2014 43rd International Conference on Parallel Processing

Xentry: Hypervisor-Level Soft Error Detection

Xin Xu Ron C. Chiang H. Howie Huang
George Washington University

***Abstract*—Cloud data centers leverage virtualization to share commodity hardware resources, where virtual machines (VMs) achieve fault isolation by containing VM failures within the virtualization boundary. However, hypervisor failure induced by soft errors will most likely affect multiple, if not all, VMs on a single physical host. Existing fault detection techniques are not well equipped to handle such hypervisor failures. In this paper, we propose a new soft error detection framework, Xentry (a sentry on soft error for Xen), that focuses on limiting error propagation within and from the hypervisor. In particular, we have designed a VM transition detection technique to identify incorrect control flow before VM execution resumes, and a runtime detection technique to shorten detection latency. This framework requires no hardware modification and has been implemented in the Xen hypervisor. The experiment results show that Xentry incurs very small performance overhead and detects over 99% of the injected faults.**

***Keywords*—Virtualization, Error detection, Hypervisor**

I. Introduction

Cloud service providers, such as Amazon EC2 and Windows Azure, utilize virtualization technology to provide infrastructure-as-a-service (IaaS). Commodity machines in those data centers are susceptible to hardware errors including soft errors (or transient faults). Unfortunately, the virtualization layer especially the hypervisor is not protected against soft errors that are temporary hardware errors caused by manufacturing defects, particle strikes, etc. [1]. These errors undermine hypervisor reliability for three reasons.

First of all, while fault isolation is provided among VMs, soft errors that occur during hypervisor executions cannot be isolated. They may lead to system-level failures and affect all the VMs as a result of error propagation. Second, soft errors are not uncommon, as several recent studies have presented non-trivial soft error rates in large-scale computer systems [2–4]. The soft error rate per processor is expected to increase by 10 to 100 fold when semiconductor manufacturing technology advances from 45 to 11 nm [5]. Third, as the hypervisor in virtualized systems becomes frequently and heavily utilized, this allows the hypervisor to become increasingly vulnerable to soft errors. For example, when four VMs are running on a physical server, the hypervisor can be activated about 650,000 times per second. Recent works also suggest to use dedicated CPU cores for the hypervisor to improve the performance of I/O intensive workloads, and these I/O cores in most cases are fully utilized [6,7].

Many fault tolerance techniques have been proposed at the application level [8–11], the OS level [12], the VM level [13,14] and the hardware level [15,16]. Fig. 1 shows a typical virtualized system, where multiple components (in solid boxes) can be protected by the proposed techniques. At the hardware level, soft errors can be detected using dual modular redundancy (DMR) [15,16]. However, DMR is hard to find in commodity servers because it requires extra hardware modules and incurs performance and energy overheads due

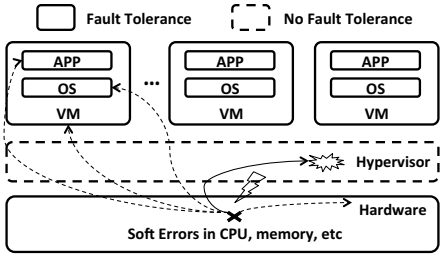


Fig. 1: Soft errors in the hardware may propagate to various levels. Fault tolerance techniques have been proposed for applications, OS, and hardware (solid boxes). The hypervisor is often not protected (dashed box). Soft errors may lead to hypervisor failures.

to synchronization and verification. At the application and OS level, current techniques are usually designed to detect soft errors when they occur in the context of applications or OSes. At the VM level, periodically checkpointing a VM is a common practice to achieve fault tolerance, but the overhead of checkpointing all the VMs is prohibitively high.

The hypervisor (shown as a dashed box in Fig. 1) is left unprotected. Some work has begun to study low-cost fault tolerance techniques to handle hypervisor failures. For example, ReHype [17] re-initiates the hypervisor upon failures while preserving VM states. However, without effective detection techniques, simply rebooting the hypervisor cannot recover from a large number of VM failures and data corruptions.

In this work, we propose that an effective error detection technique is the key to improve the reliability of the hypervisor. In particular, we aim to address three major challenges in this paper: 1) A high detection coverage is required to detect as many faults as possible. 2) Soft error propagation should be limited to minimize the impact of failures. Soft errors should be detected during hypervisor executions as early as possible, ideally before VMs resume. And 3) the performance overhead should be as low as possible. Since the hypervisor is activated frequently, the detection should complete quickly so that the application performance will not be affected.

To this end, we propose an error detection framework Xentry that is designed to enable the hypervisor to monitor possible occurrences of soft errors in runtime. Xentry utilizes two detection techniques: *VM transition detection* is designed to prevent soft error propagation across the virtualization boundary, and *runtime detection* asserts hardware and software checks during hypervisor executions to shorten the detection latency. Xentry is implemented as a light-weight software layer between the hypervisor and VMs, while relying on hardware supports to collect runtime data for error detection. The evaluation results show that our framework can achieve very high coverage (up to 99.4%) and incur very low performance overhead to the applications (2.5% on average). Our framework effectively limits soft error propagation by detecting 92.6% of soft error propagation that may have caused silent data corruptions. It can also detect errors with short detection latency (about 95% of detected faults are detected within less

0190-3918/14 $31.00 © 2014 IEEE
DOI 10.1109/ICPP.2014.43

than 700 instructions).

To the best of our knowledge, this is the first work that provides soft error detection in the hypervisor and runs as a thin software layer that is transparent to the applications. The key difference from previous works is that Xentry can detect soft errors before they propagate to guest VMs. This feature is very important for effective error detection and recovery. Also, as many prior works require hardware modifications [15,16], our software-only approach is a low-cost solution that can be easily utilized in current data centers. We are in the progress of preparing the source code for public release.

This paper is organized as follows. In Section II, we explain the basics of hypervisor operations and the reasons why the hypervisor should be protected from soft errors. Section III describes our detection framework and techniques. In Section IV, we discuss the major implementation issues. In Section V, we evaluate the proposed framework in terms of performance and effectiveness, and in section VI analyze undetected faults for future improvement. Section VII discusses related works and Section VIII concludes.

II. Background and Motivation

In this paper, our study focuses on soft errors in CPU while the hypervisor code is running (soft errors in the hypervisor context). In this section, we first discuss the impact of soft error propagation on system reliability. Then, we show that the hypervisor is vulnerable to soft errors.

