Optimizing the Performance of Virtual Machine Synchronization for Fault Tolerance

Jun Zhu,Zhefu Jiang,Zhen Xiao, *Senior Member, IEEE,* and Xiaoming Li, *Senior Member, IEEE*

Abstract—Hypervisor-based fault tolerance (HBFT), which synchronizes the state between the primary VM and the backup VM at a high frequency of tens to hundreds of milliseconds, is an emerging approach to sustaining mission-critical applications. Based on virtualization technology, HBFT provides an economic and transparent fault tolerant solution. However, the advantages currently come at the cost of substantial performance 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 performance by 31% and 21%, respectively, for some applications. Then, we present *software superpage* which efficiently maps large memory regions between virtual machines (VM). Our optimization improves the performance of HBFT by a factor of 1.4 to 2.2 and achieves about 60% of that of the native VM.

Index Terms—Virtualization, Hypervisor, Checkpoint, Recovery, Fault Tolerance.

1 Introduction

Reliable service plays an important role in mission-critical applications, such as banking systems, stock exchange systems and air traffic control systems, 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 [2], maintenance failures [3] and power outage [4], still occur in data centers. Currently, when a failure happens, it will take up to hours or days to resolve the problem, which will incur huge economic losses for some key applications.

Obviously, reliable data centers need an effective and efficient failure recovery mechanism to prevent catastrophe. Hypervisor-based fault tolerance (*HBFT*) [5][6][7], employing the checkpoint-recovery protocol [8], is an emerging approach to sustaining mission-critical applications. *HBFT* works in the primary-backup mode. It capitalizes on the ability of the hypervisor or virtual machine monitor (VMM) [9] to replicate the snapshot of the *primary VM* from one host (*primary host*) to another (*backup host*) every tens to hundreds of milliseconds. During each epoch (the time between checkpoints), 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 [10] (i.e., the newly dirtied pages, CPU state and device state.) is transferred to update the state of the *backup VM* which resides in the backup host. When the primary VM fails, its backup VM will take over the service, continuing execution from the latest checkpoint.

HBFT has two main advantages in providing fault tolerant services. First, *HBFT* employs virtualization technology and runs on commodity hardware and operating systems. It is much cheaper than the commercial fault tolerant servers (e.g., HP NonStop Server [11]) that use specialized hardware and customized software to run in fully synchronous mode. Second, *HBFT* works in the hypervisor layer and can provide fault tolerance for legacy applications and operating systems running on top of it.

However, the overhead of current *HBFT* systems is quite high during failure-free, especially for memory-intensive workloads. Lu and Chiueh reported that the performance of some realistic data center workloads experienced a 200% degradation [12]. Even with asynchronous state transfer optimization, Remus still leads to a 103% slow down compared to the native VM performance for some benchmark [5]. Kemari reported a similar performance penalty [6]. The goal of this work is to improve the performance of the primary VM during failure-free.

The performance overhead of *HBFT* comes from several sources. Output commit problem [10], e.g., a disk write operation or a network transmit operation, is a well-known source of the overhead. How to reduce this overhead is an active area of research [10][13][14]. In this paper, we address a different source of the overhead: that due to memory state synchronization between the primary and the backup machines. In a typical *HBFT* system, the hypervisor needs to track dirtied memory pages of the primary

- *Jun Zhu, Zhefu Jiang, and Zhen Xiao is with the School of Electronics Engineering & Computer Science, Peking University, China, 100871. E-mail: {zhujun, jzf, xiaozhen}@net.pku.edu.cn*
- *Xiaoming Li is with State Key Laboratory of Software Development Environment, MOST, China, 100871. E-mail: lxm@pku.edu.cn*
- *Contact Author: Zhen Xiao. This paper is an extended version of our IPDPS '10 paper [1].*

VM in each epoch [1]. The memory tracking mechanism of the shadow page table (SPT) incurs a large number of page faults [16][17], conflicting the goal of [17]: *"Reducing the frequency of exits is the most important optimization for classical VMMs"*. In addition, at the end of each epoch, all the dirtied pages have to be mapped and copied to the driver domain (Domain0) 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 optimizations to improve its performance. The following is a summary of our contributions.

First, we find that shadow page table entries (*shadow entry* for short) exhibit fine reuse at the granularity of checkpoint epochs, and that shadow entry write accesses exhibit fine spatial locality with a history-similar pattern. These observations provide the insight in building an efficient *HBFT*.

Second, we introduce two optimizations, *read fault reduction* and *write fault prediction*, for the memory tracking mechanism. They improve the performance by 31% and 21%, respectively, for some workloads.

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

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

The remainder of this paper is organized as follows. The next section presents an overview of the system architecture and some related techniques. Section 3 analyzes the behavior of shadow entry accesses across checkpoint epochs. The details of our optimizations are presented in Section 4 and evaluated in Section 5. Section 6 describes the related work and section 7 concludes.

2 Background

Logging and checkpointing are two techniques in providing fault tolerant solutions (see the excellent survey by Elnozahy et al. [13]). *HBFT* implements fault tolerance in the hypervisor and protects the whole VM that encapsulates the guest OS and the applications. It can be classified into two categories: log-based systems such as VMware FT [18] and Marathon everRun level 3 [19], and checkpoint-based systems such as Remus [5], Kemari [6] and our Taiji system [7]. Systems in the former category suffer from two shortcomings. First, they depend heavily on the the target ISA (instruction set architecture): each architecture needs a specific implementation in the hypervisor. Second, they incur a high overhead in multiprocessor environment due to shared memory access [20]. For this reason, current commercial log-based systems can work only in single processor environment.

