FTXen: Making Hypervisor Resilient to Hardware Faults on Relaxed Cores

Xinxin Jin[†], Soyeon Park[§], Tianwei Sheng[§], Rishan Chen[†], Zhiyong Shan[†], Yuanyuan Zhou[†]
[†]University of California San Diego
{xinxin, ric009, yyzhou}@ucsd.edu
[§]Whova Inc
{tianwei.sheng, soyeon.park}@whova.com

***Abstract*—As CMOS technology scales, the increasingly smaller transistor components are susceptible to a variety of in-field hardware errors. Traditional redundancy techniques to deal with the increasing error rates are expensive and energy inefficient. To address this emerging challenge, many researchers have recently proposed the idea of relaxed hardware design and exposing errors to software. For such relaxed hardware to become a reality, it is crucially important for system software, such as the virtual machine hypervisor, to be resilient to hardware faults.**

To address the above fundamental software challenge in enabling relaxed hardware design, we are making a major effort in restructuring an important part of system software, namely the virtual machine hypervisor, to be resilient to faulty cores. A fault in a relaxed core can only affect those virtual machines (and applications) running on that core, but the hypervisor and other virtual machines remain intact and continue providing services. We have redesigned every component of Xen, a large, popular virtual machine hypervisor, to achieve such error resiliency. This paper presents our design and implementation of the restructured Xen (we refer to it as FTXen). Our experimental evaluation on real systems shows that FTXen adds minimum application overhead, and scales well to different ratios of reliable and relaxed cores. Our results with random fault injection show that FTXen can successfully survive all injected hardware faults.

I. Introduction

A. Motivation

As CMOS technology scales, the increasingly smaller transistor components are susceptible to a variety of in-field hardware errors such as software errors, erratic bit errors, early-life failures, aging, and design bugs [14], [27]. Traditional hardware-level resilient solutions [7], [8], [11], [15], [35], [37]–[39], [41], [42], [44], [47] rely on redundancy techniques and have become too expensive in area, power, and performance, motivating low-cost reliability alternatives.

To address the above emerging challenge, many researchers have leveraged the fact that abundant probabilistic (such as recognition, mining and synthesis) applications and high performance computing (HPC) applications are tolerant to errors, and they proposed various ways to relax the hardware-level resilience mechanisms and to expose errors to software [20], [21], [23], [29], [32], [33].

While the previous research work has proposed promising directions to either push more performance or lower energy in hardware, so far they have looked at the existing error resilience in applications, but ignored system software. In almost all systems, nearly all applications run on top of system software such as operating systems and virtual machine hypervisors (typically for server platforms in data centers). In other words, the system software is a critical component in any software stack, providing support and services for applications, including even those resilient ones.

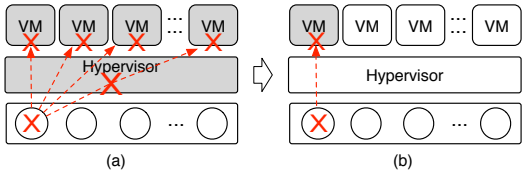


Fig. 1: (a) Current hypervisors are not fault tolerant to hardware faults. (b) The goal of our FTXen

Hardware errors can happen at any time and there is no way to control a fault to only occur during application execution time. Actually, as indicated by some recent studies [31], [36], system software is more likely to be corrupted by hardware faults and result in total system failures. When a hardware error corrupts the system software, for example, the virtual machine hypervisor, all virtual machines and applications, including those that are not running on faulty cores, would go down, regardless of whether their applications are resilient or not, as depicted in Figure 1(a).

Nowadays, more and more high performance computing is moving into the cloud platform [1]–[4], where the consequence of such system software failure could be even worse. To maximize resource utilization, typically there are hundreds of virtual machines running on one server machine. It is undesirable to let a hardware fault on one single core crash the hypervisor and take down all VMs. Even if the recovery mechanism can resume execution under some circumstances, the cost of restarting and warming up this many VMs and resuming the corresponding client states is non-trivial [19].

B. Our Contributions

To address the above problem and bring the idea of relaxed hardware closer to reality, we are making a major effort in restructuring one important part of system software, namely the virtual machine hypervisor, to be resilient to hardware faults in relaxed cores. This work focuses on the hypervisor, instead of the operating system, because in cloud platforms where reliability is crucially important, applications typically run on their own separate virtual machines. Every application has its own OS, but the hypervisor is shared among all virtual machines (and applications). Hence, error resilience in the hypervisor provides broader benefits than in the OS.

Similar to almost all previous proposals in relaxed hardware design [20], [21], [29], [33], we also need to assume

978-1-4799-8930-0/15/$31.00 ©2015 IEEE

some hardware components to be reliable (otherwise, where could they run the error checker or the recovery module?) Specifically, similar to ERSA [29], we assume a heterogeneous multicore architecture in which some cores are reliable but maybe slower, while the other cores are relaxed, i.e. with higher error rates, but faster (or more energy-efficient). We refer to the former as *reliable* cores, and the latter as *unreliable* or *relaxed* cores.

As shown in Figure 1(b), the goal of our system (referred to as FTXen) is to survive hardware faults in order to continue supporting unaffected virtual machines. More specifically, when some relaxed cores have faults, the hypervisor is still intact, and as such, only those virtual machines that are running on these cores are affected, but all other virtual machines continue running without any need for recovery.

An error resilient hypervisor is crucially important to support recent relaxed hardware architecture proposals. With a resilient hypervisor, we can place error resilient applications [29], or error resilient program components or phases (as proposed in some approximate programming models [21]) to run on relaxed cores in order to take advantage of either the faster processor speed or the lower energy cost. At the same time other applications or software components can still run on reliable cores without worrying about the underlying system software, i.e. the hypervisor, being taken down by faults in relaxed cores.

Please note that the goal of our work is not to recover affected virtual machines or applications from hardware faults on relaxed cores. Instead, our goal is to make sure that *the hypervisor itself is resilient to hardware faults on relaxed cores*. Since the hypervisor does not crash under faults, we can then leverage recent proposals [20], [25], [30], [31], [40] that exploit software techniques in order to recover applications or virtual machines that are affected by the faulty relaxed cores.

Ideally, if we can execute all hypervisor instructions on reliable cores, faults on relaxed cores cannot crash the hypervisor. Unfortunately, things are not this simple due to two reasons: (1) Many hypervisor data structures are shared by all cores. Therefore, once a faulty core erroneously changes some critical data structure, it can corrupt the whole hypervisor integrity and lead the hypervisor to crash. (2) Unlike application or virtual machines that can be bound to some specific CPU cores, many hypervisor services, such as traps, hypercalls, and context switches cannot be performed entirely on remote cores due to hardware specification.

FTXen addresses the two challenges above by completely restructuring the hypervisor. The main design principle behind the restructuring is to first identify critical data structures that are crucial for the hypervisor's integrity, and then reimplement all hypervisor operations to make sure that any hypervisor code executing on relaxed cores cannot corrupt these critical data structures, and have to communicate with reliable cores to perform updates to critical data structures. In addition, critical data structures and the communication channel need to be protected to prevent random corruption by code executed in relaxed cores.

