Proceedings of the International Multiconference on
Computer Science and Information Technology pp. 555–558

ISBN 978-83-60810-22-4
ISSN 1896-7094

Collecting program execution statistics with Qemu processor emulator

Sławomir Chyłek
Institute of Computer Science
Warsaw University of Technology
ul. Nowowiejska 15/19, Warsaw 00-665, Poland
Email: s.chylek@ii.pw.edu.pl

***Abstract*—This paper addresses the problem of improving efficiency of software implemented fault injection experiments. It is especially focused on the problem of collecting program execution statistics. The classical SWIFI depends on operating system's debugging API. We introduce a new method which is based on Qemu—a versatile processor emulator. By altering Qemu's dynamic code translation procedure we are able to collect execution statistics significantly faster than in the classical solutions. The achieved performance gain is illustrated with experimental results.**

I. Introduction

FAULT injection is a very important technique to study and improve system dependability. It gives the ability to simulate faults conditions [1] and study their effects.

Injecting faults can be realized either in hardware or in software. Hardware solutions consist of techniques like heavy-ion radiation, pin-level injection or electromagnetic interferences [2]. These solutions are costly, because they require specially designed systems or dedicated equipment. Moreover, heavy-ion radiation and electromagnetic interference have a very low repeatability, thus they are inadequate for tracing fault effects. These characteristics make hardware solutions inappropriate for everyone who intends to start software reliability tests.

On the other hand software solutions are easier to implement and assure high experiment controllability and observability. In this category we can find solutions simulating faults in low-level hardware models such as VHDL models [3]. The other group consists of runtime fault injectors [1], [4], [5], but they have limited capabilities like the lack of access to hardware hidden registers [6]. Nevertheless according to [2] both of hardware and software techniques can produce different results, yet they should be considered complementary.

Developed in our Institute FITS framework [7] is a software implemented fault injector which can simulate various hardware faults like bit-flips or stuck-at-0/1 in processor's registers and memory. Improving software with FITS is a complex and time consuming process. For every tested program it is required to run it many times to collect necessary information. We distinguish two types of a program run:

This paper was realized within Ministry of Science and Higher Education grant 4297B/T02/2007/33. I would like to thank Prof Janusz Sosnowski and Dr Piotr Gawkowski for fruitful discussions during the development of this research.

- Golden Run – a referential run during which execution statistics are collected and the correct program's result is recorded,
- Experiment Run – a run with a fault injected; execution statistics are collected on demand; a result is obtained to compare with the correct one and the execution path altered by the fault may be investigated.

After a Golden Run a series of Experiment Runs is needed to detect weak points of a program. Of course after introducing a modification into the tested program a Golden Run with following Experiment Runs have to be conducted again to verify if the modification is beneficial. Typically improving a program consists of testing many modifications and to make the whole process efficient it is crucial to minimize the overhead of every tested program run.

The core of FITS relies on operating system's debugging framework. This approach introduces some restrictions and may cause performance issues described in Section II. To improve the performance of Golden Run we have developed a new method that relies on the dynamic code translation. In this paper we outline the architecture of the new method compared to FITS's original solution, present achieved experimental results and propose directions for extending this approach.

The rest of this paper is organized as follows. In Section II we present FITS's architecture for conducting Golden Runs and point out its drawbacks. Section III introduces Qemu-based method and Section IV presents experimental results. In Section V we outline further possibilities for utilizing Qemu in fault injection techniques and Section VI will conclude this paper.

II. Golden Run under FITS

FITS is a software solution that utilizes operating system's debugging framework – in case of Windows it is Win32 Debugging API and in case of Linux it is PTrace API [1]. The general method of injecting a fault in the Experiment Run consists in:

- setting up a breakpoint at desired execution moment,
- running the tested process with FITS as monitoring process,
- during the breakpoint callback injecting a fault as altering the tested process with debugging API (e.g. overwriting registers or memory values).

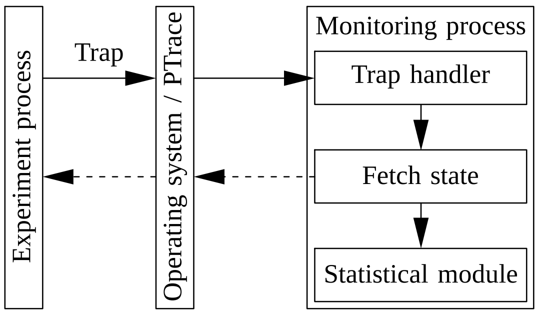

Fig. 1. Collecting processor's state in FITS

During a Golden Run statistical data of a program execution are collected. For example, these statistics include details of execution for every instruction, counters for all registers reads and writes or even the statistical of 1's and 0's in instructions arguments. To collect these data the tested program runs in the trace mode, so a debugging API callback is invoked after every executed instruction. During every callback FITS as a monitoring process investigates the state of registers in the tested program using debugging API and processes that information in a statistical module. Fig. 1 presents the diagram of this architecture.