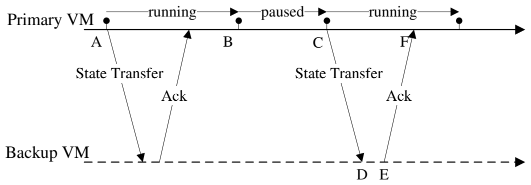


Fig. 2. General execution process of HBFT.

In this paper, we focus on checkpoint-based *HBFT* [5][6][7] which has attracted much interest recently. In such a system, the state of the backup VM is frequently synchronized with that of the primary VM. When the primary VM fails, the backup VM takes over seamlessly. Before the release of Remus, we developed a similar prototype ***Taiji*** [7] on Xen [21]. Unlike Remus which uses separate local disks for the primary VM and the backup VM, our system is deployed with Network Attached Storage (NAS). This alleviates the need to synchronize modified disk contents. Since it accesses the shared storage at the block level, file system state is maintained in case of fail over. Should shared storage become a single point of failure, RAID (Redundant Array of Inexpensive Disks) [22] or commercial NAS solutions (e.g., Synology Disk Station [23]) can be employed. In this section, we first introduce the architecture of Taiji and then its memory tracking mechanism.

2.1 Taiji Prototype

The architecture of Taiji is shown in Fig 1. The primary VM and the backup VM reside in separate physical hosts. Each host runs a hypervisor (i.e., Xen). Initially, the two VMs are synchronized by copying the state (i.e., all memory pages, CPU state and device state) of the primary VM to the backup VM. Then, the primary VM runs in repeated epoches by suspending/resuming its VCPUs. In each epoch, the primary hypervisor captures the incremental checkpoint (i.e., changed memory pages, CPU state and device state) of the primary VM and sends it to the backup host through Domain0. The output of the primary VM generated in each epoch is blocked in Domain0 until the acknowledgement of the checkpoint is received from the backup host. The hypervisor on the backup host (backup hypervisor) updates the state of the backup VM accordingly.

As a checkpoint-recovery protocol, consistent state between the primary VM and the backup VM is a prerequisite. Taiji implements checkpointing by repeated executions of the final phrase of the VM live migration [24][25] at a high frequency of tens of milliseconds. Fig 2 illustrates the general execution

[1] Memory tracking mechanism can also be implemented on nested page table [15]. Unless stated otherwise, we improve the *HBFT* implemented on SPT.

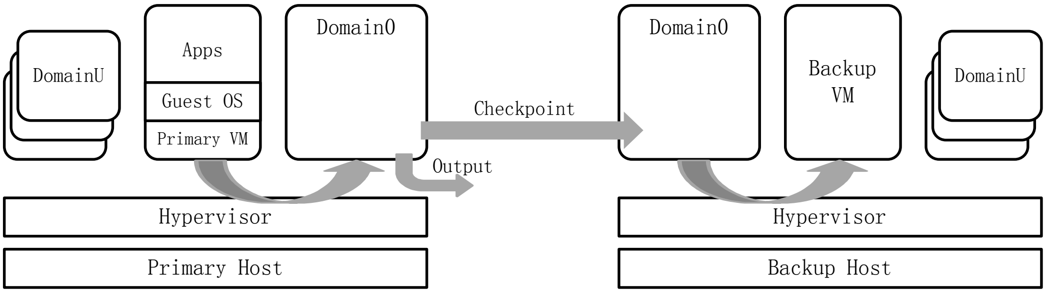


Fig. 1. The architecture of Taiji. The execution of the primary VM is divided into epochs. In each epoch, Domain0 obtains an incremental checkpoint of the primary VM, sends it to the backup host, and handles the output commit problem.

process of checkpoint-based *HBFT*. This execution process is suitable for all checkpoint-based *HBFT*, not specific to Taiji.

At the beginning of each epoch (**A**), the primary hypervisor initializes memory tracking mechanism for the primary VM. During each epoch (between **A** and **B**), the modified pages are tracked. At the same time, output state, e.g., transmitted network packets and data written to disk, is blocked and buffered in the backend of Domain0 [21]. At the end of each epoch (**B**), the guest OS is paused, and dirty memory pages are mapped and copied to Domain0. These dirty pages, as well as CPU state and device state, are then sent to the backup host (**C**) through Domain0's network driver, and the guest OS is resumed simultaneously. Upon receiving acknowledgment from the backup host (**F**), Domain0 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. The memory tracking mechanism (Section 2.2) relies on hardware page protection, so the *"running"* guest OS generates more page faults than normal. Dealing with page faults is nontrivial, especially in virtualized systems [17]. Furthermore, at the end of each epoch, the guest OS has to be paused, waiting for Domain0 to map and copy the dirty pages into Domain0's address space. Mapping physical pages between VMs is expensive [26], lengthening the *"paused"* state of the guest OS.

2.2 Memory Tracking

In order to record dirty pages in each epoch, *HBFT* employs a memory tracking mechanism which is called the *log dirty mode* in Xen.[2] The log dirty mode is implemented on the shadow page table (SPT). Fig 3 shows the details of SPT, which is the software mechanism of memory virtualization. The structure of SPT is the same as the guest page table. 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) and physical-to-machine table (P2M). The log dirty model relies on the hardware page protection of the SPT to track memory write operations by the guest OS.

