LLFI and the Art of Fault Injection: Part 1

Karthik Pattabiraman
University of British Columbia
Who am I?

• **Karthik Pattabiraman**
  – PhD from University of Illinois (UIUC)
  – Post-doc at Microsoft Research (MSR)
  – Associate Prof. at UBC

• **Research Interests**
  – Error Resilient Software Systems
  – Software engineering and Security
TAs for the Tutorial

• Abraham Chan
  – Early PhD student at UBC
  – Research interests: IoT dependability
  – Software fault injection

• Guanpeng Li
  – Last year PhD student at UBC
  – Research: application error resilience
  – hardware fault injection
What this tutorial is about?

- Fault injection (both hardware and software)

- A specific fault injection framework: LLFI
  - Developed in my group over the last 5 years
  - Integrated with the LLVM compiler framework
  - Distributed under a BSD-style license (Illinois license)
  - Used in many research papers (by us and others)
  - Adopted by industry (2 companies), national labs etc.
LLFI is an LLVM based fault injection tool, that injects faults into the LLVM IR of the application source code. The faults can be injected into specific program points, and the effect can be easily tracked back to the source code. Please refer to the paper below. NOTE: If you publish a paper using LLFI, please add it to PaperLLFI.bib

Latest commit 5a1889e on Aug 8

nankmr2012 correction made by removing 32 bit machine/OS

- bin revert of #8c1489
- config bug fix
- gui Modified Util.cpp under llvm_pass and InstrumentController.java unde...
- installer added an option for the user to run tests after LLFI installation
- llvm_passes Modified Util.cpp under llvm_pass and InstrumentController.java unde...
- runtime_lib Refer to issue B7
- sample_programs added new sample program
- test_suite removed file to restore test_suite/PROGRAMS to the previous state
- tools minor bufix of PthreadRaceConditionInjector, and added pthread_mutex...
Outline (Part 1)

- Fault Injection Goals and Techniques
- LLFI Features and Philosophy
- LLFI for hardware faults
- LLFI for software faults
- Conclusion
Dependability Evaluation

Dependability Evaluation Techniques

- Model-Based
  - Analytical
  - Simulation

- Measurement-based
  - Fault-injection
  - Real failures

Design → Prototype → Operational
Fault-injection

• Fault-injection (or fault-insertion) is the act of deliberately introducing faults into the system in a controlled and scientific manner, in order to study the system’s response to the fault
  – To estimate coverage of dependability mechanisms (e.g., detection, recovery)
  – Also used to understand inherent fault tolerance
  – To obtain reliability estimates of the system prior to deployment (requires statistical projection)
Fault-Injection Setup

Fault Injection Spec’s
- Injection Strategy: Stress-based, path-based, Random
- Injection Method: by hardware, by software
- Fault Location: CPU, Memory, disk I/O, network I/O, Other I/Os
- Injection Time: load threshold, program execution path, fault arrival rate

Workload Spec’s
- Rates and Mixes Interaction Intensity

System Under Test
- CPU
- MEMORY
- I/O

Fault Injector

Workload Generator

Load Level

Measure
Fault-injection Steps

1. Identify fault-injection points and times
2. Choose workload and platform to inject
3. Start workload on platform with instrumentation
4. Inject fault at the appropriate time and point
5. Collect the outcome of the expt.
6. Compare the outcome with the correct one
Fault-injection: Inputs/Outputs

• Inputs
  – Workload and platform to inject?
  – When and where to inject?
  – How many faults to inject (total)?

• Outputs
  – How many faults were activated?
  – How many faults cause a deviation of the outcome?
  – What is the latency of manifestation?
Measures to Compute

- What fraction of injected faults are activated?
- What fraction of activated faults manifest as failures?
- What are the average activation and failure latencies?
# Software-based Fault-Injection (SWIFI)

**Pros**
- Do not require expensive hardware modifications
- Can target applications and OS errors
- Many hardware faults do not require probes, e.g., register data corruption

**Cons**
- Restricted to inject only faults that S/W can see
- May perturb the workload that is running on the system, resulting in missing many heisenbugs
- Coarser-grained time resolution than h/w
SWIFI: Types

Compile-time
- Modify source code or machine code of the program prior to execution
- Can be used to model software defects
- Requires going thro’ compile-run cycle each time