This approach has two major drawbacks. The first one is the step execution of the tested process. It is a performance issue, because of halting execution after every instruction to invoke costly data-processing callback through debugging API. The impact is considerable because a Golden Run is approximately 20 000 times slower than executing the program normally.

The second issue is an operating system dependence. Utilizing operating system's debugging API restricts this solution to testing processes only. It is impossible to inject faults into the operating system, although it is a very interesting field for research. For example, using just FITS we are unable to answer a question what kinds of faults in kernel-space can propagate into user space processes.

III. Golden Run under Qemu-based solution

To address issues mentioned in Section II we have developed a new method for collecting statistical execution data which is based on Qemu process emulator. Qemu is a fast and versatile emulation platform with support for numerous target architectures like x86, ARM, SPARC and already has been utilized in scientific experiments [8]. Qemu's performance is achieved with dynamic code translation. The general concept consists in translating every emulated target instruction into a set of native instructions which modify the emulated processor's data structure instead of real processor registers. The support for multiple target architectures is realized with a series of backends for reading target's instructions and emulating their results on the processor's state data structure. The unique feature of Qemu is the ability to emulate not only the whole system but also a single process. In a single process emulation mode Qemu is about 20 times slower than a normal

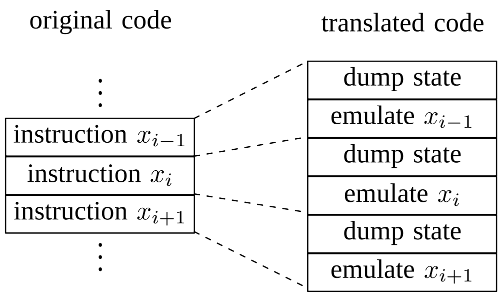

Fig. 2. Dynamic code translation

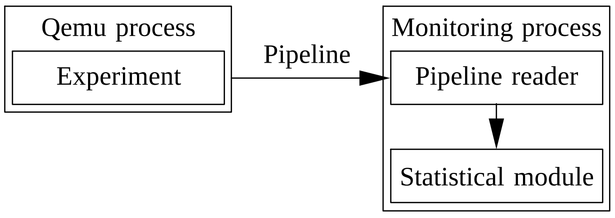

Fig. 3. Collecting processor's state with Qemu

run. For these capabilities Qemu was our basis for developing a new solution for collecting execution data statistics.

The idea behind speeding up the process of collecting execution data is the interference into Qemu's code translation procedure. We run a test program under Qemu in a single process mode with the target architecture the same as the native one, thus we make the program to be a subject of dynamic code translation. We alerted the translation procedure in such a way that before translation of every emulated target instruction a prepared code for dumping the emulated processor state is emitted. As a result during execution instructions for dumping and modifying processor's state alternate as illustrated in Fig. 2.

To ensure a good performance of processing execution state at every step we have decided to put a statistical module in a separate process. In Qemu's process the code for dumping processor's state collects all needed data and sends them through a pipeline to another process which performs statistical calculations. This solution allowed us to utilize two processor's cores during a Golden Run. The architecture of an overall Qemu-based solution is presented in Fig. 3 and exact experimental results are presented in the next section.

In that configuration performing a test is about 1 000 times slower than a normal run, which is a significant speedup compared to overhead of 20 000 times in FITS. Moreover, most of the time during a Golden Run is consumed by processing statistical data, thus it may be a starting point for further optimizations – e.g. parallel processing.

IV. Experiment results

To test the performance of Qemu-based solution we have prepared three test programs:

- `fib` – trivial implementation of the Fibonacci number evaluator – as input a Fibonacci number to compute is passed,

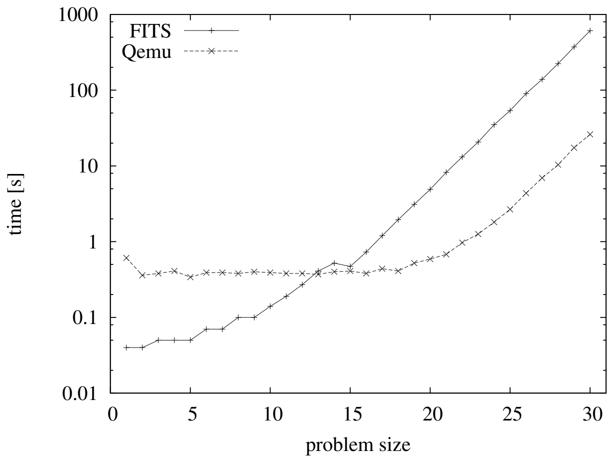

Fig. 4. Golden Run execution time for `fib`

- `aes` – AES data encryption function; modified in a such way it performs encryption of a single data block a specified number of times,
- `md5` – hash function evaluation for a file.