A. The Impact of Soft Errors Propagation

Fig. 2 shows the typical execution flow of hypervisor and VM activities. CPU can either be in the VM context or the hypervisor context. A hypervisor execution can be activated to perform privileged operations such as I/O operations. After the execution is done, the VM context will be loaded back and the VM execution resumes. The VM transitions between VM executions and hypervisor executions can be done with hardware support (hardware-assisted virtualization, e.g. Intel VMX [18] and AMD SVM [19]) or with pure software support (e.g. Xen para-virtualization). In Intel VMX, the VM execution is called *guest mode* and the hypervisor execution is called *host mode*. The transition from guest mode to host mode is *VM exit*, and the transition from host mode to guest mode is *VM entry*. For convenience, we use these terminologies to describe operations in virtual environments.

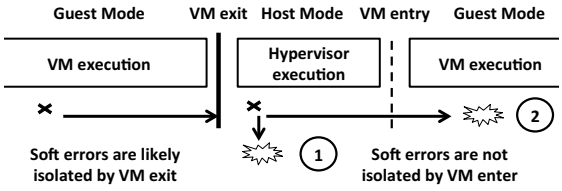


Fig. 2: Hypervisor and VM executions. Soft errors in guest mode can be isolated. Soft errors in host mode can cause system failures (Path 1). Soft errors in host mode can propagate across VM entry to VMs and applications (Path 2)

Soft errors in guest mode are likely to be isolated, but those in host mode may propagate through VM entry to VM executions. In this paper, we focus on soft errors in host mode. Hypervisor activities consist of many high-privilege and low-level operations such as context switching and interrupt handling. Soft errors may propagate through these operations, compromising the reliability of the whole system. We identified two typical soft error propagation behaviors: 1) propagating errors within host mode and 2) propagating errors across VM entries.

The first type of soft error propagates within host mode, and causes hypervisor failures before VM entry. As a result, all VMs are affected. Path 1 in Fig. 2 illustrates this case. One example is that an error occurs in the instruction pointer pointing to an invalid instruction. Although these soft errors cause hypervisor failures, they do not propagate to guest mode. If they can be detected and recovered in time, there is a good chance that hypervisor failures can be isolated and VM states can be preserved.

The second type of soft error propagates across VM entry to guest mode, and causes failures or data corruptions in VMs or applications. Path 2 in Fig. 2 illustrates this case. If this is a VM running user applications, failures are likely within this VM. If this is a control VM that the hypervisor utilizes to manage other VMs (e.g., Dom0 in the Xen hypervisor), the whole system will be affected. One example of this case is an error in the hypervisor execution emulating a privileged instruction for VMs, such as *cpuid*. When a VM is executing this instruction, a general protection exception is triggered and then trapped by the hypervisor. *cpuid* is then carried out in the hypervisor context. The results of this instruction (hold in eax, ebx, ecx, edx) will be written into the VM's VCPU structure. An error may change the instruction flow and may result in an incorrect output of *cpuid* instruction (e.g. eax). This does not cause any failures immediately in the hypervisor. But when the VM is using the incorrect (eax) value later on, a fatal failure will likely occur in the VM. Compared with the first type of behavior, the second type is more complicated since errors originate in the hypervisor but actual failures or data corruptions are in VMs. This can easily lead to incorrect diagnosis and unsuccessful recovery.

In this paper, we call the first type of soft error propagation *short latency errors*, and the second type of soft error propagation *long latency errors*. Note that, in this paper, we differentiate *long latency errors* and *short latency errors* according to the fact that if errors propagate across VM transitions rather than the duration of error propagation. An effective detection framework should detect both types of soft errors as early as possible to minimize the impact of error propagation.

B. High Frequency/Utilization of Hypervisor Activities

Hypervisor activities are very frequent, which make the hypervisor vulnerable to soft errors. To demonstrate this, we measure the number of hypervisor activities every second while applications are running in four VMs. Applications are chosen from SPEC2006 [20] and PARSEC [21] suite to represent CPU, memory, and I/O workloads. The details about the benchmark suites are explained in Section V. We conduct this experiment on both para-virtualization mode and hardware-assisted virtualization mode. A box plot is generated to show the statistics of the frequency of hypervisor activities

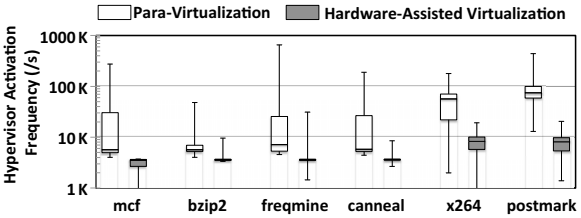


Fig. 3: The frequency of hypervisor activities. The central line on the box is the median; the box represents the data points between the 25th and 75th percentiles.; the lines extend to the maximum and minimum data points

in Fig. 3. As shown in the figure, the activation frequency is generally very high for both hardware-assisted virtualization and para-virtualization. For para-virtualization, the activation frequency is between 5,000/s and 100,000/s. The peak activation frequency is at about 650,000/s when *freqmine* is running. Para-virtualization has generally higher frequencies than hardware assisted virtualization. This is most likely because para-virtualization provides more interfaces to VMs through *hypercalls* that cause more hypervisor executions. But even in hardware-assisted virtualization mode, the activation frequency remains high. Most of them are between 2,000/s and 10,000/s.

On the other hand, the CPU utilization of the hypervisor is high in many cloud computing applications with I/O intensive workloads. Recent works have proposed using dedicated cores to run hypervisor operations for I/O intensive applications [6,7]. These dedicated cores offload hypervisor operations from guest VMs' CPUs. While this approach increases the performance of guest VMs, the dedicated cores are often fully utilized. As a result, the hypervisor running in these cores is particularly vulnerable to soft errors.

Consolidated server in data centers are expected to host tens or even hundreds of VMs. The chance of soft errors in CPUs affecting the hypervisor should not be ignored.

III. Xentry Framework

Xentry detects soft errors by identifying various error propagation behaviors as early as possible. Xentry specifically addresses three challenges mentioned in Section I. First, a high coverage is achieved by detecting various error propagation behaviors covering both *short* and *long latency errors*. Second, soft error propagation is limited by detecting *long latency errors* before VM executions resume. Third, a low overhead is achieved by leveraging hardware support for data collection.

Error behaviors are evident changes in control flow or data that are visible at the hypervisor software level. Specifically, we classify error behaviors into three types: 1) incorrect control flow (different from invalid control flow) , 2) fatal system corruptions and 3) data corruptions. *Long latency errors* usually cause incorrect control flow and data corruptions. *Short latency errors* usually cause fatal system corruptions and data corruptions. Xentry identifies all three behaviors to detect both types of soft errors.

