Understanding Reliability Implication of Hardware Error in Virtualization Infrastructure

Xin Xu *H. Howie Huang*
George Washington University

Abstract

Hardware errors are no longer the exceptions in modern cloud data centers. Although virtualization provides software failure isolation across different virtual machines (VM), the virtualization infrastructure including the hypervisor and privileged VMs remains vulnerable to hardware errors. Making matters worse is that such errors are unlikely bounded by virtualization boundary and may lead to loss of work in multiple guest VMs due to unexpected and/or mishandled failures. To understand reliability implication of hardware errors in virtualized systems, in this paper we develop a simulation-based framework that enables a comprehensive fault injection study on the hypervisor with a wide range of configurations. Our analysis shows that, in current systems, many hardware errors can propagate through various paths for an extended time before an observed failure (e.g., whole system crash). We further discuss the challenges of designing error tolerance techniques for the hypervisor.

1 Introduction

In the era of warehouse-scale computing, a typical data center consists of hundreds of thousands of servers with multicore processors, multi-level caches, and a large amount of main memory. All these components - logical circuits in CPU, memory cells in cache and memory - are not immune to soft errors [1, 2]. Technology scaling already points to a projection of escalating soft error rates in future chips [3], which combined with high server count would result in shorter MTBF (mean time between failures) and require system-level techniques on fault prevention, detection, and recovery.

On the other hand, virtualization has been widely adopted in large data centers to increase overall resource utilization. In a typical environment shown in Figure 1, a hypervisor provides an abstraction layer on top of hardware resources, working with privileged VM (e.g., Dom0 in Xen) to serve the requests from guest VMs (e.g., DomU). Note that the abstraction layer can consume a considerable amount of hardware resources (e.g., CPU

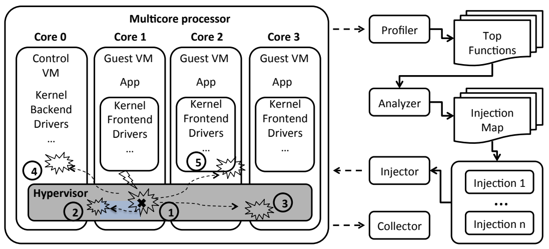

Figure 1: Virtualization architecture on a multicore processor and the proposed error injection framework. A soft error effecting the hypervisor instructions running on core 1 can be propagated on the same core (path 1 and 2), to other hypervisor instructions on core 3 (path 3), to the control VM instructions in core 0 (path 4), or to one of guest VMs in core 2 (path 5).

cycles). Virtualized servers are soon expected to host hundreds of VMs, and dedicated I/O cores will be required for data-intensive VMs [4], where it is not uncommon to see significant amount (e.g., 50%) of CPU spent for the hypervisor and Dom0, much higher than that of traditional non-virtualized OS (e.g., 2-10%) [5].

Unfortunately, while the failure of a guest VM is unlikely to affect other VMs, a hardware error induced failure in the virtualization infrastructure (the hypervisor and privileged VMs) may propagate beyond the system boundary and result in multiple VM failures, or worse, the whole-system failure. The goal of this work is to understand the characteristics of hardware error in virtualized systems through a comprehensive set of microarchitectural fault injections. To this end, we design a simulation-based fault injection framework as shown in Figure 1 and perform over 46,000 injections to characterize the reliability of the hypervisor and overall virtual systems against soft errors. Our fault injection tool is designed as a module of a full system simulator, requiring minimum modifications to the original system. It can inject soft errors into any function of the target hypervisor, removing the constraints of existing software tools [6] [7] [8].

The main contributions of our work are two-fold: First, our framework enables a comprehensive fault injection study on the hypervisor with a wide range of configurations, including para-virtualization, full virtualization, and VMs running on separate or shared cores. Our fault injection framework targets the most frequently used hypervisor functions that greatly reduces the injection scope while discovering 19% more cases than random fault injection, and also reveals some critical cases that would otherwise have been missed.

