Improving the Performance of Hypervisor-Based Fault Tolerance

Jun Zhu, Wei Dong, Zhefu Jiang, Xiaogang Shi, Zhen Xiao
School of Electronics Engineering & Computer Science
Peking University, Beijing, China
{*zhujun, dongwei, jzf, sxg, xiaozhen*}*@net.pku.edu.cn*

Xiaoming Li
State Key Laboratory of Advanced Optical Communication Systems & Networks
Peking University, Beijing, China
lxm@pku.edu.cn

***Abstract*—Hypervisor-based fault tolerance (HBFT), a checkpoint-recovery mechanism, is an emerging approach to sustaining mission-critical applications. Based on virtualization technology, HBFT provides an economic and transparent solution. However, the advantages currently come at the cost of substantial overhead during failure-free, especially for memory intensive applications.**

This paper presents an in-depth examination of HBFT and options to improve its performance. Based on the behavior of memory accesses among checkpointing epochs, we introduce two optimizations, *read fault reduction* and *write fault prediction*, for the memory tracking mechanism. These two optimizations improve the mechanism by 31.1% and 21.4% respectively for some application. Then, we present *software-superpage* which efficiently maps large memory regions between virtual machines (VM). By the above optimizations, HBFT is improved by a factor of 1.4 to 2.2 and it achieves a performance which is about 60% of that of the native VM.

***Keywords*-Virtualization; Hypervisor; Checkpoint; Recovery; Fault Tolerance**

I. Introduction

Reliable service plays an important role in mission-critical applications, such as banking system, stock exchange system and air traffic control system, which cannot tolerate even a few minutes' downtime. Although service providers have taken great efforts to maintain their services, various failures, such as hardware failures [1] and maintenance failures, still occur in data centers.

Hypervisor-based fault tolerance (*HBFT*), such as Remus [2] and Kemari [3], is an emerging approach to sustaining mission-critical applications. As a checkpoint-recovery fault tolerance mechanism [4], *HBFT* works in the primary-backup mode. It capitalizes on the ability of the hypervisor or virtual machine monitor (VMM) [5][6] to replicate the snapshot of a virtual machine (VM) from one host (*primary host*) to another (*backup host*) at frequent intervals. During each epoch, hypervisor records the newly dirtied memory pages of the *primary VM* running on the primary host. At the end of each epoch, the incremental checkpoint [7] (i.e., the newly dirtied pages, cpu state, etc.) is transferred to update the state of the *backup VM* which resides on the backup host. When the primary VM fails, its backup VM will take over the service. In this way, *HBFT* provides an economic fault tolerance solution with commodity hardware and software. Besides, without intervening in upper layers (e.g., applications or libraries), *HBFT* works in a transparent manner, even for legacy applications and operating systems.

Contact author: Zhen Xiao.

However, the overhead of current *HBFT* systems seriously affect the performance of the primary VM during failure-free, especially for memory-intensive workloads. Lu and Chiueh [8] reported that the performance of some realistic data center workloads experienced 200% degradation. Even with asynchronous state transfer optimization, in some benchmark evaluation, Remus [2] still leads to a 103% slow down compared to the native VM performance. Kemari [3] reported a similar performance penalty. In our development of *HBFT* system [9], how to improve the performance of the primary VM during failure-free was a challenge.

The performance overhead of *HBFT* results from several sources. Output commit problem [7], e.g., a disk write operation or a network transmit operation, is a well-known overhead. How to address this overhead is an active area of research [7]. In this paper, we aim to address the overhead coming from memory state synchronization that relies on Xen live migration [10]. In a typical *HBFT* system, in order to track dirtied memory pages of the primary VM in each epoch, hypervisor applies the *log dirty* mode of shadow page table (SPT) [11][1]. This mode incurs a large number of page faults, conflicting the goal of *"Reducing the frequency of exits is the most important optimization for classical VMMs"* [11]. In addition, at the end of each epoch, all the dirtied pages have to be mapped and copied to the driver domain (Dom0) before being transferred to the backup host, which further causes serious performance degradation.

Contributions. In this paper, we present an in-depth examination of *HBFT* and options to improve its performance. The following is a summary of our contributions.

First, we find that, at the granularity of checkpointing epochs, shadow page table entries (*shadow entry* for short) exhibit fine reuse, and shadow entry write accesses exhibit fine spatial locality with a history-similar pattern.

[1]Log dirty mode can also be implemented on nested page table [12]. Unless otherwise specified, we improve the *HBFT* implemented on SPT.

Second, we introduce two optimizations, *read fault reduction* and *write fault prediction*, for the log dirty mode of Xen. These two optimizations promote the performance of the log dirty mode by 31.1% and 21.4% respectively for some workload.

Finally, inspired by the advantages of superpage, we present *software-superpage* to map large memory regions between VMs. The approach significantly accelerates the process of state replication at the end of each epoch.

By the above optimizations, the primary VM achieves a performance which is about 60% of that of the native VM.

The remainder of this paper is organized as follows: The next section introduces related technologies and the general architecture of *HBFT* system. Section 3 describes the behaviors of shadow entry accesses. Section 4 presents the optimizations proposed for *HBFT*. Section 5 presents a comprehensive evaluation of our optimizations. Section 6 is about some related work, and section 7 give our conclusion.

II. Background

This section first provides an overview of the log dirty mode, the core of Xen live migration [10], and then a general architecture of *HBFT*. Readers familiar with this material may skip directly to Section 3.