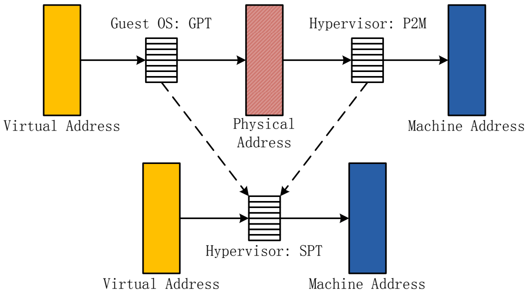


Fig. 3. The structure of shadow page table (SPT).

The log dirty mode was first designed for VM live migration to track dirty memory pages. VM live migration employs an iterative copy mechanism to ease performance degradation during migration. In the first iteration, all the VM pages are transferred to the designated host without pausing the VM. Subsequent iterations copy those pages dirtied during the previous transfer phase. These subsequent iterations are called "pre-copy" rounds. In each "pre-copy" round, the hypervisor enables the log dirty mode of SPT to record dirty pages. 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

[2] We will use memory tracking mechanism and log dirty model in this paper interchangeably.

to this page will not incur any shadow page faults in the current round.

In the current implementation, when tracking dirty pages in the next round, Xen first blows down all the shadow entries. Then, when the guest OS attempts to access a page, a shadow page fault occurs since no shadow entry exists. Xen intercepts this page fault, re-constructs the shadow entry, and revokes its write permission. By doing so, Xen makes all the shadow entries read-only. 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 access 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.

3 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 quantify this, we run SPEC CINT2006 (CINT) [27], SPEC CFP2006 (CFP) [28], SPEC Jbb2005 (SPECjbb) [29], and SPEC Web2005 (SPECweb) [30] in the primary VM, and examine 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 detailed description of benchmarks are discussed in Section 5.

3.1 Shadow Entry Reuse.

The behavior of page table entry reuse, at the granularity of instructions, has been well studied in the literature [31]. 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.

Fig 4 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 responsible for more than 95% page accesses. Even *SPECweb*, a larger workload, 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.

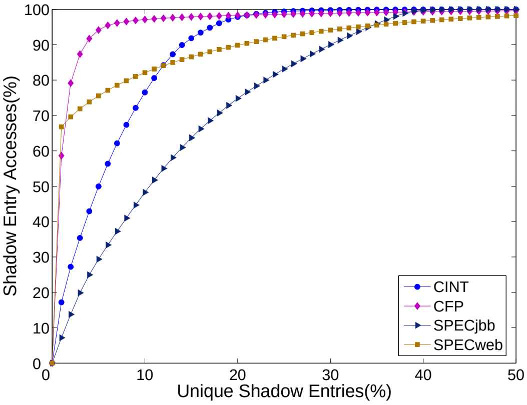


Fig. 4. Shadow entry reuse. The reuse degree is represented as shadow entry access coverage by a given percentage of unique entries.

3.2 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 [3]. 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. Note that 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 (Physical Address Extension) [32] 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.

Fig 5 provides the spatial locality of shadow entry write accesses for the workloads investigated. We divide all 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 Fig 5, 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%

[3] We follow the terminologies in the Intel Manual [32] here. The L1 page table is the last level of page table, and L2 page table is the upper level page table which points to the L1 page table.

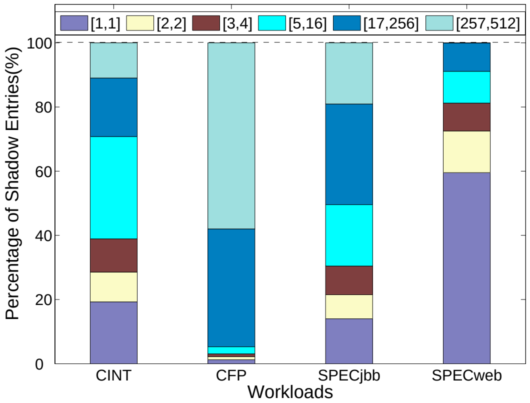


Fig. 5. Spatial locality of write accesses. The shadow entries is divided into six groups based on the length of strides where they are located.

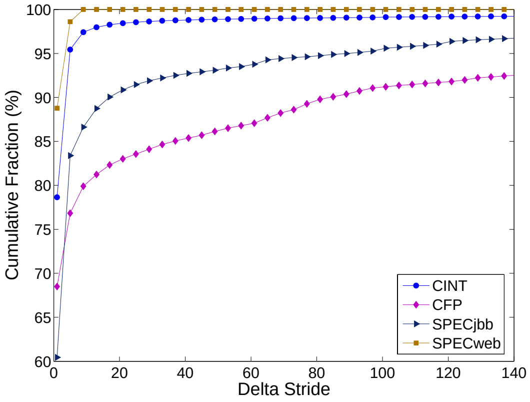


Fig. 6. History-similar pattern of write accesses. The degree of history-similarity is represented as the distribution of *delta_strides* across the whole execution of each benchmark.

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 poor spatial locality, because as a web server, it needs to deal with a large number of concurrent requests, each of which 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 different.

Fig 6 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 Fig 5, 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.

4 Optimization Details

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

4.1 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.

4.1.1 Read-Fault Reduction

Log dirty mode, first developed for VM live migration, does not take into consideration the behavior of shadow entry reuse [24], [25]. In VM live migration, at the beginning of each *pre-copy* round, all the SPTs are destroyed. This causes the first access to any guest page to result in a shadow page fault and thus write accesses can be identified by the hypervisor. The side effect of this mechanism is that the first read access to any page also induces a shadow page fault, even though only write accesses need to be tracked. This mechanism has little effect on VM live migration since the whole migrating process takes only a few number of *pre-copy* rounds before completion.