Figures from 4 to 6 present the Golden Run execution time for different test programs under FITS and Qemu. For clarity for `aes` test program instead of a number of algorithm iterations on 256-bits data block a whole amount of processed data is given. Fig. 7 presents ratio of time needed by Qemu-based solution compared to FITS for all test programs. Due to different domains of experiments graphs were normalized within x-axis to show the ratio for all probes collected during the tests.

Graphs show that Qemu-based solution consumes about 5% of time consumed by FITS to achieve the same results. The exception is a `fib` test program for small problem size, where the cost of setting up Qemu environment and dynamic translation is bigger than running a program under FITS. However, for bigger problem size the benefit is the same as in other cases.

The purpose of implementing `aes` program as an operation on a single block of data was to avoid I/O operations. This introduces diversity compared to `md5` which invokes I/O operations. However, it did not result in any major changes in behaviour since during the same amount of time (15 s.) both programs processed too small amount of data to make the lack of I/O operations significant. `aes` processed 3 times less data, which is a similar rate as obtained with testing AES and MD5 from openssl library for a 1 GB file.

V. Further development

Qemu-based solution is now utilized to speed up the Golden Run phase. However it may be possible to implement also the Experiment Run phase. It would benefit in two ways:

- perform tests on emulated architectures which differ from the running system,
- allow to inject faults into the operating system.

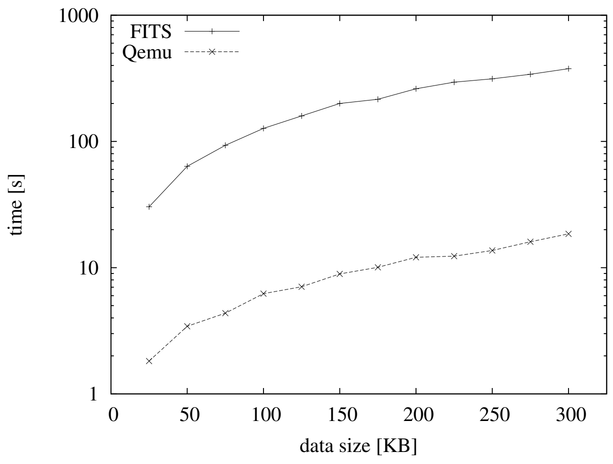

Fig. 5. Golden Run execution time for `aes`

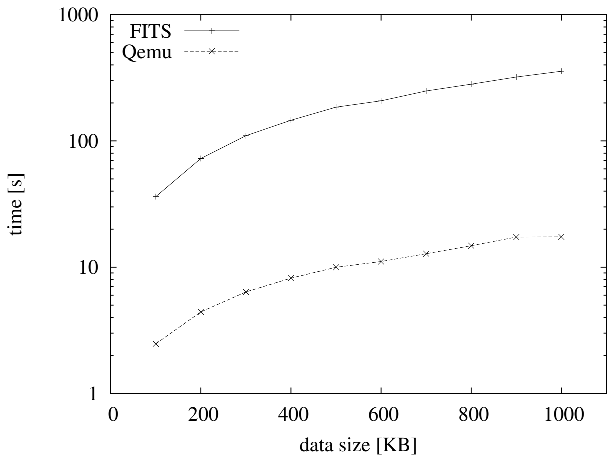

Fig. 6. Golden Run execution time for `md5`

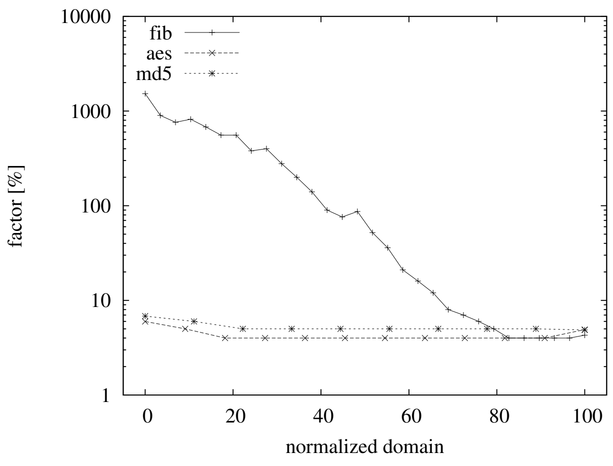

Fig. 7. Golden Run execution ratio

Performing tests on emulated architectures may be beneficial when the target system is unavailable or slower than the emulated one. Improving software with fault injection requires a significant number of Experiment Runs to determine weak points, so every speedup is desired especially when testing a complex piece of software. When we take into consideration utilizing Qemu as a backend for a distributed fault injection cluster as described in [9], [10] it could compose a powerful framework for enhancing embedded software.