A. Log Dirty Model

The log dirty mode is implemented on shadow page table (SPT) which is the software mechanism to implement memory virtualization. When running in a VM, the guest OS maintains guest page tables (GPT) that translate *virtual addresses* into *physical addresses* of the VM. The real page tables, exposed to the hardware MMU, are SPTs maintained by the hypervisor. SPTs directly translate virtual addresses into hardware *machine addresses*. Each shadow entry is created on demand according to the guest page table entry (*guest entry* for short).

The log dirty mode is designed for VM live migration to track dirty pages in each *pre-copy* round. The principle of the log dirty mode is as follows. Initially, all the shadow entries are marked as read-only, regardless of the permission of its associated guest entries. When the guest OS attempts to modify a memory page, a shadow page write-fault occurs and is intercepted by the hypervisor. If the write is permitted by its associated guest entry, the hypervisor grants write permission to the shadow entry and marks the page as a dirty one accordingly. Subsequent write accesses to this page will not incur any shadow page faults in the current round.

In the current implementation, when to track dirty pages, Xen first blows down all the shadow entries. Then, when the guest OS attempts to access a page, a shadow page fault occurs since there is no shadow entry existed. Xen intercepts this page fault, re-constructs the shadow entry, and revoke the write permission of the shadow entry. In this way, Xen makes all the shadow entries read-only before write accesses.

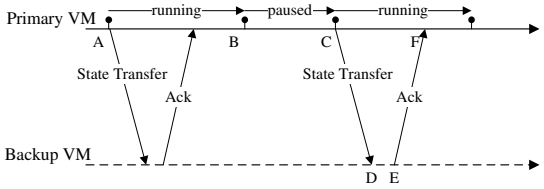

Figure 1: Execution process of HBFT.

Thus, the first write access to any page can be intercepted, and dirtied pages can be tracked.

Therefore, the log dirty mode results in two types of shadow page faults. First, when the shadow entry does not exist, both read and write accesses will generate a shadow page fault. Second, when an attempt is made to modify a page through an existing shadow entry without write permission, a shadow page write-fault occurs.

B. Architecture

Before the release of Remus, we developed a similar *HBFT* prototype ***Taiji*** [9]. Remus uses separate local disk for the primary VM and the backup VM, while *Taiji* is deployed with network-attached storage (NAS). Unless otherwise stated, our evaluations of this work are conducted on *Taiji*. In this subsection, we will introduce the general architecture of *HBFT*.

As a checkpoint-recovery fault tolerance mechanism, consistent state between the primary VM and the backup VM is a prerequisite. *HBFT* implements checkpointing by repeated executions of the final phrase of live migration at a high frequency of tens of milliseconds. Figure 1 illustrates how *HBFT* obtains a consistent state. At the beginning of each epoch (**A**), Xen initializes the log dirty mode for the primary VM. During each epoch, dirtied pages are tracked. At the same time, output state, including transmitted network packets and data written to disk, is blocked and buffered in the backend [5] of Dom0. At the end of each epoch (**B**), the guest OS is paused, and dirty memory pages and CPU state are mapped and copied to Dom0. These contents are then sent to the backup host (**C**) through Dom0's network driver, and the guest OS is resumed simultaneously. Upon receiving an acknowledgment from the backup host (**F**), Dom0 commits the buffered output state generated in the last epoch (the epoch between **A** and **B**).

In general, there are several substantial performance overheads from the above mechanism. Running in the log dirty mode, the "*running*" guest OS generates more shadow page faults than normal. Dealing with page faults is nontrivial, especially in virtualized systems [11]. Furthermore, at the end of each epoch, the guest OS has to be paused, waiting for Dom0 to map and copy the incremental checkpoint. Mapping physical pages between VMs is expensive [13], lengthening the "*paused*" state of the guest OS.

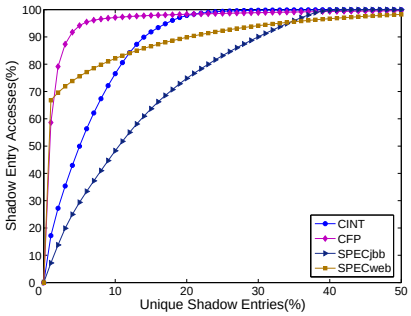

Figure 2: The degree of shadow entry reuse.

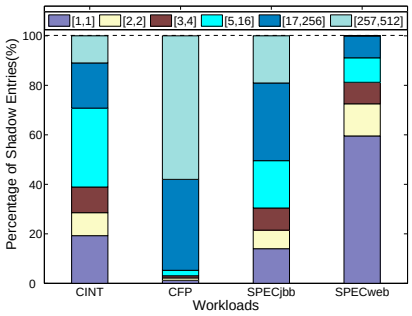

Figure 3: Spatial locality of write accesses.

III. Behavior of Shadow Entry Access

Recall that the log dirty mode results in a considerable number of shadow page faults which result in a substantial performance degradation. To motivate our optimization decisions, in this section, we provide an initial study on the behavior of shadow entry accesses, including the shadow entry reuse and the spatial locality of write accesses.

We study shadow entry accesses at the granularity of epochs, and a shadow entry is recorded at most once during a single epoch, no matter how many times it is accessed. The experiment in this section obtains one checkpoint of the guest OS every 20msec. Other experiment parameters, hardware configurations and benchmarks are discussed in Section 5.

A. Shadow Entry Reuse.

The behavior of page table entry reuse, at the granularity of instructions, has been well studied in literature [14]. We find that, even at the granularity of epochs, shadow entry accesses exhibit a similar behavior. In this paper, shadow entry reuse is defined as: if a shadow entry is accessed in an epoch, it will likely be accessed in future epochs.