Second, our analysis leads to several key observations - nearly half of injected errors result in system wide crashes including all VMs running on the machine. While some of the errors lead to quick crashes, they can also propagate to the control and guest VMs, which makes them hard to detect and protect against. We study in depth a number of critical error propagation paths.

2 Related Work

Current hardware and software methods cannot fully address the challenge of error propagation in virtualized systems. For example, hardware dual modular redundancy (DMR) has long been proposed for fault detection [9, 10], and hardware signature based methods [11, 12] have also been developed to check the correctness of various architectural states. However, few commodity servers have completely adopted these techniques with exceptions like IBM S/390 [13] and HP NonStop systems [14], although modern processors (e.g., Intel Xeon) provide some reliability features such as ECC protection for cache and memory which to some extent may mitigate the impact of memory errors. On the other hand, compiler based RMT such as SRMT [15] and DAFT [5] requires no hardware modification, but comes with high performance overhead (38% in DAFT and 19% in SRMT). Although software process-level redundancy [16, 17] may have relatively lower performance overhead, this technique is difficult to adopt in OS kernels and hypervisors. Software symptom based detection method monitors abnormal program behaviors for low cost error detection [3, 18, 19], which leverages the symptoms that are already built-in OS or hardware, such as exceptions or the event of branch miss predictions. In this work, we discuss the effectiveness of current approaches in the virtualized environment. We hope that this study will help researchers to better understand reliability implications of hardware error with respect to the hypervisor and design fault tolerance techniques accordingly.

3 Framework

The fault injection framework contains four components: 1) a *profiler* is used to profile the hypervisor and identify the most frequently used functions (i.e., top functions) as injection candidates; 2) an *analyzer* is used to analyze the top functions and generate an injection map; 3) an *injector* is used to interact with the simulator and conduct fault injection experiments; 4) a *collector* is used to collect the log and system states from a simulated serial port for in-depth error analysis. Although we focus on Xen in this work, our approach is largely applicable to other hypervisors such as KVM and VirtualBox.

Hypervisor Profiling. We profile the Xen 4.1.2 hypervisor using OProfile and UnixBench benchmark. We measure the average CPU utilization of different hypervisor functions, and identify 69 functions that cover total 90% of the CPU time. We classify these functions into four subsystems according to their functionalities.

CPU Management subsystem (CM) provides the interfaces for VMs to access physical CPU resources, including emulating architecture specific instructions and privileged instructions, scheduling virtual CPU (VCPU), and modifying control registers. Taking VCPU scheduling as an example, a (credit-based) scheduler will schedule each VCPU to run on physical CPUs exclusively for a certain period of time, where the function *csched_schedule* is used. In total, 20 functions are identified in this category.

Memory Management subsystem (MM) manages shared DRAM and ensures the isolation of each domain. The Xen hypervisor provides the pseudo-physical memory as an abstraction layer to the guest VMs. For example, the function *page_get_owner_and_reference* returns the domain which owns the memory page and the reference count of this page. In total, 28 functions are identified in this category.

Hypercall and Control management subsystem (HC) contains low-level functions that handle system calls or hypercalls. A hypercall page, essentially a memory page, is provided to each guest VM when it is started. When a system call is required in a guest VM, it will directly call the address within the hypercall page to initiate the hypercall. For example, the function *syscall_enter* is the low-level Xen routine to replace the Linux call. This function saves the current context and passes the arguments to the hypercall handler in the hypervisor. In total, 13 functions are identified in this category.

Domain Management subsystem (DM) provides a set of functions to manage VM states, for example, the function *update_vcpu_system_time* is used to update the system time of a guest VM. In total, 8 functions are identified in this category.

Analysis of Instruction Traces. In this phase, we analyze the instructions of the most frequently used functions, identify relevant registers as injection candidates, and generate an injection map. Each entry in this map consists of the target register and the timestamp for injection. Examples of entries in the injection map are shown in Table 1. The selection of target registers varies depending on instruction types. 1) For instructions without registers, we will skip all except branch instructions. If an error occurs in the address of the branch target, an incorrect instruction will be loaded. 2) For instructions with one register, this register will be selected as the injection candidate. 3) For the instructions with two registers, the source register (*src*) and the source-destination register (*src-dst*), we choose to inject the faults into *src* instead of *src-dst* to avoid the possibility of overwritten by the output of the calculation.

