2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

DualVisor: Redundant Hypervisor Execution for Achieving Hardware Error Resilience in Datacenters

Xin Xu
Department of Electrical and Computer Engineering
George Washington University
Washington, DC, USA
xuxin@gwmail.gwu.edu

H. Howie Huang
Department of Electrical and Computer Engineering
George Washington University
Washington, DC, USA
howie@gwu.edu

***Abstract*—Virtualization technology as the foundation of cloud computing provides many benefits in cost, security, and management, but all of them rely on the reliability of the underlying virtualization software - the hypervisor (or virtual machine monitor). Cloud data centers are built upon 10Ks to 100Ks commodity servers. Hardware errors in these large scale computer systems are not rare events. When hardware errors occur during the hypervisor execution, they may cause failures or data corruptions in co-located VMs, undermining the whole system reliability. In this paper, we propose DualVisor, that uses a software redundancy based fault tolerance technique to protect the hypervisor from hardware errors. DualVisor replicates hypervisor executions and data structures for error detection and recovery. In this work, we first study the need for a hardware error-resilient hypervisor. Then, we discuss the design considerations in detail. We implement a prototype in the hypervisor to demonstrate the feasibility and evaluate the performance overhead. Our preliminary results show that the performance overhead of DualVisor is fairly small (less than 6%) for tested applications.**

***Index Terms*—virtualization; reliability; hardware error; data center;**

I. Introduction

Cloud data centers typically consist of a large number of machines. With virtualization technology, each machine can host tens of virtual machines (VM) running on top of the abstraction layer of the hypervisor or virtual machine monitor (VMM). Both hardware and software reliability are major research challenges in such virtual systems. In this work, we focus on improving the reliability of the hypervisor against hardware errors.

Hardware errors may frequently occur in large-scale data centers. It is well known that various sources (e.g., particle strikes and packaging impurity) may cause temporary or permanent faults in CPUs and memory [1], [2], [3], [4], [5], [6], [7], [8]. A field study on DRAM errors suggests that the failure rate of the Jaguar system is equal to one failure for approximately every six hours [2], and multi-bit errors contribute about 50% of total memory errors. A study on the ASC Q supercomputer, which consists of 8192 CPUs, has shown that the average weekly count of CPU failures is 27.7 [9]. As manufacturing technology scales, error rates in future systems are expected to increase significantly [8].

The increasingly high error rate in CPU and memory poses a threat to virtual system reliability. In virtualized systems, a hypervisor, which runs with the highest privilege, manages the hardware resources and all the VMs. A control VM (or driver VM) provides device drivers and management interfaces to other guest VMs. Users may run their own applications inside of guest VMs. Uncorrected hardware errors may affect various software components of virtualized systems, including applications, operating systems, virtual machines, and the hypervisor.

Particularly, the hypervisor is single pointer of failure in the virtualized system, and its vulnerability to hardware errors should not be ignored. Hardware errors can affect the hypervisor and cause all-VM failure. They may even propagate to VM and applications, causing applications failures or silent data corruptions, which may easily cause incorrect diagnosis and unsuccessful recovery [10]. Previous work has been conducted to handle hypervisor failures based on the micro-reboot technique [11]. However, a VM may fail after rebooting the hypervisor. Also, it cannot handle corruptions that are difficult to detect (e.g. silent data corruptions).

In this paper, we propose DualVisor which utilizes redundant execution technique to improve the hypervisor reliability against hardware errors. DualVisor is designed based on software redundancy to protect the hypervisor in both executions and data. It can detect and recover errors within the hypervisor context before they affect VMs. We discuss the major design parameters, implement our approach in the hypervisor, and evaluate the performance overhead to applications. Specifically, we made two contributions in this paper:

- To our base knowledge, DualVisor is the first software technique designed for the hypervisor to provide both hardware error detection and recovery capabilities. DualVisor can detect data corruptions in the replicated regions (two copies are required), and allows a light-weight error recovery approach (three copies are required) instead of rebooting the hypervisor.
- We evaluate various design parameters in detail and build a system prototype to demonstrate the feasibility of DualVisor. The hypervisor runs at the lowest software stack with the highest privilege. Designing redundant execution is very challenging for the hypervisor. We conduct thorough evaluation and profiling to understand

978-1-4799-8006-2/15 $31.00 © 2015 IEEE
DOI 10.1109/CCGrid.2015.30

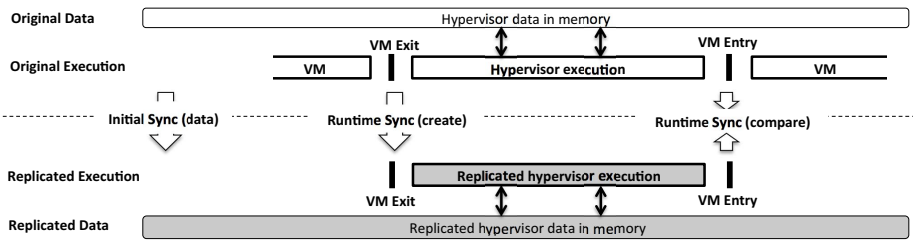


Fig. 1: Hypervisor Executions

the hypervisor behaviors, and carefully choose design parameters. DualVisor selectively replicates executions and data, covering 87% of the total number of hypervisor executions with only less than 6% performance overhead.

The paper is organized as below. Section II describes the vulnerability of the hypervisor to hardware errors. Section III presents our approach of designing redundancy for the hypervisor and discusses major design parameters. Section IV discusses the implementation in detail. Section V shows evaluation results on the performance overhead of DualVisor. Section VI discusses related work. Section VII concludes the paper and discusses the further work.

II. Hypervisor Basics and Its Vulnerability

The hypervisor is frequently activated to perform high-privilege operations, making it more vulnerable to both CPU and memory errors than one would expect. More importantly, due to the critical role of the hypervisor, errors affecting the hypervisor may cause severe consequences. In this section, we first introduce the basics of the hypervisor. Then, we discuss the hypervisor vulnerability in three areas:

- Vulnerability to CPU errors
- Vulnerability to memory errors
- Consequences if hardware errors affect the hypervisor.