Fig. 4 illustrates the structure of Xentry. There are two main components: *VM transition detection* and *runtime detection*. VM transition detection is enabled at every VM entry to identify incorrect control flow. It limits soft error propagation by detecting *long latency errors*. Runtime detection is always

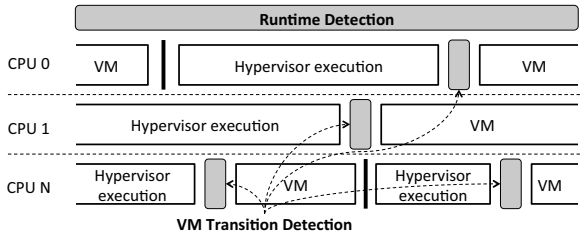


Fig. 4: Xentry fault detection framework

enabled while systems are running. It identifies fatal system corruptions and data corruptions to detect both *long latency errors* and *short latency errors*. Runtime detection improves the detection coverage and helps shorten the detection latency. The following subsections describe two techniques in detail.

A. Runtime Detection

Runtime detection monitors error behaviors along with hypervisor executions. It utilizes fatal hardware exceptions to monitor fatal system corruptions, and utilizes software assertions to monitor data corruptions.

Fatal system corruptions would halt the system due to corruptions in critical system states (such as invalid instruction pointers or memory addresses). For example, fatal system corruptions may be caused directly by fetching an invalid instruction. They may also be caused by complicated error propagation involving data corruptions and control flow changes. Fatal system corruptions are usually reported by hardware exceptions, which can be used for error detection.

Fatal hardware exceptions are utilized to detect fatal corruptions. Hardware exceptions (e.g. fatal page fault and invalid opcode) are generated by hardware platforms, making them good candidates as a low-cost detection approach. While failures may cause exceptions, exceptions do not necessarily indicate failures. Some exceptions are legal in correct executions, such as minor/major page faults and general protection exceptions. Therefore, hardware exceptions should be parsed first to filter out non-fatal ones. Another issue is that hardware exceptions cannot prevent soft error from propagating. They may occur before or after soft errors propagate to VM executions. The path 2 in Fig. 2 illustrates this case. It has been shown that hardware exceptions are strong indicators of hardware errors in applications and OS [22,23]. However, they usually do not emphasize on the strict detection latency requirement and limiting the soft errors propagation. We only use hardware exceptions to shorten the detection latency, and use VM transition detection for limiting soft error propagation.

Data corruptions. Soft errors may corrupt hypervisor variables. They are very common during error propagation. For instance, values in the registers may be changed, the destinations of memory accesses may be altered, or the instructions that operate on variables may be modified. Detecting data corruptions can help to improve the detection coverage when soft errors are not manifested in incorrect control flow.

Different from computation intensive applications, hypervisor activities involve less computations but more memory accesses, branches, etc. These operations are more difficult to verify than computations. Therefore, checking the computation

to detect errors (e.g. [24]) may not be effective in the hypervisor. Further, there are many variables in the hypervisor, and verifying all of them is too expensive. This has to be done by strategically selecting critical variables that reflect the correct execution of the hypervisor.

Software assertions are predicates that are manually inserted in the hypervisor code to detect data corruptions. To resolve two issues discussed above, software assertions must be inserted strategically with the consideration of the context.

Specifically, we use two types of assertions in hypervisors. The first type assertion applies on values with clearly defined boundaries. For example, Xen hypervisor may need to obtain the value of asynchronous interrupt or exception number before return to the guest VCPU context. This value has a predefined boundary (the number of available exceptions and interrupts). Checking this number may identify the errors in the code of obtaining this value. The code is shown in Listing 1.

Listing 1: Assertion Example 1: Valid Range

```
1 //clean up pending exceptions, and put them to VCPUs
2 int trap;
3 ....
4 for (trap = FIRST; trap < LAST; ++ trap) {
5         //obtain trap number
6 }
7 ASSERT ( trap <= LAST )
8 //put the trap number to VCPU
```

The second type of assertion applies on conditions that are critical to the correct executions. For examples, when the hypervisor changes a physical CPU to idle mode, its current virtual CPU (VCPU) should have already been in the idle mode (if without software bugs). Verifying the VCPU mode can detect errors in previous operations in this VCPU. The code is shown in Listing 2.

Selectively checking these data verifies both the checked variables and previous operations related to them. Therefore, the actual coverage of software assertions is broader than just the checked variables. Note that software assertions are designed to improve the detection coverage and to reduce the detection latency, rather than providing 100% coverage. Currently we leverage the assertions in the hypervisor code that are used for debugging and not used in normal execution. Error-free executions should not trigger any of these assertions, so they can be used as signs of soft errors. We plan to investigate more formal methods to identify critical values and integrate more assertions in the hypervisor code in future work.

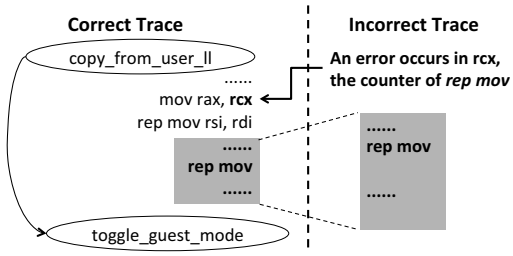
Listing 2: Assertion Example 2: Critical Conditions

```
1 void put_cpu_idle_loop(){
2         struct vcpu *v = current_vcpu;
3         ...
4         //verify VCPU is idle before idle its physical cpu
5         ASSERT(is_idle_vcpu(v));
6         //put physical cpu to idle
7         ...
8 }
```

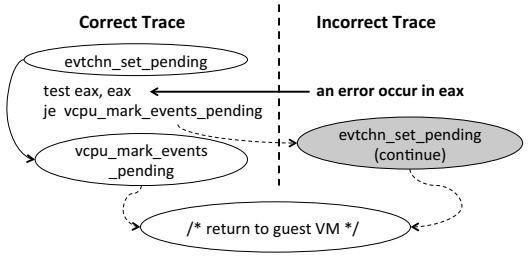
B. VM Transition Detection

Incorrect control flow. Before *long latency errors* propagate to VM executions, they may have already altered the original control flow to a valid but incorrect one. That is, the branch outcome is one of the legitimate instruction, but not the correct one depending on the branch condition. For example, for a *if* branch whose condition is true, the next instruction in correct control flow should be the one after *if* statement, but errors change it to the one after *else* statement. In either case, executed instructions are still valid. This is different from invalid control flow in which branch outcome is not a legitimate instruction (e.g. instructions that are not in the blocks after *if* or *else*).

Incorrect control flow will result in a set of detectable patterns, such as different memory operations are conducted, different numbers of instructions are executed, etc. Fig. 5 shows two examples of such incorrect control flow. In Fig. 5 (a), an error adds extra dynamic instructions into the correct executions. The fault in the condition variable of a loop, *rcx*, adds extra instructions to the original instruction trace. In Fig. 5 (b), an error changes the original branch target to a valid but incorrect one, resulting in an incorrect control flow that is usually not exercised in fault free executions. For both cases, all instructions including branch targets are valid. Therefore, only verifying the validness of control flows as in prior works [25,26] cannot identify such errors.