Figure 2 demonstrates the degree of shadow entry reuse in different workloads. Reuse is measured as the percentage of unique shadow entries required to account for a given percentage of page accesses. In the workload of *CFP*, which reveals the best shadow entry reuse, less than 5% of unique shadow entries are involved to cover more than 95% page accesses. Even *SPECweb*, a larger workload, also has a fine reuse behavior. Although *SPECjbb* has less entry reuse, nearly 60% page accesses are still carried out through only 15% unique shadow entries.

B. Shadow Entry Write Access.

In this subsection, we study the behavior of shadow entry write access. The spatial locality of write accesses is the tendency of applications to modify memory near other modified addresses. During an entire epoch, larger than 4KB (a page size) virtual memory being modified will involve more than one shadow entry being write accessed. To describe the spatial locality, write access stride (*stride* for short) is defined as consecutive shadow entries that have been write accessed in the same epoch. The *length* of a stride is defined as the number of shadow entries it contains. Usually, several strides exist in an L1 SPT. We define the average length of these strides as ***ave_stride***, used to depict the degree of spatial locality of write accesses for each SPT. And here, *ave_stride* is in the range [0,512]. (512 indicates the total number of page table entries. For a 32-bit system, the range is [0,1024]. For a 64-bit or 32-bit PAE system, it is [0,512].) A longer *ave_stride* indicates better spatial locality. The value of 512 means that all the pages covered by the L1 SPT have been modified, and 0 indicates that no shadow entry is write accessed.

Figure 3 provides the spatial locality of shadow entry write accesses for the workloads investigated. We divide all the shadow entries that have been write accessed within an epoch into six groups according to the length of the strides. For instance, [5,16] contains all the shadow entries that reside in the strides of 5 to 16 entries in length. As shown in Figure 3, the workload *CFP* exhibits best spatial locality of write accesses. More than 90% shadow entries are located in the strides of above 17 entries in length. Surprisingly, nearly 60% entries are located in the strides longer than half of an SPT. Another CPU intensive workload, *CINT*, also has fine spatial locality. However, *SPECweb* exhibits bad spatial locality, because *SPECweb*, as a web server, deals with a large number of concurrent requests of which each induces a small memory modification.

Furthermore, we find that SPT's write accesses present a history-similar pattern. That is, the *ave_stride* of a SPT tends to keep a steady value within some execution period. In order to demonstrate this property, we define ***delta_stride*** as:

$$delta\_stride = |ave\_stride_n - ave\_stride_{n+1}| \quad (1)$$

where ave_stride_n indicates the *ave_stride* of a particular SPT in the n*th* epoch. *Delta_stride* is also in the range [0,512]. A shorter *delta_stride* indicates a more history-similar pattern. It should be noted that we do not use *standard deviation* to depict this property since *ave_stride* values of an SPT between two epochs far apart may be very

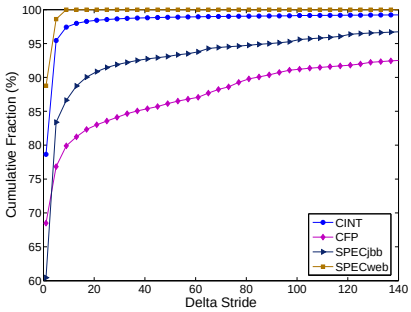

Figure 4: History-similar pattern of write accesses.

different.

Figure 4 provides the distribution of *delta_strides* across the whole execution of each benchmark. With less spatial locality, *SPECweb* still exhibits an excellent history-similar pattern. From Figure 3, we can conclude that the value of most L1 SPT's *ave_strides* is one, leading the vast majority of *delta_strides* to be zero. Even in the workload *CFP* with the lowest degree, 75% of *delta_stride* values are still below 5 shadow entries.

IV. Optimization Details

In this section, we will present our optimization details of *HBFT* implementation on Xen. We will first give our approaches to minimizing the performance overhead resulting from the log dirty mode. Then, we will present the *software-superpage* mechanism to efficiently map a large number of memory pages between virtual machines.

A. Log Dirty Mode Optimization

In the previous section, we analyzed the behavior of shadow entry accesses. Based on these observations, we propose *read-fault reduction* and *write-fault prediction* to optimize the log dirty mode of Xen.

1) Read-Fault Reduction: Log dirty mode, first developed for Xen live migration, did not take the behavior of shadow entry reuse into consideration [10]. At the beginning of each *pre-copy* round, all the SPTs are destroyed. In each round, the first access to any guest page will result in a shadow page fault and write accesses will be intercepted by Xen. The side effect of this mechanism is that too many shadow page read-faults are generated, but only write accesses are necessarily intercepted in order to record dirtied pages. The mechanism has little effect on live migration since the whole migrating process takes a few number of *pre-copy* rounds before completion.

However, for the *HBFT* system which runs in repeated checkpointing epochs at frequent intervals during failure-free, the mechanism of the log dirty mode induces too much performance overhead. Based on the behavior of shadow entry reuse analyzed in Section 3, we propose an alternative implementation.

Intuitively, we only need to scan the L1 shadow entries one by one and revoke all write permissions. Thus, all the shadow entries can be reserved for read accesses in future epochs, avoiding recreating them repeatedly. However, if there are too many L1 shadow entries and few of them will be reused in future, the intuitive approach may not outweigh the original one. In addition, when entries with writable permission are in the minority, it is also unnecessary to scan all L1 shadow entries. It is noteworthy that the guest OS is paused in this scanning period. Longer scanning time means more performance degradation on the guest OS.