Table 1. Generating an Injection Map Using the Analyzer

No	Instruction (*src, dst*)	Injection Candidate
1	add rbx, rax	rbx
2	sub 0x1, 0x10(rax)	rax
3	call 0xffffffff82000000	rip

Profiling Based Fault Injections. We choose single-bit flip soft errors in architecture level registers in CPUs, including general purpose register and the instruction pointer. Note that our method covers other components in processors, such as ROB and branch predictors, as well as in cache lines, since these non-masked faults can be simulated with errors in source registers. We leave advanced memory soft errors and other types of hardware errors as future work.

Implementation. We develop our fault injection framework as the modules that run in a full system simulator Simics [20]. The simulated system is configured with a four-core 64-bit processor, 2GB memory and a 60GB disk. The system is equipped with Xen 4.1.2 and Debian 6 with Linux kernel 2.6.32. We use one Dom0 running on one VCPU and two DomUs, each of which is assigned by one VCPU, 512MB memory and a 10GB virtual disk. The three VCPUs are attached to three physical processors respectively. This way, the activities on each domain will be limited to its own physical processor. We select a wide range of benchmarks from PARSEC [21], SPEC2006 [22] and Postmark [23] to exercise I/O (e.g., *postmark*,*freqmine*, *x264*), CPU (e.g., *canneal*, *bzip2*), and memory (e.g., *mcf*) resources. When conducting fault injections, the same benchmarks are running in two DomUs in parallel.

4 Results and Analysis

We conduct a total of 25,000 fault injections across six benchmarks with para-virtualization, targeting the most frequently-used functions (top function injection). For comparison, we conduct another 12,000 random injections. To test different VM configurations, we also conduct 3,000 injections where VMs are located in a shared core, and 6,000 injections with full virtualization.

Analyzing Crash Type. Examining the error propagation behaviors in terms of crash types is critical, as the detection and recovery mechanisms may vary depending on the crash type. In this paper, we classify the results into four types: 1) *System crash*, which represents a crash or hang in the host system. This is the worst case, since the hypervisor, the control VM, and all guest VMs

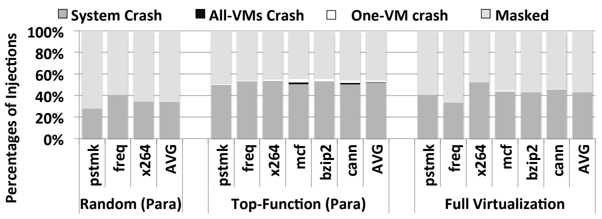

Figure 2: Comparison of Para-Virtualization (Random and Top-Function) and Full Virtualization Injections

are affected. 2) *One-VM crash*, where a failure leads to a crash in one DomU, but the Dom0 and other DomUs are not affected; 3) *All-VMs crash*, where a failure causes the crashes of all the DomUs. In this case, the hypervisor and Dom0 are still running, and no system reboot is initiated. 4) *Masked*, where the injected fault does not result in a visible crash in Dom0 or DomUs. Such fault could still lead to silent data corruption which we will investigate as part of future work.

As shown in Figure 2, our top-function injections provide a large number of crash cases at 53.3% on average, which is 19% higher than the random injection (34.1%). In total, 304 one-VM crash cases are discovered in the top-function injections. Comparatively, only two such cases in total are found in random injections. Additionally, two All-VMs crash cases are identified in the top-function injection results, which are not discovered in the random injection, and turn out to be critical for understanding the error propagation to be discussed shortly. Furthermore, our framework is able to deliver a more consistent result - less than 5% discrepancy for crash cases across different benchmarks.

Full virtualization handles the VM transitions with hardware support such as VMX (Intel) or SVM (AMD) instructions (e.g., vmread and vmwrite). Therefore, instead of using top functions, we intentionally target these instructions as well as normal instructions. As one can see from Figure 2, the errors injected in full virtualization produce similar results of different crash cases across various benchmarks. In addition, we conduct 3,000 injections by pinning all domains to a shared core, where there are an average of 45% cases as system crash, which are also close to previous configurations.