(a) Extra Code



(b) Incorrect Branch Target

Fig. 5: Examples of incorrect control flows

Machine learning based detection. To detect incorrect control flow, we develop VM transition detection based on machine learning approaches. It captures dynamic characteristics of hypervisor executions using a number of features, and automatically identifies incorrect control flow.

The key to detect incorrect control flow is identifying dynamic patterns of hypervisor executions. That is, detection techniques must be able to infer the *overall correctness* of a hypervisor execution given its context (e.g., the reasons of hypervisor executions), rather than just to check the *validness* of branch targets or other instructions.

The differences between incorrect and original control flows are subtle. These differences should be identified by an

automatic method rather than manual examination. Machine learning algorithms are proven to be effective to identify hidden patterns in a large amount of data. Therefore, we design VM transition detection based on machine learning algorithms.

Machine learning methods are usually considered as heavy weight approaches involving extensive computations. Moreover, the unique role of the hypervisor causes additional constraints to design machine learning based detection techniques. Using them in the hypervisor raises concerns about overhead and effectiveness. In the following, we discuss the design of VM transition detection with emphasis on the solutions of resolving these issues. The design of VM transition detection based on machine learning algorithms mainly consists of three aspects: 1) choosing the features that characterize dynamic execution behaviors; 2) selecting a machine learning method to classify the incorrect and correct behaviors; and 3) constructing the model for detection.

Selecting Features. Features are the information we collect to characterize the dynamic patterns of incorrect control flow. There are two basic principles when we select features: 1) The selected features must be able to characterize the dynamic patterns of control flow that cannot be revealed by static analysis (what to collect). 2) the features must be easy to collect with low overhead (how to collect).

Hypervisor executions are activated to serve requests from VMs and hardware with carefully defined VM exit reasons. These VM exit reasons can be considered as one factor characterizing hypervisor executions. Hypervisor executions contain many low level operations involving branch instructions, memory accesses, etc. The statistics of these operations may be used to characterize hypervisor executions as well.

The data described above can be easily collected by offline analysis. But for error detection, we must be able to collect them while the hypervisor is running with a low run-time overhead. Hardware performance counters as built-in components in most current systems can help to gather runtime statistics [18,19]. Using performance counters does not require significant modifications to the hypervisor, and can help to achieve low performance overhead. Therefore, we mainly leverage performance counters to collect most of features. Most systems have at least four performance counters that can collect runtime data. Combining VM exit reasons, we have five features to characterize execution patterns.

We summarize all selected features in Table I. As we discussed, VM exit reason is the most relevant feature. In full virtualization, this information can be obtained from the virtual machine control structure (VMCS). In para-virtualization, this information can be represented by function handler, and obtained using software implementation. The details of obtaining this feature is discussed in Section IV. We use performance counters to collect other four features: 1) the number of read memory accesses; 2) the number of write memory accesses; 3) the number of retired instructions; and 4) the number of branch instructions. These features characterize dynamic patterns in memory accesses, the lengths of hypervisor executions and control flow. These four events are basic performance monitoring events that are available in most x86 processors.

Note that these selected features do not explicitly represent control flow, but they implicitly capture the patterns of control flow from instruction patterns and memory access patterns. Moreover, they can capture more dynamic characteristics that control flows cannot do.

TABLE I: Selected features for VM transition detection

Features	H/W & S/W Support	Synonyms
VM exit reason	Xentry	VMER
# of committed instructions	INST_RETIRED	RT
# of branch instructions	BR_INST_RETIRED	BR
# of read memory access	MEM_INST_RETIRED.LOADS	RM
# of write memory access	MEM_INST_RETIRED.STORES	WM

Selecting machine learning methods. VM transition detection requires a light-weight algorithm to maintain low performance overhead. Many computation intensive machine learning methods are not suitable because of this very reason, e.g., the support vector machines (SVM). They require floating point and matrix computation which are too expensive for the hypervisor. The selected machine learning method should be able to generate simple (yet effective) models that can be implemented at a low cost.

On the other hand, the selected method should be able to achieve high accuracy without assuming distribution models. It is difficult to accurately predict soft error distributions. This is different from works that do not specifically address soft-error-induced failures. For example, [27] assumes certain probability distribution of data so that the generative models can be used. However, without the assumption of distribution, the generative models will suffer from low accuracy. In addition, the generative models involve a number of floating point and matrix computations, so they are too expensive for low overhead implementation. Therefore, the probability-based classifiers are not appropriate in this study.

Considering both requirements listed above, we use Decision Tree as our machine learning method. Decision Tree uses a tree-like model, consisting of a set of rules, to classify input data. The decision making process is a set of simple integer comparisons and therefore can be easily implemented with low overhead. Decision Tree also does not require any assumptions on the error probability model.

Constructing decision trees. Decision tree partitions input data (e.g. features collected from performance counters in our case) based on a set of rules (a set of branch conditions in our case) into classes (correct or incorrect in our case). Decision tree represents these rules using a tree-like structure where the leave nodes are classification results and edges represent rules of classifying features. In this section, we discuss in detail how to construct a decision tree.

A set of training input data are used to construct a decision tree, and their classification results (classes) are also given. The goal of construction process is identifying a set of rules (a tree) that can correctly classify all input data into their given classes. It may be difficult to find a tree that can achieve 100% correctness, so the process may stop after specified conditions (e.g. accuracy) are achieved.

A naive way to construct a tree is to recursively go through each feature and randomly partition data into smaller sets until specified conditions (e.g. accuracy) are satisfied. For example, one can partition a data set by specifying a rule, $10 < memorywriteaccess < 30$, into two groups (true or

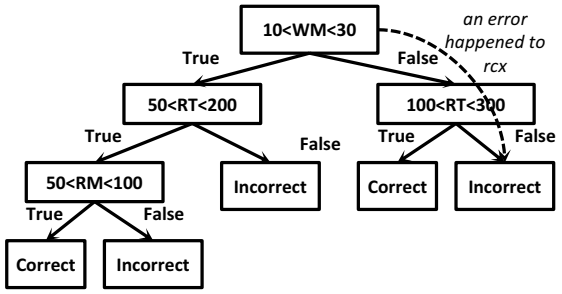


Fig. 6: A sample decision tree

false) as illustrated in Fig. 6. This splitting can be recursively conducted till the specified condition is satisfied (e.g. 99% of data are correctly classified). A tree can be built by combining all rules that are used to split data sets. However, random partition may lead to skewed trees and over-fitting issues. The worst case is each small set only contains one data point, resulting in an overly large tree.

