[Article] Fault injection tools based on Virtual Machines

Fault Injection Tools Based on Virtual Machines

Maha Kooli, Pascal Benoit, Giorgio Di Natale, Lionel Torres
Laboratoire d'Informatique, de Robotique
et de Microelectronique de Montpellier (LIRMM), France
name.surname@lirmm.fr

Volkmar Sieh
Friedrich-Alexander-University
Erlangen-Nuernberg, Germany
Volkmar.Sieh@cs.fau.de

***Abstract*—Transient and permanent faults in complex digital systems used for safety-critical applications may result in catastrophic effects. It becomes therefore extremely important to adopt techniques such as fault injection to observe the behavior of the system in the presence of faults. Several tools have been proposed in the literature that support fault injection. However, few of them allow observing complex computer-based systems. This paper presents current advances in this field, by focusing on Low Level Virtual Machine (LLVM) based fault injectors and FAUMachine. We give an overview of the LLVM environment, and two based fault injection tools: LLFI and KULFI. Moreover, we introduce FAUmachine as virtual machine that supports fault injection in different components of the system (memory, disk and network). We present limitations and difficulties of the tool, and we propose a new implementation that allows injecting faults into the register of the target processor. The paper concludes with a comparison between the fault injection tools based on virtual machine in a first level, and between the LLVM-based fault injection tools in a second level.**

***Index Terms*—Fault Injection, Virtual Machine, FAUmachine, LLVM, LLFI, KULFI**

I. Introduction

The complexity of the systems used in the safety-critical field increases significantly the number of transient and permanent faults in processors. Permanent faults are defined as faults that exist indefinitely in an element if no corrective action is taken. Thus, they model permanent hardware failures such as an ALU that stops working or a cache line that has a stuck-at fault. Transient faults are defined as faults that can appear and disappear within a given period of time during computation. They are caused by events such as cosmic rays, alpha particle strikes or marginal circuit operation caused by noise for instance.

The effect of faults can be dangerous on the behavior and the performance of the system. For this reason, it is important to integrate techniques to observe and control the effects of faults on the system in the early stages of the whole system design process. Fault injection is a powerful and useful technique to assess the system dependability on faults. It is based on the realization of controlled experiments with the goal to evaluate the behavior of the computing systems in the presence of faults. This technique can speed up the occurrence and the propagation of faults in the system in order to observe their impact on the performance of the system [1].

Several fault injection tools have been proposed in the literature. Most of them target only the hardware level of the system. This approach could be costly in term of material since it requires direct interaction with the hardware system. In addition it does not permit to observe complex computer-based systems. In our research, we are interested in evaluating the reliability of the system in an early phase of the system design, and without having information about the characteristics of the used hardware system. Fault injection tools based on virtual machine can be a good solution to achieve our objectives. First, because they permit simulating the computer without having the real hardware system. Moreover, they target hardware faults on the software level, and they allow observing complex computer-based systems, with operating system and user applications. In this paper, we will present some existing tools based on Virtual Machines (VMs), considering the virtual machine classification defined following.

Virtualization is the technology permitting to create a VM that behaves like a real physical computer with an Operating System (OS). It has an enormous effect in today's IT world since it ensures efficient and flexible performance, and permits cost saving from sharing the same physical hardware. The virtual machine where the software is running is called a guest machine, and the real machine in which the virtualization takes place is called the host machine. The words host and guest are used to make difference between the software that runs on the virtual machine and the software that runs on the physical machine.

