Efficient Checkpointing of Live Virtual Machines

Bernhard Egger, Eunbyung Park, Younghyun Jo, Changyeon Jo, and Jaejin Lee
Department of Computer Science and Engineering
Seoul National University, Seoul, Korea
{bernhard,eunbyung,younghyun,changyeon}@csap.snu.ac.kr, jaejin@snu.ac.kr

Abstract—The ability to save the state of a running virtual machine (VM) for later restoration is an important tool for home, server, and virtual desktop cloud (VDC) environments in order to achieve optimal and balanced hardware utilization. With guest memory sizes of four to eight gigabytes being the norm the time- and space-overhead of storing VM checkpoints still prevents an effective use of the technique. This work presents a method for fast and space-efficient checkpointing of VMs. Based on the observation that operating systems cache disk blocks in memory, the proposed technique transparently intercepts I/O operations and maintains an up-to-date mapping of memory pages and disk blocks containing identical data. At a checkpoint, those memory pages are excluded from the checkpoint image leading to a significant reduction of both the time and space required to take a checkpoint of a running VM.
The broad applicability and good performance of the proposed method is demonstrated by an extensive set of experiments. We have implemented the technique for para-virtualized (PV), PVHVM, and fully-virtualized (HVM) guests in the Xen hypervisor. In comparison with an unmodified Xen hypervisor, experiments with Linux and Windows guests, on average, achieve a 86%, 76%, 53%, and 47% reduction in the stored data and a 73%, 62%, 47%, and 38% shorter time required to take a checkpoint for PV, PVHVM, HVM Linux, and HVM Windows guests, respectively.

1 Introduction

Virtualization technology continues to gain importance in server, desktop, as well as embedded systems. Its use is especially prevalent in data centers and virtual desktop cloud (VDC) environments where virtualization technology is employed to run multiple instances of operating systems (OS) concurrently on one single physical host. This allows for custom-tailored environments, better use of physical resources, higher availability through live migration and better security [1]–[8].

A useful feature of virtualization technology is virtual machine (VM) checkpointing, the ability to save and restore the state of a virtual machine. VM checkpoints (also called *snapshots*) are convenient in many situations, be it for simply stopping a VM and later resuming work on a desktop computer, for migrating a VM to another host in a data center, or for performing intrusion analysis of commodity operating systems by continuously taking snapshots of the system under attack. In VDC environments, checkpointing is used to free resources while a user is not logged in. The state of the user' VM is saved to external storage when the user terminates the connection to his virtual desktop; and the VM is restored from the checkpoint as soon as the user re-establishes the link to his desktop.

To completely record the state of a virtual machine, the virtual machine monitor (VMM) is required to store the virtual CPU state, the current state of all emulated devices, and the contents of the virtual machine's memory to non-volatile memory. The memory of the VM is stored to disk in its entirety, hence the size of the snapshot is dominated by the amount of the guest's memory. With rapidly growing memory sizes for virtual machines, the size of snapshots thus becomes more and more of a concern, especially so in VDC environments. Assuming a VDC with 1000 user desktops with 8 GB of memory each and an average of 20% of users not being connected to their desktop, 200 times 8 GB = 1.6 TB are necessary exclusively to store the volatile state of inactive VMs. Furthermore, saving/restoring a single checkpoint to/from disk at a speed of 200MB per second requires at least 8 GB / 200MB/s = 40 seconds.

In order for checkpoint-restore technology to be applied effectively in VDC environments, both the size of the checkpoints as well as the time required to save and restore the checkpoints must be reduced. Several techniques have been proposed to exclude memory not currently in use [4] or to compress the checkpoint data [9]. The former technique, however, is no applicable to fully-virtualized systems and the overhead of (de-) compressing memory data typically increases the time required to save/restore a checkpoint.

This paper proposes a fast and space-efficient method to save and restore checkpoints of a VM. The proposed technique not only reduces the size of the checkpoint images considerably, but also the time required to save checkpoints is significantly shorter than existing methods.

The proposed method is based on the observation that it is not necessary for the VMM to store the entire contents of the VM's memory because a significant portion of the data in memory is already available on external storage. This is caused by modern operating systems using the better part of the unused memory as a cache commonly known as the *page cache* [10]. This page cache contains data that has been recently read from (or is to

be written to) block devices in order to reduce the large access latency of external devices.

In a VM environment, all I/O operations to devices pass through the VMM. In the proposed technique, the VMM transparently intercepts I/O operations to block devices and maintains data structures that map the blocks of non-volatile storage (a virtual disk or a partition) to the VM's memory pages. At a checkpoint, memory pages that are known to be identical copies of disk blocks are excluded from the VM memory image.

We have implemented the proposed technique in the most recent Xen hypervisor 4.5 [11] for para-virtualized (PV), fully-virtualized (HVM), and PVHVM guests. Experiments with an extensive set of benchmarks on Linux and Windows guests demonstrate the broad applicability and good performance of the proposed method.

