2010 Third International Conference on Dependability

Software-Implemented Fault Injection at Firmware Level

Peter Tröger
Hasso-Plattner-Institute
University of Potsdam
Potsdam, Germany
peter.troeger@hpi.uni-potsdam.de

Felix Salfner and Steffen Tschirpke
Department of Computer Science
Humboldt University Berlin
Berlin, Germany
salfner/tschirpk@informatik.hu-berlin.de

***Abstract*—Software-implemented fault injection is an established method to emulate hardware faults in computer systems. Existing approaches typically extend the operating system by special drivers or change the application under test. We propose a novel approach where fault injection capabilities are added to the computer firmware. This approach can work without any modification to operating system and / or applications, and can support a larger variety of fault locations. We discuss four different strategies in X86/X64 and Itanium systems. Our analysis shows that such an approach can increase portability, the non-intrusiveness of the injector implementation, and the number of supported fault locations. Firmware-level fault injection paves the way for new research directions, such as virtual machine monitor fault injection or the investigation of certified operating systems.**

***Keywords*-fault injection; firmware; extensible firmware interface; system management mode; X86**

I. Introduction

An in-depth study of fault tolerance capabilities of a system design can be accomplished at various stages including the conceptual design phase, system design, or prototype phase. In order to address the problem that true failures are a rare event during experimental runs, fault injection is frequently applied. System components are intentionally modified in order to trigger single component failures. Once a component is in an erroneous state, subsequent reactions of the system are monitored and evaluated in order to assess and quantify fault tolerance capabilities of the system under test.

Several approaches to fault injection have been proposed in the past. They vary by the layer where faults are injected and how the injection is performed. Major criteria for selecting an appropriate fault injection technique are the assumed fault model, the number of tests that have to be performed in a given time, controllability of the fault injection, and the number of changes necessary to implement fault injection in the tested system.

The representativeness of fault injection experiments is limited if the system under test has to be modified. In such case, tests might not be performed under conditions similar to the run-time environment of the productive version. This can be a relevant aspect in safety-critical applications, such as control systems for critical infrastructures. Such systems frequently rely on certified software that must not be modified. Hence, the alterations necessary to implement fault injection should be minimal. A second limitation of most injection techniques is their non-portability across operating systems and/or hardware platforms.

In order to cope with portability and non-intrusiveness aspects of fault injection, we develop a novel approach to fault injection that operates on firmware level, i.e., below the operating system. The goal of this paper is to explore how such firmware-based approach can be implemented focusing on the wide-spread X86/X64 and Itanium architecture (see Section III) and to analyze properties and limits of the presented approach (see Section IV).

II. Related Work

Fault injection can be realized either in simulation or in real hardware / software, depending on the chosen fault model. In this paper we only consider software-based approaches, a broader overview is given in [1]. *Injection on hardware level* is appropriate for fault classes with tight constraints on timing resolution, fault location (e.g., cache cells), or fault type (e.g., circuit bridging faults) [2].

Software fault injection emulates both software and hardware faults by programmatic modification. It can target different layers of software, such as the operating system, a virtual runtime environment, or the application itself. Beside the generation of faulty software binaries (compile-time injection), most approaches rely on the injection at runtime, i.e. FIAT and DOCTOR [1].

A special variation is *software-implemented fault injection (SWIFI)*. SWIFI emulates specifically transient and permanent hardware faults by programmatic changes, for example in registers or memory cells. It is applicable to the hardware and software environment of the real system under test; it can be easily expanded with respect to fault types, and does not bear the risk to permanently damage the hardware as true physical fault injection. On the other hand SWIFI requires modifications that can limit the representativeness of results obtained. It can also only target fault locations that are accessible for software.

There are several examples for SWIFI solutions. One is the FERRARI system [3] that uses software trap handlers to

978-0-7695-4090-0/10 $26.00 © 2010 IEEE
DOI 10.1109/DEPEND.2010.10

emulate CPU, memory, and bus faults. The tested process triggers the UNIX *ptrace()* facility in order to run in a special trace mode.