Runtime
- Modify the program or its data during runtime
- Can be done through the debugger, kernel or with support from compiler
- No need to go through compile-run cycle each time
Compile-time Injection

• Modify program’s code prior to execution
  – Model hardware transient faults in machine code
  – Also, allows for modeling of software errors
  – Typically only inject into the first dynamic instance of an instruction

• Main advantage: Take advantage of static analysis of the code to customize the injection
Runtime Injection

• Advantages
  – Can inject faults without recompiling - speed
  – Faults can occur deeper in the execution. e.g., one-millionth iteration of a loop
  – Fault can depend on runtime conditions. e.g., if memory usage exceeds a threshold, inject fault
Outline (Part 1)

• Fault Injection Goals and Techniques

• LLFI Features and Philosophy

• LLFI for hardware faults

• LLFI for software faults

• Conclusion
Why yet another fault injector?

• **Difficult to customize existing injectors**
  – Inject into specific instructions
  – Inject into a specific variable
  – Inject into specific code constructs

• **Difficult to understand the results**
  – Difficulty in fault injection customization
  – Difficult to study the propagation of errors
  – Difficult to map result back to source code
LLFI

• **A fault injector based on LLVM (http://llvm.org)**
  – Intermediate representation (IR) level injection
  – Hybrid compile-time and runtime injection

• **Features**
  – Easy to customize the fault injection
  – Easy to analyze fault propagation
  – Accurate compared to assembly level injection
LLFI: Hybrid Compile/Runtime Injection

• **Source-level instrumentation of programs**
  - Integrated with a compiler framework, LLVM
  - Precise targeting of selected code constructs

• **Fault-injection is done at program runtime**
  - Avoid going through the compile-cycle every time
  - Ability to use run-time information to inject faults

• **Tracing of faults after injecting them**
  - Map the fault back to the source code and data
LLFI: Workflow

**Compile-time**

- Source code
  - LLVM Compiler
  - IR code
  - Instrument
  - Injection executable

**Run-time**

- Profiling executable
  - Profile
  - Dynamic counts of injection locations

- Run-time options
  - Inject Fault
  - Run injection executable
  - Injection Results
  - Statistics of injected faults
Why LLVM Compiler?

• Supports wide variety of front- and back-ends
• Provides high-level features in the IR code
Factorial Example: Original

```c
1 define i32 @main(i32 %argc, i8** %argv) nounwind {
2 entry:
3 %"alloca point" = bitcast i32 0 to i32
4 %0 = getelementptr inbounds i8** %argv, i64 1
5 %1 = load i8** %0, align 1
6 %2 = call i32 (...)* @atoi(i8* %1) nounwind
7 br label %bb1
8
9 bb:
10 %3 = mul nsw i32 %fact.0, %i.0
11 %4 = add nsw i32 %i.0, 1
12 br label %bb1
13
14 bb1:
15 %i.0 = phi i32 [ 1, %entry ], [ %4, %bb ]
16 %fact.0 = phi i32 [ 1, %entry ], [ %3, %bb ]
17 %5 = icmp sle i32 %i.0, %2
18 br i1 %5, label %bb, label %bb2
19
20 bb2:
21 %6 = call i32 (i8*, ...)* @printf(i8* noalias getelementptr
22 inbounds ([4 x i8]* @str, i64 0, i64 0), i32 %fact.0) nounwind
23 br label %return
24
25 return:
26 ret i32 undef
27 }
```
Factorial Example: Instrumented

```plaintext
1 define i32 @main(i32 %argc, i8** %argv) nounwind {
2   entry:
3       call void @initInjections(i8* getelementptr
4           inbounds ([(1 x 18)* @NameStr, i32 0, i32 0])
5           "%alloca point" = bitcast i32 0 to i32
6       %fi2 = call i32 @injectFault0(i32 2, i32 0, i32 0 "%alloca point")
7       %0 = getelementptr inbounds i8** %argv, i64 1
8       %fi3 = call i8** @injectFault1(i32 3, i32 0, i8** %0)
9       %1 = load i8** %fi3, align 1
10      %fi4 = call i8* @injectFault2(i32 4, i32 0, i8* %1)
11      %2 = call i32 (...)* @atoi(i8* %fi4) nounwind
12      %fi5 = call i32 @injectFault0(i32 5, i32 0, i32 2)
13      br label %bb1
14
15  bb: ; preds = %bb1
16      %3 = mul nsw i32 %fi8, %fi1
17      %fi6 = call i32 @injectFault0(i32 6, i32 0, i32 3)
18      %4 = add nsw i32 %fi1, 1
19      %fi7 = call i32 @injectFault0(i32 7, i32 0, i32 4)
20      br label %bb1
21
22  bb1: ; preds = %bb, %entry
23      %i.0 = phi i32 [ 1, %entry ], [ %fi7, %bb ]
24      %fact.0 = phi i32 [ 1, %entry ], [ %fi6, %bb ]
25      %fi8 = call i32 @injectFault0(i32 8, i32 0, i32 %fact.0)
26      %fi1 = call i32 @injectFault0(i32 1, i32 0, i32 %i.0)
27      %5 = icmp sle i32 %fi1, %fi5
28      %fi9 = call i1 @injectFault3(i32 9, i32 0, i1 %5)
29      br i1 %fi9, label %bb, label %bb2
30
31  bb2: ; preds = %bb1
32      %6 = call i32 (i8*, ...) @printf(i8* noalias getelementptr
33           inbounds ([4 x i8]@.str, i64 0, i64 0), i32 %fi8) nounwind
34      %fi10 = call i32 @injectFault0(i32 10, i32 0, i32 %6)
35      br label %return
36
37 return: ; preds = %bb2
38      call void @postInjections()
39}
```
Features of LLFI