The contributions of this paper are as follows:

- We introduce a fast and space-efficient checkpointing technique for virtual machines. By excluding memory pages whose contents are available on non-volatile storage from the checkpoint image both the space as well as the time required to take a checkpoint are significantly reduced.
- We have implemented the proposed technique in the Xen 4.5 hypervisor. We support para- and fully-virtualized guests and have conducted the experiments with Microsoft Windows and Linux guest operating systems. We identify and correctly handle all scenarios that could lead to a corruption of the VMM's disk-to-memory mapping data structures and thus might result in a corrupted memory image.
- We compare the performance of the unmodified Xen VMM with three virtualization levels PV, PVHVM, and HVM, both for individual checkpoints as well as incremental checkpoints. Compared to unmodified Xen, we achieve an average reduction in the required disk space ranging from 47% (HVM - Windows guest) to 86% (PV - Linux guest) and a reduction in the checkpointing time from 38% (HVM - Windows guest) to 73% (PV - Linux guest). The proposed method also outperforms incremental checkpointing by a 57% reduction in disk space and a 26% reduction in time to take a checkpoint.
- The optimized restore procedure from space-optimized checkpoint images incurs an acceptable overhead or even outperforms unmodified Xen.

The rest of this work is organized as follows: Section 2 discusses related work. Section 3 provides some background on virtualization and VM checkpointing in particular. Section 4 motivates the proposed technique, and Sections 5 and 6 discuss the implementation of the proposed technique for the different virtualization techniques. The experimental setup is described in Section 8, and Section 9 presents the results. Finally, Section 10 concludes the paper.

2 Related Work

While the available CPUs and I/O bandwidth on a physical machine can be fairly time-shared or multiplexed, allocating or reclaiming memory dynamically is not easy due to the fact that the memory has an inherent weakness as a shared resource. Reclaiming memory from a running VM without seriously affecting its performance is not easy as the VMM has limited knowledge of the memory contents and thus the importance of a specific memory page to the VM. A common technique to reclaim memory is Ballooning [4], in which the VMM communicates with a driver running inside the VM. This ballooning driver requests non-shared memory from the guest operating system. Since this memory is not used to hold any meaningful data, the VMM can reclaim the associated memory pages and assign them to another VM. Most major VMMs, such as KVM [12], VMware [13], VirtualBox [9], and Xen [14], make use of the ballooning technique. Ballooning can also be used in conjunction with space-efficient checkpoints by excluding the memory allocated to the balloon driver. Allocating a lot of memory to the balloon driver, however, reduces the amount of memory available for the page cache and thus will lead to slightly decreased performance of the VM after restoration. Our technique, on the other hand, directly identifies duplicated pages and does not affect the performance of the VM after restoration.

Transcendent memory [15], [16] is another approach to efficient memory management between multiple VMs, and in terms of dealing with free memory it resembles our approach. It maintains a system-wide global memory pool that contains memory not allocated to any VM. Para-virtualized guests can access that memory indirectly through an API. Several approaches have been proposed such as efficient memory provisioning [17], memory sharing between multiple VMs [18], [19], or hot spot migration [20]. Transcendent memory cannot be used to reduce the space or time of checkpoints.

We track a VM's I/O operations to infer a mapping between the external storage and the pagecache of the guest in order maintain information about duplicated pages in the VM's memory and external storage. The works of Jones [21] and Lu [22] use a similar approach. Both approaches focus on the working set size (WSS) estimation of a VM in order to allot the exact amount of memory needed by the VM. The page miss ratio curve [23] used to estimate the WSS is based on the reuse distance [21]. The reuse distance is tracked by detecting promotion and eviction of pages to the page cache. While external requests from or to secondary storage can be traced transparently in the VMM, the eviction of a page from the page cache is hidden from the VMM. Geiger uses heuristics to detect that pages have been evicted from the page cache, whereas in the VM memory access tracking approach the guest is modified to notify the VMM of any page evictions from the page cache.

Fast resume from a VM checkpoint image is equally important to give the best experience to users. Zhang [24] proposed fast restoration of VM using working set estimation. In their work, a VM is resumed early after only restoring the workingset of the checkpointed image, the remaining contents of the image are fetched in the back-

ground while the VM is already running. This technique focuses only on reducing the restoration time and does not address reducing the size of the checkpoint image. Egger *et al.* [25] have recently proposed a method to reduce the restoration time further, below that of original Xen. Their idea is a combination of the method presented here and Zhang's technique of early VM re-initialization.

The approach proposed in this paper is independent from other approaches that try to reduce the memory footprint of a VM or its snapshot. VirtualBox [9] compresses the checkpoint image; our technique can be combined with data compression to achieve even smaller checkpoint images. In comparison to previous work [26], this work extends support for PVHVM guests and the hardware-assisted virtualized memory translation capabilities of modern CPUs. While previous work suffered from a 3- to 4-fold increase in the restoration time this work includes an optimization that performs on par with unmodified Xen.