A. Hypervisor Basics

Fig. 1 (the upper part) shows a typical execution flow of the hypervisor in current systems. After a virtualized system starts, the hypervisor states are in memory (e.g. memory region X). A hypervisor execution may be activated by VMs or hardware. The transition between the hypervisor and VMs can be assisted by hardware (e.g. Intel VMX [12] and AMD SVM [13]). In Intel VMX, the transition from VM executions to hypervisor executions is *VM Exit*, and the transition from hypervisor to VM is *VM Entry*. Right after VM exit, the execution context of the hypervisor will be loaded to CPU (e.g. CPU X), and the hypervisor execution will start operating on the hypervisor data in memory. The hypervisor execution follows the paths that are carefully defined. For example, Intel VMX architecture defines about 60 reasons that may trigger a VM exit. The hypervisor has function handlers to perform corresponding operations [14], [15]. After the hypervisor execution is finished (VM entry), the CPU context is switched back to VMs.

B. CPU Errors

The vulnerability of the hypervisor to CPU errors comes from the high frequency of hypervisor activities. The hypervisor instructions are frequently running in CPUs. We conduct a number of experiments to examine the frequency of hypervisor executions. The experiments run in a server with one Xeon E5506 processor with 12GB memory. We measure the number of hypervisor executions per second when the VMs are running. The results are shown in Fig. 2.

Fig. 2 (a) shows the box plot of hypervisor frequency when two VMs are running. The hypervisor is generally activated more than 5,000/s for CPU and memory intensive benchmarks, such as *mcf and bzip2* [16] and *freqmine and canneal* [17]. For I/O intensive *postmark* [18] and mixed-workload *x264* [17], the hypervisor is activated at a much higher frequency. Fig. 2 (b) shows the normalized frequency as the number of VMs increases. When the number of VMs increases from one to two, the frequency increases accordingly (upto 1.9x in *canneal*).

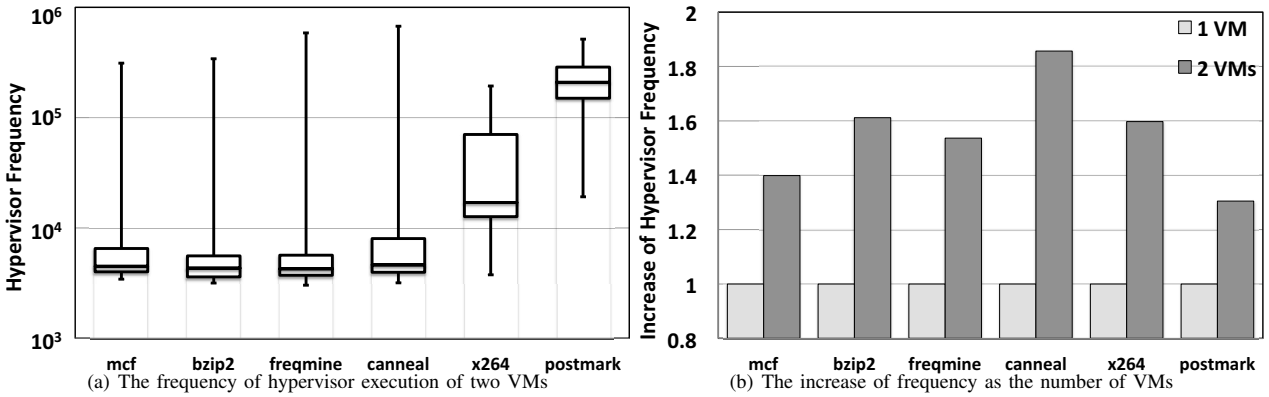
Not only the frequency is high, but also the utilization can be also high on physical cores. Research has been proposed to use dedicated cores to run hypervisor operations to reduce the overhead to VMs and applications for future virtual systems [19]. In those systems, one can expect that the hypervisor may fully utilize the dedicated cores. Such high frequency and high CPU utilization of dedicated cores greatly increase the hypervisor vulnerability to CPU errors.

C. Memory Errors

The vulnerability of the hypervisor to memory errors comes from two aspects. First, the lifetime of hypervisor data can be longer than VMs, resulting a higher probability of being affected by memory errors. Hypervisor data are allocated in memory from the beginning of system start till a reboot or power-off, and soft errors may occur at any time.

Second, memory operations are frequent in hypervisor executions. We measure the ratio of memory instructions in the hypervisor activities, using Xen 4.1.2 with two VMs in Simics, a full system simulator [20]. Our experiments show that 71.8% of hypervisor instructions are memory related. This also contributes to the vulnerability of the hypervisor to memory data.

Note that, handling hypervisor memory errors is more complicated than VM memory errors. VM memory errors can be handled with low-cost techniques, such as memory page



(a) The frequency of hypervisor execution of two VMs

(b) The increase of frequency as the number of VMs

Fig. 2: VM Exit Frequency

off-line. However, because the hypervisor runs at the lowest level of the virtual system software stack, errors in hypervisor data cannot be handled easily.

D. Consequences of Hardware Errors

We conduct fault injection experiments to study the impact of hardware errors on the hypervisor [21]. We find that errors in the hypervisor may propagate to various components in the system, causing system failures, VM failures, application failures and even data corruptions. although the memory footprint and the CPU utilization of the hypervisor might be smaller than the total of all VMs, the vulnerability of the hypervisor to hardware errors should be minimized in order to build a reliable cloud-computing ecosystem.

III. DualVisor Design

Currently, virtualization provides fault tolerance techniques at the VM level to handle failures. For example, checkpointing VMs is a standard technique for improving VM reliability. While users can selectively protect VMs and change configurations such as checkpointing intervals, they often use VM-checkpoint with little consideration of hardware errors (e.g. hardware error rate). Therefore, many VM-level techniques, such as VM checkpointing and replication [22], need to run constantly or periodically in the lifetime of the protected VM to ensure the reliability. This reduces the overall system utilization, as it requires additional computing, storage, and networking resources.

To tackle the challenge of providing hardware error resilience in virtualized systems, we propose an automatic management framework as illustrated in Fig 3 (in dashed boxes). The goal of this framework is to automatically monitor the current vulnerability, adjust the level of protection, and provide sufficient protection for virtualized systems. This automatic management module consists of three major components:

- Hardware error reporting module to report the current system reliability based on modeling and prediction
- Fault tolerance techniques for guest VMs that can be easily controlled and configured
- Fault tolerance techniques for the hypervisor that can also be controlled by the administrators.