Xception by Carreira et al. [4] relies on the advanced hardware capabilities of modern processors. The toolkit utilizes hardware performance monitoring to raise a processor debug exception when the injection condition occurs. The exception handler then performs the actual injection. Several other SWIFI approaches, such as FTAPE by Tasi and Iyer, also rely on special operating system drivers. This approach has the comparatively lowest interference with the system under test - it does not require special trace execution, new software trap handlers, or any direct modification of the tested application.

Our broader analysis showed that existing SWIFI approaches always modify system software to some extent. Either the tested application has to run in a special trace mode, the operating system has to be modified or extended by a driver, or the application itself has to be modified. Our new concept specifically addresses this interference problem by establishing the fault injector as part of the *computer firmware*. This allows leaving operating system and application stack untouched, while still having the benefits of a SWIFI approach. Since many relevant server infrastructures - even in critical environments - rely on X86/X64 computers today, we focus on this hardware architecture in the following analysis of possible realizations.

III. Solution Space

The standard firmware in X86/X64 systems is still the *Basic Input Output System (BIOS)*, which was invented over 20 years ago for the first IBM personal computer. The BIOS is responsible for initializing and abstracting the computer hardware, in order to allow an operating system boot loader to work on different hardware platforms. With modern operating systems, the BIOS interfaces are no longer used after startup, since native software drivers take over control. Extending BIOS firmware with fault injection functionality would demand some source-code modification – but any standard BIOS software is subject to strict licensing. One possible alternative is open-source firmware, with *Coreboot* as most prominent example.

A completely different option recently became available with the introduction of Itanium processors. Intel and other companies started to develop an alternative firmware concept in order to circumvent classical BIOS limitations such as the 16 bit code base and the proprietary implementation strategy [5]. The result of this effort is the *Extensible Firmware Interface (EFI)* standard. The EFI specifications define programming interfaces that allow extension and configuration of a computer's firmware in a more flexible way than it was the case with traditional BIOS. Beside the exclusive usage of EFI in Itanium systems, it is also the default firmware for

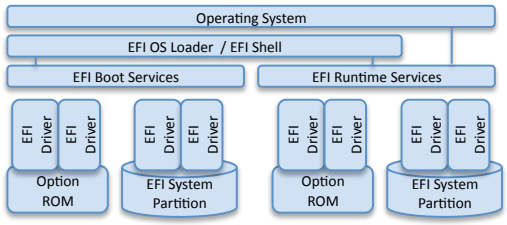

Figure 1. Extensible Firmware Interface Stack

all recent Apple computers and many other recent X86/X64 systems.

Figure 1 shows the relation between EFI and other system components. All services are implemented by EFI drivers, which are either stored in the motherboard option ROM or can be loaded from a dedicated hard disk partition. EFI drivers provide the firmware's hardware abstraction, e.g., for console interaction, PCI bus access, or memory access of the boot loader. EFI-based applications, especially the operating system boot loader, use the driver-implemented functionality.

EFI boot service drivers are only available until the initiation of an operating system startup has been signaled. *EFI runtime service drivers* remain in memory to provide firmware functionality to the operating system.

A second dimension of the solution space, beside the firmware extension strategy, is the execution mode of fault injection code.

A fault injector implementation always runs as 'out-of-order' code during normal processor operation, for example as interrupt-triggered driver, trap handler, or debugger code. Since our goal is to inject faults without any software modification on operating system level or above, we propose to run the fault injector in the *System Management Mode (SMM)*. Besides the well-known real mode and protected mode, SMM is the most privileged execution mode of X86-compatible processors. It is originally intended for special BIOS software that has to be regularly executed, such as power and fan management functions or the handling of hardware error events such as memory parity faults.

The SMM processor mode is triggered by a special interrupt, the *System Management Interrupt (SMI)*, or by sending a special APIC message to the processor. The SMI is a non-maskable interrupt that takes precedence over all other interrupts. With the switch to SMM, the CPU freezes all activities in the current mode of operation. The interrupt handler routine has now full access to computer resources. All I/O and system machine instructions are allowed, which is a difference even to the Ring 0 protected mode code. Memory is accessible in 32bit real mode addressing. When the SMI handler has completed its operation, it executes a special instruction that causes the processor to resume the operation mode that was active before.