Observation #1: **Soft errors in the hypervisor may cause various types of failures** - more than half of the errors can lead to system-wide crashes that will affect all the VMs running on the shared host, and soft errors may even cause all VM failures without triggering system-wide reboot.

Analyzing Fault Location that is defined as the function where a fault is injected. Recall that we classify the Xen functions into four subsystems and here we aim to understand the reliability characteristics of each subsystem. In Figure 3, the left side shows the percentages of

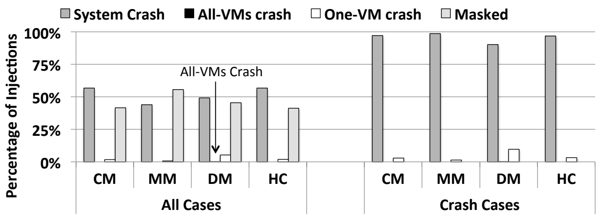

Figure 3: Results by Fault Locations

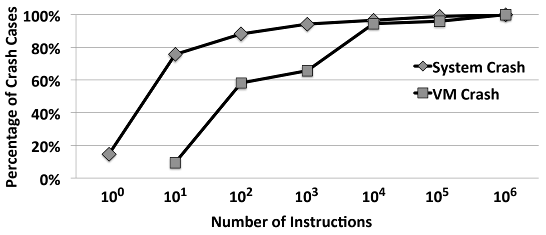

Figure 4: CDF of Crash Latency

all results, and the right side filters out the masked cases and shows the percentages of only crash cases.

Observation #2: All Xen subsystems are vulnerable to soft errors, as more than 40% of error injections leads to crashes. The *CM* (CPU management) and *HC* (hypercall/control) subsystems have the highest percentages (over 58%) of crash cases, while the *MM* (memory management) with a relatively lower percentage of 44.5%. Interestingly, the *DM* (domain management) subsystem accounts for more one-VM and All-VMs crash cases. Nevertheless, this result shows that no one subsystem is less critical than others, thus a good fault tolerance mechanism for a hypervisor should provide a complete coverage over critical kernel functions.

Analyzing Crash Latency that is calculated as the number of instructions when a fault is activated and a fatal exception is captured by the system. For multicore processors, only the instructions on the core where the fault is injected are calculated. It is possible that several exceptions are triggered before system crash. Here we calculate the latency till the first exception is triggered (a conservative estimation). Figure 4 shows the cumulative distribution (CDF) of the latency of crash cases.

Observation #3: Most crash cases have relatively short latency (<=100 instructions), 88% for system crash and 60% for one-VM crash cases, respectively. **Observation #4: There still exist a considerable amount of the crashes with long latency (>10,000 instructions)**, about 5% in both cases. These errors that are likely to spread to DomU/Dom0 should be effectively detected as early as possible for successful recovery.

Analysis of Failure Location that is the domain where a fault is manifested to a fatal error symptom, such as a fatal exception or an infinite loop. When a failure occurs, the error handling routines in the hypervisor and Linux kernel may print out the debug message that we examine to pinpoint the failure location. This helps us evaluate the effectiveness of the built-in fault detection mechanisms in Xen and Linux. In Figure 1, we have identified five possible error propagation paths, leading to five failure locations: 1) Immediate (*Imm*); 2) *Same hypervisor function*; 3) *Another hypervisor function*; 4) *Dom0* and 5) *DomU*, where the faults propagate to the Dom0 and DomU kernel respectively. In general, Dom0 and DomU failures usually have much longer latency.

Observation #5: A fault that results in a system crash may become visible quickly for half of the injections, and propagate down the execution path for the rest. In the first case, because 50% of the failures are detected quickly, chances are that system states are intact and the system recovery is possible. As shown in Figure 5, most (70%) of the errors in hypercall instructions fall into this category. On the other hand, on average 51.2% system crashes involve error propagation to the same or different hypervisor function.