Previous research has shown that it may be possible to model or predict errors in CPU and memory [23], [24], [7], [9], [25]. This can be utilized to construct the first component - hardware error reporting module, although efforts are still required for system implementation. VM-level fault tolerance techniques, such as checkpointing VMs, are already available in current virtual systems. They can be easily configured by the hypervisor to construct the second component. However, the third component in this framework has not been properly addressed, which is the focus of this work.

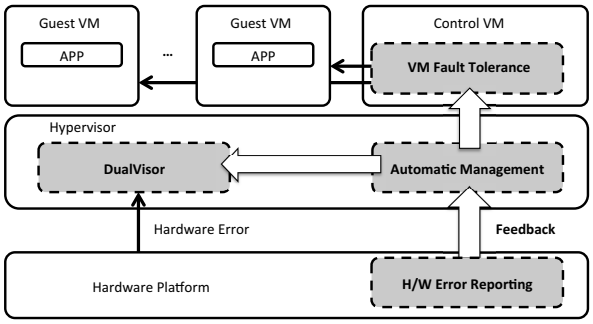


Fig. 3: Automatic Framework with DualVisor

As a stand-alone fault tolerance technique, DualVisor protects the hypervisor from hardware errors. As a critical component in the automatic management framework (as shown in Fig. 3), DualVisor can be easily configured by software. The automatic management module can now configure the level of protection to both guest VMs and the hypervisor based on the feedback of current hardware reliability.

In the following, we first describe the overall design of DualVisor. Then, we discuss each component in detail. DualVisor has many design parameters that should be carefully chosen to balance the cost (e.g. performance overhead and re-engineering cost) and the benefits (e.g. protection strength). We will discuss and evaluate the trade-off, and explain our considerations for each component.

A. Hypervisor Redundant Executions

DualVisor uses redundant execution to protect the hypervisor. Replication is done in both execution and data. Note that while redundancy [26], [27] is not a new idea, this work focuses on the challenges of designing and implementing software redundant executions for the hypervisor, especially how to achieve the redundancy with minimal overhead.

Fig. 1 (the lower half) illustrates the design concept of DualVisor. Note that this is the high-level design concept, and we will discuss the design parameters in detail shortly in this section. In general, we replicate both hypervisor executions and hypervisor data. This replication can be done for all hypervisor operations or for just selected ones. Replicas are periodically synchronized with each other to create redundant inputs for replicas or to compare states for error checking. There are three types of synchronizations to enable redundancy.

The first type is *initial synchronization*. After the hypervisor is initialized, a redundant copy of hypervisor data should be created and stored in a different memory region. This synchronization is mainly used to set up the redundant execution data. Therefore, the initial synchronization is required only once.

The second type is *runtime synchronization (create)*. It creates the execution context and inputs for the replicated execution, so that two replicas receive the same context and inputs (e.g. the VM exit reason). This runtime synchronization is required on each VM Exit that will be replicated (protected). Two parameters should be carefully considered for designing this runtime synchronization. The first parameter is which (or all) hypervisor activities should be replicated? The second one is where should we create the redundant execution (in the same CPU or two different CPUs). We will discuss these parameters later in this section.

The third type is *runtime synchronization (compare)*. Hypervisor executions generate the outputs containing the returned values to VMs and the modified hypervisor states. This runtime synchronization (compare) is required to check these outputs for error detection. It is enabled at the end of the hypervisor execution. We can replicate all data at once in this step, or only replicate a portion of hypervisor data depending on some parameters (e.g. their importance or vulnerability). We will discuss this design parameter shortly.

When there is no error, the outputs of replicas will be the same, and the system will continue as normal execution. If there is an error, two replicas will return difference states (e.g silent corruptions in data). The runtime synchronization (compare) can identify this difference and invoke proper error handling procedures. Errors may cause fatal failures immediately in one execution before runtime synchronization (compare), and the hypervisor execution may not continue to the error handling routines. In this case, fatal failures will be reported by hardware exceptions (e.g. machine check exception, MCE). Exception handlers in the hypervisor will handle this fatal failure immediately (before synchronization). Therefore, fatal failures can still be correctly handled.

The hypervisor is at the lowest level of the software stack in virtual systems. This creates difficulties to adopt existing infrastructure to carry out this design (e.g., *fork* and *ptrace*) [28], [29]. We need to modify the hypervisor software to enable redundant executions. We already list several design options when we explain the overall design above. In the following, we explain these design parameters in detail. Then, we discuss our considerations when choosing these parameters. Basically, we would like to consider following questions:

- What should we replicate?
- Where should we replicate them? (in which CPU context?)
- How should we replicate data?
- How should we replicate executions?
- When should we synchronize the replicated states?

B. The Sphere of Replication

We first examine what hypervisor executions should we include into the scope of replication. Should we replicate all hypervisor executions or only a part of them? By replicating more hypervisor executions, we increase the protection level. But it requires more efforts in re-engineering the hypervisor and potentially higher overhead. The trade-off is clear, but there is no quantitative data that can guide us to make decisions. To better understand this trade-off, we profile the hypervisor executions and collect detailed statistics.

We utilize the experimental setup in Section V to measure the number of VM exits for each VM exit reason. We run the application inside one VM in the test machine. We also collect the results when the VM is booted. Fig. 4 shows the profiling results. Although there are sixty VM exit reasons defined by Intel VMX, only eight reasons are frequently triggered. These eight reasons account for 99.6% of total number of VM exits on average. This is because many VM exit reasons are defined to handle abnormal behaviors when the system and VMs are in incorrect state such as crashes. These VM exit reasons are unlikely used in normal executions. Among these triggered VM exit reasons, two of them, external interrupt (EXT_INT) and non-maskable interrupt (NMI), are most frequently used, accounting for 28% and 45% respectively. We further profile the type of NMI and find that the most frequent NMI is page fault exception.

Based on this profiling result, we choose to selectively replicate frequently utilized VM exits rather than replicating all of them. Specifically, we replicate five VM exit reasons: NMI (for page fault exception), external interrupt, pending virtual interrupt, CPUID, and control register access (the bottom five types shown in Fig. 4). By replicating only these five VM exits, we cover about 87% of VM exits on average. We will explain in detail the implementation for each VM exit in Section IV.