We implement FTXen on the popular open source Xen hypervisor [9]. We evaluate FTXen on real systems for performance measurements, and use the QEMU emulator to inject hardware faults. Our experimental results show that our FTXen can successfully survive injected faults and add little performance overhead on popular HPC applications, allowing them to take full advantage of faster processors in relaxed cores. In addition, it scales well with different ratios of reliable cores vs. relaxed cores.

II. Background

Our work is based on the para-virtualized Xen on the x86_64 architecture, as shown in Figure 2. The current Xen hypervisor is not resilient to hardware faults. FTXen restructures Xen's components, interfaces, and data structures to make it resilient to errors on relaxed cores. Before going into our restructure, we first briefly describe the architecture of Xen and its vulnerability in each component to hardware faults.

In a Xen/x86 system, only the hypervisor runs with full processor privileges (ring 0 in the x86 four-ring model). Guests are restricted to run in ring 3. A guest virtual machine running on Xen is called a domain. There is a privileged domain, known as Domain0. All other unprivileged domains are referred as DomainU.

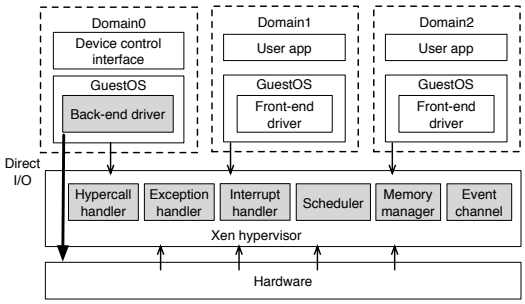


Fig. 2: Xen architecture and vulnerable components.

III. FTXen

Privileged Domain Domain0 provides control interfaces to support and manage other domains. Furthermore, another common task of Domain0 is to handle device I/O. Virtual devices under Xen use a split device driver architecture, as shown in Figure 2: the front end runs in an unprivileged domain and only issues requests to the back end. The back end, which runs in Domain0, is responsible for validating parameters and submitting I/O requests to physical device drivers.

Hardware device drivers are the most vulnerable part. Because all I/O requests are actually passed to Domain0, an error occurring in a driver can cause Domain0, and subsequently all the other domains, to crash.

Hypercall The guest OS needs to invoke hypervisor services to perform privileged operations, such as updating its page tables, using hypercalls. Hypercall arguments and return values are passed through general purpose registers. Even a single bit error can change the hypervisor control and data flow during its execution. For example, when the hypervisor executes the hypercall request to update a guest page table, if it

```
static void csched_acct(void* dummy)
{
    if ( list_empty(&sdom->active_sdom_elem) )
    {
       list_add(&sdom->active_sdom_elem,
                    &csched_priv.active_sdom);
    }
}
```

(a) A relaxed core writes a wrong pointer into a shared list

```
static void csched_acct(void* dummy)
{
     list_for_each( iter_sdom, next_sdom,
                         cshed_priv.active_sdom)
      {
           ......
      }
}
```

Fatal page fault !

(b) Fatal page fault due to the wrong pointer read

Fig. 3: A fault on relaxed core leads to fatal page fault in another core.

mistakenly writes a wrong machine frame number to the guest page table entry, the guest may randomly access a memory belonging to other guests or even the hypervisor, taking down other virtual machines.

Exception During execution, both a guest OS and Xen can trigger hardware exceptions. If a guest OS is responsible for the exception, Xen rebuilds a trap frame on the guest OS's kernel stack, and routes the exception back to the guest; otherwise, Xen calls its own exception handler.

One common cause of exception is page fault. Since the hypervisor is very vulnerable to the pointer, the bit error in an address register can generate a completely illegal address and cause a fatal page fault, bringing down the whole system.

Interrupt and event channel Xen virtualizes interrupts as events, and delivers them to the target domain asynchronously through the event channel mechanism.

Event notifications from Xen to a guest domain are raised by updating a guest's private data structure. Since the hardware does not know on which core each guest currently runs, it can deliver the event to any core. So the hypervisor executing on a relaxed core can update some private data structure of a guest that is currently not running on that core. If this core is faulty, it can take down that guest too, even though it does not run on this core.

Scheduler The main job of the scheduler is to schedule virtual CPUs (vCPUs) on a particular physical CPU, since each domain can have one or more vCPUs, but each physical CPU can run only one vCPU at one time.

In addition, the scheduler may also need to do global load balancing, which means it may directly operate on any vCPU running on other CPUs. As such, a fault on one CPU can propagate to another in the scheduling routine.

Memory Protection In a x86 system, Xen reserves a large private space in the middle of the whole virtual address space. Any memory access from a guest to this region is protected, using existing page-level protection mechanism. However, the current page level protection cannot protect a problematic access from the hypervisor code itself running on a faulty core. In the next section we will describe how to extend the existing page level protection to support fault isolation from faulty relaxed cores.

Assumption on Faults We assume the main memory is protected by error correcting code (ECC). As mentioned earlier, we also assume the reliable cores have hardware-mechanism to detect and recover from hardware faults and thus is error free. So the only concern here is the relaxed cores that have higher error rates in exchange for faster speed. We assume relaxed cores are unreliable except memory management unit (MMU), instruction cache, interrupt and exception logic.

MMU performs address translation and access permission checks. The permission bits are defined in each page table entry. FTXen relies on this page protection to prevent services on relaxed cores from corrupting critical data structures (Section IV). The instruction cache is also protected by ECC to prevent arbitrary instruction execution. The reliable interrupt and exception logic ensures that CPU will correctly trap to FTXen on privileged operations.

IV. FTXEN

The goal of FTXen is to be resilient to faults on relaxed cores. As an essential condition for error-resilience in software, the critical data structure needs to be intact in case of faults. This is especially true for system software because their main job is to control, and to manage, not to compute. So most of their data structures can affect the system's execution and behavior, not just the results like in resilient applications. Therefore, protecting the integrity of these shared data structures against faults on relaxed cores is crucially important.

Figure 3 shows an example depicting how a fault on a relaxed core can corrupt a critical data structure in Xen, and result in Xen and all virtual machines crashing. The `csched_acct()` function is in the hypervisor scheduling routine, and operates on a global data structure `csched_priv.active_sdom`. When a relaxed core writes a wrong pointer to this variable and later another core reads the pointer, a fatal page fault happens and results in a system crash.

FTXen isolates hardware faults on relaxed cores and achieves error resilience by restructuring the hypervisor memory layout and hypervisor service mechanisms to guarantee that updates to critical data structures are only performed on reliable cores. This ensures that a fault on relaxed cores cannot corrupt the hypervisor's critical data structures and would leave the other cores intact to support other virtual machines.