• Easy to customize the fault injection

• Easy to analyze the fault propagation

• Easy to use
Easy Fault Injection Customization

• Fault injection instruction selector
  – Based on instruction type
    • Include: add + cmp
    • Include: all; Exclude: load
  – Based on custom instruction selector

  – Include backward/forward trace
Easy Fault Injection Customization

• Fault injector
  – Common fault injectors
    • Bit-flip, stuck-at-0/1, etc.

  – Custom fault injectors
    • Specified by user as C function
Easy Analysis

• **Trace the value of every instruction**
  – Obtain golden run and fault injection run
  – Can include forward and backward dependencies
  – Can limit the trace for performance reasons

• **Perform a comparison**
  – Data diff:
    • Instruction ID: 20/add: val 3 => 11
  – Control diff:
    • Instruction ID: 22/cmp: 22 -> 23 => 22 -> 24
Easy Analysis

• Graphical output of trace differences as dot file
LLFI: Easy to Use

• Has both a CLI and GUI

• Can be scripted through Python/Yaml interface

• Provides canned scripts for common operations

• Easy installation scripts
LLFI: Easy to Use (Java GUI)
Easy to Use (Web-based GUI)
Outline (Part 1)

• Fault Injection Goals and Techniques
• LLFI Features and Philosophy
  • LLFI for hardware faults
  • LLFI for software faults
• Conclusion
Soft Error

• What:
  – High-energy particle strike
• Where / When:
  – Flip-flop in execution units
  – Memory storage
• Consequence:
  – Cause bit-flip in flip-flop
  – Introduce logic error
Failure Types

• Crash
• Hang
• Silent Data Corruption (SDC)
• Benign
Fault Injection Levels

- Gate level
  - Accurate but slow
- Instruction level (LLVM IR)
  - Easy mapping
  - Closed to both source code and assembly
- Source code level
  - Inaccurate but fast
In LLVM IR

```llvm
<label>:76
%77 = load i32* %i, align 4
%78 = sext i32 %77 to i64
%79 = load i32** %mark, align 8
%80 = getelementptr inbounds i32* %79, i64 %78
%81 = load i32** %80, align 4
%82 = icmp ne i32 %81, 0
br i1 %82, label %99, label %83
```
Setting up the YAML File

defaultTimeOut: 300

compileOption:
  instSelMethod:
    - insttype:
      include:
        - all
      exclude:
        - ret

regloc: destreg

runOption:
  - run:
    numOfRuns:
    100

fi_type: bitflip
Exercise

• What is the SDC rate of CMP instructions?
  • In instruction selector: specify “icmp” and “fcmp”

• Steps:

  1. Navigate to the Hardware folder.

  2. Compilation from C file to IR.
     
     ```
     clang -S -emit-llvm binarytree.c -o binarytree.ll
     ```

  3. Instrumentation
     
     ```
     instrument --readable binarytree.ll -lm
     ```

  4. Profiling
     
     ```
     profile llfi/binarytree-profiling.exe 6
     ```

  5. Fault Injection
     
     ```
     injectfault llfi/binarytree-faultinjection.exe 6
     ```
Fault Injection Result

- Golden output:
  - llfi/baseline

- Program output in fault injection runs:
  - llfi/std_output

- Error output in fault injection runs (Hang/Crash):
  - llfi/error_output

- Fault injection location etc:
  - llfi_stat_output

- LLFI indexed IR
  - llfi/binarytree-llfi_index.ll
Outline (Part 1)

• Fault Injection Goals and Techniques
• LLFI Features and Philosophy
• LLFI for hardware faults
• LLFI for software faults
• Conclusion
Software Fault Model

• What:
  – Common software bugs that are difficult to test using traditional testing

• Why:
  – Developer oversight
  – External environment factors

• Where / When:
  – In the software, during execution
Examples of Supported Faults

- Data Corruption
- File I/O Buffer Overflow
- Buffer Overflow Malloc
- Function Call Corruption
- Invalid Pointer
- Race Condition
Example: Buffer Overflow Malloc Injection

- Difficult to test with unit testing
- Implementation
  - Under allocate malloc and calloc by 40 bytes
  - Example:
    - malloc(1024) -> malloc(984)
    - Subsequent memory accesses may result in buffer overflow
Setting up the YAML File

defaultTimeOut: 30

compileOption:
    instSelMethod:
        <content of inst selector>...
    regSelMethod: customregselector
    customRegSelector: Automatic

runOption:
    - run:
        numOfRuns: 100
        fi_type: AutoInjection

Specify the instruction selector

Set the register selector to Automatic

Set this to AutoInjection
Instruction / Register Selector

compileOption:
  instSelMethod:
    - customInstselector:
      include:
        - BufferOverflowMalloc(Data)
    - funcname:
      include:
        - all
      exclude:
        - main

regSelMethod: customRegselector
  customRegSelector: Automatic

Instruction selector (Related to your fault)

Function selector
Activity

• How do buffer overflow malloc corruptions outside of the main() function affect the application? Perform 100 fault injections.

  – What is the crash/hang rate?

  – What is the SDC rate?

Note: The basic YAML file is set up. You need to make some small changes.
Trace Propagation

tracingPropagation: True

tracingPropagationOption:
  maxTrace: 250
  debugTrace: True/False
  generateCDFG: True

- Trace the register values at every instruction
- Generates a golden and faulty trace file
- Needed for error propagation analysis
YAML File - Trace Propagation

compileOption:
  instSelMethod:
    <content of inst selector>...
  regSelMethod: customregselector
  customRegSelector: Automatic

  tracingPropagation: True
  tracingPropagationOption:
    maxTrace: 250
    debugTrace: True/False
    generateCDFG: True

runOption:
  - run:
    numOfRuns: 100
    fi_type: AutoInjection
Error Propagation Analysis (EPA)

• How much of the faulty trace differs from the golden trace?

• Use the graph visualization tool!
EPA Commands

tracediff [goldentrace] [faultytrace] > diffReport.txt

traceontograph diffReport.txt llfi.stat.graph.dot > tracedGraph.dot

zgrviewer tracedGraph.dot
Outline (Part 1)

• Fault Injection Goals and Techniques
• LLFI Features and Philosophy
• LLFI for hardware faults
• LLFI for software faults
• Conclusion
Conclusion

• LLFI is an easy-to-use and customizable framework for fault injection
  – Both hardware and software faults
  – Hybrid compile-time, runtime-model
  – Easy analysis of fault injection results
  – Released under BSD style license

http://github.com/DependableSystemsLab/llfi
Outline (Part 1)

• Fault Injection Goals and Techniques
• LLFI Features and Philosophy
• LLFI for hardware faults
• LLFI for software faults
• Conclusion
Outline (Part 2)

• LLFI Philosophy and Architecture

• Writing Hardware Fault Injector

• Writing Software Fault Injector

• Applications and conclusions