In order to revoke write permissions efficiently, we use a bitmap *marker* for each SPT. Take x86 32-bit PAE (512 entries per SPT) for example. Each bit of an eight-bit marker corresponds to one eighth of the L1 SPT and indicates whether there are writable entries in the segment. At the beginning of each epoch, all the markers are initialized to zero, which means no writable entries exists. During the period of execution, when a shadow page write-fault occurs, its associated bitmap is set according to the position of the shadow entry. At the end of this epoch, which segment is to be scanned is decided by the marker. Due to the fine spatial locality of most applications, those entries with writable permission tend to cluster together, making scanning process efficient. Thus, the paused period at the end of each epoch can be kept in an acceptable length.

Though optimized for *HBFT* systems, our *read fault reduction* is also beneficial for live migration. We are planning to merge these modifications into the upcoming version of Xen.

2) Write-Fault Prediction: In order to track dirty pages, hypervisor intercepts write accesses by revoking the write permission of shadow entries. First access to any page results in a shadow page write-fault. Handling page faults incurs non-trivial overhead, especially for applications with large writable working sets [10]. We consider improving log dirty mode further by predicting which entries will be write accessed in an epoch and granting write permission in advance.

When a shadow entry is predicted to be write accessed in the epoch, the page pointed to by this entry is marked as dirty, and the entry is granted with write permission, which will avoid shadow page write-fault if the page is really modified later. However, prediction faults will produce more "dirty" pages which consume more bandwidth to update the backup VM. The *FDRT* technique proposed in [8], which transfers incremental checkpoint at a fine-grained dirty region within a page, can pick out fake dirty pages before transferring, but it incurs more overhead of computing a hash value for each page.

Based on the behavior of shadow entry write accesses analyzed in the previous section, we develop a prediction algorithm which is called ***Histase*** (***hi***story ***st***ride b***ase***d) and relies on the regularity of the system execution. In the

following, we will answer two questions: (1) how to predict write accesses effectively? (2) how to rectify prediction faults efficiently?

To describe the behavior of write accesses, Histase maintains ***his_stride*** for each SPT, which is defined as:

$$his\_stride = his\_stride * \alpha + ave\_stride * (1 - \alpha) \quad (2)$$

where *his_stride* is set to zero initially and the *ave_stride* obtained from the previous epoch forces *his_stride* to adapt to the new execution pattern, and $0 \leq \alpha < 1$.

There are two points to be explained for this equation. First, the parameter α provides explicit control over the estimation of SPT's historical stride behavior. At one extreme, $\alpha = 0$ estimates *his_stride* purely based on the *ave_stride* from the last epoch. At the other extreme, $\alpha \approx 1$ specifies a policy that *his_stride* is estimated by a long history. In this paper, we set $\alpha = 0.7$ by default. Second, Histase builds upon the optimization of *read fault reduction*. When scanning L1 SPTs at the end of each epoch, *ave_stride* can be calculated at the same time with trivial CPU overhead.

When a valid shadow page write-fault occurs, Histase tries to promote more writable permission based on *his_stride*. Heuristically, those shadow entries within *his_stride* forwards and *his_stride*/3 backwards become candidates of which those that are not allowed to be writable are ignored. Understandably, when a page is modified, the pages forwards also tend to be modified for spatial locality. However, predicting backwards is somewhat obscure. In practice, some applications traverse large array reversely with small probability. In addition, user stack grows towards lower addresses in some operating systems [14]. Thus, Histase also predicts a smaller number of backward shadow entries.

Prediction faults are inevitable. In order to rectify them, Histase takes advantage of an available-to-software bit (called *Predict* bit in Histase) and *Dirty* bit of L1 shadow entry. When a shadow entry is granted with write permission due to prediction, Histase sets its *Predict* bit and clears the *Dirty* bit. Whenever the entry is write accessed in the future of this epoch, its *Dirty* bit will be set automatically by the MMU. At the end of each epoch, Histase checks each predicted entry. Those entries without *Dirty* bit set are picked out as fake dirty pages.

Faulty predictions result in more shadow entries with write permission, which will make the scanning period at the end of each epoch more time-consuming. Fortunately, since Histase takes effect whenever a shadow page write-fault happens, the predicted entries are close to those pointing to actual dirty pages. With the help of the *marker* proposed in *read fault reduction*, scanning process stays efficient.

B. Software-Superpage

In this subsection, we introduce *software-superpage*, and show how it improves memory state transfer between VMs.

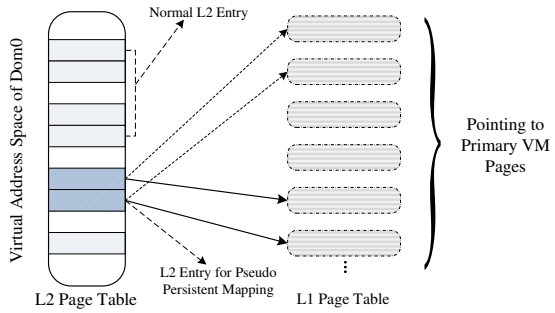

Figure 5: Software-superpage mapping to the primary VM's entire memory pages.

The Xen hypervisor is not aware of any peripherals. It reuses the Dom0's drivers to manage devices, including the network driver. At the end of each epoch, all the dirty pages have to be mapped into Dom0's address space for read-only accesses before being sent to the backup VM through the network driver. The overhead of *mapping*/*unmapping* memory pages between VMs is rather large. Since the primary VM is *paused* in this period, this overhead results in serious performance degradation. Evidently, reducing the *mapping*/*unmapping* overhead can improve the performance of the Primary VM significantly.

