A Fault Injection System Based on QEMU Simulator and Designed for BIT Software Testing

LI Yi, XU Ping
Key Laboratory of Science and Technology on Reliability and Environment Engineering
Beihang University
Beijing, China
liyi@dse.buaa.edu.cn, pingxu@buaa.edu.cn

WAN Han
State Key Laboratory of Virtual Reality Technology and Systems
Beihang University
Beijing, China
wanhan@cse.buaa.edu.cn

***Abstract*— An important step in the development of dependable systems is the validation of their fault tolerance properties. Fault injection has been widely used for this purpose. This paper presents a simulator implemented fault injection and monitoring environment based on the QEMU platform, called BitVaSim, which is targeted for the embedded development boards equipped with PowerPC or ARM processor together with Built-In Test software operating environment. BitVaSim takes advantage of simulation and do no harm or irruption to either the real hardware or the software, in addition, all the simulated parts are reachable so that more fault modes are available to achieve. BitVaSim uses abstract key-value pairs to describe the functional fault modes, and then simulates the hardware board as while as realistic faults incurred by hardware into the simulator, in order to monitor the activation of the faults and their impact on the target system especially the BIT system behavior in detail. Fault injection interfaces are configured to implement failure mode matching and fault conditions triggering to inject faults on demand in simulator runtime. Faults injected by BitVaSim can affect any process running on the target system (including the kernel), and it is possible to inject faults in applications for which the source code is not available. Experimental results are presented to demonstrate the accuracy and potential of BitVaSim in the evaluation of the dependability properties of the complex computer systems and the BIT system.**

Keywords- Built-In Test; Fault Injection; Fault Modeling; QEMU; Simulator; Software Testing

I. INTRODUCTION

Built-In Test (BIT) is a mechanism that permits a machine to test itself, and most embedded systems run diagnostic tests to check the health of the system. Furthermore, diagnostics are also used to confirm a fault that might have been detected during normal operations. Almost all avionics and safety-critical devices are now incorporate BIT in order to enhance safety and reliability. Moreover, the core function of BIT system is performed by BIT software. This in turn is driving research to support the evaluation of dependability for the BIT system. [1]

Fault injection technology is an effective way for BIT software validation, which implemented by manually inject faults to the system incorporates BIT. After observing whether BIT can detect and isolate the faults, we can give a conclusion that how the BIT software matched the design requirements, and then suggest the improvements for the design. [2]

Recently, the researches on fault injection techniques can be grouped into hardware-based, software-based and simulation/emulation-based fault injection method [3][4]. The advantages and disadvantages of them have been discussed in former papers [5][6][7]. Here we focus on these fault injection techniques from BIT software's dependability point of view. To meet BIT software testing requirements, the testing process is not allowed to interfere into the tested targets, including underlying hardware board, operating environments, and BIT software itself. Since traditional fault injection techniques have specific deficiencies on the particularity of BIT software testing, there are growing demands for effective fault injection techniques [8][9] to be applied in the design of BIT systems, especially the BIT software.

QEMU [10][11] is a versatile emulation platform with support for numerous target architectures like x86, ARM, MIPS and allowing to run a variety of unmodified guest operating systems. This paper outlines a fault injection system based on QEMU platform which simulates the whole embedded hardware environment to support BIT software and run it as guest application. The evaluation proves that our method fulfills the general requirements in BIT software testing. Furthermore, this system enhances the flexibility and the robustness of the fault injection tools with economical resource cost. In this method, fault injection parameters such as fault models, fault locations, and temporal/spatial characteristics of the fault can be flexibly configured.

The rest of this paper is organized as follows. Section II gives a brief description to the design of the simulation environment. The implement of the simulator based fault injection system including the steps of fault modeling, fault simulation and fault injection is described in Section III. And then, preliminary experiments are given in Section IV that demonstrates the effectiveness of this fault injection system. Finally, Section V concludes the paper and our future work are prospected.