C. Replicating CPU Context

The second design parameter is where to execute redundant copy (in which CPU context). The redundancy can be either temporal (in the same CPU) or spacial (in two different CPUs).

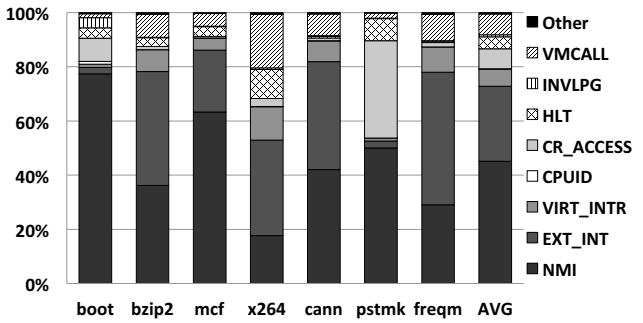


Fig. 4: Distribution of VM exit reasons. NMI: non-maskable interrupt, EXT_INT: external interrupts, VIRT_INTR, pending virtual interrupt, CPUID: CPUID instructions, CR_ACCESS: control register access, HLT: HLT instructions, INVLPG: INVLPG instructions, VMCALL: hypercall.

Fig. 5 illustrates the difference between (a) original executions without replication, (b) temporal redundancy, and (c) spatial redundancy.

Spacial redundancy (Fig. 5 (c)) can be achieved by using another physical core for redundant execution. In this way, two copies can be executed in parallel. Hypervisor executions that have a relatively longer execution time may benefit from this approach. However, this approach requires setting up the context in another CPU core. The CPU context of the hypervisor executions includes the VM exit reason and guest VM data structures, such as guest VCPU registers and domain information. This information should be replicated among two redundant executions. Setting up this context will cause extra delay. Also, executing the redundant copy requires an idle CPU. We can pre-allocate idle CPUs solely for redundancy, or we can identify them at system runtime. In a server with light workload, it might be easy to identify the idle CPU. But it may be difficult to identify idle CPUs in highly consolidated servers. Because DualVisor can be easily managed by software, we can leave this problem to the automatic management framework for further optimization.

Temporal redundancy (Fig. 5 (b)) can be achieved by executing the redundant copy in the same physical CPU core right after the original execution. The results from two copies can be synchronized at the end of the execution of the redundant copy. Two copies will share the same CPU context. Note that our fault model is single bit flip error in CPU. A soft error may occur any one of the two copies. In this case, the error can be detected. If an error occurs in the non-replicated region, the error may not be detected.

It is important to understand the execution time of various steps in both spacial and temporal redundancy in detail to make the trade-off. Towards this goal, we implement both techniques and measure the latency of each step involved in both techniques. For temporal redundancy, we need to create memory space for replicated data. We create this space on the function stack (allocation on stack). After that, we copy the original data to the newly created space (memcpy), and then carry out the redundant execution. Then, we compare

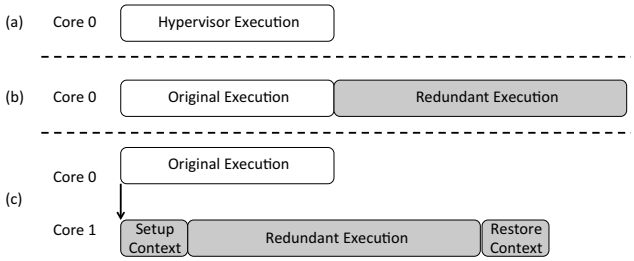


Fig. 5: Spatial and Temporal Redundancy. (a) Original execution without redundancy; (b) Temporal redunancy; (c) Spatial redundancy

TABLE I: Detailed Latency Measurement

Operation	**Temporal** (ns)	**Spatial** (ns)
Allocation on stack	28	28
Memcmp	1326	1326
Memcpy	31	31
Set context	N/A	750
Restore context	N/A	142
Send IPI	N/A	1900
Total	1385	4177

data for error checking (memcmp). For spacial redundancy, other than these operations, we need to activate the redundant execution in another CPU core. We implement this using inter-processor interrupt (IPI). In this newly activated core, we save the previous context and set up the new context. Then, the redundant execution starts. After the execution is done, we restore the previous context.

We measure the time of each operation on a Intel Xeon 5506 processor, except for the execution itself because its time may vary depending on VM exit reasons. The side-by-side comparison is shown in Table I. The results show that the spacial redundancy has significant higher overhead (about 2x higher). Sending IPI takes a relatively long time (almost half of the total time in spatial redundancy). The actual impact on performance overhead depends on the length of original VM exits. Based on the numbers in Table I, it is possible to use different techniques for different VM exits, depending on the execution time of original VM exits. In our implementation, we use temporal redundancy for all VM exits that we select to protect. Note that the numbers here cannot be directly transferred to the impact on application execution time, because the hypervisor is only activated periodically when applications are running.

D. Replicating Data

The third design parameter is how to replicate data for redundant executions. We need to create a shadow copy of original data, so that they can be used by the replicated hypervisor execution. There are several potential solutions for creating these shadow data: creating a contiguous memory space for entire hypervisor memory data, using a page table, or selectively replicating a critical portion of hypervisor data.

We can allocate a contiguous memory region and copy over all original data. We need to ensure that two regions have the exact same layout, so that the address of any data structure in the shadow regions will differ from the original data by a constant offset. The replicas can be referenced using this offset and the original data addresses. Of course this approach requires significant efforts to re-engineer the hypervisor software (e.g. through compiler techniques or by re-writing the hypervisor software).

Another potential method is replicating memory data by adding a page-table mechanism, which allows the hypervisor to manage its own data. We can add a very thin layer under the original hypervisor, and create a page table to manage original and replicated data. This method does not require a contiguous memory region, but still requires many efforts to modify the hypervisor software. Note that the nested virtualization technique [14] may be leveraged to implement this, but it requires careful modifications to make the extra layer extremely thin. Otherwise, the extra virtualization layer becomes a new single-point of failure.

Both methods are based on the notion of replicating all hypervisor memory data. However, data are not equally important. The data that are more frequently used by the hypervisor execution can be more critical than those are rarely used. For example, the guest VM CPU context can be a critical data structure because it is the input for most of hypervisor executions. In contrast, some local data that are only used when handling errors will unlikely be used in normal executions. Comparatively, the former can be more critical than the latter. We can selectively replicate these critical data structure that are frequently used in hypervisor executions. This method potentially reduces the re-engineering cost and the performance overhead, since it only replicates selected data. But we need to carefully select the data structures to provide sufficient protection.