The simplest method to eliminate the overhead is to map the entire memory pages of the primary VM into Dom0's address space persistently, avoiding the *map*/*unmap* operations at the end of each epoch. However, the virtual address space required in Dom0 must be equal to the primary VM's memory size. This is not a problem for the 64-bit address systems, but the 32-bit systems with limited address space (4G) still account for a great proportion nowadays. In addition, many 32-bit legacy applications are still in use. Therefore, persistent mapping is not practical when the primary VM is configured with a large memory.

Software-superpage, designed as a pseudo-persistent mapping, reduces the *map*/*unmap* overhead to a low level. Our design builds upon two assumptions. First, Dom0 is non-malicious and can be granted with read-only access to the primary VM's entire physical memory. Second, because of balloon driver [6] or memory hotplug [15], a system's memory pages may be changed. We first assume that the primary VM's memory size keeps constant when being protected, then at the end of this subsection, we will relax this assumption.

Figure 5 illustrates the design details. For brevity, we take 32-bit PAE system for example. In the initialization phase of fault tolerance, Dom0 allocates L1 page tables (PT) for pseudo-persistent mapping. These L1 PTs point to the primary VM's entire memory pages, from zero to the maximum size. For example, if the primary VM's memory size is 4G, then 2,048 L1 PTs in Dom0 are allocated, each covering 2M physical memory pages. In our design, we

allocate a smaller number of L2 PT entries (PTE) than 2,048 L2 PTEs to point to these L1 PTs. For example, 32 L2 PTEs (*i.e.*, 64M virtual address space of Dom0). At any time, among these 2,048 L1 PTs, at most 32 of them are actually installed into these L2 PTEs. We call these L1 PTs *officeholding* PTs. Those uninstalled L1 PTs are called *nonofficeholding* PTs, and are pinned in Dom0's memory giving Xen an intuition that these pages are being used as L1 PTs. When coping a dirty page, Dom0 first checks these L2 PTEs mappings. If the L1 PT that covers the dirty page has been installed into an L2 PTE, the page can be accessed directly. Otherwise, an L2 PTE is updated to point to the L1 PT, promoting the L1 PT to *officeholding*. In this way, we *map*/*unmap* memory pages as superpage mapping with a limited virtual address space.

In order to reduce the times of updating L2 PTEs, we employ an LRU algorithm [14] to decide which L2 PTE should be updated. When an L1 PT is accessed, its associated L2 PTE is marked as the youngest. A *nonofficeholding* L1 PT is always installed into the oldest L2 PTE. This policy is based on the observation that the set of pages that have not been modified are less likely to be modified in the near future. With fine temporal locality of memory accesses, the majority of pages can be directly accessed with its L1 PT already being installed.

The advantages of *software-superpage* are two-fold. On one hand, for fault tolerance daemon, it provides an illusion that all the memory pages of the primary VM are mapped into Dom0's address space persistently. It eliminates almost all *map*/*unmap* overhead. On the other hand, it does little disturbance to the other parts residing in the same virtual address space of Dom0 because only a small part of virtual address space is actually used to access the entire memory pages of the primary VM.

Outstanding Issues. In the design, we make an assumption that the primary VM's memory pages stay constant. However, in a typical virtualization environment, page changes may take place. Transparent page sharing [6][16] is a prevalent approach to harnessing memory redundancy. Pages are shared among VMs if they have identical or similar content. The shared pages except the referenced one are reclaimed from the VMs, and when sharing is broken, new pages will be allocated to the VMs. In addition, the first version of Xen PV network driver used a page flipping[2] mechanism which swapped the page containing the received packet with a free page of the VM [5].

To cope with these challenges, an event channel is employed in Dom0. If any of the primary VM's pages is changed, hypervisor sends a notification to Dom0 through the event channel. Upon notification, Dom0 updates the corresponding L1 shadow entry to point to the new page.

[2]Menon et al. [17] has shown that page flipping is unattractive.

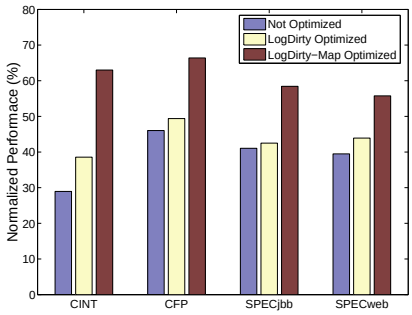

Figure 6: The performance of the primary VM with different configurations.

V. Evaluation

The optimizations presented in the previous section are implemented on Xen-3.3.1, with Dom0 and the primary VM running XenoLinux version 2.6.18 configured with 32-bit PAE kernel.

All the experiments are based on a testbed consisting of two HP ProLiant DL180 Servers, each with two quad-core 2.5GHz processors (8 cores in total), 12G memory and a Broadcom TG3 network interface. The machines are connected via switched Gigabit Ethernet. We deploy the primary VM on one of the two machines, and the backup VM on the other. The primary VM is configured with 2G memory and a single virtual CPU pinned to a physical core of one of the CPU sockets. The Dom0 is configured with the remaining 10G memory and 4 virtual CPUs that are pinned to different cores of the other CPU socket.

In this paper, we focus on the performance overhead resulting from synchronizing memory state between the primary VM and the backup VM in each epoch. The snapshot of virtual disk and network output commit, which are two other components of the *HBFT* system, are disabled in these experiments to eliminate their influence. Besides, the length of an epoch is sensitive to system performance. Throughout the paper, we set an epoch 20 msec as default.