II. SETUP THE SIMULATION ENVIRONMENT

A. Full-System Simulator

The full-system simulator QEMU provides a solution to simulate the entire target hardware system, including the

Published by Atlantis Press, Paris, France.

processor cores, memories, interconnection buses, and some of the peripheral devices such as NIC, external bus interface, serial devices, CD-ROM, etc.. That is why the complete software stacks from real systems can run on the simulator without any modification. The full system simulation can provide a controlled environment, such as good control over the time and fault location. Especially it provides good observability over the system internal state and behavior. Moreover, it provides opportunity to comprehensively evaluate the vulnerability of different micro-architectural structures against different fault models without undesired changes or intrusions in the testing system.

B. Functional Modules

In BitVaSim, QEMU is heavily modified, and additional five modules are integrated into it. These modules are modeling and configuration module, fault simulation module, fault injector module, monitor and feedback module, so as to make it has fault injection and fault information monitoring capabilities.

1) Modeling and configuration module: this module the configures target hardware platform to simulate and parses fault models into fault sequences expressed in corresponding data structure.

2) Hardware simulation module: core module of the simulator, it simulates the behavior of the hardware that composed of the target hardware board, including the processor core, memories and peripheral devices. Therefore, it can execute the application binary and generate the corresponding response.

3) Fault simulation module: we simulate the processor's hardware faults in this module. By accessing and modifying the processor state through BitVaSim interfaces this module simulates the fault occurrence.

4) Fault injection module: the fault injector reads the fault sequences from the configuration file, then using "fault checkpoints" to specific time and places to inject faults, and finally the faults are injected according to the trigger conditions.

5) Monitor and feedback module: it provides the ability for simulator to monitor and control the fault inject process as well as the simulator runtime information.

C. Simulate the Target Hardware Platform

The Guest OS and applications are designed for specific embedded hardware board, which must be implemented in the simulator. Simulate a hardware board go through with the following steps.

1) Instantiate CPU cores: select the target CPU model and call cpu_arm_init/cpu_ppc_init to instantiate a ARM or PowerPC CPU core into the simulation routine for the target mother board.

2) Define address map: assign address for memory and I/O devices with calling the corresponding functions.

g_new and memory_region_init_ram is used to malloc memory space for RAM and initial the memory region, which can be divided into sub-regions with the help of function memory_region_add_subregion.

memory_region_init_io is used to initial memory region for I/O, that all the read/write operations to this address area are performed by its binding callback function.

3) Wire up all the devices: qdev_create and qdev_init may help to create, and then initial a known device.

Some other devices may be wired up by calling their specific interface functions, such as sysbus_create_simple to create a simple device on system bus and so on.

4) Load kernel/OS images: arm_load_kernel is called to load an ARM kernel file in the simulator, on the other hand, load_elf is used for load an ELF image file.

After all, this board should be registered by qemu_register_machine with board info as its argument. So that the hardware platform can be simulated by specific its board name and CPU architecture when launching QEMU program.

III. BUILD THE FAULT INJECTION SYSTEM

To implement a scalable fault injection system for BIT software testing, we use software modeling to define the fault model in order to deploy the fault injection test effectively. General hardware faults are simulated in the simulator and organized with inline functions, which can be invoked by injection module, meanwhile perform the fault function. Furthermore, users can extend BitVaSim to inject customized fault if they can comply with the fault specification define below. On the other hand, it is very important to control and monitor the running simulator with fault injection, including obtains the experiment result feedback. This feature will help to test the BIT system effectively.

A. Fault Modeling in XML Language

XML is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. Using XML-based formalization approach to build fault models can decouple the experiment UI from tools, and it is an effective way to reduce complexity and decouple the testers with developers [12]. Furthermore, the faults which are intended to be injected can be pre-configured, make it possible for automatically carry out fault injection experiment and gets the results feedback conveniently.

The basic characteristics of fault-set must be considered how to model a wide variety of fault modes. Keywords such as fault location, fault target, fault mode, fault parameters and fault time, are extracted. All hardware faults can be portrayed by the permutations and combinations of these keywords. The fault mode characterization is carried out by enumerating where, when, for how long the fault needs to be injected and what kind of corruption has to be made.