Utilizing Qemu as a fault injector at operating system level would make possible not only testing behaviour of operating system with hardware faults, but also to simulate errors in peripheral devices as Qemu also simulates.

The only concern for Qemu as a fault injection framework is uncertainty whether an emulated processor will behave the same way as a real processor after injecting a fault. This may be achieved by comparing results of fault injection experiments produced by FITS and those from Qemu's emulation. Another benefit of such research could be an increased quality of Qemu's emulation. This is because any detected difference in behaviour may result in a fix in Qemu, so it behaves the same as a real processor.

VI. Conclusion

Fault injection is a way to study and increase software reliability by experiments. The software fault injection techniques are complementary with less available hardware techniques, which makes them the first tool to use when there is a need to improve a piece of software. That makes work on these techniques justified.

Utilizing a processor emulator like Qemu outlines new directions for implementation of fault injection. It gives the ability to speedup some tasks and has a potential to extend the functionality of existing fault injectors.

The presented method of collecting execution data statistics is a mean to reduce time spent on improving software and it has a potential for further extensions. Working on efficient fault injection experiments makes possible to improve with this technique more and more complex applications.

References

[1] H. Madeira, R. R. Some, F. Moreira, D. Costa, and D. Rennels, "Experimental evaluation of a cots system for space applications," pp. 325–330, 2002, iD: 1.

[2] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G. H. Leber, "Comparison of physical and software-implemented fault injection techniques," *Computers, IEEE Transactions on*, vol. 52, no. 9, pp. 1115–1133, 2003.

[3] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson, "Fault injection into vhdl models: the mefisto tool," pp. 66–75, 1994, iD: 1.

[4] A. Baldini, A. Benso, S. Chiusano, and P. Prinetto, "BOND": An interposition agents based fault injector for windows nt," pp. 387–395, 2000, iD: 1.

[5] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "Ferrari: a tool for the validation of system dependability properties," pp. 336–344, 1992, iD: 1.

[6] D. Gil, J. Gracia, J. C. Baraza, and P. J. Gil, "Analysis of the influence of processor hidden registers on the accuracy of fault injection techniques," in *High-Level Design Validation and Test Workshop, 2004. Ninth IEEE International*, 2004, pp. 173–178.

[7] P. Gawkowski and J. Sosnowski, "Developing fault injection environment for complex experiments," in *On-Line Testing Symposium, 2008. IOLTS '08. 14th IEEE International*, 2008, pp. 179–181.

[8] K. Onoue, Y. Oyama, and A. Yonezawa, "A virtual machine migration system based on a cpu emulator," pp. 3–3, 2006, iD: 1.

[9] J. Sosnowski, A. Tymoczko, and P. Gawkowski, "An approach to distributed fault injection experiments," pp. 361–370, 2008.

[10] ——, "Developing distributed system for simulation experiments," *Information Systems Architecture and Technology, Information Systems and Computer Communication Networks*, pp. 263–274, 2008.

[11] A. Lesiak, P. Gawkowski, and J. Sosnowski, "Error recovery problems," in *Dependability of Computer Systems, 2007. DepCoS-RELCOMEX '07. 2nd International Conference on*, 2007, pp. 270–277.

[12] P. Gawkowski and J. Sosnowski, "Analyzing fault effects in fault insertion experiments," pp. 21–24, 2001, iD: 1.

[13] ——, "Using software implemented fault inserter in dependability analysis," pp. 81–88, 2002, iD: 1.

[14] H. Madeira, D. Costa, and M. Vieira, "On the emulation of software faults by software fault injection," pp. 417–426, 2000, iD: 1.

[15] J. Carreira, H. Madeira, and J. G. Silva, "Xception: a technique for the experimental evaluation of dependability in modern computers," *Software Engineering, IEEE Transactions on*, vol. 24, no. 2;, pp. 125–136, 1998.

[16] A. da Silva, J. F. Martinez, L. Lopez, A. B. Garcia, and V. Hernandez, "Xml schema based faultset definition to improve faults injection tools interoperability," pp. 39–46, 2008, iD: 1.