Applications benefiting of efficient VM checkpoints are not limited to VDC environments. OS debugging or intrusion analysis by means of a VM also profit from fast and space-efficient checkpointing [7], [27]–[29].

3 Background

3.1 Virtual Machine Monitors

Virtualization allows the execution of operating systems inside a virtualized hardware environment. This virtualized environment is provided by a *virtual machine monitor* (VMM) or *hypervisor*. VMMs support para-virtualized, fully-virtualized, or both environments. In a para-virtualized environment, the VMM exposes certain APIs to the VM. The VM communicates with the VMM through this *para-API*. The weakness of this approach is that the guest operating system needs to be modified in order to take advantage of the para-API. Nevertheless, para-virtualization is a pervasive virtualization technology supported by most major VMMs due to its benefits such as improved I/O performance [30], [31], better memory management [17]–[19], or load balancing [20]. Full virtualization, on the other hand, provides a complete emulation of the underlying hardware and does not require guest modification. This approach is more difficult to implement, and emulating certain functionality may lead to reduced performance in the guest. It is still the prevalent emulation method because guest operating systems that cannot be modified easily (such as Microsoft Windows) can be supported as well.

Recent versions of the open source Xen Hypervisor [14] offer para-virtualized, full virtualization, and the so-called PVHVM virtualization. With PVHVM, a full-virtualized guest uses specialized para-virtualized drivers to access hardware and can thus achieve nearly the same performance as PV guests.

3.2 VM Checkpointing

VM checkpointing refers to the process of saving the state of a virtual machine to non-volatile storage so that it can be completely stopped and restored in its exact state at a later time. Checkpoints can be taken of running as well as stopped VMs. A snapshot of a running VM includes the VM's memory contents and the state of its devices including the CPU(s). A checkpoint of a stopped VM is usually limited to creating a snapshot of the current state of its external storage. In this work, we focus on improving the process of checkpointing a running system.

A snapshot of a running system comprises the context of the virtual CPUs (VCPU), virtual devices connected to the VM (such as network adapters, keyboard, and the display), as well as a copy of the memory allocated to the VM. Typically, it is assumed that the external storage connected to the guest (i.e., its virtual disk) is not modified until the guest is restored. This is a reasonable assumption since modifying the contents of the disk while the guest is not running will most likely make it impossible to resume the guest. Modern operating systems often cache disk blocks in memory, and modifying the disk behind the guest's back would lead to inconsistencies in that cache. The following paragraphs describe the differences in context of checkpointing for the different virtualization technologies.

Para-virtualized environments. When the VMM checkpoints a VM, the para-virtualized guest is first notified through the para-API. The guest itself carries out several tasks including disconnecting devices, disabling interrupts, and marking page tables. After finishing the preparatory work, the guest transfers control to the VMM. The VMM uses the privileged domain *dom0* to save the state of the VM to persistent storage. In addition to the contents of the VM's volatile memory, the state of the virtual devices owned by the guest, and context of the VCPUs are stored as well. Restoring a guest running in a VM is the exact opposite process. The domain *dom0* first creates a new user domain, allocates the necessary amount of memory, assigns and restores the context of the *VCPUs* and the virtual devices. Control is then transferred to the guest which reconnects the devices to its event queue, re-enables interrupts and then continues executing where it left off before the checkpoint. Since the guest is involved in the checkpoint/restore process, it is possible to exclude unallocated memory pages from the checkpoint image by modifying the code of the guest OS.

Fully-virtualized environments. In fully-virtualized environments, the guest is not aware that it is being checkpointed; this, for example, hinders the exclusion of free memory pages from the checkpoint image. The VMM stops the guest, and saves the state of all virtual devices owned by the guest as well as the contents of the VM's volatile memory to disk. The guests page tables can be saved as-is since they do not contain physical machine addresses (the VMM uses *shadow page tables* or *hardware assisted paging* to fully virtualize memory accesses, see Section 6). Restoring a fully-virtualized guest is similar to a para-virtualized one. In a fully-virtualized environment, however, the VMM has to re-

TABLE 1
Amount of duplication between memory and disk after large I/O operations

	Total Memory Size	
	512 MB	1024 MB
Linux	466 MB (91%)	966 MB (93%)
Windows	353 MB (84%)	860 MB (84%)

initialize the devices itself since the guest expects the devices to operate as if it had never been checkpointed.

PVHVM environments. PVHVM guests are checkpointed and restored in a similar way as fully-virtualized guests, only the device drivers follow the para-virtualized approach. In the current implementation in Xen, the guest is aware that it is being checkpointed and we can thus exclude free memory pages from the checkpoint image.

4 Space-Efficient Checkpointing

To achieve space as well as time efficient checkpointing, the proposed method eliminates duplicates of pages in the checkpoint file that are also available on the external storage of the VM. The key idea is to detect promotion of memory pages into the guest's page cache by transparently intercepting I/O requests to external storage by the guest.