Alternatively, the random partition process can be optimized using entropy in information theory. Intuitively, a set of data with similar features have a low entropy. The splitting procedure during tree construction aims to maximize the expected deduction D in entropy, which is the information volume per node. Assuming a binary decision tree, the function D is defined as $D(T, T_L, T_R) = Entropy(T) - (P_L \cdot Entropy(T_L) + P_R \cdot Entropy(T_R))$, where T is the original dataset, and T_L and T_R are datasets split into left and right tree respectively. P_L and P_R means the ratio of data points in T go to T_L and T_R respectively.

We use an example to illustrate our rule constructing procedure. We first collect statistics of incorrect and correct hypervisor executions. Each sample contains five features and one label (correct or incorrect). These data are used as a training data set to build the decision tree.

We iterate through each feature to select a cut point to split the dataset. Suppose a dataset T have 15 data points, 10 of them are correct and 5 of them are incorrect. The entropy of T relative to this classification is $Entropy(T) = -(10/15)\log_2(10/15) - (5/15)\log_2(5/15) = 0.276$. We want to choose a cut point on RT such that D is maximized. Assume that there are two choices, cutting at RT=100 and 200. When RT=100, T_L has 7 data points, 5 of them are correct and 2 of them are incorrect; T_R has 8 data points, 5 of them are correct and 3 of them are incorrect. When RT=200, T_L has 10 data points, all of them are correct; T_R has 5 data points, all of them are incorrect. The information gain D are 0.004 and 0.276 when cutting at 100 and 200, respectively. Thus, RT=200 will be selected as the cutting point to separate the dataset. The process will be repeated with different statistics until reaching the desired quality or all cases are tested. Fig. 6 shows a resulting tree using the aforementioned learning process. The tree can be summarized in a set of rules and can be utilized to identify incorrect control flows.

When building a decision tree, investigating all features may be time consuming. In addition, optimizing the selection at each single node does not guarantee an optimized global result. Therefore, we use the *random tree* algorithm, an alternative decision tree building process with extra randomization factors [28], to generate rules. More specifically, when the random tree method deciding a split, it randomly choses and considers $\lfloor \log_2(\text{number of features}) \rfloor + 1$ features at each node, which is three in our case.

To construct the VM transition fault detector, we need to collect training data set consisting of both correct execution data and incorrect execution data. We use a full system simulator to construct our model that will be explained later. To collect the performance counter statistics, we analyze the traces of fault injection runs that are generated by the simulator. We use the five features to characterize hypervisor execution behaviors, and both tree algorithms to generate detection rules. We conduct about 23,400 fault injections and fault-free runs to collect training samples. Each sample contains a label (correct or incorrect) and the statistics of five features. In total, the training data set contains 12,024 samples (10,280 samples are labeled as correct, and 1,744 are labeled as incorrect). These data are used to construct the classification tree model. We then conduct another set of fault injections (about 17,700) and fault-free runs to obtain the testing data set. In total, the testing data set contains 6,596 samples (5,295 samples are correct and 1,301 samples are incorrect). We utilize the implementation of machine learning algorithms in WEKA [28] to construct models. The results show that the random tree algorithm achieves slightly high accuracy (98.6%) than decision tree (96.1%). Random tree induces some randomness when partitioning the features. The slightly higher accuracy may be resulted from the better partitioning caused by the induced randomness. These results show that the proposed classification method with selected features are effective as an error detection technique. Due to the space limit, we omit the evaluation results and discussions on various features, tree depth, and training set size.

IV. Implementation

We implement Xentry in Xen 4.1.2. It contains about 2,000 lines of source code. Its static code size is much smaller than nested virtualization (5,500 lines of source code in [29]). Because VM transition detection is only enabled at the beginning and the end of hypervisor executions, its runtime overhead is also smaller than nested virtualization. The runtime detection of Xentry can be implemented by inserting assertions in software. The major implementation issues come from VM transition detection. The hypervisor runs at the lowest level at the software stack in virtual environments. It is difficult to employ existing software tools (such as redundant processes using *fork()* in OS Kernel [10,11]) to implement detection techniques.

Xentry is implemented by only modifying the hypervisor software so that it can be easily deployed in various systems. Xentry functions as an interface between the hypervisor and other domains (guest VMs and hardware). It intercepts all VM exits to prepare for data collection by instructing performance counters, and then allows original hypervisor execution to continue. It enables VM transition detection at every VM entry. Conceptually, this implementation is similar to *shim*, which is a method to solve software compatibility issues [30]. To implement the framework, three major practical issues should be addressed: 1) intercepting hypervisor executions; 2) accessing performance counters on each hypervisor execution; 3) enabling VM transition detection.

Intercepting hypervisor executions. Xentry needs to intercept all hypervisor executions to obtain VM exit reasons and to instruct performance counters. VM exit reasons falls into five categories: 1) Common interrupts that are generated by hardware devices such as disk and network devices. The interface of handling these interrupts in Xen is *do_irq()*; 2) Interrupts generated by Advanced Programmable Interrupt Controller (APIC) such as performance counter interrupts and inter-processor interrupts. There are ten interrupt handlers in this category; 3) Software interrupt and tasklet: these are handled by two Xen functions *do_softirq()* and *do_tasklet()*. 4) Exceptions: 19 exceptions are handled by exception handlers respectively; 5) Hypercalls: there are 38 hypercalls in current Xen 4.1.2. The hypercall is triggered by calling the hypervisor page where the hypercall handlers are registered. We replace the hypercall page entries with Xentry. There are small pieces of codes that are not covered by Xentry, including the code of Xentry itself and VM transition handling such as stack operations. In this case, we rely on the hardware exceptions and the software assertions.

Accessing performance counters. Performance counters are initialized on all processors when the system is booted. Data collection is only enabled at VM exit. On each VM exit, performance counters start to collect data right before the original entry function is called. At each VM entry, performance counters are disabled and data are collected by Xentry. Logical cores do not share performance counters, so data are accurate even if simultaneous multi-threading is enabled [18].

Enabling VM transition detection. VM transition detection is applied after original hypervisor executions relinquish control to Xentry. The rules generated by Random Tree are essentially a series of branches with conditions, which can be easily implemented.

After a fault is detected, proper recovery procedures should be enabled. It is possible to re-initialize the hypervisor to recover from soft errors [17]. Also, it may be possible to develop new techniques with even lower overhead (e.g., stop and re-initiate hypervisor executions with errors). In this paper, we mainly focus on the implementation of detection methods, leaving integrating the recovery techniques as our future works.