Figure 4 shows the overall system architecture of FTXen. First, it separates critical data structures into *shared* data regions and gives different access permissions to different types of cores: relaxed cores have only read accesses to

this region, whereas reliable cores can read and write this region. Second, it modifies the services on relaxed cores to forward operations that need to update the shared data region to be performed on reliable cores. The communication channel between relaxed and reliable cores is also designed in some way to deal with corrupted requests from faulty relaxed cores.

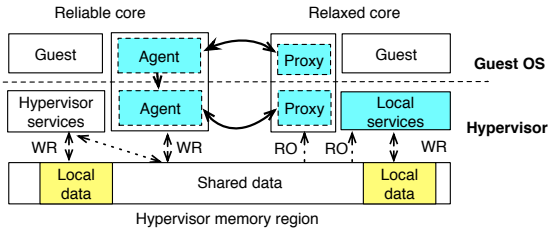


Fig. 4: FTXen system architecture.

A. Hypervisor Critical Data Protection

In the original Xen, some hypervisor memory regions are shared by all cores. Among them, one of the most important is the `idle_page_table` data structure, which is the root page table containing the address mapping of the hypervisor's private memory space and is shared by all cores. As shown in Figure 5(a), when Xen allocates a new root page table for a vCPU, it copies the mapping of the hypervisor memory region from the `idle_page_table`, except for those locally used for each vCPU (i.e., guest page table, per-domain mappings). By doing such, every vCPU has the same view of the hypervisor memory space except for those two local regions. All the vCPUs have the same access permissions to the hypervisor memory.

To be resilient to hardware faults on relaxed cores, FTXen first separates the memory regions shared across all vCPUs from those used only by each vCPU as shown in Figure 5 (a). Then it restricts write accesses to the shared regions by relaxed cores. Please note that although the "guest page table" region is only for per guest use, it is still *read-only* to relaxed cores because FTXen only allows reliable cores to update page tables in order to protect shared data. Relaxed cores can still read the shared regions and can write/read their own local regions. Such design is to avoid unnecessarily forwarding operations that only require read accesses to shared data to be performed on reliable cores, thus reducing the load on reliable cores.

To achieve the functionality above, FTXen needs to address three major design issues:

(1) How to give different memory access permissions to reliable cores and relaxed cores: Instead of using one `idle_page_table` for all cores, FTXen creates a separate `idle_page_table` for each core by copying the one in CPU0 (reliable core) at FTXen initialization time. While the address layout is the same as `idle_page_table` shown in Figure 5 (a), each page table marks different access permissions to the hypervisor memory, depending on this core's reliability type (reliable/relaxed) and the memory type (shared/local), as shown in Figure 5 (b). It leverages the page-level protection method supported by x86_64 processors in a way similar to previous work [17], [46] for other purposes.

Similarly, as demanded by a guest OS, FTXen creates a separate root page table for the running vCPU. Only reliable cores can allocate and set up the new table by copying the `idle_page_table`; in contrast, for a guest running on the relaxed core, it needs to issue a hypercall to request one reliable core to do so on behalf of them (the hypercall restructuring is discussed in Section IV-B).

To minimize the amount of shared data (and thereby reduce the load on reliable cores), we also revise some shared data structures to become local data. For example, arrays for maintaining each core's information, such as interrupt request queues (irq_stat[NR_CPUS]) for each CPU, are divided to separate per-cpu data structure, with each one stored in the corresponding core's local data region as shown in Figure 5 (a).

(2) How to manage memory access permissions for dynamic memory allocation (heap): Since the heap area used for hypervisor services can be accessible from all cores, if one faulty core accidentally writes a wrong data in heap, it can propagate the fault to all. Therefore, FTXen initially gives only read access to relaxed cores for the heap region. Then, it allows only reliable cores to do the memory allocation and reset the corresponding pages to be writable by marking the page table entries on behalf of the relaxed cores. Later, dynamic memory allocation requests from a guest OS pass through hypercalls, and thus FTXen restructures hypercalls to make only reliable cores do these allocation steps for relaxed cores. The details about hypercall restructuring is discussed in Section IV-B.

To reduce the effort of remotely performing memory allocations, FTXen enables relaxed cores to perform the memory allocation for small objects (those smaller than a page size) on its own. Specially `xmalloc()/xfree()` are used to allocate or reclaim small objects from a reserved shared memory pool. FTXen lets each core have its own local memory pool, which is initialized with contiguous pages with both read and write permissions only to the core, meaning that any faults on it will be isolated. The initialization of the memory pool is performed by reliable cores. Later `xmalloc()` can immediately return small memory objects without any communication with reliable cores.

(3) How to enforce shared-data protection:

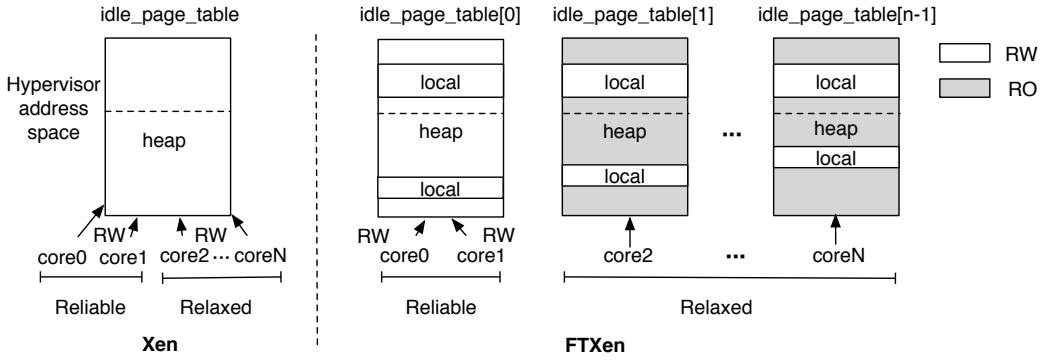
The page fault handler is of great importance to FTXen because it enforces the shared data protection against accesses from relaxed cores. Therefore FTXen allows only reliable cores to handle page faults generated by relaxed cores (the communications between relaxed cores and reliable cores are handled by hypervisor services restructured in FTXen, and the details are in Section IV-B).

Once a hardware fault on a relaxed core leads to an update to shared data, the hardware will throw a protection page fault, and the hypervisor exception handler is invoked only on reliable cores. It checks where the address belongs to and if the permission type matches. If it detects an illegal attempt (e.g. by a relaxed core), it denies the request and marks the corresponding core as "faulty". Later it can avoid scheduling any VMs on this core.

Please note that we do not modify memory layout used for guest OS and applications because Xen already restricts the

Virtual Address	Description	Category	Permission for relaxed cores
0xffff800000000000 - 0xffff803fffffffff	Machine-to-physical address translation table (map to guests)	shared	read-only
0xffff810000000000 - 0xffff817fffffffff	Guest page table	local	read-only
0xffff820000000000 - 0xffff827fffffffff	Guest private data (e.g., GDT, LDT)	local	read-write
0xffff828000000000 - 0xffff8283ffffffff	Machine-to-physical address translation table	shared	read-only
0xffff828400000000 - 0xffff8287ffffffff	Machine page frame information array.	shared	read-only
0xffff828800000000 - 0xffff828bffffffff	I/O address remapping area	shared	read-only
0xffff828c80000000 - 0xffff828cbfffffff	hypervisor text, static data, and per-cpu data	local/shared	read-write/read-only
0xffff830000000000 - 0xffff83ffffffffff	1:1 mapping to all physical memory (hypervisor heap)	local/shared	read-write/read-only

(a) Hypervisor memory layout



(b) Protecting hypervisor shared memory regions.