4.1 Motivation

Even for virtual machines, these days memory sizes of several gigabytes are the norm. The operating system and the running applications often occupy only a small fraction of the total available memory, leaving most of the memory unused. Modern operating systems use this unused memory to cache recently read data as well as data yet to be written to disk thereby hiding the long access latency to external storage. As demonstrated in Table 1, depending on the usage scenario and the size of the memory, the amount of memory allocated to this *page cache* can exceed 90% of the physically available memory if the guest VM processes large amount of I/O requests [26]. The data in Table 1 was generated by saving the guest's memory after booting the system up and copying a large amount of data. We then compared the memory contents to the data on disk. While the experimental setup may not be an overly realistic usage scenario, it shows the amount of duplication between memory and disk in an extreme case.

Up to this day, VMMs typically store a one-to-one image of the guest's memory in the snapshot. Table 2 lists the size of the VM snapshot (memory image plus device state) for major VMMs such as VirtualBox [9], VMware [13], and Xen [14]. The snapshots for VMware and Xen are both a bit larger than the VM's memory size due to the inclusion of the volatile state for its VCPUs and devices. VirtualBox also stores the entire contents of its memory, but compresses the data. The presented technique here is orthogonal to data compression and

TABLE 2
Size of snapshots of a VM with 4 GB of memory

VMM	Snapshot Size (MB)	Remarks
VirtualBox	1542	compressed
VMware	4234	
Xen	4100	

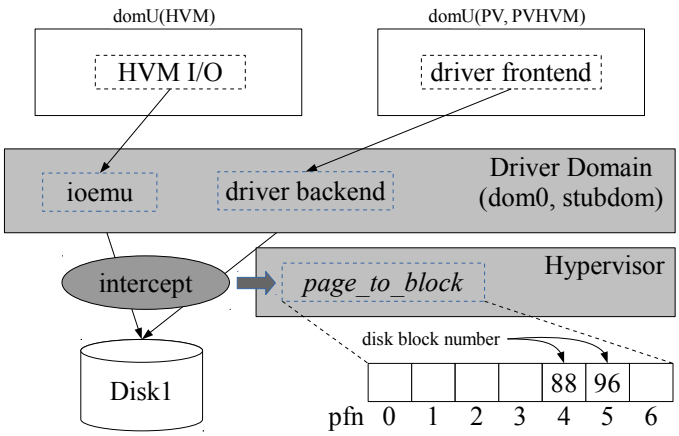


Fig. 1. Disk I/O flow in the Xen VMM and the page-to-block map

can be included in VirtualBox to achieve yet smaller checkpoint images.

The results from Table 1 and 2 suggest that there exists a big potential to reduce the size of VM snapshots. Instead of storing the duplicated data in the VM snapshot itself, a reference to the external storage medium suffices. Such an approach not only reduces the size, but also shortens the time required to save a snapshot because the time required to store a snapshot is dominated by the amount of data to be saved to external storage. The proposed method also affects restoration. Instead of being able to read the memory contents sequentially from disk, the contents need to be composed from the snapshot image and data spread over the VM's external storage. Without further precautions this may lead to an increase in restoration time, however, we show that a simple optimization is able to reduce the restoration times below that of unmodified Xen for PV and PVHVM guests and a minimal increase for HVM guests.

There exist other opportunities to further reduce the size of snapshots, such as excluding free memory pages and memory that is not mapped into the guest's address space from the snapshot image. The former can only be exploited in para-virtualized and PVHVM environments since the VMM itself has no knowledge about which mapped memory pages the guest is currently using and which represent free pages. The latter can be excluded from the snapshot both in para-virtualized and fully-virtualized environments by inspecting the guest's page tables and checking which virtual pages are not mapped.

4.2 Transparent I/O Interception

In virtualized environments, the VMM virtualizes some hardware resources in order to make them visible to

guests running in VMs as if they were dedicated. For some types of hardware resources such as network interface cards (NIC) or disks, concurrent accesses to these hardware resources by multiple VMs need to be arbitrated by the VMM. While accesses to physical memory are implicitly multiplexed by the memory controller and the hardware bus arbitration, simultaneous requests to the physical disk or NIC without meditation of software may cause unknown behavior or even device crashes. For this reason, I/O operations to secondary storage without hardware virtualization support are always under control of the VMM or a driver in a privileged domain. At those points, it is thus possible to intercept all I/O requests transparently.