A. Workload Overview

We evaluate our optimization techniques with a variety of benchmarks representative of real-world applications. Table 1 lists the workloads. Among them, *SPECjbb* and *SPECweb* are server applications and candidates for fault tolerance in the real world. The server of *SPECweb* runs in the primary VM, and two separate client machines and one backend simulator (BeSim) are connected with the primary VM via switched Gigabit Ethernet. *CINT* and *CFP* are also presented for reference points.

We run each workload three times and the average value is presented in this paper.

Table I: Workloads Description.

Workload	Description
CINT	SPEC CPU2006 integer benchmark suite.
CFP	SPEC CPU2006 floating point benchmark suite.
SPECjbb	SPECjbb2005 configured with Java 1.6.0. A benchmark that is used to evaluate the performance of Internet servers running Java applications.
SPECweb	SPECweb2005 configured with Apache 2.2. A benchmark that is used to evaluate the performance of World Wide Web Servers.

B. Overall Result

Figure 6 shows the performance of the primary VM which runs different workloads, and the performance is normalized to that of the native VM running in Xen (*baseline*). We present the following configurations: 'Non Optimized' refers to the unoptimized *HBFT*. 'LogDirty Optimized' refers to the version with only the log dirty mode optimized, including *read fault reduction* and *write fault prediction*. 'LogDirty-Map Optimized' refers to the optimized version with both the optimized log dirty mode and the *software-superpage* map. We do not compare the performance with that of the applications running in native operating systems (i.e., non-virtualized environment), because the overhead resulting from Xen hypervisor is rather small [5] and is not the focus of this work.

As shown in Figure 6, *CINT* suffers the worst performance degradation when running in the unoptimized *HBFT*, yielding only about 30% of baseline performance. Relative to the other workloads, *CINT*, which has a larger writable working set, leads to high overhead by log dirty mode and memory mapping. By our optimizations, the performance of *CINT* improves by 33.2% and 84.5% respectively by applying the optimized log dirty mode and *software-superpage* mechanism.

Relative to the optimized log dirty mode, *software-superpage* optimization gains more improvement for all workloads. By our optimizations, the primary VM is improved by a factor of 1.4 to 2.2 and it achieves a performance which is about 60% of that of the native VM. In the following, we will examine each optimization in detail.

C. Log Dirty Mode Improvement

1) Experimental Setup: The log dirty mode is the mechanism to record which memory pages are dirtied. To better understand its improvement, we quantify its performance in an isolated environment. At the beginning of each epoch, we make all memory pages of the primary VM read-only with a *hypercall*. Then, the primary VM resumes running for an epoch of 20msec. After that, we repeat the above procedure without engaging other components of *HBFT*.

2) Log Dirty Mode Efficiency: We evaluate the log dirty mode with the following configurations: 'OriginXen' refers to the Xen with the unoptimized log dirty mode, 'RFRXen' refers to the version with *read fault reduction* optimization and 'WFPXen' refers to the optimized version with *write fault prediction*.

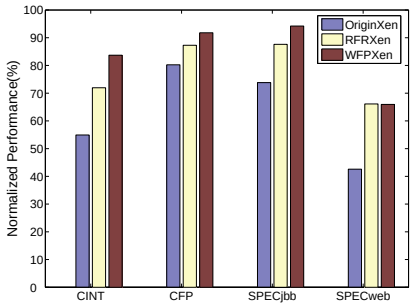

Figure 7: The performance of the log dirty mode, normalized to the native VM.

Performance. Figure 7 compares the performance of the primary VM running in different versions of the log dirty mode with that of the native VM.

The results show that the log dirty mode of OriginXen incurs non-trivial performance degradation, ranging from 19.8% on *CFP* to 57.4% on *SPECweb*.

RFRXen, which exploits the behavior of shadow entry reuse, improves the performance of *CINT* by 31.1% relative to OriginXen. It should be noted that though *SPECweb* experiences a large number of requests with short session, it still derives much benefit from RFRXen, gaining 55.2% improvement.

Based on RFRXen, WFPXen improves the log dirty mode further by well predicting shadow page write-faults. As expected, *CINT*, *CFP* and *SPECjbb* are improved further, by 21.4%, 5.6%, and 8.9% respectively, since they have fine spatial locality as demonstrated in Figure 3. Especially, *SPECjbb* achieves nearly 95% of baseline performance. However, the applications with poor spatial locality yield little improvement. *SPECweb* even suffers one score of degradation (from 402 in RFRXen to 401 in WFPXen). We will analyze these further by reduction of shadow page faults and by prediction accuracy.

Reduction of Shadow Page Faults. To determine where our optimization techniques differ from OriginXen, Figure 8 demonstrates the average count of shadow page faults per epoch with different configurations. RFRXen almost reduces the average count of shadow page read-faults to

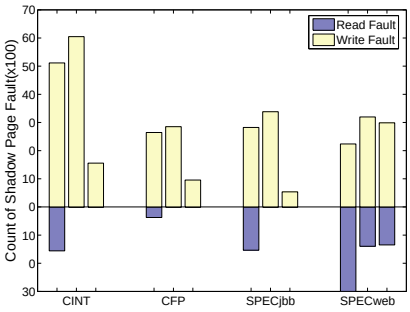

Figure 8: shadow page faults. From left to right, OriginXen, RFRXen and WFPXen.