However, for the *HBFT* system which runs in repeated checkpointing epochs (rounds) at frequent intervals during failure-free, the mechanism of the log dirty mode induces too much performance overhead. Through extensive experiments, we find that these overhead comes from frequent shadow page faults in each execution epoch because all the SPTs have been destroyed at the beginning of each epoch. Dealing with shadow page faults in virtualization environment incurs non-trivial performance degradation [17]. Based on the behavior of shadow entry reuse analyzed in Section 3, we propose an alternative implementation.

Instead of destroying the SPTs at the beginning of each epoch, we can just make them read only. This avoids the overhead of recreating them in the next epoch and yet is sufficient to trap write accesses in order to record dirty memory regions. However, making the SPTs read only requires checking all L1 shadow entries one by one and revoke their write permissions.

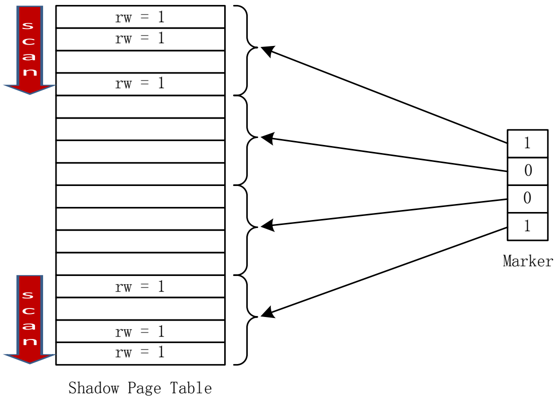


Fig. 7. Selectively check shadow page table entries by the marker and revoke write permissions at the beginning of each epoch.

This scanning process can also be time-consuming. If the number of L1 shadow entries is large and few of them is reused in the future, this intuitive approach may not outweigh the original one. In addition, when entries with write permissions are in the minority, it is also inefficient to scan all L1 shadow entries in order to identify them. 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. Fig 7 illustrates our mechanism. Each bit of the four-bit marker corresponds to one fourth of the L1 SPT and indicates whether there are writable entries in the segment. At the beginning of each epoch, we check the marker to identify segments with writable entries and only need to scan those segments to revoke their write permission. We then clear the corresponding bitmap in the maker to zero. 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. In Fig 7, there are shadow page write-faults generated through entries in the first and the fourth parts of the SPT, and the first bit and the fourth bit of the marker are set accordingly. At the end of this epoch, we only need to check the shadow entries in the first and fourth parts. 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. In our current implementation, we use a eight-bit marker for each SPT. How to select the number of bits for the marker properly is our future work.

4.1.2 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 a non-trivial overhead, especially for applications with large writable working sets [24]. 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 indeed modified later. However, prediction errors will produce false "dirty" pages which consume more bandwidth to update the backup VM. The *FDRT* technique proposed in [12], which transfers incremental checkpoint at a fine-grained dirty region within a page, can pick out false dirty pages before the transfer, but at the expense 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 0≤ α <1. *his_stride* is initialized to the first value of *ave_stride* and the *ave_stride* obtained from the previous epoch makes *his_stride* adapt to the new execution pattern. 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. Histase builds upon the bit marker 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 makes nearby pages writable based on *his_stride*. Heuristically, those shadow entries within *his_stride* forwards and *his_stride*$/3$ backwards are made writable except those that are not allowed to be writable. Understandably, when a page is modified, the pages forwards also tend to be modified due to spatial locality. However, the justification for predicting backwards is less clear. In practice, some applications traverse large array reversely with small probability. In addition, user stack grows towards lower addresses in some operating systems [31]. 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. If the entry is indeed write accessed in the rest of this epoch, its *Dirty* bit will be set by the MMU automatically. At the end of the epoch, Histase checks each predicted entry and picked out those without the corresponding *Dirty* bit set as fake dirty pages.

Faulty predictions result in more shadow entries with write permissions, which will make the scanning process at the end of the epoch more time-consuming. Fortunately, since Histase takes effect only when 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.

4.2 Software Superpage

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

The Xen hypervisor is not aware of any peripherals. It reuses the Domain0's drivers to manage devices, including the network driver. At the end of each epoch, all the dirty pages have to be mapped into Domain0'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 Domain0's address space persistently, avoiding the *map/unmap* operations at the end of each epoch. However, the virtual address space required in Domain0 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, Domain0 is non-malicious and can be granted with read-only access to the primary VM's entire physical memory. Second, because of balloon driver [16] or memory hotplug [33], 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.

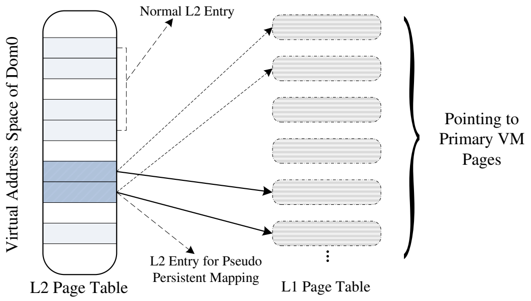

Normal L2 Entry
Virtual Address Space of Dom0
Pointing to Primary VM Pages
L2 Entry for Pseudo Persistent Mapping
L2 Page Table
L1 Page Table

Fig. 8. Software superpage. L1 page tables point to the primary VM's entire memory pages, and limited L2 page table entries are used to point to L1 page tables.