Figure 1 shows the flow of I/O requests to disk in the Xen hypervisor for para-virtualized, PVHVM and fully-virtualized guests. An I/O request from a para-virtualized or PVHVM guest running in the user domain *domU* (Figure 1 on the right) is forwarded directly to the VMM's driver domain (*dom0, stubdom*). I/O requests to the physical disk can be intercepted in the driver backend in the VMM's driver domain for the disk. For fully-virtualized guests (Figure 1 on the left), I/O requests go through and can be intercepted in *ioemu* located in the VMM's driver domain

Mapping page frames to disk blocks. The mapping of a VM's memory contents to data on disk is maintained by transparently tracking all I/O requests from the guest to external storage. The gathered information is stored in a so-called $page\_to\_block$ map which is maintained separately for every running VM. Since the number of memory pages is typically much smaller than the number of disk blocks, the map is indexed by the guest's memory page index, the so-called *page frame number* or *PFN*. The data stored in the map is the 8-byte disk block number. The $page\_to\_block$ map is updated whenever the guest issues an I/O request. Both the space and runtime overhead of maintaining the $page\_to\_block$ map are relatively small: for a VM with four gigabytes of memory, the $page\_to\_block$ map requires eight megabytes of memory. I/O operations are costly operations by themselves; in most cases the table update are hidden by the I/O latency (See section 9).

Figure 1 shows the contents of the $page\_to\_block$ map after the guest has read the disk blocks number 88 and 96 into its memory pages at index 4 and 5, respectively.

Modifications to pages in the page cache through memory write operations are hidden from the VMM and thus render the mapping in the $page\_to\_block$ map invalid. In order to detect in-memory modifications to such pages, we either have to validate at checkpoint time that the memory pages have not been modified or intercept all write operations to pages known to be duplicated on disk. The former approach is taken with para-virtualized and PVHVM guests, the latter with HVM guests. The following sections describe the implementation for the different virtualization technologies in detail.

4.3 Restoring from Space-Optimized Checkpoints

To restore a VM from an unoptimized checkpoint image, the VMM first creates the guest VM's domain with the necessary VCPUs and devices and then loads the contents of the entire checkpoint image into the guest VM's memory. Unless the checkpoint image is severely fragmented on disk, this operation will run at almost optimal speed. Restoration of a VM from a space-optimized checkpoint proceeds in the same order, however, the contents of memory pages known to be duplicated on disk are scattered all over the VM's disk. In a naïve implementation, the VMM processes the entries on the $page\_to\_block$ map one by one which leads to an excessive overhead on rotating magnetic disks caused by high number of read head repositionings. By sorting the entries in the $page\_to\_block$ map by the *disk block number* most of this overhead can be avoided and, compared to unmodified Xen, the restoration process takes less (for PV and PVHVM guests) or only marginally longer (for HVM guests). With the advent of solid-state disk drives (SSDs) that do not exhibit a synch overhead, we expect at least equal performance also for fully-virtualized guests. The experimental results in Section 9 discuss the results along with the overhead caused by sorting the $page\_to\_block$ map.

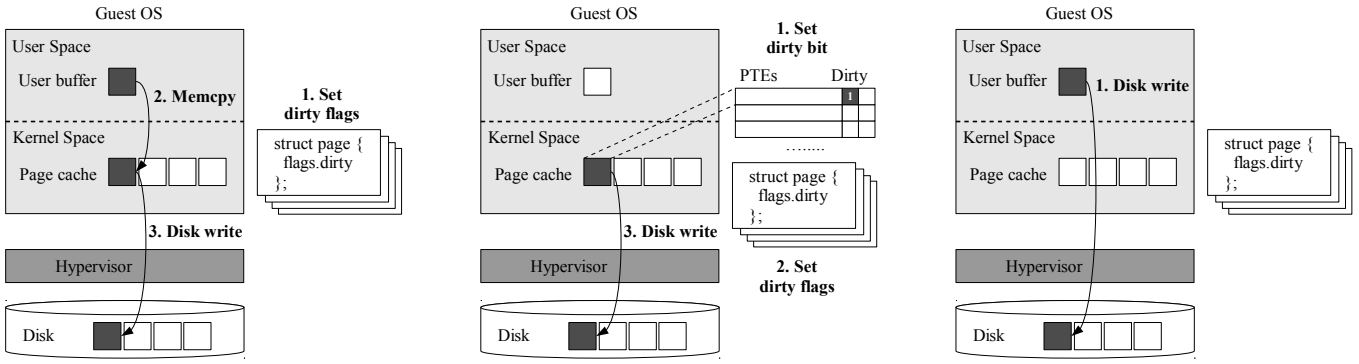
5 Para-virtualized and PVHVM Guests

For PV and PVHVM guests, there exist two types of pages that do not need to be included in a snapshot: pages that contain data that is duplicated on disk and pages that are free or not in use by the VM.

5.1 The Page Cache

The operating system maintains consistency between the page cache and the disk by periodically flushing dirty pages (i.e., pages whose contents have been modified) to disk. At the moment of the checkpoint, however, dirty pages may be present in the page cache. There are four distinct operations that can render a page in the page cache dirty: write system calls, memory mapped I/O, direct I/O, and I/O at a sub-page granularity. The following paragraphs describe each of these operations in detail and show how to identify dirty pages so that the VMM can include them in the memory image of the checkpoint.