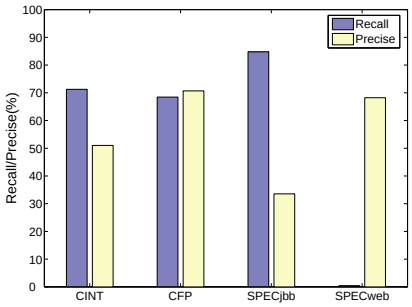

Figure 9: Accuracy of shadow page write-fault prediction.

zero for most applications investigated. However, *SPECweb* still suffers considerable read-faults. *SPECweb* consumes more SPTs since many concurrent processes exist and the working set is very large. For the constraint of memory size reserved for SPTs, Xen has to destroy some of SPTs for newly allocated ones, even though those SPTs will be used in the near future. Besides, destroying and allocating SPTs are common since most of the processes have a short lifetime. The majority of shadow page read-faults come from non-existed shadow entries, and many read-faults remain in *SPECweb*. We are investigating to resize some resource limits of Xen to cope with the larger working set of today's applications.

Another thing to note is that most applications running in RFRXen experience more shadow page write-faults per epoch compared with OriginXen (e.g., 934 more for *CINT*) because the elimination of most shadow page read-faults makes the system run faster. As a result, more application instructions are issued during a fixed epoch, which incurs more shadow page write-faults.

Prediction Accuracy. Histase combines the behaviors of spatial locality and history-similar pattern to predict shadow page write-faults. To better understand efficiency of Histase, we borrow the terminologies from information retrieval: ***recall*** is the number of true predictions divided by the total number of dirty pages in each epoch and ***precise*** is the number of true predictions divided by the total number of predictions.

Figure 9 shows that Histase behaves differently among the workloads. As expected in Figure 3, the applications with fine spatial locality benefit well. Take *CFP* for example. Histase predicts 68.5% of shadow page write-faults, with false predictions being only 29.3%.

Histase predicts few shadow page write-faults in *SPECweb* because of its poor spatial locality, as demonstrated in Figure 3. However, Histase still predicts with high *precise* since it bases its prediction on history-similar pattern, and retains application performance.

Table II: Software-superpage hit ratio.

Workload	*CINT*	*CFP*	*SPECjbb*	*SPECweb*
Hit Ratio	97.27%	97.25%	97.80%	79.45%

D. Software-Superpage Evaluation

With limited virtual address space of Dom0, *software-superpage* eliminates almost all of the memory mapping operations, reducing the primary VM's *paused* period drastically. Throughout this paper, we allocate a fixed 64M virtual address space in Dom0 in order to map all the memory pages of the primary VM (2G).

The performance of *software-superpage* mainly depends upon how effectively we use limited virtual address to map dirty pages. The LRU algorithm is intended to unmap the pages that are least likely to be dirtied again, and here we evaluate how well it achieves that goal.

Table 2 shows the mapping hit ratio for different workloads running in the primary VM. The hit ratio reveals the probability that a newly dirtied page has already been mapped into the limited virtual address space. Due to the memory access locality, *software-superpage* performs well for most workloads, with a hit ratio of over 97%. This mechanism works not so well for the workloads with poor locality, which is confirmed by the hit ratio of *SPECweb*. Nevertheless, it has mapped nearly 80% of the dirty pages accessed.

With a high hit ratio, *software-superpage* eliminates almost all of the mapping operations, reducing the length of the *paused* state greatly. As shown in Figure 6, *software-superpage* improves the performance of the primary VM by at least 30% relative to the unoptimized *HBFT*.

VI. Related Work

Hypervisor-based fault tolerance is an emerging solution to sustain mission-critical applications. Bressoud and Schneider [18] proposed the pioneering system with lockstep method which depends upon architecture-specific implementation. Lockstep requires deterministic replay on the backup VM and is not convenient for multi-core systems. Recently, based on Xen live migration, Remus [2] and Kemari [3] provide an alternative solution. However, like

most checkpoint-recovery systems, both Remus and Kemari incur serious performance degradation for the primary VM. We develop a similar *HBFT* system, *Taiji*. In this paper, we abstract a general architecture of these systems and illustrate where the overhead comes from.

How to address the overhead of *HBFT* has attracted some attention. Closest to our work is Lu and Chiueh's [8]. They focused on minimizing the checkpoint size transferred at the end of each epoch by fine-grained dirty region tracking, speculative state transfer and synchronization traffic reduction using an active backup system. We improve the performance of the primary VM by addressing the overhead of memory page tracking and the overhead of memory mapping between virtual machines. Though the focuses are different, these two studies are complementary to each other.

Checkpoint-recovery mechanism has been used in various areas [4][19][7] to tolerate failures. Many researchers have been engaged in reducing checkpointing overhead. For example, incremental checkpointing [20] is exploited by reducing the amount of data to be saved. The objective of our work is to optimize the checkpointing implementation based on hypervisor, which present a different challenge.

Prefetching is a well-known technique widely applied in computer systems. There is a large body of literature on prefetching for processor caches, which can be viewed in two classes: those that capture strided reference patterns [21], and those that make prefetching decision on historic behavior [22]. In addition, many researchers have focused on reducing MMU walks by prefetching page table entries into TLB. *Distance prefetching* [23], which approximates the behavior of stride based mechanism and tracks the history of strides, is similar to our Histase prefetching.

Interestingly, our *software-superpage* optimization borrows the idea of temporary kernel mappings [24]. Every page in high memory can be mapped through fixed PTEs in the kernel space, which is also an instance of mapping large physical memory by limited virtual addresses.