Modern processors with hardware virtualization support

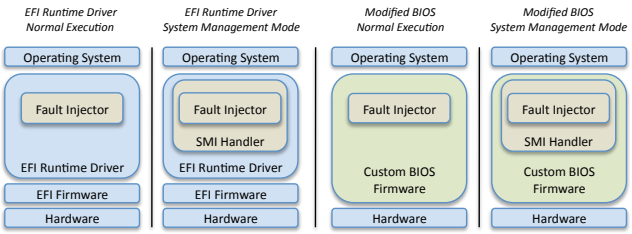

Figure 2. Solution space for firmware-based fault injection

can also be influenced by SMM in the sense that the currently active logical processor is interrupted and the state is saved. All software layers above firmware level are now transparently suspended, starting from the host operating system itself.

The combination of both concepts (custom BIOS vs. EFI, Non-SMM vs. SMM) results in four strategies for firmware-based fault injection in X86-alike systems, as shown in Figure 2.

IV. Analysis

In the following section, we evaluate the identified implementation strategies with respect to typical properties of SWIFI.

A. Supported Fault Locations

The fault location denotes the specific hardware part that can be set to an erroneous state under the given fault model. An analysis of related work showed that for most approaches the focus is on the main processor. Some of the SWIFI solutions also support memory cell or I/O driver fault injection.

In our firmware-based approach, both non-SMM variants support all explicit modifications (such as register changes) that are possible for kernel-mode software running in protected mode. This is also the case for driver-based related SWIFI solutions.

With the usage of SMM, the set of possibilities for processor fault injection is substantially extended. Several X86/X64 processor registers are saved before the SMI handler starts executing. Parts of the saved state map are changeable in an SMI handler implementation, which allows the manipulation of other general purpose registers; status registers, and even the instruction pointer on bit-level. Other processor registers are not automatically saved and restored, but can still be changed. This includes FPU registers, cache configuration registers, control registers and the trap controller state. A similar classification exists for registers in the Itanium processor [6].

The extended set of fault locations with the SMM approach enables a set of new opportunities for hardware fault experiments. Manipulations in the saved processor state information might even lead to machine halting on SMM exit, since the processor hardware itself tries to detect invalid conditions. This might or might not act as another possible fault type, for example to emulate a processor hardware crash fault. The injection from a SMM handler could theoretically also support processor caches, if they remain untouched when SMM is enabled by the processor. According to the architectural documentation of SMM [6], this is a model-specific property of the processor revision.

When memory cells are targeted as fault location, SMM approaches are limited to 32bit real-mode addressing, whereas non-SMM approaches can access the entire protected-mode address space. SMM handlers also have write access to the memory of a potentially running virtualization hypervisor. This is possible since logical processors are also suspended on SMM entry. SMM-based fault injection therefore offers the possibility for *hypervisor fault injection*, a topic that is –to the best of our knowledge– not covered by dependability research at the moment.

For I/O devices as targeted fault location, both the non-SMM and the SMM firmware approaches are feasible. They support the direct access to hardware devices by port-mapped and memory-mapped I/O, without any involvement of the operating system itself.

From the viewpoint of supported fault locations, the SMM-EFI approach needs to be favored. It makes the fault injector available even before the operating system loader starts to operate. SMM supports a broader range of manipulations, since neither the restrictions of the protected mode nor operating system security mechanisms are active during SMM interrupt handling.

B. Portability

From the viewpoint of portability, the usage of BIOS alternatives demands support for the very specific combination of chip set, memory controller, and other parts of the motherboard. EFI, in contrast, is a standardized interface that tries to solve the portability problem for firmware source code. With the envisioned spreading of EFI as default firmware in X86/X64 systems, more and more hardware platforms will support the EFI-based realization of a fault injector. We therefore favor these solutions for the sake of portability. The level of support for SMM handler code in EFI runtime drivers is still under investigation.

C. Fault Trigger

The fault trigger is an explicit condition that, once met, leads to the injection of a hardware fault. There is a distinction between the mechanism that checks the condition, and mechanism that subsequently activates the fault injection code.

Classical examples for SWIFI activation mechanisms are timeouts (with unpredictable fault effects, therefore suitable for transient faults and intermittent hardware faults), exceptions resp. traps, and inserted code. Firmware-based

fault injection theoretically adds the possibility for hardware interrupts as fault trigger condition, but since EFI does not support the hooking on hardware interrupts, the custom BIOS replacement provides the better solution in this case.