The hypervisor data structures can be classified into three types: global data structures that are shared by all VM exits, VM related data structures that are only shared by VM exits from the same VM, and local temporary data that are private to each VM exit or function (temporary pointers to global data are not included). In general, the critical data structures that we replicated are local data and VM related data (e.g. guest VM CPU context, guest VCPU information). We do not replicate the global data structures as they are shared among all VMs, and can cause non-deterministic problems (explained in Section III-E). To replicate VM related data (e.g. guest VCPU), we allocate extra memory for these data structure. The memory allocation is usually done once when a VM is created (initial synchronization in Fig. 1). During the execution of VM exit, a redundant copy will be created by copying the original data to the pre-allocated memory region (runtime synchronization for creation in 1). An example is shown in Listing 1. For some local variables requiring small memory space, we allocate memory space on the function stack during the hypervisor execution. An example is shown in Listing 2.

Listing 1: Data Replication Example 1: VM Data

```
1  struct vcpu *alloc_vcpu (...){
2          /* create original VCPU data */
3          ...
4          v = alloc_vcpu_struct();
5          ...
6          /* Allocate on heap for replication */
7          v->shadow_for_regsiter =
8                  xmalloc(struct cpu_user_regs);
9          v->shadow_for_trap =
10                 xmalloc(struct hvm_trap);
11         v->shadow_for_vcpu = alloc_vcpu_struct();
12         ...
13 }
```

Listing 2: Data Replication Example 2: Local Data

```
1  /* Define original local data */
2  unsigned long cr; /* control register number */
3  unsigned long gp; /* general purpose register */
4  ...
5  /* Allocate replications on stack*/
6  unsigned long shadow_for_cr;
7  unsigned long shadow_for_gp;
8  ...
9  }
```

In hardware assisted virtualization, there is a special data structure, virtual machine control structure (VMCS). VMCS is a critical data structure containing information about the guest VM current VM exit, such as the VM exit reason and the VM exit qualification. The hypervisor uses special instructions (e.g. *vmread* and *vmwrite* in Intel VMX) to access VMCS. We use these instructions as a hint for replication. That is, we replicate data after reading from VMCS (*vmread*), and compare data before writing to VMCS (*vmwrite*). This can also be done automatically using compiler techniques. In our current implementation, we manually identify these data and create replications.

E. Replicating Executions

During VM exits, we replicate the hypervisor executions. Original and redundant executions access and update their own data. We compare the replicated data at the end of the VM exit. We classify executions into two types: with side effects or without side effects. Some executions may trigger interrupts in guest VMs or modifying the global data and VM related data. We consider these executions have side effects. We cannot replicate these operations because that will affect the correctness of the hypervisor or VM state. We only replicate the executions without side effects (e.g. only reading from the global data and VM related data or writing to local data). We compare the outputs of replicated executions before these side effects occur. We call such executions (executions without side effects) that we replicate as Sphere of Replication (SoR).

Fig. 6 illustrates two types of VM exits requiring different implementations of SoR. Some VM exit reasons have side effects in various places during their executions. Such a VM exit will be split into several regions without side effects, and

each of them is a SoR, as shown in Fig. 6 (a). For example, for the VM exit reason of pending virtual interrupt, injecting the virtual interrupt to guest VM will cause side effects. Therefore, the actual injection is only done once. We replicate and verify the operation of obtaining the virtual interrupt information (e.g. modifying local data that are read from VMCS). Some only have side effects at the end of the execution. For these VM exits, we can simply replicate the entire execution without affect the system correctness. For those cases, the SoR can be larger, sometimes are from the beginning to the end of the hypervisor execution. Figure 6 (b) illustrates this case. The VM exit reason CPUID falls into this category, and we will explain it in detail in IV.

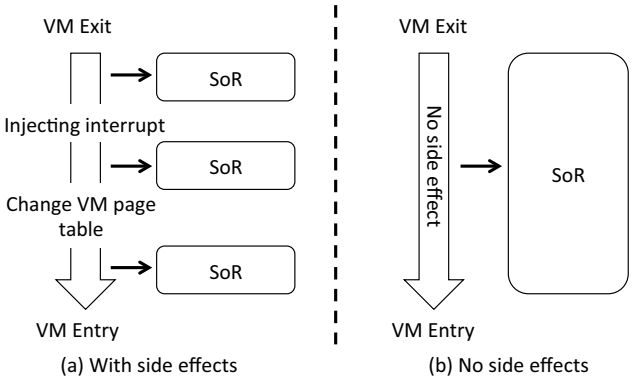


Fig. 6: Two examples of replicating executions

There might be a concern about the deterministic execution between two replicated SoRs. Hypervisor executions are inherently multi-threaded. One of the redundant SoR may be preempted by another VM exit, and these two hypervisor executions may modify a shared data structure. This may cause discrepancy between two copies of preempted the hypervisor execution. Fig. 7 illustrates this scenario. D is the original data, and D' is its replica. Both VM exits A and B will access D during their executions. A' and B' are their replicated executions respectively accessing D'. In one execution, VM exit A accesses D before VM exit B. In the replicated execution, VM exit A' accesses D after VM exit B'. Because the order of executions are different, D and D' will not be the same even if there is no error.

This non-deterministic problem may only occur when two different VM exits are interleaved and try to access shared data structures. Therefore, the solution can be either preventing interleaved hypervisor executions or preventing shared data structures between interleaved hypervisor executions.

One approach to preventing interleaved hypervisor executions is using a global lock for the hypervisor, we called it hypervisor lock, so that hypervisor executions will not be interleaved. However, this implementation will cause performance overhead because it essentially serializes the hypervisor execution. In fact, a similar hypervisor lock is implemented in [30] for each VM, so that only one VM exit per VM is allowed at a time. But it may still allow multiple VM exits

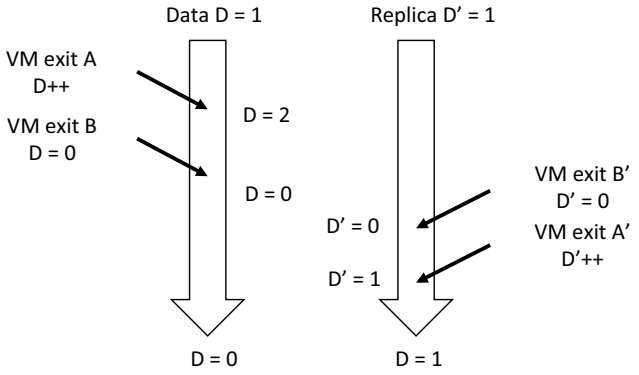


Fig. 7: Non-deterministic executions

from different VMs.