V. Evaluation

A. Experiment Infrastructure

To evaluate performance overhead, we use a single-socket server equipped with a 4-core Intel Xeon E5506 processors (8 logical cores), 12GB memory and 1 TB SATA disk. Four guest VMs are running the same benchmarks. Each VM is assigned with 1 VCPU and 2GB memory.

To evaluate the detection effectiveness, we conduct fault injections based on a full system simulator. Fault injection is a common method for evaluating fault tolerance techniques. In this work, we use a full system simulators, Simics [31], to conduct fault injections. Simics can accurately simulate system behaviors. It have been widely used in academia and industry for fault injection, software debugging, etc. Compared with real systems, simulation provides better capabilities in terms of controlling and monitoring fault injection processes.

We develop our fault injection framework as the modules that run in Simics. The simulated system is configured with a 4-core 64-bit processor (each core is comparable to a Intel Pentium-4 64-bit processor), 2GB memory, 60GB disk, Xen 4.1.2 and Debian 6 with Linux kernel 2.6.32. We configure one Dom0 with one VCPU and two para-virtualized DomUs each of which is assigned with one VCPU, 512MB memory and a 10GB virtual disk.

We select a wide range of benchmarks from PARSEC [21], SPEC2006 [20] and Postmark [32] to exercise I/O (e.g., postmark, freqmine from PARSEC and x264 from PARSEC), CPU (e.g., canneal from PARSEC, bzip2 from SPEC2006), and memory (e.g., mcf from SPEC2006). The reason that we select these benchmarks is to exercise different functions of the hypervisor, because the hypervisor is the software under test rather than the benchmarks. When conducting fault injections, the same benchmarks are running in two DomUs in parallel.

B. Fault Model

In this paper, we focus on soft errors in CPU because combinational logic circuits in CPU are usually not protected by ECC or parity checking. Note that even if the circuits are protected by ECC, uncorrected errors may still occur when the number of errors are beyond the ECC capabilities. We currently use the single bit-flip fault model in the architectural register state, including general purpose registers, instruction and stack pointers and flags. We adopt the common practice that assumes one single-bit flip soft error may occur at a time. Note that since soft errors are random events, it is very unlikely that two single-bit flip errors occur in CPU concurrently.

Soft errors may occur before reading or writing a register. Soft errors will not be activated if they occur before new-values are written (non-activated errors). Non-activated errors do not affect the system correctness even without any fault tolerance techniques, and the execution behaviors are as same as correct executions. Only soft errors occurring before reading registers can be activated (activated errors).

We randomly select the regions when applications are running as injection points. On each fault injection run, only one fault is injected. After a fault is injected, we allow the simulation to continue to observe if it can be detected.

C. Performance Overhead

Faults are rare events. Systems are running in fault-free mode for most of the time. Therefore, the low performance overhead in fault-free mode is very important. We evaluate the performance overhead of Xentry in the fault free mode. Each benchmark is executed for 10 times. After each run, the average running times of four VMs are collected.

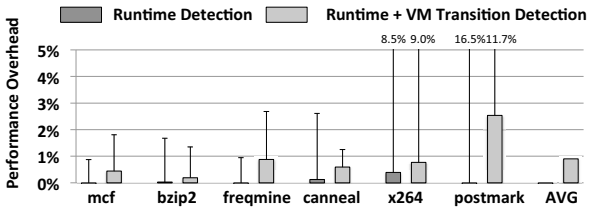


Fig. 7: Normalized performance overhead of Xentry

The average and maximum performance overheads are shown in Fig. 7. The shaded box is the overhead of solely

using runtime detection, and the empty box is the overhead of using both runtime detection and VM transition detection. The numbers are normalized to the average run time with an unmodified Xen 4.1.2 hypervisor. It is not surprising to see runtime detection only incur very small overhead, since only selected variables are verified. The overhead of runtime detection and VM transition overhead is also low. Four benchmarks, *mcf, bzip2, freqmine, and canneal* all have performance overheads lower than than 1%. The average overhead can be as low as 0.19% for *bzip2*. The *postmark* shows highest performance overhead (11.7% maximum). But the average overhead is 2.5%. Overall, Xentry can achieve low overhead mainly for two reasons: 1) we leverage existing hardware supports to offload heavy tasks such as data collection (performance counters) and runtime error monitoring (hardware exceptions); 2) VM transition detection is designed with low complexity so that the classification results can be quickly obtained by traversing limited branches.

D. Overall Detection Coverage

We evaluate the overall detection effectiveness of Xentry using fault injection experiments. We conduct 30,000 fault injections, and about 17,700 injected errors cause failures or data corruptions. We summarize the results of these errors by the detection techniques.

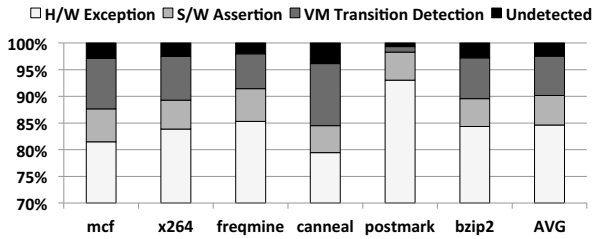


Fig. 8: Overall detection results

Fig. 8 shows the overall detection results. To show the result distribution more clearly, the y-axis starts from 70%. As the figure shows, the overall detection coverage is up to 99.4%, and the average is 97.6%. Most of errors (85.1%) are detected by the hardware exceptions because fatal system corruptions are common in hypervisor low-level operations. About 5.2% of injected faults on average are detected by software assertions. 6.9% of injected faults are detected by the VM transition detection, and these faults are all *long latency errors* that otherwise propagate to VM executions without our framework. The results suggest that the proposed detection techniques are effective in detecting various error behaviors, and the overall coverage is high.

E. Detecting Long Latency Errors

Long latency errors propagate across VM entry to guest mode. Based on our fault injection results, *long latency errors* may cause four types of consequences: 1) one VM failure: faults propagate to a VM, and hang or crash this VM; 2) all VM failure: faults propagate to the control VM or next hypervisor executions, and cause system failures affecting all VMs; 3) APP crash: faults propagate to the application running in the VM, and cause applications to exit abnormally such as segmentation faults; 4) APP SDC: faults propagate to the application running in the VM, and the application exits normally. But the result produced by the application is different from the one produced by the correct execution. Since there is no visible failures involved in this case, it is extremely harmful for systems. As we explained previously, existing methods have difficulties to detect, diagnose, or recover from such faults. Our framework can detect these faults.