Fig 8 illustrates the design details. For brevity, we take 32-bit PAE system for example. In the initialization phase of fault tolerance, Domain0 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 Domain0 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 Domain0). At any time, among these 2,048 L1 PTs, at most 32 of them are actually installed into these L2 PTEs. Those uninstalled L1 PTs are pinned in Domain0's memory giving Xen an intuition that these pages are being used as L1 PTs. When coping a dirty page, Domain0 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 referenced on demand. In this way, we *map/unmap* memory pages as superpage mapping (mapping 2M virtual address space once) with a limited virtual address space.

In order to reduce the times of updating L2 PTEs, we employ an LRU algorithm [31] to decide which L2 PTE should be updated. When an L1 PT is accessed, its associated L2 PTE is marked as the youngest. A demanded 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 recently 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 Domain0'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 Domain0 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 [16], [34] is a prevalent approach to harness 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[4] mechanism which swapped the page containing the received packet with a free page of the VM [21].

To cope with these challenges, an event channel is employed in Domain0. If any of the primary VM's pages is changed, hypervisor sends a notification to Domain0 through the event channel. Upon notification, Domain0 updates the corresponding L1 shadow entry to point to the new page.

5 Evaluation

The optimizations presented in the previous section are implemented on Xen-3.3.1, with Domain0 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. The Domain0 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. The system performance is sensitive to the length of an epoch. Unless otherwise specified, we set an epoch 20 msec as default, and the primary VM is configured with one CPU core. In section 5.5, we will evaluate the sensitivity of the length of epoch and the number of primary VM's CPU cores.

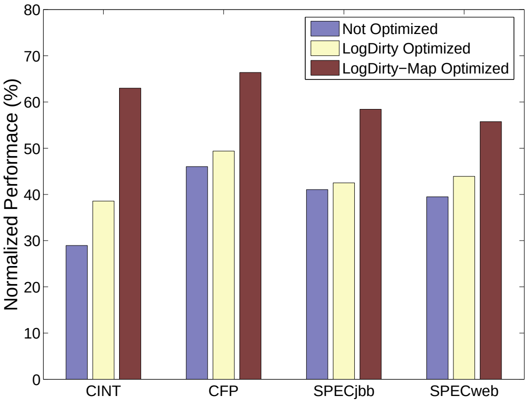


Fig. 9. The overall performance of the primary VM normalized to performance of the native VM.

5.1 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.

5.2 Overall Result

Fig 9 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) since the overhead resulting from virtualization is not the focus of this work. In practice, virtualization has been widely deployed for its key benefits, such as server consolidation and isolation.

As shown in Fig 9, *CINT* suffers the worst performance degradation when running in the unoptimized *HBFT*, yielding only about 30% of baseline performance. This is because it has a large writable

[4] Menon et al. [35] has shown that page flipping is unattractive.

TABLE 1
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.

working set which leads to high overhead by log dirty mode and memory mapping. With our optimized log dirty mode and the additional *software superpage* mechanism, the performance of *CINT* improves by 33.2% and 84.5%, respectively.

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

5.3 Log Dirty Mode Improvement

5.3.1 Experimental Setup

The log dirty mode is the mechanism to record which memory pages are dirtied. To better understand its impact on performance, we study this mechanism in isolation as follows: at the beginning of each epoch, we mark all memory pages of the primary VM read-only with a *hypercall*. Then, the primary VM resumes running for an epoch of 20msec. The other components of *HBFT* are disabled during the above procedure.

5.3.2 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 the *read fault reduction* optimization and 'WFPXen' refers to the optimized version with the additional *write fault prediction*.

Performance. Fig 10 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 substantial 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 predicting shadow page write-faults.

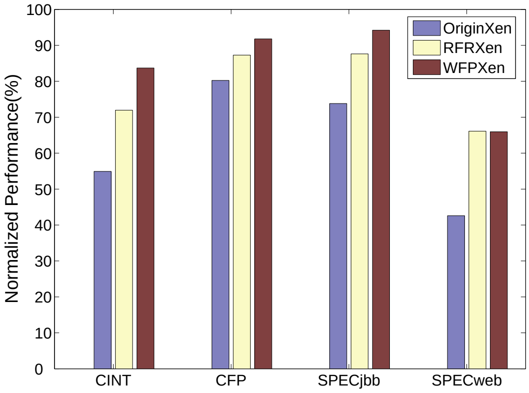


Fig. 10. The performance of memory tracking mechanism normalized to performance of the native VM.

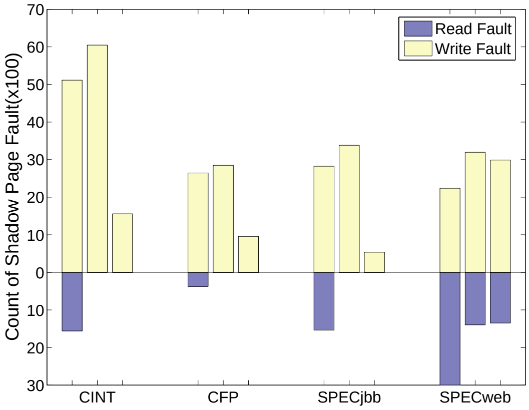


Fig. 11. Count of shadow page faults. Each bar indicates the total count of shadow page faults, and the upper half indicates the count of shadow page write-fault and the lower half indicates the count of shadow page read-fault. From left to right, OriginXen, RFRXen and WFPXen.

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 Fig 5. 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 the reduction of shadow page faults and by prediction accuracy.