Considering the notion of virtualization, a classification of VMs into two main categories has been proposed in [2]:

- *System Virtual Machines* provide a complete environment that supports the execution of a complete operating system. System VMs are able to provide a platform to run programs in which the real hardware is not available for use, and to run multiple OS environments concurrently on the same computer with a strong isolation. The virtual machine can provide an ISA that is different from the one of the physical machine. In this situation the whole software is virtualized, therefore the VM has to emulate both the application and the OS code.
- *Process Virtual Machines* is a virtual platform that executes a single process. It is created when the process is started and deleted when it terminates. Its goal is to provide an independent programming environment platform which abstracts away details of the underlying

hardware or operating system, and enables a program to execute in the same way on any platform. Process VMs using different guest and host ISAs are implemented using an interpreter, which fetches, decodes and emulates the execution of individual guest instructions. Since this process is relatively slow, the dynamic binary translation can provide better performance by converting guest instructions to host instructions in blocks rather than instruction by instruction, and saving them in a cache for later reuse.

The rest of the paper is structured as following. Section 2 introduces a process virtual machine named LLVM. The first subsection gives an overview of the tool and its functionalities. The second and the third subsections present two fault injection tools based on LLVM named respectively LLFI and KULFI. Section 3 introduces a system virtual machine named FAUmachine. The first subsection gives an overview of the tool and its functionalities, the second subsection describes the fault model supported by FAUmachine, and the third subsection presents our contribution to implement new feature not supported by FAUmachine, which is fault injection in CPU registers. Section 4 compares in a first level the two fault injection tools based of virtual machine, and in a second level the LLVM-based fault injection tools. Section 5 concludes the paper.

II. LLVM

A. Overview

LLVM (Low Level Virtual Machine) is a compiler framework designed to support transparent, life-long program analysis and transformation for arbitrary programs, by providing high-level information to compiler transformations at compile-time, link-time, run-time, and in idle-time between runs [3].

LLVM research project [4], started in 2000, is developed at the University of Illinois, UrbanaChampaign, with the objective to provide a modern Static Single Assignment (SSA) based compilation strategy, able to support static and dynamic compilation of programming languages. LLVM has been the support for many sub-projects used in academic research.

LLVM uses the LLVM Intermediate Representation (IR) as a form to represent code in the compiler. It symbolizes the most important aspect of LLVM, because it is designed to host mid-level analysis and transformations found in the optimizer section of the compiler. The LLVM IR is independent from the source language and the target machine. It is easy for a front end to generate, and expressive enough to permit important optimizations to be performed for real targets.

Figure 1 describes the LLVM compiler [5]. The front end parses, validates and diagnoses errors in the input code. Then it translates the parsed code into LLVM IR. This IR is sent into a code generator to produce native machine code.

As a source code front ends, the LLVM compiler supports several programming languages, such as C, C++, Objective-C, Fortran, Python, ect. As a machine code backends, it supports many instruction set, such as ARM, MIPS, PowerPC, SPARC, x86/x86-64, ect. This means that LLVM permits to define an abstraction layer to make the information obtained at software level and the information obtained at hardware level, compatible and easily exchangeable.

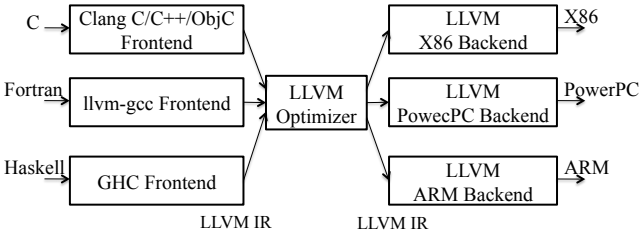

Fig. 1. LLVM Based Compiler [5].

As mentioned above, LLVM is provided with a set of sub-projects and tools for compilation and code optimization. We describe in the following subsections, two fault injection tools based on LLVM.

B. LLFI

LLFI [6] is an LLVM based fault injection tool, that permits to inject faults into the LLVM intermediate level of the application source code. Using LLFI, the faults can be injected at specific program points and data types. The effect can be easily tracked back to the source code. LLFI is typically used to map fault characteristics back to source code, and understand program characteristics or source level for various kinds of fault outcomes. The reason why LLFI injects faults at this level, is that the LLVM intermediate code is at a higher level than the assembly code, and is able to encode more information than the source code. In fact, at the assembly level, it is not easy to track back the fault behavior to the source level. This problem could be solved with a fault injection at the source code level. However, this solution does not allow modeling hardware faults because many hardware faults, that affect some control flow instructions and registers are masked at the lower layers of the system and can not be simulated at the application layer.