- Fault definition: using XML element tag to define the fault keywords and store their attributes' value.
- Document format constraint: using XML schema document to express a set of rules. An XML document must conform to the schema, in order to consider whether it is a 'valid' expression according to the schema.

Published by Atlantis Press, Paris, France.

- Build fault model in XML schema: build serialized fault models stored by XML document based on the definition and constraint mentioned above.

TABLE I. shows all the XML elements and attributes used for faultset and injection process definition.

TABLE I. XML ELEMENTS AND ATTRIBUTES FOR FAULTSET DEFINITION

Label	***Description***
<INJECTION>	start describe a injection process, and faults should be included as sub key-elements
<fault>	start describe a fault
id="N", N is a natural number	the fault ID
<component>	target component to inject this fault, e.g. "CPU"
<target>	target point to inject the fault, e.g. "GPR01"
<mode>	fault mode, the manner by which a fault is observed
<params>	fault parameters, e.g. "address", "bit", "err-code"
<trigger>	how to trigger this fault
<time>	fault injection time
type="[permanent | intermittent | transient | specific]"	fault type, e.g. permanent, intermittent, transient or specific factor
<duration>	the duration time of intermittent fault
<interval>	the interval time of intermittent fault
<when-begin>	the start time of transient fault
<when-over>	the end time of transient fault

B. Fault Simulation Functions in Simulator

All the hardware faults simulated in the simulator is grouped into classes [13]: data storage, control, and I/O fault. As well as, their behaviors or equivalent effects are simulated in different groups of function.

1) Data Storage Fault: the data storage units are the individual flip-flops, registers, and memory cells.

The simulated hardware faults for these data storage units are stuck at 0 or 1, bit-flip, coupling. Therefore, the fault simulated are as follows: set the faulty bits to zero or one; flip the faulty bit in storage; extraneous write the coupled bit in storage cell.

In addition, the simulation function generates a fault mask according to the fault mode, and then conducts AND, OR, XOR operation between the mask and the faulty bit(s). The sample usage of fault masks and operations are enumerated as follows.

$\text{memory} \rightarrow \text{ram}[addr_1]$ & = 0xFFFFFDFF // for stuck-at-0
$\text{memory} \rightarrow \text{ram}[addr_2]$ | = 0x00000080 // for stuck-at-1
$\text{memory} \rightarrow \text{ram}[addr_3]$ ^ = 0x00000003 // for bit-flip

2) Control Fault: several control fault are developed, including incorrect instruction, extraneous instruction, illegal interruption or exception, etc..

To simulate these control faults, the functions are developed to reach relevant fault effects: alternative the decode statement, not execute certain statement or execute commands instead of the correct one. For example we simulate an illegal interruption through the fault function with the following algorithm body.

$\text{cpu}_{\text{arch}} \rightarrow \text{excp\_index} := \text{excp\_index}$
$\text{cpu}_{\text{arch}} \rightarrow \text{error\_code} := \text{error\_code}$
$\text{do\_interrupt}\ (cpu_{arch})$

3) I/O Fault: each I/O device is abstracted by a data structure in the simulator which registers its initialize, interruption and read/write functions. So we simulate I/O faults in these entrances interface. Fault modes for I/O device including initialization failed, read/write error, bad interrupt handling and so on.

C. Fault Injection in Simulator Runtime

According to the parameters like fault location, fault occur time, we set up "fault checkpoints" at the places where may occur faults in simulator. TABLE II. lists some primary fault checkpoints.

TABLE II. FAULT CHECKPOINTS SETTING

Fault checkpoint	***Corresponding fault modes***
Accessing registers	Register bit-flip
	Register bit stuck-at-0
	Register bit stuck-at-1
	Register extraneous load
	Register miss load
CPU execution	Incorrect instruction
	Extraneous instruction
	Illegal interruption / exception
Accessing memory	Memory bit-flip
	Memory bit stuck-at-0
	Memory bit stuck-at-1
	Memory cells coupled
I/O device initialize	Device initial error
I/O operations	Read/write error

The basic fault injection workflow is shown in Figure 1.