The *write* system call. When a process issues a write system call, the file descriptor, the user buffer and the size of the write operation are passed as parameters. The file descriptor points to a specific file, and the user buffer is normally located in the user heap or global buffer memory as shown in Figure 2 (a). If a page cache hit occurs the kernel marks the affected pages dirty in its internal data structures and then performs a simple memory copy from the user buffer to the pages in the page cache. If the pages are not present in the page cache the kernel first allocates a sufficient number of pages in the page cache, marks them dirty, and then performs the memory copy operation. The interpretation of a page's



(a) Write system call (b) Memory-mapped I/O (c) Direct I/O

Fig. 2. Write Operations : write system call, memory-mapped I/O, and direct I/O.

dirty flag is thus that the contents of the page differ from the data on disk. Unless the VMM contains guest-specific code, modifications to internal data structures of the guest are not possible. The VMM is thus not aware dirty pages in the page cache.

PV/PVHVM guests perform preparatory work prior to being checkpointed. We insert an additional phase that scans the dirty bit of the kernel's page cache data structures and returns that information to the VMM such that it can exclude dirty pages from the *page_to_block* map. An alternative would be to write all dirty pages in the page cache to external storage before performing the snapshot. We chose not to follow this approach as it would prevent the user from taking precise snapshot at very specific moments.

Memory mapped I/O. The pages involved in memory mapped I/O are part of the guest kernel's page cache and the user is allowed read/write access to these pages (Figure 2 (b)). Since memory write operations go undetected by the VMM, the guest kernel itself is not immediately aware of modifications to pages used for memory mapped I/O. To detect write operations to memory mapped pages in the page cache, the kernel periodically scans the dirty bit of the corresponding page table entries (PTE) and sets the *dirty* flag in the page cache data structures accordingly.

When a guest is checkpointed it is possible that some dirty bits in the PTEs have not yet been propagated to the corresponding dirty flags in the page cache. The VMM is aware which memory pages contain the guest's page tables and scans the user address space for *dirty* bits. The corresponding entries in the *page_to_block* map are then removed.

Note that the kernel often maps memory pages into its own address space. The Linux kernel, for example, maps the entire kernel address space with the *dirty* bit set. We therefore do not consider the PTEs of kernel memory as candidates for PTE scanning. Since most modern operating systems including Microsoft Windows and Linux use separate address spaces for user and kernel processes, it is not difficult to distinguish the PTEs of kernel memory pages.

Direct I/O. Direct I/O refers to file system I/O operations that bypass the OS-level page cache (Figure 2 (c)). When a self-caching application opens a file for unbuffered I/O, pages that have already been cached for that file are dropped from the page cache. The user process is responsible to provide buffers for direct I/O. These buffers are located in the user's address space, and the application is responsible for maintaining consistency between the data in memory and on disk.

The proposed method transparently traces all I/O operations, hence the memory pages used as buffers in direct I/O are also recorded in the *page_to_block* map. However, since these user mode buffers are not part of the page cache we conservatively assume that all pages used for direct I/O are dirty and include them in the checkpoint image.

Sub-page I/O granularity. In many modern operating systems including Linux, the majority of disk I/O operations are performed at memory page granularity (typically 4 KB) as opposed to the smaller granularity of disk blocks. There are several reasons for this, the most compelling being the ever increasing trend towards page-based file systems, more efficient manageability of the memory, and compatibility for other kernel components, such as virtual memory management. However, depending on the type of the file system or the underlying block device, finer-grained block-based I/O operations (typically 512 or 1024 bytes) may be necessary to handle file system metadata (e.g., superblocks, meta data, journal data, and so on). To cache I/O operations performed at block granularity, a page is logically divided into several subpages that are managed separately. We do not track this type of I/O operations because the number of such blocks is rather low and thus does not justify the considerable overhead of handling them properly. The page split into sub-pages are identified during the preparatory work and included in the checkpoint image.

5.2 Free Pages

Pages that are not in use by the guest at checkpoint time do not need to be saved. Ignoring free pages does not hamper correctness because no assumptions about the

contents of free pages are made. We identify free pages as part of the preparatory work and exclude them from the checkpoint image.

5.3 Tracking I/O vs. Processing Data in the Guest

For PV/PVHVM guests transparent tracking of I/O is not absolutely necessary. Instead, the guest could provide the necessary data about what pages contain data duplicated on external storage to the VMM at checkpoint time. In the current implementation of the proposed technique, we chose to track I/O operations and update the mapping at checkpoint time as described above in order to keep the modifications of the guest to a minimum.

6 Implementation for Fully-virtualized Guests

The major difference between PV/PVHVM and HVM environments is that in the latter the VMM cannot obtain information about the contents of memory pages from the guest. For example, allocated but unused memory pages cannot be excluded from HVM guest snapshots since the memory of a VM is a black box to the VMM. Instead, the VMM itself must track all I/O operations to infer duplication between the VM's memory and the external storage and maintain a consistent $page\_to\_block$ map.