Software-superpage is inspired by the advantages of superpage which has been well studied in literature [25]. Superpage has been adopted by many modern operating systems, such as Linux [24] and FreeBSD [26]. These studies rely on hardware implementations of superpages which restrict to map physically continuous page frames. Swanson et al. [27] introduced an additional level of address translation in memory controller so as to eliminate the continuity requirement of superpages. Our *software-superpage* mechanism, which also avoids the continuity requirement, is designed to reduce the overhead of mapping memory pages between VMs.

VII. Conclusion

One of the disadvantages of *HBFT* is that it incurs too much overhead to the primary VM during failure-free. In this paper, we first analyze where the overhead comes from in a typical *HBFT* system. Then, we analyze memory accesses at the granularity of epochs. Finally, we present the design and implementation of the optimizations to *HBFT*. We illustrate how we address the following challenges, including: a) analyzing the behavior of shadow entry accesses, b) improving the log dirty mode of Xen with *read fault reduction* and *write fault prediction*, and c) designing *software-superpage* to map large memory region between VMs. The extensive evaluation shows that our optimizations improve the performance of the primary VM by a factor of 1.4 to 2.2 and the primary VM achieves about 60% of the performance of that of the native VM.

Acknowledgment

We are very grateful to the anonymous reviewers for their valuable feedback that significantly improved the quality of this paper. This work is supported by the National Grand Fundamental Research 973 Program of China under Grant No.2007CB310900, the MoE-Intel Joint Research Fund 4523362, and the National 985 Program for Peking University phase two funding.

References

[1] N. Rafe, "Minor outage at facebook monday." CNet News, January 2009.

[2] 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 (NSDI '08)*, Berkeley, CA, USA, 2008.

[3] T. Yoshiaki, S. Koji, K. Seiji, and M. Satoshi, "Kemari: Virtual machine synchronization for fault tolerance," 2008.

[4] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher *et al.*, "Recovery oriented computing (ROC): Motivation, definition, techniques," Tech. Rep., 2002.

[5] 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," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, NY, USA, 2003.

[6] C. A. Waldspurger, "Memory resource management in vmware esx server," in *Proceedings of the 5th Symposium on Operating System Design and Implementation (OSDI '02)*, New York, USA, 2002.

[7] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys (CSUR '02)*, vol. 34, no. 3, pp. 375–408, 2002.

[8] M. Lu and T.-c. Chiueh, "Fast memory state synchronization for virtualization-based fault tolerance," in *Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '09)*, Estoril, Lisbon, 2009.

[9] "http://net.pku.edu.cn/vc/files/ft/index.html."

[10] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI '05)*, Berkeley, CA, USA, May 2005.

[11] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '06)*, New York, USA, 2006.

[12] AMD, *AMD Pragrammer's Manual, Volume2*, 2007.

[13] A. Burtsev, K. Srinivasan, P. Radhakrishnan, L. N. Bairavasundaram, K. Voruganti, and G. R. Goodson, "Fido: Fast inter-virtual-machine communication for enterprise appliances," in *Proceedings of the 2009 USENIX Annual Technical Conference (USENIX '09)*, San Diego, USA, 2009.

[14] A. S.Tanenbaum, *Modern Operating Systems*. Prentice Hall, 2002.

[15] D. Hansen, M. Kravetz, and B. Christiansen, "Hotplug memory and the linux vm," in *Proceedings of the Ottawa Linux Symposium*, pp. 278–294.

[16] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat, "Difference engine: Harnessing memory redundancy in virtual machines," in *8th USENIX Symposium on Operating System Design and Implementation (OSDI '08)*, San Diego, CA, USA, 2008.

[17] A. Menon, A. L. Cox, and Z. Willy, "Optimizing network virtualization in xen," in *Proceedings of the 2006 USENIX Annual Technical Conference (USENIX '06)*, Berkeley, CA, USA, 2006.

[18] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault tolerance," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, New York, USA, 1995.

[19] J. Hursey, T. I. Mattox, and A. Lumsdaine, "Interconnect agnostic checkpoint/restart in open mpi," in *Proceedings of the 18th ACM International symposium on High Performance Distributed Computing (HPDC '09)*, Garching, Germany, 2009.

[20] G. Bronevetsky, D. J. Marques, K. K. Pingali, R. Rugina, and S. A. McKee, "Compiler-enhanced incremental checkpointing for openmp applications," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*, Salt Lake City, UT, USA, 2008.

[21] F. Dahlgren, M. Dubois, and P. Stenstrom, "Fixed and adaptive sequential prefetching in shared memory multiprocessors," in *Proceedings of the 1993 International Conference on Parallel Processing (ICPP '93)*, 1993.

[22] J. Doug and G. Dirk, "Prefetching using markov predictors," *IEEE Transactions on Computers*, vol. 48, no. 2, pp. 121–133, 1999.

[23] G. B. Kandiraju and A. Sivasubramaniam, "Going the distance for tlb prefetching: an application-driven study," in *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA '02)*, Washington, DC, USA, 2002.

[24] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*. O'Reilly & Associates, 2003.

[25] T. H. Romer, W. H. Ohlrich, A. R. Karlin, and B. N. Bershad, "Reducing tlb and memory overhead using online superpage promotion," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA '95)*, New York, USA, 1995.

[26] N. Juan, I. Sitaram, D. Peter, and C. Alan, "Practical, transparent operating system support for superpages," in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, New York, USA, 2002.

[27] M. Swanson, L. Stoller, and J. Carter, "Increasing tlb reach using superpages backed by shadow memory," in *Proceedings of the 25 th Annual International Symposium on Computer Architecture (ISCA '98)*, Barcelona, Spain, 1998.