- The simulator inspects the fault sequence that intends to be injected at the runtime: when the simulation arrives at a fault checkpoint such as read/write register, ALU computation, and etc... Then, search in the customized fault sequence to find whether there contains associated faults need to be injected.
- Pattern recognition: judge whether there's a fault need to be injected at this fault checkpoint.
- Checking with fault trigger conditions: comparing the fault trigger condition (e.g. trigger type, fault inject time) with associated parameters from simulator's runtime environment (e.g. program counter, execution time), inject the fault when satisfy the condition.

- Dispatch corresponding fault simulation procedure to perform fault injection: the fault can be injected by dispatching fault simulation procedure when the two testing step mentioned above have been passed.

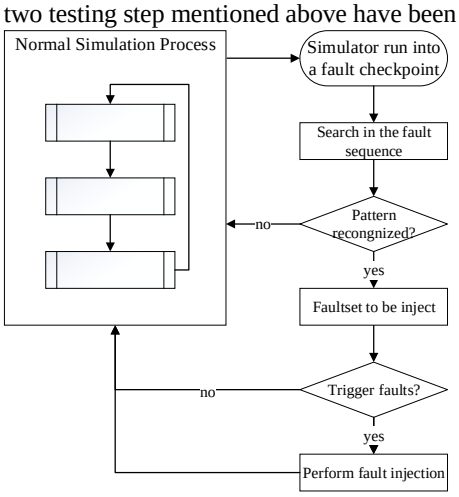

Figure 1. Fault Injection Workflow

The faults can be triggered by pre-configured fault sequence including fault time and other useful parameters. In order to simulate the random fault, the user can trigger fault manually through the fault monitor interface when the simulator is running.

D. Fault Monitoring in Control Mode

The simulator's running status and fault injection messages can be traced by the simulator monitor. The original QEMU monitor is a lightweight, text-based command line interface, through which we query the status of virtual CPU, memory, devices, etc.. Furthermore, we extended the commands of QEMU monitor to query fault sequence and trigger a fault manually.

```
(BitVaSim) fault reload conf/usecase.cfg.xml
The configuration file is successful reloaded.
(BitVaSim)
(BitVaSim) info faults
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
mode: register:stuck-at-1
component: CPU
target: CP15:c1_sys
PERMANENT
params:
        bit: 0000000000000001
        bit: 0000000000000004
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
mode: memory-cell:bit-flip
component: MEMORY
target: Nand Flash
TRANSIENT, start: 200000, end: 202000
params:
        address: 000000001000abcd
        bit: 0000000000000008
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

-----------------------Statistics-----------------------
        FAULT MODE              |       COUNTS
------------------------------------------------------------
      register:stuck-at-1       |            1
     memory-cell:bit-flip       |            1
------------------------------------------------------------
(BitVaSim)
```

Figure 2. Example for the Usage of Fault Monitoring

The command 'info faults' is used to query all the injected faults; as while as, 'fault query [fault-mode]' is used to query those injected faults under specific fault-mode; Moreover, the command 'fault reload [file-path]' is used to reset/add/cancel all or part of injected faults by reload the fault configuration file.

IV. EXPERIMENTS AND EVALUATION

Generally, the CPU Built-In Test is achieved by functional testing, in other word, run a program on the CPU with its full set of instructions, and then compares the resulting output with the standard result.

We use the program bitcount in MiBench benchmark to verify whether the simulator can detect the fault that has been injected to it. Here we injected register fault and fast IRQ fault into BitVaSim and observed the results. As shown in Figure 3. , the benchmark's running result on the simulator that has injected these faults is compared to the standard output. Running results show the differences between the simulator with faults injection and the simulator without faults injection. Furthermore, this can be used to verify the correctness of BIT program.

```
Bit counter algorithm benchmark

Optimized 1 bit/loop counter          > Time:   0.060 sec.; Bits: 18563087
Ratko's mystery algorithm             > Time:   0.020 sec.; Bits: 17272864
Recursive bit count by nybbles        > Time:   0.070 sec.; Bits: 17116098
Non-recursive bit count by nybbles    > Time:   0.020 sec.; Bits: 18244704
Non-recursive bit count by bytes (BW) > Time:   0.010 sec.; Bits: 18730970
Non-recursive bit count by bytes (AR) > Time:   0.020 sec.; Bits: 16962481
Shift and count bits                  > Time:   0.090 sec.; Bits: 17759895

Best  > Non-recursive bit count by bytes (BW)
Worst > Shift and count bits
```