Another solution is allowing interleaved VM exits, as long as they are not accessing shared data structures. In fact, in our current implementation, this approach is more feasible and potentially incurs smaller overhead than the hypervisor lock. As we discussed in Section III-D, the hypervisor data structures can be classified into three types: global data structure, local temporary data, and VM related data structures. Local temporary data will not be shared by VM exits. Therefore, the non-deterministic problem will not occur in those data. Only global data and VM related data may cause non-deterministic problems. However, our design for replicating executions can prevent this problem.

We have two approaches for replicating executions (Fig. 6) for hypervisor executions with or without side effects. For those hypervisor operations without any side effects, such as CPUID, they will not commit any changes into global shared data structure before the results from two copies are compared (they only read from shared data structures). The deterministic execution can be achieved between these copies without requiring additional techniques. For those hypervisor operations with side effects, we only replicate the data and operations before these side effects occur. That is, we essentially replicating local variables in SoRs that are not shared among hypervisor executions. Therefore, the deterministic results can also be guaranteed. Therefore, our approach will not cause any non-deterministic problems in shared data.

F. Synchronization

The synchronization can be done in all hypervisor data structures (e.g. guest VM context, VMCS and global data). We can further optimize this process by comparing only the field that should be modified for the specific VM exit reasons. This selective comparison may speedup the synchronization. If we selectively compare a portion of data, it is possible that the errors cause incorrect states in the regions that are not compared. These errors can be detected when corrupted states are used for later hypervisor executions. If corrupted states are never used, they will not affect system correctness.

IV. Implementation

In the following, we explain the details of replicated data and executions in these selected five VM exits.

Non-maskable interrupt. Most of NMIs are used to signal abnormal events, such as errors and failures, which are not triggered in normal executions. Based on our profiling results, most of NMIs are because of page fault during normal execution. Therefore, our replication is only done for handling page fault. In the case of page fault, the hypervisor first needs to read the faulting address and error code from VMCS. Then, the hypervisor checks if this page fault is a real page fault that should injected into the guest VM. We replicate the page fault information (local data), such as exception type, faulting address and error code, after the hypervisor obtain them from VMCS, and replicate the operations on those information. However, because injecting the page fault into the guest VM will cause side effects, we do not replicate this.

External interrupt. The hypervisor reads the interrupt vector from VMCS and injects it into the guest VM CPU registers. We replicate the interrupt vector (local data) and its operations before interrupt injection. The hypervisor will also handle the interrupt and deliver an event through the event channel. Because this operation has side effect, we did not replicate it.

Pending virtual interrupt. The hypervisor checks if there is a pending virtual interrupt, and modify the corresponding guest VM states. We replicate the interrupt information (local data) and its operations before modifying the guest VM state.

CPUID. CPUID VM exit takes inputs from guest VM CPU registers, retrieves corresponding CPU information, and writes it back to guest VM CPU registers. Retrieving CPU information does not have side effects, and can be replicated as a whole. In this case, we replicate the guest VM CPU registers (VM data) and guest VM information (VM data) at the beginning, and then replicated this hypervisor execution. At the end of the VM exit, we compare the guest VM CPU registers to make sure two copies have the same value.

Control register access. Depending on the type of access, the hypervisor will either modify the control register (write access) or modify a general purpose register based on control register value (read access). We replicate the input values (local data) and related operations before modification.

V. Performance Overhead

Redundancy incurs runtime performance overhead, which affects all VMs. A very high overhead may prevent the practical use of DualVisor. Therefore, we conduct a study to evaluate the overhead based on temporal redundancy implementation. Note that the runtime cost of DualVisor can be further optimized within the automatic management framework described in Fig. 3. We evaluate overhead in three aspects:

- The overhead of redundant executions to the VM exit execution time
- The overhead of redundant executions to the application execution time.
- The increase of the overhead as the number of VMs increase

The experiment is done in a server with one Xeon E5506 CPU and 12GB memory. Each Hardware-assisted VM is assigned with 1 VCPU and 2GB memory.

Replication may have different impact on different VM exits due to the variation of data sizes and replicated executions. To understand the detailed impact of redundant execution on VM exit executions, we measure the execution time of the VM exit handler in the Xen hypervisor when DualVisor is enabled or disabled. We collect these results when a VM is being booted or running a application. As shown in Fig. 8, redundancy executions have various impact on the execution time of VM exits. Four VM exits show small overhead (less than 7%), and the virtual interrupt has almost no overhead. The virtual interrupt is relatively short with a few of operations, and we only replicate the data and executions related to the virtual interrupt numbers before injecting them to VMs. Therefore, the overhead is not noticeable. Comparatively, CPUID has the highest relative overhead (116%). This is because the SoR of CPUID is relatively large (the entire execution) and the original execution is relatively short. As a results, the overhead of replication is even higher than the original execution. Therefore, the overhead is higher than 100%. Note that these results are measured for VM exits rather than applications. The overhead on applications (shown in the next figure) is smaller, as the hypervisor executions are only activated periodically during VM executions.

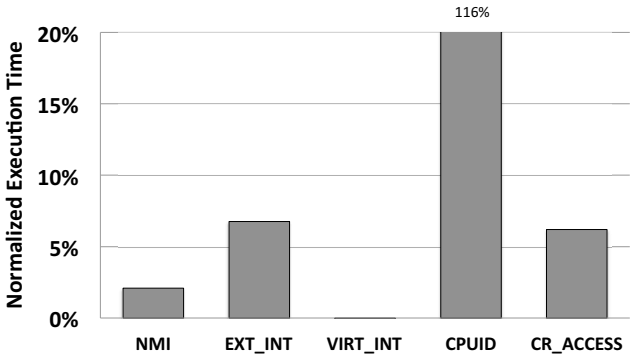


Fig. 8: Overhead to VM Exit Handler Execution Time

The other type of overhead is measured by the execution time of applications inside of VMs. The results are collected when one VM or two VMs (with the same application) are running. Experiments are repeated by 10 times for each application. The average overheads caused by redundancy are shown in Fig. 9. The numbers are normalized to the average execution times when the original Xen hypervisor is running.