The goal behind LLFI is to identify source level heuristics that permits to identify optimal locations for high coverage detectors of faults causing Egregious Data Corruptions (EDC). EDC are application outcomes that deviate significantly from the error-free outcome [6]. Non-EDC are application outcomes with small deviation in output. EDC and non-EDC define the Silent Data Corruptions (SDC), which is the outcomes that result from any deviation from the fault free outcome.

LLFI supports the fault injection of transient hardware fault occurring in the processor, which are the result of cosmic ray or alpha particle strikes affecting flip flops and logic elements. LLFI considers faults in the functional unit (the ALU and the address computation for loads and store. However, faults in the memory components, in the control logic of the processor and in the instructions are not considered by the tool, which

is a limitation of the approach. In the new released version of LLFI, there is consider of permanent faults that occur in the processor, such as stuck-at.

C. KULFI

KULFI [7] (Kontrollable Utah LLVM Fault Injector), developed by Gauss Research Group at School of Computing, University of Utah, Salt Lake City, USA, is an LLVM based fault injection tool, that permits to inject random single bit errors at instruction level. It allows injecting faults into both data and address registers. It permits to simulate faults occurring within CPU state elements, providing a finer control over fault injection. For example, it enables to the user to define some relevant options related to the fault injection mechanism, such as the probability of the fault occurrence, the byte position in which the fault could be injected, the suitable choice whether the fault should be injected into the pointer register or the data register.

KULFI considers the injection of both dynamic faults and static faults. Dynamic faults represent the transient faults and are injected to a fault site randomly selected during program execution. Static faults represent the permanent faults and are injected to a fault site selected randomly before the program execution.

III. FAUmachine

A. Overview

FAUmachine, an open source tool [8] developed in the Friedrich Alexander University of Erlangen-Nuremberg in Germany, is a virtual machine that permits to install a full operating system (Linux, OpenBSD, Mac OS X) and run them as if they are independent computers. It supports CPUs 80286, 80386, pentium, pentium II and AMD64 as microprocessors.

FAUmachine differs from the standard virtual machines, like QEMU [9] or VMWare [10] in many aspects. The main motivation behind the FAUmachine project is to build a virtual machine that provides a realistic hardware simulator able to simulate hardware faults [11]. It permits to inject faults and observe the whole operating system or application software. Thanks to the virtualization, FAUmachine permits a high simulation speed for both complex hardware and software systems [12]. FAUmachine also supports automated tests, including the specification of faults to be injected.

Listing 1. Injecting a stuck-at 1 in bit 0 of the memory address 0xfff0fff

```
architecture behaviour of fi is
 signal err : boolean;
 begin
        process
         begin
          wait for 1000 ms;
                shortcut_bool_out(err,
                        ":pc:mem0:u3",
                        "stuck_at_1/0xfff0fff/0");
                err <= true;
                wait;
  end process;
end behaviour;
```

The fault injection mechanism could be simulated either through a Graphical User Interface (GUI) provided by the virtual machine, or using VHDL scripts where the type, the location, the time, and the duration of fault should be defined (e.g. Listing 1). VHDL is just used as a language to describe the faults in the fault injection experiment. It is not used to describe the whole system and simulate the architecture. Injecting faults using VHDL script is more efficient and yields to significant results because the experiments are automated. In fact, the user has the possibility to define a test bench, which is a set of predefined stimuli and responses that are fed to and awaited from the system under test. In addition, several number of faults could be defined and injected simultaneously during one experiment. FAUmachine gives also the opportunity of running experiments deterministically. It means that each run can be repeated for an arbitrary number of times, with exactly the same system under test conditions as to the temporal flow of events [13].