Fig. 5: **FTXen memory layout and shared data protection**

access from VMs to the hypervisor's private memory. So any error at the application level or the guest OS level will not take down the hypervisor. What FTXen is doing is to protect against error propagated by the hypervisor code executed on a faulty relaxed core.

B. Service Restructuring

While relaxed cores have limited access permissions on hypervisor shared memory region, they still need to provide hypervisor services, including hypercall handling for guest requests, vCPU scheduling, interrupt and exception handling, etc. We call these services *global hypervisor services*, and the other services that access only local memory regions as *local hypervisor services*.

1) Local Hypervisor Services: Local hypervisor services can still be performed by relaxed cores without any concern because they do not require updates to critical hypervisor data structures. Below, we show a few examples of such local services:

Hypercall to switch guest OS kernel stack The hypercall handler only needs to write the guest's kernel stack pointer and segment register. Although a hardware fault may result in a wrong update and a guest crash, it will not affect the hypervisor and other guests.

Local time calibration It calibrates a local CPU time and updates the current vCPU time.

Local vCPU scheduling Without dynamic load balancing across CPUs, a local scheduler can deschedule a current vCPU and select the next vCPU. Such scheduling does not interfere with other cores and only modifies the status of those vCPUs that are mapped to this core.

2) Global Hypervisor Services: For global hypervisor services, FTXen needs to 1) restructure the service mechanisms to request reliable cores to perform the services on behalf of relaxed cores, and 2) make the communication between relaxed and reliable cores error resilient to avoid error propagation. Figure 4 and Figure 6, respectively, show the overall architecture and the working flow between the two.

A proxy runs on a relaxed core and sends a service request with the corresponding request ID (task ID) and its related parameters to a reliable core. For this interprocessor data transfer, we use shared memory. For each guest running on relaxed cores, FTXen allocates a dedicated buffer shared with each agent running on reliable cores. The proxy puts the request into the buffer and waits for the corresponding agent to perform the service.

An agent running on reliable cores polls the buffer to see whether there is any incoming request. During each iteration, it performs an integrity check on the buffer's data structure (including check if the buffer pointer is valid by comparing it with its stored local value and the range, etc). Once an agent sees the request, it first checks whether it is a valid request. For example, it checks if a pointer passed by a proxy is supposed to be within a legal range, according to its semantics (e.g., the pointer to the exception frame of the guest should be within its

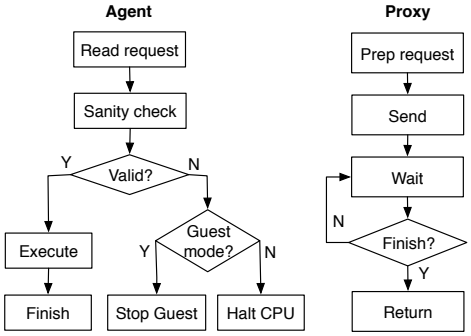


Fig. 6: FTXen proxy-agent model for global hypervisor services. A proxy running on relaxed cores sends a request through a dedicated 1-1 buffer to a reliable core, and an agent on reliable core performs the requested service on behalf of the proxy after validating the request.

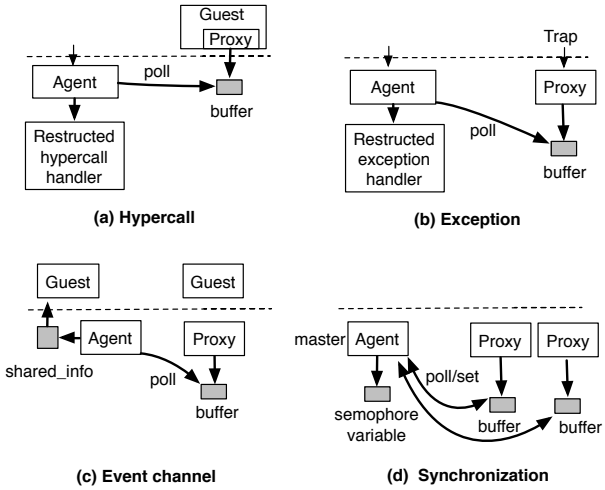


Fig. 7: Global hypervisor service restructuring

per-cpu region). If an exception handling request indicates that a page fault happens on write-protected hypervisor shared data region, the request may be corrupted as a result of a hardware fault on the corresponding relaxed core.

If the request is illegal, the agent rejects it, and if necessary, will halt the relaxed core or the guest to avoid the fault propagating further. If the request is legal, the agent will perform the requested service, and write any return results back to the proxy buffer. If the agent needs to directly write the relaxed core's or guest's memory location, referred by a pointer argument, it first checks whether it is a valid pointer by using the existing Xen internal functions (i.e. `_copy_from_user()`, `_copy_to_user()`) before writing the memory location.

Below, we discuss how FTXen restructures each global hypervisor service in more detail. Different restructuring techniques are applied to different services by considering who requests the services (e.g., hardware, guests, or hypervisor on relaxed cores), where is the best place for a proxy and an agent to run in order to achieve error resilience (e.g., on guest OS or at the entry of Xen hypervisor service routine), etc. Figure 7 illustrates the restructuring design for different global services.

Hypercall FTXen restructures hypercalls in a similar way with exception-less system call [43]. As shown in Figure 7 (a), FTXen runs a proxy in *guest OS* to *hijack the hypercall* before it traps into the hypervisor. By doing so, we can prevent any hardware faults in relaxed cores from contaminating the hypervisor. The proxy then writes the call ID and related parameters into a proxy buffer dedicated to the guest to ask an agent to perform this hypercall. For hypercalls, the agent is implemented as a daemon running in hypervisor and constantly polling the buffer.

For certain hypercalls, it is challenging for an agent to directly update the guest memory using the pointer passed by a guest, since the memory region is not mapped to the remote core. For example, when a guest issues a hypercall `HYPERVISOR_mmu_update` to update its page table, the remote agent receives only the guest virtual address pointing to the guest page table entry. To solve this issue, FTXen implements *software MMU* inside the hypervisor, walking the guest page table to find the target machine address, and remaps it to the hypervisor memory on a reliable core. To reduce the software translation overhead, FTXen also introduces a *software TLB* which caches recently translated mappings for each guest.

Some hypercall handlers need to perform a hardware context switch on the relaxed cores to complete the hypercall service. The hardware context switch involves writing CR3, flushing TLB, etc., which are only related to local context and local memory regions so that the relaxed core can perform them without concern about fault isolation. Therefore, FTXen runs a handler in each relaxed core to do the required switch operation, and allows reliable cores to call them through IPI.