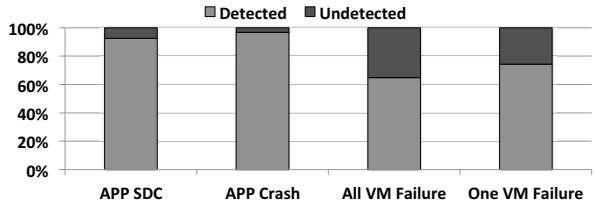


Fig. 9: Detection coverage of long latency errors

Fig. 9 shows the percentage of *long latency errors* that can be detected by our framework. The results are grouped according to the consequences if these errors were not detected. VM transition detection can successfully detect 92.6% of APP SDC cases and 96.8% of APP crash cases. We discuss the undetected cases in next section. Note that all cases in Fig. 9 are very difficult to detect with existing approaches. On the other hand, our framework can detect them with high coverage and low overhead, especially for SDC cases.

F. Detection Latency

The detection latency is measured by the number of instructions between error activation and detection. We utilize three techniques to detect error behaviors. We collect the detection latencies for all detected cases and group them according to detection techniques.

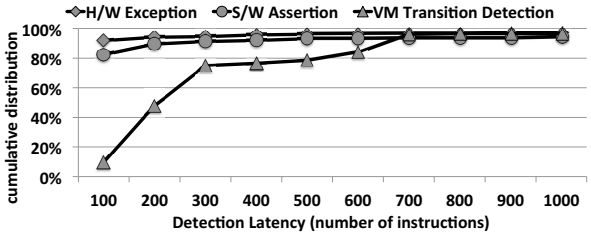


Fig. 10: CDF of detection latency

Fig. 10 shows the cumulative distribution of the detection latency. While most of cases (95%) detected by VM transition detection have less than 700 instructions, hardware exceptions and software assertions have generally shorter latencies. These results suggest that the later two techniques can help to reduce the detection latencies. Note that detection latencies shown here are varied, but all these faults are detected before starting VM executions.

Above results show that Xentry is an effective and low-cost detection framework. Xentry is able to achieve the three goals described in the beginning of Section III. Moreover, application SDCs are usually difficult to detect, but Xentry can effectively detect 92.6% of them.

VI. Discussion

Undetected Faults. There are a small percentage of undetected faults and the distributions are shown in Table II. We

analyze the undetected faults to identify the reasons. Apart from the errors that are mis-classified by the VM transition detection (10%), we find that the majority of undetected faults do not change the original control path. Rather, the errors only change data values. We find that two types of operations are more evident than others. The first case is stack values (20%). Errors corrupt the stack, and therefore incorrect values are pushed to or restored from the stack. This is difficult to detect since errors are activated after VM entry. Some of such errors may be captured by inserting more software assertions, but not all of them. These faults can be detected with more expensive approaches such as selective redundancy at software level or compiler level. Specifically, the values can be duplicated when they are pushed on to the stack, and verified when they are popped from the stack. The duplication can be done by adding instructions to the hypervisor directly in software or in compiler. The second case is timing values (53%). When guest VMs need to obtain system times, the hypervisor sends time values to the requesting domains. The time related values cannot be verified by directly duplicating instructions, since replicated *rdtsc* instructions may not generate exact same values. However, duplication may still help to check the correctness of time-related values. For example, two adjacent *rdtsc* may have a small variation in their output values. Checking this variation may help detect errors.

TABLE II: Undetected faults

Mis-Classify	Stack Values	Time Values	Other Values
10%	20%	53%	17%

False Positive. Due to the statistical nature of machine learning algorithms, false positive cases usually cannot be avoided. In our system, this may cause unnecessary recovery. To minimize the overhead, Xentry need to utilize light-weight software recovery solutions. Hypervisor executions are performed with carefully defined reasons and data structures (e.g. VCPU and domain data structures). It may be possible to recover hypervisor executions by preserving and restoring those data structures and activation reasons upon positive detection. In fact, some hypercall failures can be recovered by re-execution [17], which can be leveraged to further reduce the overhead. We assume that the recovery techniques will preserve the critical hypervisor data (e.g. VCPU and domain information) and the VM exit reason by making a redundant copy at every VM exit. If there is a positive detection (correct or false), these critical data and the VM exit reason will be restored and the hypervisor execution is re-initiated.

Since this paper focuses on the error detection techniques, we leave the implementation of the recovery technique as future work. In this work, we conduct the experiments to estimate the overhead of such technique, and then use these data to estimate the performance overhead of false positive cases. We measure that copying critical data structures will take about 1,900ns in a Xeon E5506 2.13GHz CPU. Recovery includes restoring the critical hypervisor data and re-executing the hypervisor execution, essentially doubling the original execution time. We collect a trace of hypervisor executions using the physical system setup described in Section V. We use the false positive rate (0.7%) obtained in Section III to randomly select hypervisor executions as false positive cases. This is repeated by 100 times for each application. Then, we estimate the overhead of fault-free executions (with false positive cases) with respect to Xen executions. We also measure the CPU utilization of the Xen when VMs are running using OProfile [33], and then calculate the overhead of recovery with respect to applications.

As Fig. 11 shows, the overheads for each application are very small (2.7% on average) and the difference between the maximum and minimum overheads are less than 0.03%. *Mcf and bzip2* have even smaller overheads, and both are about 1.6%. *Postmark* has the highest overhead 6.3%. This experiment shows that false positive cases can be handled with reasonable overhead if Xentry were combined with a light-weighted recovery approach.

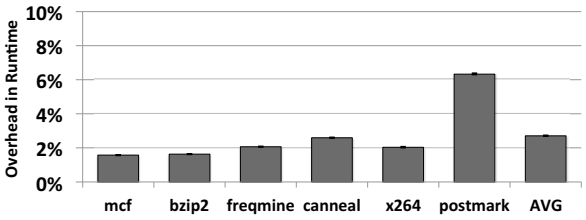


Fig. 11: Recovery overhead with False Positive Cases

VII. Related Works

Behavior Based Detection: Abnormal behaviors have been used to detect errors in previous works [22,25,26,34,35]. For example, SWAT [22] uses several symptoms such as high OS activities and fatal traps to detect errors in the application context. A hardware checkpointing mechanism is assumed for error diagnosis and recovery. Control flow violation can also be considered as one type of error behavior that is used for error detection [25,26]. Another group of works utilize classification methods to automatically identify faults [27,36]. An approach with two stages of classification is utilized to identify faults in high performance computing applications [27]. A Markov model is utilized in [36] to identify the abnormal behaviors in large-scale distributed systems.