B. Fault Model

FAUmachine supports the fault injection in several components of the system:

- *Memory*: transient bit-flip faults, and permanent stuck-at and coupling faults, the
- *Disk, CD/DVD drive*: transient and permanent block faults, and transient and permanent whole disk faults, and the
- *Network*: transient, intermittent and permanent send and receive faults.

The fault injection mechanism in FAUmachine is implemented inside the simulators of different components. Each component has a specialized fault type. For example, the simulator of the network interface card permits to specify the percentage of packet loss. The simulator of memory faults permits to define the bit and the address where the fault is injected.

C. Fault Injection in CPU Registers

1) Concept

FAUmachine does not support injecting faults in the CPU registers. However, simulating faults in these locations is interesting for us in our research, since we want to target all the components of the system. Running fault injection experiments in all possible elements of the system enable us to achieve a detailed evaluation of the system reliability. This was our motivation to study the possibility of implementing the fault injection in the CPU registers inside FAUmachine. Implementing the bit flip and the stuck-at faults requires the modification of the Just-In-Time (JIT) compiler of FAUmachine. The C-code of the simulated hardware of FAUmachine tries to reflect the faulty behavior of the real hardware, which makes possible to add new fault injection capabilities without the need of rewriting a big part of FAUmachine.

2) Implementation

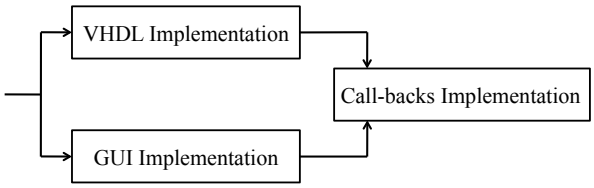

Fig. 2. Implementation of the CPU Registers Fault Injection Process.

The implementation of the whole process is divided into three main tasks as shown in figure 2:

- *VHDL Representation of the fault:* We define the structure of the function needed to implement the VHDL script when setting up the fault injection experiment. The function is defined as following:
  shortcut_bool_out(Sig, PathToComponent, "FaultType-/RegisterName/Bit");
  where:
  - *Sig*: the signal to the fault
  - *PathToComponent*: the path to the instantiated component, which should be the cpu register
  - *FaultType*: the fault type (bit-flip or stuck-at)
  - *RegisterName*: the name of the CPU register where we want to inject the fault (EAX, EBX, ...)
  - *Bit*: is the bit that we want to change the value
- *GUI Representation of the fault:* We implement the user interface that permits to insert the fault parameters. It is added to the existing fault injection interface provided by FAUmachine as a new row with the specific parameters.
- *Call-backs Implementation:* We defined a set of methods that are responsible for the fault injection mechanism. These methods are implemented in the very generic Intel CPU, so that all the x86 CPUs inherit the new fault injection facility. Figure 3 presents the structure describing the methods and the relationships between them.

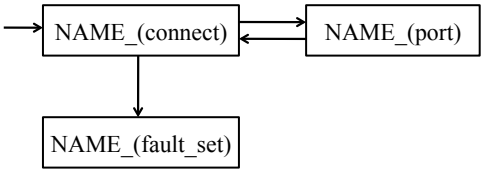

Fig. 3. The Structure describing the Methods and their Relationships.

The function *NAME_(connect)* is called when the shortcut of the fault is established. It stores the register, the bit number and the fault type into the fault structure previously defined. These information are fetched from the shortcut when the function *NAME_(port)* is called. No fault injection is recognized at this level. Fault injection is done when the signal connected is set to 1. The signal change is propagated via the callback functions *NAME_(fault_set)*, which should do the real fault injection. When *NAME_(fault_set)* is called, it knows already which bit to change in which register thanks to the information stored in the fault structure.

3) Discussion

The implementation described above represents only the infrastructure needed when the fault injection function is called. Many other internal mechanisms need to be handled. One challenge is the modification of the JIT compiler.