Exception Exceptions may be caused in user mode or in hypervisor mode. We use one type of frequently happened exception page fault to illustrate the exception handler restructure.

At a page fault, hardware triggers an exception, and a proxy catches this and sends a handling request to the agent on a reliable core, as shown in Figure 7 (b). It reads the page fault linear address recorded in CR2 and writes it to a proxy buffer dedicated to the guest. Different from the hypercall proxy, the proxy for exception handling is running inside the hypervisor, since hypervisor first catches the hardware exception. To avoid any harmful effect from hardware faults to be propagated in the hypervisor mode, the proxy is implemented at the entry of the hypervisor service and immediately obtains necessary context information. After it is copied into a proxy buffer, the agent running on the reliable core polls the buffer in the same way as handling a hypercall.

The agent running in the hypervisor first validates the request. If it is legal, it creates an exception frame on the guest OS stack on the relaxed core, and transfers the control to the guest OS's own page fault handler. The handler then reads the exception frame and also loads the page into memory by itself, which is safe since the memory region belongs only to the relaxed core (i.e., the guest). Finally, to update the page table entry, the guest OS issues hypercall

HYPERVISOR_mmu_update.

Event Channel When one guest sends an event to another, in the original Xen, the hypervisor running on the sender core sets an event flag in a special hypervisor page (i.e., shared_info) belonging to the receiver guest. To be resilient to errors on relaxed cores, FTXen has to restructure it by having the sender (proxy) set a flag in its own buffer, and having the receiver (agent) poll the flag (Figure 7(c)) and update its own event flag.

Synchronization in Hypervisor For *each* semaphore, FTXen assigns one reliable core as the master to coordinate synchronization. A proxy running on some other core performs synchronization by setting a flag into its proxy buffer, and an agent running on the master polls the buffers and updates its local synchronization variable to keep track of the status, as shown on Figure 7 (d). To prevent deadlock and hang, each master uses a global timer to check if a relaxed core exceeds a synchronization time limit and, if so, it forces the core to release and lets others continue.

Scheduler For load balancing and global vCPU scheduling, the hypervisor needs to update vCPU run-queue on other cores. FTXen allows only reliable cores to perform global load balancing for all vCPUs including those running on relaxed cores. Specifically, when a scheduler on a reliable core needs to migrate vCPUs in/out of a relaxed core, it updates the vCPU data structure (the migration flag, host core ID, etc), and adds/removes it into/out of the vCPU run-queue on a new host core.

Device I/O and Interrupts Since the back-end driver located in Domain0 handles all I/O requests coming from the front-end drivers in DomainU, FTXen statically binds Domain0 to reliable cores to prevent any fault propagation through Domain0. I/O Advanced Programmable Interrupt Controllers (APIC) [26] are connected with external devices and responsible for delivering the interrupts generated by devices to corresponding cores. FTXen statically redirects all I/O interrupt signals to reliable cores.

C. Implementation Details

We implement FTXen by modifying Xen-3.4.4. We use Linux version 2.6.18 for both Domain0 and DomainU in our experiments. We also slightly modify the DomainU Linux kernel for the hypercall hijacking before entering the hypervisor.

In the initializing stage, FTXen builds the initial root page table called idle_page_table on CPU0. On 64 bit paging mode, the modern x86 processor may use 1G- or 2M-sized super pages instead of traditional 4K pages, which would prevent FTXen from setting the access permission with 4K granularity. Therefore, FTXen splits the super pages into 4K-sized pages and restructures page tables to have more levels accordingly. After writing the address of the root page table to CR3, each core boots with a separate instance of the root page table with different access permission, as we discussed in Section IV-A.

Xen-3.4.4 supports 40 hypercalls in total. We examined all hypercalls and restructured 9 of them, some of which provide more than one service depending on their parameters.

Hypercalls	Reason to restructure
mmu_update	Update guest page table
update_va_mapping	Update guest page table
mmuext_op	MMU operations updating shared objects: e.g., guest page table allocation
memory_op	Guest memory allocation from hypervisor heap
set_gdt	Set guest GDT (page-frame array)
update_descriptor	Update GDT/LDT entry, need to validate the descriptor
grant_table_op	Set up a grant table for inter-domain page sharing
event_channel_op	Manage inter-domain event-channel
multicall	Access the restructured hypercalls above to batch multiple hypercalls

TABLE I: Restructured hypercalls

Table I lists the restructured hypercalls and the reasons why they need to be restructured. Most of these hypercalls are related to memory management since they need to update shared data structure such as page tables. After separating the shared data and the local data, the other 31 calls perform only local services, so we do not need to restructure them.

D. Performance and Scalability Concern

Since many hypervisor operations are shifted from relaxed cores to reliable cores, performance and scalability might be a concern. However, as our results on a real machine with different ratios of reliable cores vs. relaxed cores show, this concern is not significant, mainly because of a few reasons: (1) Hypervisor operations and services are not invoked frequently and/or are very light-weight (for example, simply passing the exceptions to a guest OS), which is exactly the reason why virtual machines such as Xen and VMWare do not add much overhead over bare machines [9], [48]. (2) To minimize the loads on reliable cores, as mentioned earlier, FTXen performs all services that do not require modification to shared data locally on relaxed cores. For example, 31 of the 40 hypercalls are still performed locally on relaxed cores without any burden to reliable cores. (3) In addition, as also discussed earlier, we have broken some shared data structures so each core has its own copy, allowing more operations/services to be performed locally. (4) The software TLB avoids a significant amount of virtual address translation by walking multiple levels of guest tables for unreliable cores. (5) The load from relaxed cores is evenly distributed among all reliable cores to achieve load balancing.

V. Evaluation Methodology

A. Fault Injection

To evaluate the error resilience of FTXen, we use fault injection. Both transient and permanent faults may affect the hypervisor's states. We do not intend to study the probability of these two different types of faults corrupting the hypervisor state, as many previous works have done [31]. Instead, we focus on the comparison in terms of error-resilience between the original Xen and FTXen. Therefore, to reduce the experiment time, we inject permanent, stuck-at faults. A prior study on fault behaviors [18] shows that most device-level

faults can be modeled via single bit errors. In a processor, the faults in register files can propagate to most instructions, influence the hypervisor's control flow, change the load/store address and corrupt its data. As such, similar with previous work [22], [29], we randomly inject stuck-at bit errors in registers. These registers include 16 general purpose registers (GPRs), the instruction pointer register (RIP), the instruction register (IR), and the control register CR3.

We considered several aspects when choosing the simulation platform: (1) multicore simulation support; (2) acceptable simulation time; and (3) fine fault injection granularity. Using SIMICS+GEMS for micro-architecture level timing simulation can achieve an accurate fault injection model [31], but causes a significant slowdown(per CPU) of 5600 [13], [34]. In our experiments there is one *extra layer*-the hypervisor, so the simulation time of SIMICS would be even worse than previous work. As such, we select QEMU [10], which supports multicore simulation and the simulation time is acceptable.