(a) Sample output

```
Bit counter algorithm benchmark

Optimized 1 bit/loop counter          > Time:   0.120 sec.; Bits: 19221181
Ratko's mystery algorithm             > Time:   0.060 sec.; Bits: 15758605
Recursive bit count by nybbles        > Time:   0.080 sec.; Bits: 18877205
Non-recursive bit count by nybbles    > Time:   0.070 sec.; Bits: 15711469
Non-recursive bit count by bytes (BW) > Time:   0.060 sec.; Bits: 17835885
Non-recursive bit count by bytes (AR) > Time:   0.060 sec.; Bits: 15311271
Shift and count bits                  > Time:   0.280 sec.; Bits: 16724232

Best  > Ratko's mystery algorithm
Worst > Shift and count bits
```

(b) Fault output

Figure 3. Running Benchmark Program for CPU Built-In Test

On the other hand, we observe the simulator's ability in the exception injection. In this observation, several exceptions, such as illegal instruction exception and interruption, have been injected into O.S. which is running on the simulator. As shown in Figure 4. , here the program is running on the simulator and then the exceptions are injected into the system. The results show that the machine self-test abnormalities and data storage exception in the simulator level injection will inevitably lead to fatal operation system failure. In addition, the exception in the system call and self-trapping will destroy the application running on the O.S., which leads to its failure. While external input interrupt will not produce the damaging results.

Published by Atlantis Press, Paris, France.

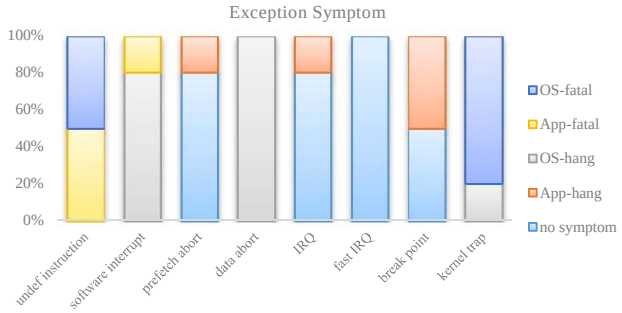

Figure 4. Fault Injection Experiment on Exception Symptom

Moreover, memory test is undergoing to validate all the faults injected to specific memory addresses can be report in this way. Here we inject several faults to memory, stuck-at-1 to 3rd bit (00000008) of memory address 0x00b00004, stuck-at-0 to 5th (00000060) and 6th bit (00000070) of 0x00f0f0f0, and bit-flip to its 7th bit (00000080). The Figure 5. shows that all these faults are figured out and reported with failing address and error bits.

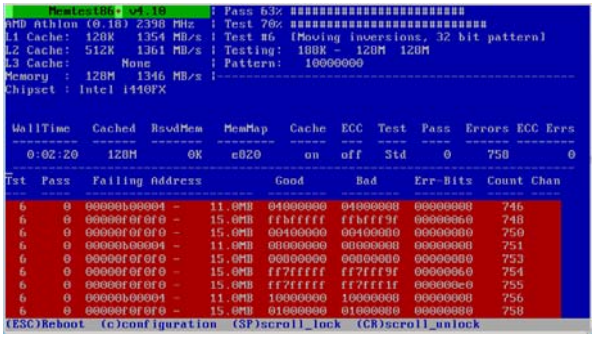

Figure 5. Fault Injection Experiment on Memory Test

Finally, we have tested all the faults simulated in BitVaSim which can be configured and injected correctly when the simulator is running. Furthermore, the faults to be injected in an experiment can be configured and queried by monitor as well. Under the help of fault monitor and guest operating system, we can detect the injected faults and observe their effect to the system.