Exceptions, traps, and inserted code can only be implemented by extending or modifying the tested system. This is contradictory to our goal of non-intrusive failure injection.

In the case of using an SMM handler for fault injection, an SMI is the only available starting point for the injection. SMIs can be triggered from software directly or indirectly using I/O controller chips. Such controllers support a large set of SMI-triggering events including the power button, real-time clock timers, serial / USB port activities, or NMIs.

Using the I/O controller to trigger an SMI enables the re-use of well-known condition checks from other SWIFI approaches, but would require code in the operating system. Therefore a trade-off exists between minimal intrusiveness (where only hardware timers are available as trigger) and maximum flexibility in fault injection triggering (at the cost of portability).

We conclude that firmware-based fault injection would not provide any advantage if a rich set of fault triggers is the primary concern. This property is independent from the chosen implementation strategy. With respect to all investigated properties, we decided for the EFI runtime driver approach.

D. Initial Experiments

In a set of initial experiments, we tested a collection of four EFI-enabled hardware configurations ranging from laptops to dual-core server machines. EFI also has dedicated support for the realization of SMM handlers as special EFI runtime drivers. We figured out that EFI runtime drivers are supported in all tested hardware combinations, while the implementation of the necessary SMM protocols is not available on all tested platforms. It should be noted that our fault injector prototype was portable between different EFI vendor shells, processors and chip set types.

A second option for EFI development is the use of hardware emulation environments. Beside the EFI SDK possibilities (simulator, EFI firmware on USB stick), there is also an EFI-enabling solution for QEMU, a well-known PC hardware emulator.

More detailed experiences from our initial implementation were left out due to space constraints. Specific implementation results and measurements will be presented in subsequent publications.

V. Conclusion and Outlook

We presented our ongoing research for a novel way to emulate hardware faults in computer systems by firmware-based fault injection. We compared this concept with existing Software-Implemented Fault Injection (SWIFI) approaches, and explained that firmware-based fault injection can offer a new level of non-interference with the tested system. Our fault injection concept works independent from the operating system type. This offers unique possibilities for fault injection experiments such comparisons across operating systems, special external device tests, or even virtualization hypervisor fault tolerance analysis. Firmware-based fault injection is the best choice in cases where a manipulation at the software level is no option.

We analyzed four different realization strategies based on technologies available in the X86/X64 and Itanium processor families. We argued that the combination of Extensible Firmware Interface (EFI) implementation with the usage of the System Management Mode (SMM) provides the best possible solution in terms of non-intrusiveness and supported fault locations. When portability is the major criterion, an EFI fault injector without SMM seems to offer the best solution. As with most SWIFI solutions, both approaches are mainly suited for transient faults.

Our analysis in this paper focused on intrusiveness and portability, leaving out system monitoring for fault effects. SMM allows the access to low-level monitoring data, such as the processor hardware event counters. Additionally, our approach can be seamlessly combined with existing SWIFI solutions. This might also provide elaborated monitoring and fault trigger capabilities.

Future work will include extended experiments with our time-triggered EFI-SMM injector prototype. We will also continue to investigate the support for SMM and EFI in different hardware combinations. A further research topic is the investigation of hypervisor fault injection in order to study dependability aspects of hardware virtualization.

References

[1] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault Injection Techniques and Tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[2] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault Injection for Dependability Validation: A Methodology and Some Applications," *IEEE Trans. Softw. Eng.*, vol. 16, no. 2, pp. 166–182, 1990.

[3] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "FERRARI: A Flexible Software-Based Fault and Error Injection System," *IEEE Trans. Comput.*, vol. 44, no. 2, pp. 248–260, 1995.

[4] J. a. Carreira, H. Madeira, and J. a. Gabriel Silva, "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers," *IEEE Transactions on Software Engineering*, vol. 24, no. 2, pp. 125–136, 1998.

[5] V. Zimmer, M. Rothman, and R. Hale, *Beyond BIOS: Implementing the Unified Extensible Firmware Interface with Intel's Framework (Computer System Design)*. Intel Press, 2006.

[6] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developers Manual - Volume 3B: System Programming Guide, Part 2*, 9 2009.