All six benchmarks show very small overhead, less than 1% when one VM is running and less than 6% when two VMs are running. The performance overhead is slightly higher for I/O intensive applications, *postmark*. Overall, we consider this overhead is reasonable for redundant execution. The reason why we can achieve such low overhead is selective replication. We only replicate five VM exits (but covering 87% of the total number of VM exits), and only replicate critical data structures

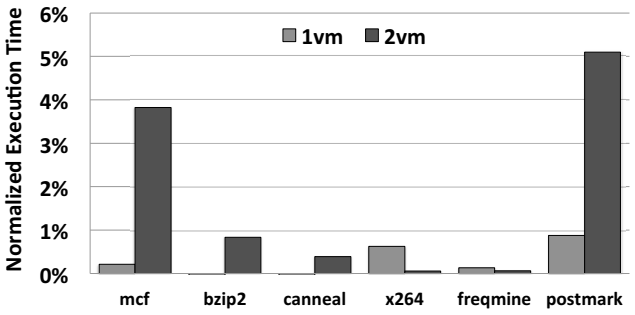


Fig. 9: Overhead to Application Execution Time

and their related executions. Note that we choose selective replication to trade off the cost and the protection level. It is possible to provide stronger protection by replicating more VM exits and more data, with a higher cost accordingly.

To better understand the scalability of our approach, we measure the performance overhead in terms of execution time when the number of VMs increases. Fig. 10 shows the results. All VMs are running *mcf* (CPU and memory intensive) at the same time. We do not use I/O intensive application here because running four I/O intensive VMs will cause significant interference. As we can see from the figure, the overhead increases to 3.8% when there are two VMs running. But it does not increase further when there are four VMs. This happens because DualVisor incurs relatively small overhead, compared to the performance interface from co-located VMs.

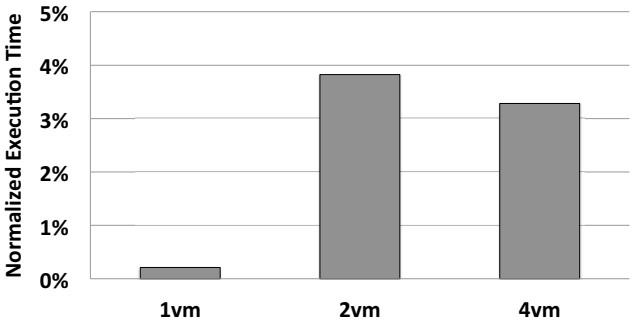


Fig. 10: Overhead as the number of VMs increases (mcf)

VI. Related Work

Redundancy can be done in hardware [26], [27] or software [28], [29] for applications [28], [29] or VMs [22], [31]. This work is different from previous research as we focus on the detailed analysis, design and implementation of software redundancy in the context of the hypervisor.

Hardware redundancy has been proposed for CPU errors in [26], [27], leveraging simultaneous multi-threading or multi-core processors. Hardware DMR can achieve nearly 100% coverage to CPU errors at the cost of area, performance and energy. Due to its high cost, hardware redundancy are not available in medium and low-end computer systems, which are the majority of cloud computing data centers. Hardware redundancy is unlikely incorporated into these commodity systems in the near future.

Software redundancy has been used to improve the reliability of applications. Redundancy can be achieved using compiler techniques [32] or operating system supports [28], [29]. Our method also uses software redundancy. Hypervisor runs at the lowest level at the software stack in the system. This unique role creates many difficulties for implementing software redundancy. For example, operating system supports (e.g. *ptrace and fork*) for software redundancy are not available for hypervisor. Compiler techniques may be possible for automatically creating redundant copies. In our current implementation, we manually modify the code. But some information (such as *vmwrite* and *vmread*) can be leveraged as a hint for automatically inserting software code. We plan to further investigate this approach in our future work.

Redundancy can also be applied at virtual machine level. Remus [22] leverages live migration techniques to replicate VM states to a remote physical host. If failures occur in any replication, the other one can continue correctly. Since Remus is implemented at the VM level, it does not require modifications to applications. But Remus has high performance overhead, so it is more suitable to selectively protect user applications. Our method aims at improving hypervisor reliability, which can be beneficial to all VMs. It helps improve the overall virtual system reliability. Therefore, it is orthogonal to the these VM-level techniques.

The hardware error problem has been studied in virtualized systems [25], [21], [10], [11]. In [25], a VM power model and a VM failure model are designed to manage the cost of power and failures when dynamic voltage and frequency scaling (DVFS) is utilized in virtualized systems. The models use DVFS states to estimate VM power and reliability. It can be leveraged to strengthen our automatic management framework (Fig. 3 to provide feedback information from hardware. In [21], a fault injection framework is designed to study error propagation behaviors. The results in that paper suggest soft errors affecting the hypervisor may cause various types of failures, suggesting new approaches are required for correct diagnosis and recovery. In [10], a soft error detection framework, Xentry, is designed and implemented. Xentry prevents the error propagation from the hypervisor to VMs by detecting them with very short latency. In [11], a recovery techniques is designed to recover the hypervisor failure, preserving running VM states. Our works is different from previous works by integrating both detection and recovery capabilities using redundant executions. DualVisor can detect silent data corruptions which are difficult to detect in current systems. DualVisor recovers errors while the hypervisor is running instead of rebooting the whole hypervisor, minimizing the impact to running VMs. Also, its protection strength can be adjusted (by adjusting the number of redundant copies) according to the system reliability requirements.

VII. Discussion and Conclusion

In this paper, we demonstrate the need for a hardware-error-resilient hypervisor. We describe the design of DualVisor, a fault-tolerance technique that is designed specifically for the hypervisor. We discuss various design parameters in detail, and implement DualVisor in the hypervisor software. Our experimental results show that the performance overhead of software redundancy is small for CPU and memory intensive applications. The discussion and results in this paper demonstrate the benefits and feasibility of an error resilient hypervisor. For our future work, we would like to extend our early effort in designing the automatic management framework [33], and integrate DualVisor into this framework. Our current implementation of redundancy is done with detailed inspection of the hypervisor code. We would also like to investigate intelligent and automated methods to deploy redundancy in the hypervisor.

