1:
        i0 = phi [1, entry], [i1, BB]
        fact0 = phi [1, entry], [fi10, BB]
        cond = sle i0 , n1
        br cond, label BB, label Return

    Return:
        print fact0 }

Replace all uses of original with return val