Xentry also falls into this category. One major difference is that Xentry is specifically designed for hypervisor focusing on isolating error propagation. This capability is very important in the hypervisor context to prevent from multiple VM failures and data corruptions. Previous works cannot achieve this. Moreover, the machine learning approach is carefully designed to satisfy unique requirements of the hypervisor. Unlike [27], it does not rely on specific distribution model, and still can achieve high accuracy and low false positive rate. Also, Xentry is designed based on unique hypervisor error behaviors. To detect *long latency errors*, it captures dynamic patterns of control flow to verify its correctness. This can capture errors that cannot be identified by simply checking control flow validness as in [27,36].

Redundancy: Redundancy can be implemented in hardware [15,16,25], software [10,11] or compiler [8,9]. Hardware redundancy generally incurs significant performance and energy overheads as well as high design and production cost. They are often not available in commodity processors. Redundancy can also be implemented in the software level [10,11], leveraging tools from operating systems, such as *fork()* in PLR [10] and *ptrace* in RAFT [11]. These tools

are not available in the hypervisor. Redundancy can also be implemented in compilers [8,9]. The performance overhead is generally high (38% in DAFT [9] and 19% in SRMT [8]). More importantly, the hypervisor executions consist of many low-level operations such as handling interrupts and emulating instructions. These low-level events are not handled in those compiler based methods.

Checkpoint/Restart: Recovery techniques have been proposed in the VM and the application level. At the VM level, checkpointing is a common technique to improve reliability. VM level checkpointing schemes save the VM state which can be recovered in the same or a different server later on [14,37]. However, checkpointing involves intensive memory and I/O operations, which incur significant performance overhead and resource contention. Checkpointing procedures may briefly pause the protected VM to dump VM memory data. Even if the pause may be very short (in ms), it periodically causes downtime or delay in service. Therefore, they are often designed for protecting a single VM rather than all VMs across whole data centers. The application-level checkpointing schemes, e.g., MPI checkpointing/restart schemes [38], are only suitable for specific applications.

Hypervisor Fault Tolerance: Most of previous works do not focus on the hypervisor reliability with a few of exceptions. For example, ReHype re-vitalizes the hypervisor upon failures based on micro-reboot techniques [17]. It is transparent to applications and incurs minimum overhead in fault-error runs. It relies on effective detection techniques. Soft errors in the hypervisor context may propagate to VM executions and cause VM failures, applications failures, or silent data corruptions. Our detection framework can effectively detect these errors within short detection latencies and prevent them from corrupting VM executions. Therefore, our work is orthogonal to ReHype. Tinychecker [39] uses nested virtualization technique and is designed to detect malicious hypervisor behaviors that are caused by software bugs. In contrast to Tinychecker, Xentry focuses on hardware errors and leverages hardware performance counters to monitor VM transition behaviors. It is less intrusive to the hypervisor, incurs less overhead, and minimizes the chance of the framework being affected.

VIII. Conclusion

In this paper, we have designed Xentry, a soft error detection framework, in the hypervisor. Xentry detects close to 99% of faults. Particularly, we have designed VM transition detection based on machine learning algorithms to identify incorrect control flow. It effectively limits soft error propagation by detecting 92.6% of SDCs. Xentry greatly improves fault isolation capability in the virtual systems. Because Xentry leverages existing hardware support for data collection in runtime, the performance overhead is very low. Xentry is designed for hypervisors and can be easily implemented and deployed in many virtualized data centers. In future work, we plan to develop new techniques to further increase the detection coverage and reduce the false positive rate.

Acknowledgment

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

References

[1] S. S. Mukherjee *et al.*, "The soft error problem: An architectural perspective," in *HPCA*, 2005.

[2] S. Michalak *et al.*, "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.

[3] X. Li *et al.*, "A realistic evaluation of memory hardware errors and software system susceptibility," in *USENIX ATC*, 2010.

[4] V. Sridharan *et al.*, "A field study of dram errors," 2012.

[5] M. Snir *et al.*, "Addressing failures in exascale computing," Argonne National Laboratory (ANL), Tech. Rep., 2013.

[6] N. Har'El *et al.*, "Efficient and scalable paravirtual i/o system," in *USENIX ATC'13*, 2013, pp. 231–242.

[7] A. Landau *et al.*, "Splitx: split guest/hypervisor execution on multi-core," in *WIOV'11*, 2011.

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

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

[10] Shye, Alex et al., "Using process-level redundancy to exploit multiple cores for transient fault tolerance," in *DSN 2007*, 2007, pp. 297–306.

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

[12] A. Lenharth *et al.*, "Recovery domains: an organizing principle for recoverable operating systems," in *ASPLOS*, 2009.

[13] C. Clark *et al.*, "Live migration of virtual machines," in *NSDI*, 2005.

[14] Wang, Long et al., "Checkpointing virtual machines against transient errors," in *IOLTS, 2010*, July, pp. 97–102.

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

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

[17] Le, Michael et al., "Rehype: enabling vm survival across hypervisor failures," in *VEE*, 2011.

[18] Intel, "Intel 64 and ia-32 architectures software developers manual."

[19] AMD, "Amd64 architecture programmers manual."

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

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

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

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

[24] S. K. S. Hari *et al.*, "Low-cost program-level detectors for reducing silent data corruptions," in *DSN*, 2012.

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

[26] R. Vemu *et al.*, "CEDA: Control-Flow Error Detection Using Assertions," *IEEE Trans. Comput.*, vol. 60.

[27] G. Bronevetsky *et al.*, "Automatic fault characterization via abnormality-enhanced classification," in *DSN*, 2012.

[28] M. Hall *et al.*, "The weka data mining software: an update," *ACM SIGKDD Explorations Newsletter*, 2009.

[29] F. Zhang *et al.*, "Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in *SOSP*, 2011.

[30] Microsoft, "Understanding shims," http://technet.microsoft.com/en-us/library/dd837644(v=ws.10).aspx.

[31] Wind River, "Simics Full System Simulator," Website, 2006, http://www.simics.net.

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

[33] "Oprofile," http://oprofile.sourceforge.net/.

[34] N. J. Wang *et al.*, "Restore: Symptom based soft error detection in microprocessors," in *Dependable Systems and Networks (DSN), 2005 35th Annual IEEE/IFIP International Conference on*, 2005, pp. 30–39.

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

[36] J. Gao *et al.*, "Modeling probabilistic measurement correlations for problem determination in large-scale distributed systems," in *ICDCS*, 2009.

[37] Cully, Brendan et al., "Remus: high availability via asynchronous virtual machine replication," in *NSDI*, 2008, pp. 161–174.

[38] L. Bathen *et al.*, "Havoc: A hybrid memory-aware virtualization layer for on-chip distributed scratchpad and non-volatile memories," in *DAC*, 2012.

[39] Tan, Cheng et al., "Tinychecker: Transparent protection of vms against hypervisor failures with nested virtualization," in *DSN Workshop 2012*, June, pp. 1–6.