For the bit flip, the JIT needs to generate slow code only in a small time interval around the time of the bit flip. The overhead of injecting a bit flip in the CPU register will not be so big in average. In figure 4, we compare the compilation process with and without fault injection. Before and after the fault injection, we could use the standard optimized code. When we want to inject the bit flip, the CPU is executing one block. So this block must be splitted into single instructions in order to be able to inject faults at any instruction we want, not only at the beginning of the block. When the bit flip fault is activated, we throw away all code compiled before and generate new single-instruction code. Then we can count instructions and inject the fault into the 100th instruction. We will have a small delay between activating the fault and the real fault injection but this will not be noticed. When the bit-flip is done we can then again throw away all generated single-instruction code and switch back to the standard optimized code generation.

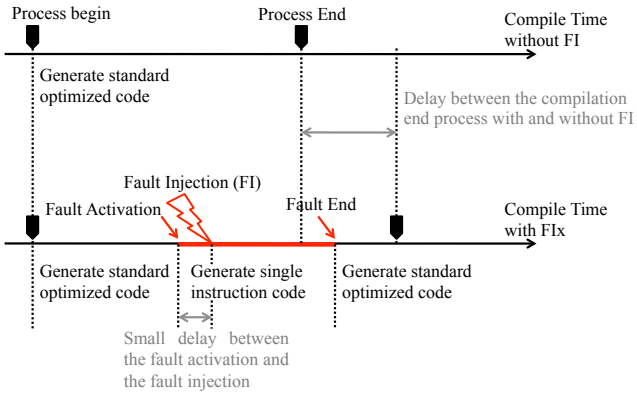

Fig. 4. Compilation Process with and without the Injection of a Bit Flip Fault.

Concerning the stuck-at faults, the JIT needs also to do some more tasks. For example, if we wanted to execute the instruction *"add $1, %eax"* without any fault injection, this would be compiled similar to listing 2. However, when a stuck-at 1 fault is injected into the register %eax in bit 2, this would be compiled similar to listing 3. In general, the stuck-at fault injection will perform well. There is only a small overhead (about less 5%) when writing into the register affected by the fault injection.

Listing 2. Compiling instruction without fault injection

```
T0 = env->reg_eax;
T1 = 1;
do_add();
env->reg_eax = T2;
```

Listing 3. Compiling instruction with injection of stuck-at 1 into register %eax in bit 2

```
T0 = env->reg_eax;
T1 = 1;
do_add();
env->reg_eax = T2;
env->reg_eax |= 1 << 2; /* Fault-Injection */
```

IV. Comparison

Based in our research objectives, we compare in this section the presented fault injection tools based on virtual machines, FAUmachine and LLVM-based in a first step, and the LLVM-based fault injection tools, LLFI and KULFI in a second step.

A. FAUmachine vs LLVM-based Fault Injection Tools

Our research objective is to evaluate the reliability of the system in an early design phase of the system, without previous information about the characteristics of the hardware. We target complex computer-based systems with the operating system and the user application. In a second stage of our research, we want to validate our approach with a finer specification of the hardware system. We evaluate the reliability of the whole system with the observation of the hardware faults effect on the behavior of the global system.

LLVM-based fault injection tools aim to study the effect of faults on a high-level code. It permits to assess the reliability of the system targeting the user application level independently from the hardware system. However, FAUmachine allows injecting hardware faults in the virtual machine and observing their effects on the operating system. Thanks to the virtualization, FAUmachine can be used as a validation tool to observe the behavior of the whole system without having the real hardware system. Unlike LLVM, FAUmachine is not independent from the hardware system but it supports a specific set of processors. As a conclusion, FAUmachine and LLVM-based fault injection tools are complementary. They are both interesting in our research project since they permits to target complex computer-based systems on both the user application and the operating system level.