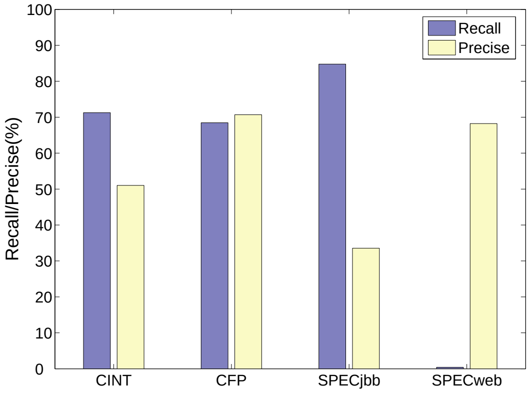


Fig. 12. Shadow page write-fault prediction

Reduction of Shadow Page Faults. To determine where our optimization techniques differ from OriginXen, Fig 11 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 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. Due to 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-existing 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.

A somewhat counter-intuitive result is that most applications running in RFRXen experience more shadow page write-faults per epoch than in OriginXen (e.g., 934 more for *CINT*). This is 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 the effectiveness of Histase, we use the terminologies from information retrieval [36]: ***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. In this experiment, recall can be seen as a measure of completeness, whereas precision can be seen as a measure of exactness.

Fig 12 shows that Histase behaves differently among the workloads. As expected in Fig 5, the applications with fine spatial locality benefit the most. 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 Fig 5. However, Histase still predicts with high *precise* since it bases its prediction on history-similar pattern, and retains application performance.

TABLE 2
Mapping hit ratio of software superpage.

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

5.4 Software Superpage Evaluation

With limited virtual address space of Domain0, *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 Domain0 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 Fig 9, *software superpage* improves the performance of the primary VM by at least 30% relative to the unoptimized *HBFT*.

5.5 Sensitivity to Parameters

5.5.1 Length of Epoch

The performance of the primary VM and the output latency is sensitive to the epoch length since the output generated in each epoch is delayed until the primary host receives the acknowledgement from the backup host [10]. Shorter checkpoint epoch means shorter output latency but higher performance degradation to the primary VM, while longer checkpoint epoch means better performance but longer output latency.

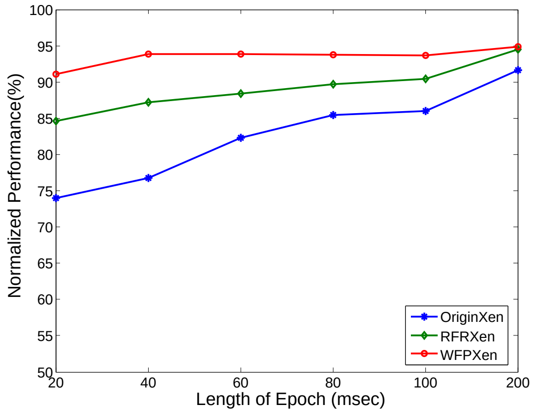


Fig. 13. Performance of memory tracking mechanism. The epoch length ranges from 20msec to 200msec.

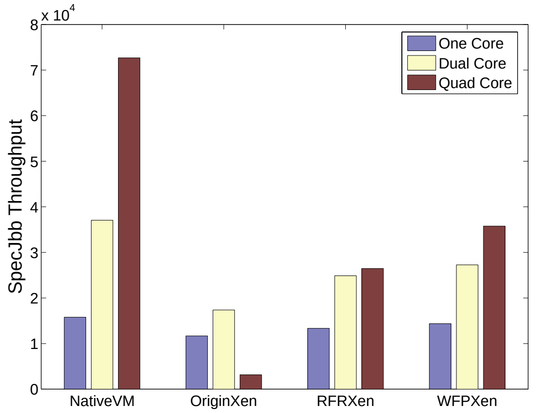


Fig. 14. The throughput of SPECjbb running with different memory tracking mechanism.

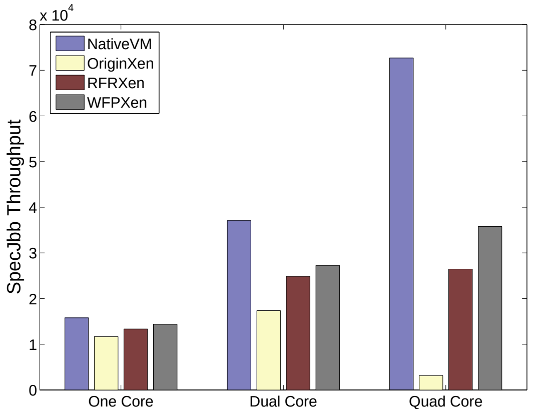


Fig. 15. The throughput of SPECjbb running with different number of CPU cores.

TABLE 3
Mapping hit ratio of software superpage.

CPU Cores/JVM instances	Mapping Hit Ratio
One	97.80%
Two	95.98%
Four	91.26%

Fig 13 shows the performance of memory tracking mechanism when the epoch length ranges from 20msec to 200msec. It presents the normalized performance of *SPECjbb* running with memory tracking enabled, relative to its performance in the native VM. We can see that our optimizations accelerates the performance of the primary VM under all configurations, but the gain decreases as the epoch length increases.

In this paper, we focus on the implementation of the memory tracking mechanism. How to select the epoch length is left to the applications. For applications with high performance requirements (e.g., HPC applications [37]), a long epoch length is appropriate. But for I/O intensive applications, such as web servers, shorter epoch is necessary to maintain the responsiveness of user requests. Additionally, for some complicated applications, it is also appropriate to choose dynamic checkpoint scheduling, such as aperiodic checkpointing [38] and cooperative checkpointing [39].