We develop an automatic fault injection framework on QEMU. For repeatability we run the workload and randomly capture 10 VM snapshots (after initialization) for each evaluated application. Then, on the second round, we load the snapshots one by one and randomly inject 1-bit stuck-at register error before executing the next instruction. We run each snapshot 10 times, so for each workload, we inject 100 register faults in total.

We test if the hypervisor is still alive by using ssh service to connect to Domain0. If ssh fails, we consider the hypervisor/Domain0 crashed under one injection. For further examination on the crash causes, we check the crash messages or attach GDB to QEMU. If the ssh connection succeeds, we assume the hypervisor successfully survived the injected fault.

Our evaluation measures FTXen's error resilience capabilities, and therefore we focus on faults that finally cause system failures, instead of transient faults which may lead to silent data corruption. In reality, FTXen captures transient faults that may corrupt critical data and crash the machine in original Xen as well. Moreover, the accuracy of our fault injection can be improved by injecting faults to other components, such as register bus, latches, etc. But due to the lengthy simulation time, we cannot use a finer-grained simulation framework.

B. Performance Evaluation on Real Systems

We also evaluate FTXen performance overhead on various application-level benchmarks, including 5 recognition, mining and synthesis (RMS) applications (*canneal, streamcluster, ferret, bodytrack, and x264*) from PARSEC benchmark suit [12], SPEC CPU2006 [5] and SPECjbb2005 [6].

We run benchmarks with Xen and FTXen respectively, on a real multicore machine with Intel(R) Core(TM) i7-2600 processors and 16GB memory. The default CPU frequency of all cores is 1.6GHZ.

We also measure the potential performance improvement when running these applications on faster relaxed cores by adjusting the CPU frequency of three cores to 2.4GHZ while keeping the frequency of the one reliable core on 1.6GHz.

VI. Evaluation Results

A. Error Resilience

We compare the hypervisor crash rate on three configurations: (1) Original Xen, (2) Original Xen but pin Domain0 to the reliable core, and (3) FTXen. The reason we also evaluate configuration (2) is because Domain0 is a critical part of the hypervisor and implementing this is very easy. By comparing (2) and (3), we can understand the necessity for doing all the other complex restructuring.

As a starting point for easy understanding of hypervisor crash behavior, we first use a micro test application *loop*, which executes an empty loop in DomainU. This micro application has very little interaction (no hypercall, no page fault, etc.) with the hypervisor. But even in this case, we observe a very high hypervisor crash rate as 82% with the original Xen. Pinning Domain0 to the reliable core can reduce the crash rate by 10%, but the majority of the injected faults still result in crashes of the whole system (hypervisor and all VMs). On the other hand, as seen in the last column in Table II, FTXen is error-resilient.

We then run applications from SPEC CINT2006 with fault injections on relaxed cores. The results are shown in Table II. The hypervisor's crash rate is as high as 95% with original Xen, but FTXen can survive all injected faults in all cases.

App	Xen	Pin-Dom0	FTXen
loop	82%	69%	0
h264ref	97%	91%	0
omntpp	95%	87%	0
perlbench	95%	79%	0
bzip2	97%	87%	0
astar	91%	82%	0
gcc	97%	82%	0
hmmer	97%	89%	0

TABLE II: Hypervisor crash rates of applications under different configurations (Xen, Xen with pinning Domain0 to reliable cores, and FTXen). We only show the results for 7 SPEC CPU2006 applications here to save space. The results for the others are similar.

Location	Xen	Pin-Dom0	FTXen
GPR	92%	80%	0
RIP	83%	80%	0
IR	100%	100%	0
CR3	50%	47%	0

TABLE III: Hypervisor crash rates of different fault injection locations. CR3 masks more injected faults than other registers in original Xen. FTXen survives unmasked faults in all registers.

Table III shows the impact of the injected faults on different registers. CR3 is more tolerant to injected faults in original Xen because only bit 12-51 of CR3 is used for address translation and thus the fault injections on other bits are masked by the register. However, the majority of faults cannot be masked and result in hypervisor crash. For example, all the faults injected to IR result in hypervisor crash in original Xen, mostly because the erroneous instructions give rise to fatal illegal opcode trap. But FTXen can successfully survive such faults.

B. Hypervisor Crash Analysis

To show more insights for understanding, we examine the applications' crash causes under configuration Pin-Dom0, i.e.

Benchmark	Description	Xen	FTXen	FTXen(R)
canneal	Simulate cache-aware annealing to optimize routing cost of a chip design	1.00	0.99	1.38
streamcluster	Online clustering of an input stream	1.00	0.98	1.47
ferret	Content-based similarity search of feature-rich data	1.00	0.99	1.46
bodytrack	Tracks a human body with multiple cameras through an image sequence	1.00	1.00	1.53
x264	H.264/AVC(Advacned Video Coding) video encoder	1.00	0.99	1.40
SPECint2006	Evaluate CPU integer performance	1.00	0.99	1.46
SPECfp2006	Evaluate CPU floating point performance	1.00	1.00	1.48
SPECjbb2005	Evaluate the perf. of servers running typical Java business applications	1.00	0.99	1.45

TABLE IV: Normalized performance of Xen, FTXen and FTXen (R) on real machine with 1 reliable core and 3 relaxed cores. The difference between Xen and FTXen shows the overhead of FTXen. The FTXen(R) column shows the performance improvement on the relaxed architecture.

pinning Domain0 to the reliable core. Because all applications have similar results, we randomly sample 200 crashes for our analysis. We categorize the crash causes into 4 types, as shown in Figure 8.

The first dominating type is deadlock (41%), which typically results in hypervisor hang. This is not surprising because the hypervisor needs to frequently synchronize operations among CPU cores. If a faulty relaxed core fails to release a lock or semaphore, the other cores can wait there forever. In FTXen, we eliminate such deadlocks by reimplementing synchronization operations as described earlier in Section IV-B2.

The second dominating type is page fault (26%), which is triggered by invalid memory accesses from the hypervisor. Because we inject register bit errors, it can change some load/store addresses. Such wildly arbitrary memory accesses can also happen while executing hypervisor code. FTXen extends the memory protection mechanism to guarantee that the hypervisor running on relaxed cores cannot write shared data. Therefore, when an illegal access occurs, the control transfers to the page fault handler on the reliable core to perform all necessary updates to shared data such as page tables. So the error does not affect the hypervisor.

The third one is fault trap (26%), including general protection fault, invalid operation fault, double fault, etc. These can be triggerd by illegal software operations or a buggy fault handler routine (becasue the injected fault can affect the exception handler routine). FTXen migrates the exception handler to the reliable core. In original Xen, a fatal exception occurring in hypervisor indicates the system integrity is severely damaged In FTXen, these fatal exceptions can corrupt only this CPU's local data, leaving the integrity of the hypervisor and other cores intact.