B. Comparing LLVM-based Fault Injection Tools: LLFI vs KULFI

KULFI and LLFI are open source fault injection tools based on LLVM. They were developed concurrently and they share several similar features [7]. In figure 5, we provide a table comparing the two tools on many aspects. Both LLFI and KULFI support the injection of single bit fault into the intermediate code level of the application (IR). KULFI is easier to control in term of simulating the fault injection experiments. It provides also more precise control over the fault injection process than LLFI.

	LLFI	**KULFI**
Principal Function	Identify source code level heuristics of EDC causing faults	Simulate faults occurring within CPU state elements
Fault Model	Transient hardware faults that occur in the processor Permanent hardware faults (stuck-at)	Static faults: permanent faults, injected during compile time Dynamic faults: transient faults, injected during program execution
Fault Injection	Inject fault in the intermediate code level of the application (IR) Inject a single bit fault into the destination register	Inject fault in the intermediate code level (IR) (LLVM bitcode level) Inject a single bit faults into both data and address registers
Feedbacks	Uses more recent version of LLVM	Provides more precise control over the fault injection process Easier to control

Fig. 5. Comparison between LLFI and KULFI.

V. Conclusion

In this paper, we presented some LLVM-based fault injection tools, that are able to inject single bit faults into the intermediate code level of the application, and to observe their effect on the user application. At the moment these LLVM-based fault injection tools inject transient and permanent faults, such as stuck-at or bit flip faults.

In this paper, we presented FAUmachine, a virtual machine that is able to support fault injection in several components of the system and to observe the effect of faults on the performance and the behavior of the whole system. We presented our contribution to add a new feature to FAUmachine, which is the implementation of fault injection in the CPU registers.

In future work, we aim to adapt these fault injection tools to support the fault model we defined at software level, such as an instruction or a variable used in place of another.

Acknowledgment

This work has been supported by the joint FP7 Collaboration Project CLERECO (Grant No. 611404).

References

[1] H. Ziade, R. Ayoubi, and R. Velazco, "A survey on fault injection techniques," vol. 1, no. 2, pp. 171–186, July 2004.

[2] J. E. Smith and R. Nair, "The architecture of virtual machines," *Computer*, vol. 38, no. 5, pp. 32–38, May 2005. [Online]. Available: http://dx.doi.org/10.1109/MC.2005.173

[3] C. Lattner and V. Adve, "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*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673

[4] C. Lattner. The LLVM compiler infrastructure. [Online]. Available: www.llvm.org/

[5] L. Chris, "LLVM," in *The Architecture of Open Source Applications*, A. Brown and G. Wilson, Eds., vol. I, ch. 11. [Online]. Available: http://www.aosabook.org/en/llvm.html

[6] A. Thomas and K. Pattabiraman, "LLFI: An intermediate code level fault injector for soft computing applications," 2013.

[7] V. C. Sharma, A. Haran, Z. Rakamarić, and G. Gopalakrishnan, "Towards formal approaches to system resilience," in *Proceedings of the 19th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2013, to appear.

[8] (2003-2013) FAUmachine. [Online]. Available: www.FAUmachine.org/

[9] QEMU. [Online]. Available: http://wiki.qemu.org
[10] (2014) VMWare. [Online]. Available: www.vmware.com/
[11] H. Hxer, M. Waitz, and V. Sieh, "Advanced virtualization techniques for FAUmachine," in *Proceedings of the 11th International Linux System Technology Conference*, September 2004, pp. 1–12.
[12] S. Potyra, V. Sieh, and M. D. Cin, "Evaluating fault-tolerant system designs using FAUmachine," in *Proceedings of the 2007 Workshop on Engineering Fault Tolerant Systems*, ser. EFTS '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1316550.1316559
[13] M. Sand, S. Potyra, and V. Sieh, "Deterministic high-speed simulation of complex systems including fault-injection." in *DSN*. IEEE, 2009, pp. 211–216.