6.1 Reverse Mapping and Unaligned I/O

Unlike in paravirtualized environments, a reverse mapping from disk blocks to memory pages is required with full virtualization. The following sequence of I/O requests illustrates the need for a reverse map (pfn denotes the index to a 4 KB memory page, the $blocknum$ points to a 512-byte disk block):

1) READ [pfn:1, block num:16, size:4K]
2) READ [pfn:3, block num:16, size:4K]
3) WRITE [pfn:2, block num:16, size:4K]

After the guest OS has issued the two read operations, the $page\_to\_block$ map contains two references to block $\#16$ at indices 1 and 3. The subsequent write operation to block $\#16$ must first invalidate the entries at positions 1 and 3 and then add a new reference to block $\#16$ at index 2 of the $page\_to\_block$ map. To implement this invalidation efficiently, the VMM maintains a hash map providing a reverse mapping from disk blocks to memory pages. The disk block number is used as the key, and the data is a linked list containing the indices of all associated memory pages.

This reverse mapping is also used to keep track of I/O requests that are not aligned at page granularity:

1) READ [pfn:1, block num:16, size:4K]
2) READ [pfn:2, block num:24, size:4K]
3) WRITE [pfn:3, block num:20, size:4K]

After the two read operations have been issued, the $page\_to\_block$ map contains two references to block $\#16$ and $\#24$ at indices 1 and 2, respectively. A subsequent write operation to block $\#20$ must invalidate both entries since it spans the two entries in the $page\_to\_block$ map. In order to correctly detect and invalidate such entries, the offset to the next lower 4-KB boundary is also recorded in the reverse $page\_to\_block$ map.

6.2 Maintaining Consistency

To detect in-memory modifications to pages that are currently tracked by the $page\_to\_block$ map, the VMM marks all such pages *read-only* in the physical memory management unit (MMU). There are two supported techniques to provide the illusion of access to physical page tables to HVM guests: shadow page tables and hardware-assisted paging (HAP). In the following, we describe how to handle both of these cases.

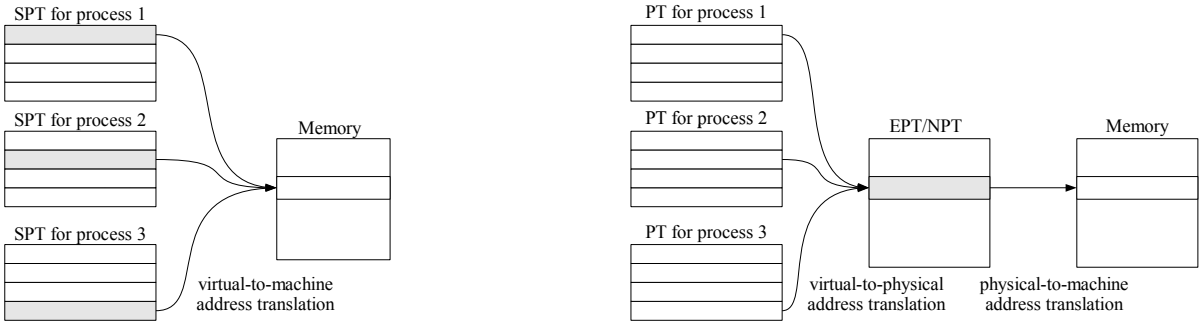
Shadow page tables. On older processors that do not support HAP, the fully-virtualized guest's physical page tables are monitored and mirrored by the VMM. The (virtualized) page table base register managed by the guest allows the VMM to traverse the guest's page table structure and create a *shadow copy* of each page. The page available to the guest contains pseudo-physical addresses (PFNs) which are then translated by the VMM to machine-physical addresses (MFNs) in the shadow page tables.

To detect modifications to its page table structure, the VMM marks all memory pages containing page tables of the guest *read-only*. A subsequent write operation to such a page will trigger a page fault which is intercepted and allows the VMM to update the corresponding shadow page table. There are a number of subtleties associated with shadow page tables; these are, however, not important for our work and outside the scope of this paper.

To intercept write accesses to pages tracked in the $page\_to\_block$ map, we mark all entries in the shadow page pointing to such pages *read-only*. This happens whenever the $page\_to\_block$ map gets updated. Similarly, whenever the guest OS maps a page into its address space, we check if it is present in the $page\_to\_block$ map and, if so, mark it read-only. As soon as the guest tries to modify to a write-protected page, a page fault exception is raised. The VMM detects that the guest is about to modify a page in the $page\_to\_block$ map, deletes the relevant entry from the map, re-maps the page as read-write, and restarts the aborted write operation.

Hardware Assisted Paging (HAP). Modern processors provide hardware-assisted virtualized memory translation. HAP is termed extended page tables (EPT) in Intel CPUs and nested page tables (NPT) in AMD processors [32]. HAP supports a global mapping of physical-to-machine pages per domain (i.e., a guest VM), we can thus simply flip the *read-only* bit in the physical-to-machine page table and do not need to track all entries in shadowed page tables pointing to a physical memory page. The page fault caused by a write operation to a protected page is intercepted by the VMM, and the corresponding entry deleted from the $page\_to\_block$ map.