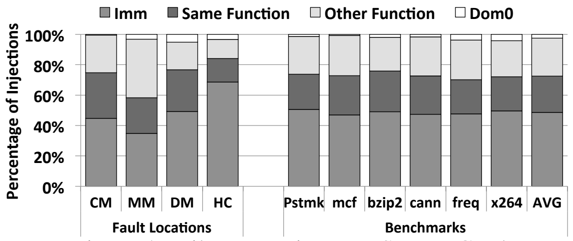

Figure 5: Failure Locations For System Crash

Observation #6: Error propagation is highly likely in the memory management subsystem, followed by the CPU and domain management subsystem. In 2.5% of the cases, a failure happens in the Dom0, which also explains the previous observation on long crash latency, and indicates the difficulty of system recovery in events of corrupted states in either the hypervisor or Dom0. The distributions of failure locations among benchmarks are relatively consistent with small variance.

Analysis of Error Propagation. Here we analyze crash cases and present eight representative cases with examples shown in Table 2.

Corruption in registers (example #1). In this example, the injected fault changes the register *rbx* to an invalid address. As a result, the fatal page fault exception is triggered when this address is being used in current instruction. Since the crash is captured right after the injection, only the register that carries the fault is corrupted. If carefully managed, one may be able to recover the states of the hypervisor, as well as those of the Dom0 and DomUs. All *Imm* failures belong to this category.

Corruption in local variable (#2). In this case, the fault propagates from a register to the address of the local variable in the hypervisor function. The general protection exception is triggered at the first time this variable is

Table 2. Examples of Error Propagation

No	Instruction Trace	Error Propagation	Crash Type	Failure Location	Latency	Symptom	Corrupted States
1	*test 0x1, 0x16b0(**rbx**)*	*rbx* is changed to an invalid address	System Crash	Imm	1	XEN-FPF	Register
2	push **rbx** ... pop rbx *mov (rbx), rcx*	*rbx* is not used immediately after injection, and the crash occurs when *rbx* is used as an address	System Crash	Same Function	39	XEN-GPF	Local variable
3	sub 0xb8,**rsp** ... ret *mov 0x98(rax), rbx*	The stack pointer is changed to a valid but incorrect value. After the function returns, the caller function accesses the inaccessible stack	System Crash	Other Function	58	XEN-GPF	Function stack
4	cmp 0x1fff,**rsi** ... *je 0xffff82c4801034f9* *mov* sysret jb 0xffff82c480103c5e	The branch destination is altered. The Dom0 stacks and hypervisor data are corrupted. After the context is switched back to Dom0, incorrect functions are loaded.	System Crash	Dom0	60	KBUG-ATM	Hypercall return
5	mov **rax**,rcx *rep mov* ... *(Dom0 accesses shared memory)*	The loop count (*dcx*) of repeat mov instructions is altered, and incorrect number of strings are moved to the extra segment. When Dom0 CPU tries to access the segment, the failure occurs.	System Crash	Dom0	791,986	SS	Shared memory
6	mov **rax** ,0x8(rdx) sysret *mov rax, rcx*	*rax* holds a valid but incorrect return address, and the exception is triggered when the context is switched back to Dom0. Hypervisor data are safe.	System Crash	Dom0	211,178	XEN-FGM	Dom0 data
7	lea rdi, 8(**rbp**) *je 0xffff82c48100e23b* *data corruptions* sysret /*to DomU*/ iret /*to application*/	The address of a variable is changed, followed by wrong branch destination. Subsequently application data are corrupted. After application context is load, the failure is triggered	One-VM Crash	DomU	5,377	UPF	Hypercall data DomU data Application data
8	call 0xffff82c480103be0 *loopne 0xffff82c40017ed7e* *continue looping*	The fault changes the control path and the CPU goes to an infinite loop.	All-VMs Crash	Other Function	N/A	Unknown	Unknown

used. The propagation latency is 39 instructions, and no other states are affected.