5.5.2 CPU Core Numbers

To demonstrate our optimizations in multi-core VMs, we evaluate the performance of the memory tracking mechanism and the efficiency of software superpage with *SPECjbb* suite running in the primary VM configured with different number of CPU cores. In each test, we set the number of JVM instances as the number of CPU cores of the primary VM. In this experiment, the length of epoch is set to 20msec.

Fig 14 shows the throughput of *SPECjbb* configured with different memory tracking mechanism, and Fig 15 shows the throughput of *SPECjbb* configured with different number of CPU cores. As shown in Fig 14, ”NativeVM” configuration, which does not enable memory tracking, shows the best throughput of SPECjbb running in a VM with different number of CPU cores. From Fig 15, we find that in the un-optimized(”OriginXen”) implementation of memory tracking, the relative performance descends drastically with the increase of core numbers. The throughput is surprisingly no more than 5% of the best throughput when the primary VM is configured with four cores. Our optimizations accelerates the performance of memory tracking under all configurations. Even in the four-core primary VM, our optimized memory tracking mechanism promotes the performance of *SPECjbb* to nearly 50% of the best performance.

Table 3 presents the mapping hit ratio of software superpage when the primary VM runs SPECjbb in it. With the increase of JVM instances, the working set of the primary VM becomes larger and the spatial locality of modified pages in each epoch becomes worse. The mapping hit ratio of software superpage decreases accordingly because software superpage relies heavily on spatial locality of modified pages in each epoch. Nevertheless, the mapping hit ratio remains higher than 90% for SPECjbb.

6 Related Work

Hypervisor-based fault tolerance is an emerging solution to sustain mission-critical applications. Bressoud and Schneider [40] proposed the pioneering system with the 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 [5] and Kemari [6] 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 [12]. 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 to tolerate failures [41], [42], [13]. Many researchers have been engaged in reducing checkpointing overhead. For example, incremental checkpointing [43] 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 presents 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 [44], and those that make prefetching decision on historic behavior [45]. In addition, many researchers have focused on reducing MMU walks by prefetching page table entries into TLB. *Distance prefetching* [46], 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 [47]. 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 the literature [31]. Superpage has been adopted by many modern operating systems, such as Linux [47] and FreeBSD [48]. These studies rely on hardware implementations of superpages which restrict to map physically continuous page frames. Swanson et al. [49] 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.

7 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.

Acknowledgments

The authors would like to thank the anonymous reviewers for their invaluable feedback. This work is supported by the National Grand Fundamental Research 973 Program of China under Grant No.2007CB310900, the MoE-Intel Joint Research Fund MOE-INTEL-09-06, and the SKLSDE-2010KF of State Key Laboratory of Software Development Environment.

References

[1] J. Zhu, W. Dong, Z. Jiang, X. Shi, Z. Xiao, and X. Li, "Improving the Performance of Hypervisor-Based Fault Tolerance," in *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS '10)*, 2010.

[2] N. Rafe, "Minor Outage at Facebook Monday," 2009.

[3] R. Miller, "IBM Generator Failure Causes Airline Chaos," 2009.

[4] R. Miller, "Codero Addresses Lengthy Power Outage," 2010.

[5] 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)*, 2008.

[6] Y. Tamura, K. Sato, S. Kihara, and S. Moriai, "Kemari: Virtual Machine Synchronization for Fault Tolerance," 2008.

[7] "Taiji." [Online]. Available: http://net.pku.edu.cn/vc/files/ft/index.html

[8] J. Gray, "Why Do Computers Stop and What Can Be Done About It?" in *Proceedings of the 3rd Symposium on Reliability in Distributed Software and Database System (SRDS '86)*, 1986.

[9] R. P. Goldberg, "Survey of Virtual Machine Research," *IEEE Computer*, vol. 7, no. 6, pp. 34–45, 1974.

[10] E. N. Elnozahy and W. Zwaenepoel, "Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit," *IEEE Transactions on Computers*, vol. 41, no. 5, pp. 526–531, 1992.

[11] W. Bartlett, "HP NonStop Server: Overview of an Integrated Architecture for Fault Tolerance," in *2nd Workshop on Evaluating and Architecting System Dependability*, 1999.

[12] 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)*, 2009.

[13] 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*, vol. 34, no. 3, pp. 375–408, 2002.

[14] J. Nakano, P. Montesinos, K. Gharachorloo, and J. Torrellas, "ReViveI/O: Efficient Handling of I/O in Highly-Available Rollback-Recovery Servers," in *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA '06)*, 2006.

[15] AMD, *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, 2006.

[16] C. A. Waldspurger, "Memory Resource Management in VMware ESX Server," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, p. 181, 2002.

[17] 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)*, 2006.

[18] VMware, "Protecting Mission-Critical Workloads with VMware Fault Tolerance," 2009.

[19] "Marathon." [Online]. Available: http://www.marathontechnologies.com/

[20] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen, "Execution Replay of Multiprocessor Virtual Machines," in *Proceedings of the fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '08)*, 2008.

[21] 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," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, p. 164, 2003.

[22] D. A. Patterson, G. Gibson, and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data (SIGMOD '88)*, 1988.

[23] "Synology." [Online]. Available: http://www.synology.com

[24] 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 Symposium on Networked Systems Design and Implementation (NSDI '05)*, 2005.