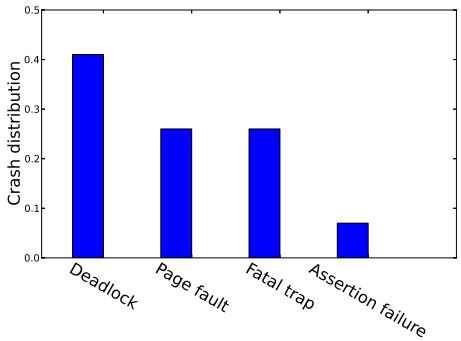


Fig. 8: Distribution of crash causes

The last one is assertion failure (7%). The hypervisor inserts assertions to check the correctness of system state at runtime. A failure of assertion is usually caused by the corruption of hypervisor data structures and further results in Xen panic(), i.e. a voluntary halt to the whole system. FTXen solves this problem by separating the local and global data structures so that the data corruption can only impact faulty cores.

C. Performance Evaluation

Table IV shows the results of a series of benchmarks. The performance is normalized to the results with the original Xen, the higher the better. Xen and FTXen refer to the results on SMP platform, while FTXen (R) represents the results on simulated relaxed cores.

For the evaluated resilient applications (mining, recognition, etc.) that motivate the relaxed hardware architecture, compare the relative performance between Xen and FTXen with all 1.6GHz cores, FTXen introduces only 0.8% overhead on average. If the relaxed cores can be faster (2.4GHz), we could see that these applications can gain 1.45 times of performance improvement on average. SPEC CINT, SPEC CFP and SPECjbb have similar performance with RMS applications. The results indicate that FTXen can effectively enable computationally intensive, memory intensive server workloads with relaxed multicore architecture.

D. Configuration Ratios of Reliable vs. Relaxed Cores

In order to estimate FTXen's reliable core ability to serve multiple relaxed cores, we test the application performance with multiple running virtual machine instances. Figure 9 shows the normalized average performance of a subset of SPEC INT2006 concurrently running on from 1:1 to 1:7 varying ratios of reliable cores vs. relaxed cores. With even a ratio of 1:7, the performance overhead of FTXen is still small (only 3.8%). It indicates that FTXen should be able to support systems in data center environment without the need of using too many reliable cores per machine.

E. OS Operation Overhead

To more precisely measure the restructuring overhead of FTXen compared with the original Xen, we further perform series of micro OS benchmark lmbench [45] experiments. The goal of these experiments is to study the FTXen's overhead in a more fine granularity way—the OS operations. lmbench

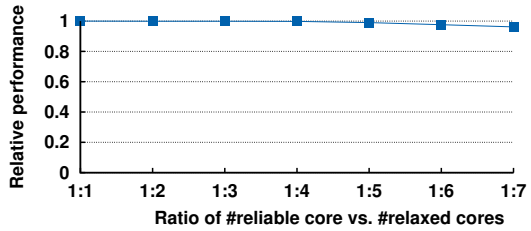


Fig. 9: Normalized average performance of a subset of SPEC INT2006 with different reliable core to relaxed cores ratios. Other applications have similar results.

Config	null call	null I/O	stat	open close	slct TCP	sig inst	sig hndl	fork proc	exec proc	sh proc
Xen	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
FTXen	1.00	0.97	1.00	0.98	1.00	0.99	1.00	0.83	0.84	0.85
FTXen(R)	1.56	1.45	1.53	1.46	1.50	1.51	1.53	1.25	1.26	1.28

TABLE V: lmbench: Normalized processes performance

contains 37 micro benchmark which measure the OS performance. Table V, VI,and VII show the results. The key result is: FTXen invokes *zero* overhead on these OS operations on the relaxed architecture, which demonstrates the effectiveness of our performance optimization.

In the process benchmark result (Table V), *fork*, *exec* gain smaller benefits from relaxed architecture because both of them needs to heavily update page tables.

Table VI measures the context switch time for a different number of processes with a different working set size. As the working set size increases, the pure overhead introduced by context switch becomes smaller and smaller since cache misses dominate. With HPC workloads in data centers, most working set sizes are large.

In the file system latency measurements (Table VII), the *page fault* and *mmap* show larger latency than others because they cause two OS operations—one IPI for page fault and one hypercall hijacking for guest page table updates.

VII. Related Work

Relaxed Hardware Ersa [29] proposed a low-cost robust system architecture for probabilistic applications that have algorithmic or cognitive resilience to errors. Relax [20] and Encore [23] exploit such application characteristics to recover from hardware faults in software. Similarly, Truffle [21] proposes an approximate programming model to identify approximating components in software to lower the hardware energy and cost. SWAT [25], [30], [31], [40],Shoestring [22] and ReStore [49] leverage software-level symptom-based techniques to detect hardware faults. Li *et al.* [32] proposed a lightweight recovery method to achieve the best-effort application-level correctness.

While the above research work has proposed promising directions, they all aimed at providing error resilience in applications. Complementary to the prior work, our work focused on making system software error-resilient.

Config	2p 0K	2p 16K	2p 64K	8p 16K	8p 64K	16p 16K	16p 64K
Xen	1.00	1.00	1.00	1.00	1.00	1.00	1.00
FTXen	0.67	0.74	0.75	0.71	0.75	0.73	0.77
FTXen(R)	1.01	1.02	1.02	1.08	1.14	1.11	1.14

TABLE VI: lmbench: Normalized context switch performance

Config	0K crt	0K dlt	10K crt	10K dlt	mmap	prot fault	page fault	100fd selct
Xen	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
FTXen	0.97	0.96	0.97	1.00	0.75	0.51	0.75	0.98
FTXen(R)	1.43	1.44	1.46	1.48	1.13	0.82	1.10	1.48

TABLE VII: lmbench: Normalized file & VM system performance. (crt: create, dlt: delete) Although FTXen invokes some overhead on prot fault, it does not have any weight in the real world.

Fault Containment Hive [16] and Cellular Disco [24] are multi-cellular kernels that are designed to contain faults on multiprocessor architecture. The system is partitioned to a set of cells. Each cell is a fault containment unit configured with its own physical resources and runs as an independent multiprocessor kernel by treating a multi-processor machine as a distributed system of many nodes. With such architecture, it is hard for them to support multi-threaded applications that share memory. FTXen supports fault containment at virtual machine level. Each virtual machine can support applications in the same way as Xen without any restrictions, e.g., these applications can be multi-threaded with shared memory.

Some work [28], [46] separates Linux drivers from kernel to prevent faulty drivers from crashing the kernel. They consider *driver software bugs* more and do not target handling failures introduced by wild hardware faults. While we have learned some useful lessons from the above work, FTXen differs significantly in that we restructure every component in the hypervisor to make it resilient to hardware errors on relaxed cores.