Corruption in function stack (#3). Here the stack pointer is changed by the fault to a valid but incorrect address. Although this does not affect the data in current function, the caller function's stack and data are corrupted. An exception is triggered after function return.

The above three examples cause the failures that have happened within the hypervisor. For the following examples (with the exception of the last one, All-VMs crash), although the errors are originated in the hypervisor, the failures happen in either the Dom0 or a DomU.

Corruption in the hypercall (#4). Here the fault changes the hypervisor data, and subsequently alters the control path. As a result, the hypervisor exits to the Dom0 earlier than the correct instruction trace. Therefore, there are multiple data corruptions in the hypervisor data including the return values of the hypercall. But the hypervisor returns to the Dom0 in a short period of time (60 instructions), and the Dom0 states in the hypervisor stack are not corrupted. A crash will likely happen when the Dom0 uses the incorrect return values.

Corruption in the shared memory (#5). In this case, the hypervisor is serving a request from a DomU (the hypervisor request handler is running on the same physical CPU as the DomU). The fault changes the value of the string length, resulting in additional string *mov* operations. The repeated *mov* will change the stacks, resulting in corrupted values in shared memory data. During this time, the Dom0 initiates another request to the hypervisor to access shared memory data, and the fault in stack segments is reported by the Dom0 kernel. Here the fault propagates across both domains and CPUs.

Corruption in Dom0 states (#6). In this example, when the Dom0 issues a hypercall, the hypervisor will save the states of the Dom0 to its stack and restore after the hypercall is served. In this case, the fault is injected into the instruction that operates on the stack containing the Dom0 states. Specifically, it alters the return address of the Dom0. Therefore, after the system returns to the Dom0 context, the invalid instruction address will trigger an exception. We find that although this case has a long latency (over 200K instructions), the return address is not used for address or computation. Therefore the hypervisor state is correct and only Dom0 states are corrupted.

Corruption in application (#7). Here the fault changes the branch outcome and corrupts the data in the hypercall. When the context is switched back to the application, the incorrect return address is loaded and triggers the failure in the DomU. The failure is isolated in the DomU and will cause this DomU to crash. This case is different from the previous case #6 in that the hypercall data are corrupted and the error propagates through hypercall and DomU kernel to the application. But the crash is contained within the DomU.

All-VMs crash (#8). There are two such cases where

Table 3. Summary of representative fault detection techniques

Fault Detection Schemes				Failure Cases				
Approach	Technique	Perf Overhead	COTS HW Support	Imm	Same Function	Another Function	Dom0	DomU
Hardware	AR-SMT [24]	16.7%	No	✔	✔	✔	N/A	N/A
	SRT [9]	unreported	No	✔	✔	✔	N/A	N/A
	CRT [10]	unreported	No	✔	✔	✔	N/A	N/A
	DDFV [11]	1.8%	No	*	*	*	*	*
	Argus [12]	4%	No	*	*	*	*	*
Compiler	SRMT [15]	19%	N/A	*	*	*	?	?
	DAFT [5]	38%	N/A	*	*	*	?	?
Software	PLR [16]	26%	N/A	✘	✘	✘	✘	✘
	RAFT [17]	3.54%	N/A	✘	✘	✘	✘	✘
Cross Layer	Restore [18]	5%	Yes	✔	+	+	−	−
	SWAT [19]	5%	Yes	✔	+	+	−	−
	Shoestring [3]	15.8%	Yes	✔	+	+	−	−

✔- Detectable; ✘- Undetectable; * - Requires extensive modifications; + - Detectable and recoverable for some cases; − - Detectable and not recoverable; ? - not clear; N/A - not applicable

the fault causes the failure of both DomUs, while the Dom0 seems to be functioning. The reason is that a fault alters the control flow in the hypercall, and the hypervisor eventually goes to an infinite loop.

5 Discussions

To summarize, we have identified several important error propagation behaviors: 1) an error may propagate from one CPU to another CPU through shared data structures in the hypervisor; 2) an error may propagate from the hypervisor to the Dom0/DomU kernel, and potentially to the applications; 3) an error may propagate to the Dom0/DomU through hypercall return values, shared memory, and stacks that hold VM states. In Table 3, we analyze a number of representative methods in terms of applicability to a hypervisor and effectiveness in the virtualized environment. Despite significant extensions required, we believe that symptom-based error detection methods are promising as they require no special hardware support and can provide spatial or temporal redundancy within a hypervisor.