[25] M. Nelson, B. Lim, and G. Hutchins, "Fast Transparent Migration for Virtual Machines," in *Proceedings of the 2005 USENIX Annual Technical Conference (USENIX '05)*, 2005.

[26] 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)*, 2009.

[27] "CINT2006." [Online]. Available: http://www.spec.org/cpu2006/CINT2006/

[28] "CFP2006." [Online]. Available: http://www.spec.org/cpu2006/CFP2006/

[29] "SPECjbb2005." [Online]. Available: http://www.spec.org/jbb2005/

[30] "SPECweb2005." [Online]. Available: http://www.spec.org/web2005/

[31] A. Silberschatz, P. Galvin, and G. Gagne, *Operating System Principles*. Wiley India Pvt. Ltd., 2006.

[32] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2009.

[33] D. Hansen, M. Kravetz, B. Christiansen, and M. Tolentino, "Hotplug Memory and the Linux VM," in *Proceedings of the 2004 Ottawa Linux Symposium*, 2004.

[34] 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 *Proceedings of the 8th USENIX Symposium on Operating System Design and Implementation (OSDI '08)*, 2008.

[35] A. Menon, A. L. Cox, and W. Zwaenepoel, "Optimizing Network Virtualization in Xen," in *Proceedings of the 2006 USENIX Annual Technical Conference (USENIX '06)*, 2006.

[36] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Addison-Wesley Reading, MA, 1999.

[37] M. F. Mergen, V. Uhlig, O. Krieger, and J. Xenidis, "Virtualization for High-Performance Computing," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 2, p. 8, 2006.

[38] Y. Ling, J. Mi, and X. Lin, "A Variational Calculus Approach to Optimal Checkpoint Placement," *IEEE Transactions on Computers*, vol. 50, no. 7, pp. 699–708, 2001.

[39] A. J. Oliner, L. Rudolph, and R. K. Sahoo, "Cooperative Checkpointing: A Robust Approach to Large-Scale Systems Reliability," in *Proceedings of the 20th Annual International Conference on Supercomputing (ICS '06)*, 2006.

[40] T. C. Bressoud and F. B. Schneider, "Hypervisor-Based Fault Tolerance," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, 1995.

[41] C. Studies, D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kcman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft, "Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies," pp. 1–16, 2002.

[42] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman, "The LAM/MPI checkpoint/restart framework: System-initiated checkpointing," *International Journal of High Performance Computing Applications*, vol. 19, no. 4, pp. 479–493, 2005.

[43] 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)*, 2008.

[44] 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.

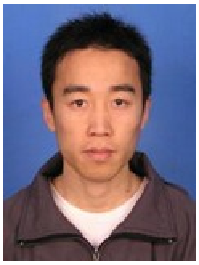
[45] D. Joseph and D. Grunwald, "Prefetching using Markov predictors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA '97)*, 1997.

[46] G. B. Kandiraju and A. Sivasubramaniam, "Going the Distance for TLB Prefetching: an Application-Driven Study," in *Proceedings 29th Annual International Symposium on Computer Architecture (ISCA '02)*, 2002.

[47] D. Bovet and M. Cesati, *Understanding the Linux Kernel*. O'Reilly Media, Inc, 2005.

[48] J. Navarro, S. Iyer, P. Druschel, and A. Cox, "Practical, Transparent Operating System Support for Superpages," in *Proceedings of the 5th USENIX Symposium on Operating System Design and Implementation (OSDI '02)*, 2002.

[49] M. Swanson, L. Stoller, and J. Carter, "Increasing TLB Reach Using Superpages Backed by Shadow Memory," in *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA '98)*, 1998.



Jun Zhu is a PhD student in the School of Electronics Engineering & Computer Science, Peking University. He received his bachelor degree in the School of Computer Science at Beijing University of Aeronautics and Astronautics in 2006. His research interests include distributed system, fault tolerance, operating system and virtualization. His recent research aims to provide fast recovery from hardware failures and software failures for mission-critical services.

Zhefu Jiang is a graduate student studying at Computer Network and distributed System Laboratory, Peking University. He received his bachelor degree from HuaZhong University of Science and Technology, majoring in Computer Science. During his student life, he focus on exploring system level developing techniques, especially OS and Virtualization development. He is also a student who shares a great interest in Open-Source techniques such as Xen, Linux and other GNU projects.

Zhen Xiao is a Professor in the Department of Computer Science at Peking University. He received his Ph.D. from Cornell University in January 2001. After that he worked as a senior technical staff member at AT&T Labs - New Jersey and then a Research Staff Member at IBM T. J. Watson Research Center. His research interests include cloud computing, virtualization, and various distributed systems issues. He is a senior member of IEEE.

Xiaoming Li , a professor of Peking University, received his Ph.D. in Computer Science from Stevens Institute of Technology (USA) in 1986 and has since taught at Harbin Institute of Technology and Peking University. He has founded the Chinese web archive WebInfoMall (http://www.infomall.cn), the search engine Tianwang (http://e.pku.edu.cn), the peer-to-peer file sharing network Maze (http://maze.pku.edu.cn), and other popular web channels. He is a member of Eta Kappa Nu, a senior member of IEEE, currently a Vice President of Chinese Computer Federation, International Editor of Concurrency (USA), and Associate Editor of Journal of Web Engineering (Australia). He has published over 100 papers, authored Search Engine Principle, Technology, and Systems (Science Press, 2005), and received numerous achievement awards from the Ministry of Science and Technology, Beijing Municipal Government, and other agencies.