VIII. Conclusions

Relaxed hardware architecture is a promising direction in addressing the emerging challenge of the high error rate in hardware with a low cost. To enable such architecture, one of the major issues is the error resilience of system software. This work makes a first step in restructuring the virtual machine hypervisor to be resilient to errors on relaxed cores, making the relaxed hardware proposal one step closer to reality, especially for data centers or cloud platforms where virtual machines are often used to support HPC applications. Our experimental results have shown that FTXen can survive all injected faults with little performance overhead on real machines. In addition, it also scales well under different configuration ratios of reliable cores vs. relaxed cores.

Acknowledgment

We thank all anonymous reviewers for their feedback and suggestions. We also thank Haogang Chen for his helpful discussion. This work is supported by the National Science Foundation under grant CSR-1217408, CSR-1321006 and expedition.

REFERENCES

[1] Google cloud platform. https://cloud.google.com/products/compute-engine/.

[2] High performance computing in the cloud. https://aws.amazon.com/hpc/.

[3] High performance computing on microsoft azure for scientific and technical applications. http://research.microsoft.com/en-us/projects/azure/high-perf-computing-on-windows-azure.pdf.

[4] Ibm hpc management suite for cloud. http://public.dhe.ibm.com/common/ssi/ecm/en/dcs03007usen/DCS03007USEN.PDF.

[5] SPEC CPU2006. http://www.spec.org/cpu2006.

[6] SPECjbb2005. http://www.spec.org/jbb2005.

[7] N. Aggarwal and P. Ranganathan. Configurable isolation: building high availability systems with commodity multi-core processors. In *In Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 470–481. ACM Press, 2007.

[8] T. M. Austin. Diva: a reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 32, 1999.

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, 2003.

[10] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, 2005.

[11] D. Bernick, B. Bruckert, P. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. Nonstop advanced architecture. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 12–21, 2005.

[12] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[13] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.

[14] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *Micro, IEEE*, 25(6):10–16, 2005.

[15] F. A. Bower, P. G. Shealy, S. Ozev, and D. J. Sorin. Tolerating hard faults in microprocessor array structures. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, DSN '04, pages 51–, Washington, DC, USA, 2004. IEEE Computer Society.

[16] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault containment for shared-memory multiprocessors. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 12–25, New York, NY, USA, 1995. ACM.

[17] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 2–13, New York, NY, USA, 2008. ACM.

[18] G. Choi, R. Iyer, and D. Saab. Fault behavior dictionary for simulation of device-level transients. In *Computer-Aided Design, 1993. ICCAD-93. Digest of Technical Papers., 1993 IEEE/ACM International Conference on*, pages 6–9, 1993.

[19] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. Curios: Improving reliability through operating system structure. In *USENIX Symposium on Operating Systems Design and Implementation*, December 2008.

[20] M. de Kruijf, S. Nomura, and K. Sankaralingam. Relax: an architectural framework for software recovery of hardware faults. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 497–508, New York, NY, USA, 2010. ACM.

[21] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 301–312, New York, NY, USA, 2012. ACM.

[22] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: Probabilistic soft error reliability on the cheap. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 385–396, New York, NY, USA, 2010. ACM.

[23] S. Feng, S. Gupta, A. Ansari, S. A. Mahlke, and D. I. August. Encore: low-cost, fine-grained transient fault recovery. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 398–409, New York, NY, USA, 2011. ACM.

[24] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular disco: Resource management using virtual clusters on shared-memory multiprocessors. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, pages 154–169, New York, NY, USA, 1999. ACM.

[25] S. K. S. Hari, S. V. Adve, and H. Naeimi. Low-cost program-level detectors for reducing silent data corruptions. In *Proceedings of the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.

[26] Intel 64 and ia-32 architectures software developer manuals. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[27] The international technology roadmap for semiconductors 2009. http://www.itrs.net/Links/2009ITRS/Home2009.html.

[28] A. Kadav, M. J. Renzelmann, and M. M. Swift. Tolerating hardware device failures in software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, 2009.

[29] L. Leem, H. Cho, J. Bau, Q. A. Jacobson, and S. Mitra. Ersa: error resilient system architecture for probabilistic applications. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, 2010.

[30] M.-L. Li, P. Ramachandran, S. Sahoo, S. Adve, V. Adve, and Y. Zhou. Trace-based microarchitecture-level diagnosis of permanent hardware faults. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 22–31, 2008.

[31] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. Understanding the propagation of hard errors to software and implications for resilient system design. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, 2008.

[32] X. Li and D. Yeung. Application-level correctness and its impact on fault tolerance. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 181–192, 2007.

[33] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. Flikker: saving dram refresh-power through critical data partitioning. *SIGARCH Comput. Archit. News*, 39(1):213–224, Mar. 2011.

[34] M. Marty, B. Beckmann, L. Yen, A. Alameldeen, M. Xu, and K. Moore. ISCA tutorial. http://research.cs.wisc.edu/gems/tutorial.html, 2005.

[35] M. Mueller, L. C. Alves, W. Fischer, M. L. Fair, and I. Modi. Ras strategy for ibm s/390 g5 and g6. *IBM J. Res. Dev.*, 43(5):875–888, Sept. 1999.

[36] E. B. Nightingale, J. R. Douceur, and V. Orgovan. Cycles, cells and platters: an empirical analysisof hardware failures on a million consumer pcs. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, 2011.

[37] M. D. Powell, A. Biswas, S. Gupta, and S. S. Mukherjee. Architectural core salvaging in a multi-core processor for hard-error tolerance. In *In Proc.of the 36th Annual International Symposium on Computer Architecture*, 2009.

[38] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, 2000.

[39] E. Rotenberg. Ar-smt: a microarchitectural approach to fault tolerance in microprocessors. In *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*, june 1999.

[40] S. K. Sastry Hari, M.-L. Li, P. Ramachandran, B. Choi, and S. V. Adve. mswat: low-cost hardware fault detection and diagnosis for multicore systems. In *Proceedings of the 42nd Annual IEEE/ACM International*

Symposium on Microarchitecture, MICRO 42, pages 122–132, New York, NY, USA, 2009. ACM.

[41] E. Schuchman and T. N. Vijaykumar. Rescue: A microarchitecture for testability and defect tolerance. In *In ISCA 05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 160–171. IEEE Computer Society, 2005.

[42] P. Shivakumar, S. Keckler, C. Moore, and D. Burger. Exploiting microarchitectural redundancy for defect tolerance. In *Computer Design, 2003. Proceedings. 21st International Conference on*, pages 481–488, 2003.

[43] L. Soares and M. Stumm. Flexsc: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[44] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. Exploiting structural duplication for lifetime reliability enhancement. *SIGARCH Comput. Archit. News*, 33(2):520–531, May 2005.

[45] C. Staelin and H. packard Laboratories. lmbench: Portable tools for performance analysis. In *In USENIX Annual Technical Conference*, pages 279–294, 1996.

[46] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, 2003.

[47] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery using simultaneous multithreading. In *Proceedings of the 29th annual international symposium on Computer architecture*, ISCA '02, 2002.

[48] C. A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, Dec. 2002.

[49] N. Wang and S. Patel. Restore: symptom based soft error detection in microprocessors. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, June 2005.