Note that the framework that implements DualVisor is not replicated, and therefore is vulnerable to hardware errors. To protect the framework from fatal failures, we can leverage existing hardware supports, such as MCE. Errors may not cause fatal failures of the framework, but cause incorrect detection, e.g., identifying a correct execution as incorrect. We consider this is a benign case because an error does occur and is reported. We assume there is only one error in the system because two errors occur at the same time is very unlikely. Therefore, when an error affects the hypervisor execution, the framework will function correctly. In our current implementation, because we selectively replicate data and execution, the non-replicated portion of the hypervisor will still be vulnerable. As a part of our future work, we plan to investigate the vulnerability of non-replicated portion, and increase the sphere of replication where it is necessary.

Acknowledgment

This work is supported by National Science Foundation grant CNS-1350766.

References

[1] T. J. O'Gorman, J. M. Ross, A. H. Taber, J. F. Ziegler, H. P. Muhlfeld, C. J. Montrose, H. W. Curtis, and J. L. Walsh, "Field testing for cosmic ray soft errors in semiconductor memories," *IBM Journal of Research and Development*, vol. 40, pp. 41 –50, jan. 1996.

[2] V. Sridharan and D. Liberty, "A study of dram failures in the field," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 76:1–76:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389100

[3] ——, "A study of DRAM failures in the field," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC, 2012, pp. 76:1–76:11.

[4] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM Errors in the Wild: A Large-Scale Field Study," in *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, ser. SIGMETRICS, 2009.

[5] X. Li, M. C. Huang, K. Shen, and L. Chu, "A realistic evaluation of memory hardware errors and software system susceptibility," in *Proceedings of the USENIX conference on USENIX annual technical conference*, ser. USENIXATC, 2010.

[6] X. Li, K. Shen, and M. C. Huang, "A memory soft error measurement on production systems," in *In USENIX Annual Technical Conf*, 2007.

[7] A. A. Hwang, I. A. Stefanovici, and B. Schroeder, "Cosmic rays don't strike twice: Understanding the nature of dram errors and the implications for system design," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 111–122.

[8] M. Snir, R. Wisniewski, J. Abraham, S. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson *et al.*, "Addressing failures in exascale computing," Argonne National Laboratory (ANL), Tech. Rep., 2013.

[9] S. Michalak, K. Harris, N. Hengartner, B. Takala, and S. Wender, "Predicting the number of fatal soft errors in los alamos national laboratory's asc q supercomputer," *Device and Materials Reliability, IEEE Transactions on*, vol. 5, pp. 329–335, 2005.

[10] X. Xu, R. Chiang, and H. H. Huang, "Xentry: Hypervisor-level soft error detection," in *The 43rd International Conference on Parallel Processing (ICPP14)*, Minneapolis, MN, Sep. 2014.

[11] M. Le and Y. Tamir, "Rehype: enabling vm survival across hypervisor failures," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE, 2011, pp. 63–74.

[12] Intel, "VT (Virtualization Techology)," http://www.intel.com/technology/virtualization/.

[13] "AMD-V," http://sites.amd.com/us/business/it-solutions/virtualization/Pages/amd-v.aspx.

[14] "KVM," http://www.linux-kvm.org/.

[15] 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.*, vol. 37, pp. 164–177, 2003. [Online]. Available: http://portal.acm.org/citation.cfm?id=1165389.945462

[16] Standard Performance Evaluation Corporation, "SPEC Benchmarks," *http://www.spec.org*, 2006.

[17] C. Bienia, S. Kumar, J. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," *Princeton University Technical Report TR-811-08*, January 2008.

[18] J. Katcher, "Postmark: A new file system benchmark," 1997.

[19] A. Landau, M. Ben-Yehuda, and A. Gordon, "Splitx: split guest/hypervisor execution on multi-core," in *WIOV*. USENIX, 2011.

[20] Virtutech. (2006) Simics Full System Simulator. [Online]. Available: http://www.simics.net

[21] X. Xu and H. H. Huang, "Understanding reliability implication of hardware error in virtualization infrastructure," in *10th Workshop on Hot Topics in System Dependability (HotDep 14)*. Broomfield, CO: USENIX Association, Oct. 2014.

[22] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: high availability via asynchronous virtual machine replication," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'08, 2008, pp. 161–174.

[23] J. C. Smolens, B. T. Gold, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk, "Fingerprinting: bounding soft-error detection latency and bandwidth," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS, 2004.

[24] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers, "The case for lifetime reliability-aware microprocessors," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ser. ISCA, 2004.

[25] X. Xu, K. Teramoto, A. Morales, and H. H. Huang, "Dual: Reliability-aware power management in data centers," *Cluster Computing and the Grid, IEEE International Symposium on*, vol. 0, pp. 530–537, 2013.

[26] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ser. ISCA, 2000, pp. 25–36.

[27] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed design and evaluation of redundant multithreading alternatives," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ser. ISCA. Washington, DC, USA: IEEE Computer Society, 2002, pp. 99–110.

[28] Y. Zhang, S. Ghosh, J. Huang, J. W. Lee, S. A. Mahlke, and D. I. August, "Runtime asynchronous fault tolerance via speculation," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO, 2012, pp. 145–154.

[29] A. Shye, T. Moseley, V. J. Reddi, J. Blomstedt, and D. A. Connors, "Using process-level redundancy to exploit multiple cores for transient fault tolerance," in *Dependable Systems and Networks (DSN), 2007 37th Annual IEEE/IFIP International Conference on*, 2007, pp. 297–306.

[30] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen, "Execution replay of multiprocessor virtual machines," in *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM, 2008, pp. 121–130.

[31] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault tolerance," in *Proceedings of the fifteenth ACM symposium on Operating systems principles*, ser. SOSP '95, 1995.

[32] Y. Zhang, J. W. Lee, N. P. Johnson, and D. I. August, "DAFT: decoupled acyclic fault tolerance," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT. ACM, 2010, pp. 87–98.

[33] X. Xu and H. Huang, "Towards virtualized systems with hardware error tolerance (poster)," in *5th Asia-Pacific Workshop on Systems (APSys)*, 2014.