V. CONCLUSION AND FUTURE WORK

This paper proposed a QEMU-based fault simulation and injection method, which can configure the fault injection test according to the users' requirement. BitVaSim executes tests, and automatically inject faults into hardware devices in a virtual machine. We have implemented a prototype BitVaSim using QEMU together with the fault specification that written in XML. In addition, unmodified operating systems and applications, especially the BIT system can run on the prototype without intrusion. This method also has the whole control to the entire environment, which contributes to the fault injection process's monitor and results' efficient feedback. With the dynamic binary translation, BitVaSim results in a 5%~7% reduction in performance in comparison to a pure QEMU execution, due to the fault injection capability.

In the future, we intend to consider the reproducibility of the system test and to design a fault injector library for more easily injecting and testing. It's expected to build a robust simulator-based fault injection system that fulfills the general requirements in BIT software testing, and then introduce this method in Prognostic and Health Management (PHM) system validation.

ACKNOWLEDGMENT

This research was supported by the Technological Foundation Project of China National Defense Science and Engineering Bureau under grant No.Z132012A004.

REFERENCES

[1] Wang Yichen, Xu Ping. "Build-In-Test Design and Test for Embedded Software". Computer Engineering. 2009, Vol.35, No.17.

[2] Xu Ping, Kang Rui. "The Research of Fault Injection System's Framework in the Testability Experiment Validation". Control Technology. 2004, 23(8): 12-14.

[3] Haissam Ziade, Rafic Ayoubi, Raoul Velazco. "A Survey on Fault Injection Techniques". The International Arab Journal of Information Technology, Vol. 1, No. 2, July 2004.

[4] Mario Garcia Valderas, Marta Portela Garcia, Raul Fernamdez Cardenal, Celia Lopez Ongil, Luis Entrena, "Advanced Simulation and Emulation Techniques for Fault Injection". IEEE Int. Symp. Industrial Electronics ISIE 2007 (2007), pp. 3339-3344.

[5] Jean Arlat, Yves Crouzet, Johan Karlsson, Peter Folkesson, Emmerich Fuchs, Günther H. Leber. "Comparison of Physical and Software-Implemented Fault Injection Techniques". IEEE TRANSACTIONS ON COMPUTERS, VOL. 52, NO. 9, SEPTEMBER 2003.

[6] Sun Junzhao, Wang Jianying, Yang Xiaozong. "The Present Situation for Research of Fault Injection Methodology and Tools". JOURNAL OF ASTRONAUTICS. Vol.22, No.1, Jan. 2001.

[7] Mei-Chen Hsueh, Timothy K. Tsai, Ravishankar K. Iyer. "Fault Injection Techniques and Tools". IEEE Journals, Volume: 30, Issue: 4. 1997.

[8] Sand. Matthias, Potyra. Stefan, Sieh. Volkmar. "Deterministic High-Speed Simulation of Complex Systems Including Fault-Injection". In: Kaaniche, Mohamed (Ed.): Proc. of the 39th Intern. Conf. on Dependable Systems and Networks ((DSN 2009), Estoril, Portugal, 29.06. - 02.07.2009). 2009, pp. k.A..

[9] Michael Le, Andrew Gallagher, Yuval Tamir. "Challenges and Opportunities with Fault Injection in Virtualized Systems". First International Workshop on Virtualization Performance: Analysis, Characterization, and Tools Austin, Texas, April 2008.

[10] QEMU User Manual. Website, 2012. http://wiki.qemu.org/Manual/.

[11] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator". USENIX 2005 Annual Technical Conference, FREENIX Track, Pp. 41–46 of the Proceedings.

[12] A. da Silva, J.F. Martinez, L. Lopez, A. B. Garcia, V. Hernandez. "XML Schema based Faultset Definition to Improve Faults Injection Tools Interoperability". Third International Conference on Dependability of Computer Systems, 2008. Digital Object Identifier: 10.1109/DepCoS-RELCOMEX.2008.26. Publication Year: 2008, Page(s): 39-46.

[13] Charles R. Yount, Daniel P. Siewiorek. "A Methodology for the Rapid Injection of Transient Hardware Errors". IEEE Transactions on Compters, Vol. 45, No. 8, August 1996.