As part of future work, we will extend the fault injection framework to investigate silent data corruption [25–27] that also has a significant impact on the reliability of virtualized systems. In addition, there is a clear need for an error-resilient hypervisor that provides strong protection for kernel functions and prevents error propagation across VMs, while incurring minimal performance overhead. We are developing a prototype based on the observations from this study.

6 Acknowledgment

We thank the HotDep reviewers and Flavio Junqueira for their helpful suggestions. This work is supported by National Science Foundation grant CNS-1350766.

References

[1] Tezzaron, "Soft errors in electronic memory-a white paper," 2004.

[2] Biswas, Arijit et al., "Explaining cache ser anomaly using due avf measurement," in *HPCA 2010*.

[3] Feng, Shuguang et al., "Shoestring: probabilistic soft error reliability on the cheap," in *ASPLOS 2010*.

[4] Landau, Alex et al., "Splitx: split guest/hypervisor execution on multi-core," in *WIOV 2011*.

[5] Zhang, Yun et al., "Daft: decoupled acyclic fault tolerance," in *PACT 2010*.

[6] Reddi, Vijay Janapa et al., "Pin: A binary instrumentation tool for computer architecture research and education," in *WCAE 2004*.

[7] Gu, Weining et al., "Characterization of linux kernel behavior under errors," in *DSN 2003*.

[8] Hsu, Israel et al., "Using virtualization to validate fault-tolerant distributed systems," in *PDCS 2010*.

[9] Reinhardt, Steven K. et al., "Transient fault detection via simultaneous multithreading," in *ISCA 2000*.

[10] Gomaa, Mohamed et al., "Transient-fault recovery for chip multiprocessors," in *ISCA 2003*.

[11] Meixner, Albert. et al., "Error detection using dynamic dataflow verification," in *PACT 2007*.

[12] Meixner, Albert et al., "Argus: Low-Cost, Comprehensive Error Detection in Simple Cores," in *MICRO 2007*.

[13] Spainhower, Lisa et al., "IBM S/390 parallel enterprise server G5 fault tolerance: a historical perspective," *IBM J. Res. Dev.*, vol. 43, pp. 863–873, Sep. 1999.

[14] Bernick, David et al., "Nonstop advanced architecture," in *DSN 2005*.

[15] Wang, Cheng et al., "Compiler-managed software-based redundant multi-threading for transient fault detection," in *CGO 2007*.

[16] Shye, Alex et al., "Using process-level redundancy to exploit multiple cores for transient fault tolerance," in *DSN 2007*.

[17] Zhang, Yun et al., "Runtime asynchronous fault tolerance via speculation," in *CGO 2012*.

[18] Nicholas J. Wang and Sanjay J. Patel, "Restore: Symptom based soft error detection in microprocessors," in *DSN 2005*.

[19] Li, Man-Lap et al., "Understanding the propagation of hard errors to software and implications for resilient system design," in *ASPLOS 2008*.

[20] Simics Full System Simulator, http://www.simics.net.

[21] Bienia, Christian, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[22] Standard Performance Evaluation Corporation, SPEC CPU2006, http://www.spec.org/cpu2006/.

[23] Katcher, Jeffrey , "PostMark: a new file system benchmark," Network Appliance, Tech. Rep. TR3022, Oct. 1997.

[24] Rotenberg, Eric., "AR-SMT: a microarchitectural approach to fault tolerance in microprocessors," in *FTCS 1999*.

[25] Reis, George et al., "Swift: Software implemented fault tolerance," in *CGO 2005*.

[26] Wappler, Ute et al., "Software encoded processing: Building dependable systems with commodity hardware," in *SAFECOMP 2007*.

[27] Correia, Miguel et al., "Practical hardening of crash-tolerant systems," in *USENIX ATC 2012*.