Figure 3 shows the difference between shadow page



(a) Shadow page tables (b) Hardware-assisted paging (EPT/NPT)

Fig. 3. Virtual-to-physical memory address translation with shadow page tables and hardware-assisted paging.

tables and HAP support. Shaded parts indicate the entries that must be marked *read-only*.

6.3 Implementation on Other Virtualization Platforms

The implementation details for para- and fully-virtualized guests presented here are specific to Xen, but in principle applicable to any virtualization platform that keeps some form of control over the VMs' memory and is involved in I/O multiplexing from the virtual to the physical devices. All forms of traditional para- and full virtualization fall under this category. KVM, for example, uses QEMU for I/O device emulation, hence I/O can be intercepted in the QEMU backend. The newer PVHVM technique and hardware-virtualization are also be supported. In the latter case with no guest OS support the respective device drivers in the guest operating systems must be adapted to inform the VMM about I/O operations.

7 Incremental Checkpointing

With incremental checkpointing, only the changes since the previous checkpoint are recorded. Several studies on VM live migration [33], [34] employ incremental checkpointing for their purposes. When migrating a VM from one machine to another, the memory contents are copied to the target machine iteratively. In the first iteration, the entire memory contents are transferred. In each successive iteration, only the pages that have been modified while the previous iteration was running are transferred until a threshold is reached.

The Xen VMM uses shadow page tables (SPT, see Section 6) and a dirty map to track pages that have been changed. In a first step, the dirty map is cleared and SPT mode is enabled. Write operations cause the dirty bit in the SPT to be set from where it is propagated to the dirty map by a periodically running scan. In the proposed method, we leverage this feature to implement incremental checkpointing of memory. At every checkpoint, we collect the dirty pages by inspecting the dirty map and clear the dirty bits in the SPTs. Then, SPT mode is enabled to track modifications to memory pages ending with the next checkpoint. In order to restore a guest to one of the incremental checkpoints, it is necessary to maintain a history of which pages have been recorded in which checkpoint file. Figure 4 shows the history for three incremental checkpoints.

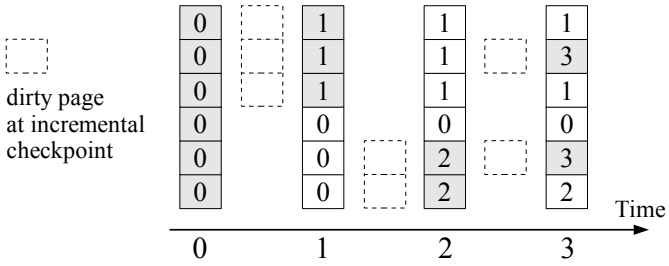


Fig. 4. Incremental checkpointing

TABLE 3
Benchmark Scenarios

Benchmark	Description
Postmark	file system benchmark (file size 4KB 4MB; 1500 files; 10000 transactions)
Make	compilation of the Linux kernel tree (make bzImage; make modules; make modules_install)
Copy	copying a large file
Gzip	compressing an Apache access log file
Download	downloading a disk image from a server
Movie	playing a movie
Software Installation	installation of LibreOffice, Mozilla Thunderbird, and OS-specific security updates
Desktop - Internet	browsing several webpages and watching videos on Youtube using Mozilla Firefox
Desktop - Office	reading, editing, and saving files using LibreOffice
Idle	idle system
Bootup	checkpoint taken immediately after booting up

Our proposed method of minimizing duplication between the memory checkpoint and the guest's virtual disk can be combined with incremental checkpointing. At each checkpoint, pages that have already been flushed to disk or added to the free page list are removed from the set of pages that have been modified compared to the previous checkpoint and do not need to be saved. The combined method outperforms incremental checkpointing in all situations (see Section 9).

TABLE 4
Base results of checkpointing

	Space[MB]	Checkpoint time[sec]	Restoration time[sec]
1GB	1022	9.00	6.96
2GB	2048	14.30	13.84
4GB	4100	26.73	27.01

8 Experimental Setup

We have conducted an extensive series of experiments with several benchmark applications representing various usage patterns. The following measures are of interest: (1) compared to the current Xen implementation, how does the proposed method perform in terms of disk space and runtime? (2) compared to incremental checkpointing of the entire guest memory, how does the proposed method without and with incremental checkpointing perform in terms of disk space and runtime? (3) what is the effect on checkpoint restoration time? and finally (4) how does the proposed technique perform with incremental checkpoints?

We have implemented the proposed technique for PV, PVHVM, and HVM environments for the Xen VMM and support all versions from 4.1.2 to the newest 4.5. The experiments have been conducted on a host machine with an Intel Core i7x8 with 16 GB of RAM. The guest operating systems was executed in a domU environment with four virtual CPUs and 4 GB of RAM. The Linux guests run the 64-bit Ubuntu 12.04 Desktop version (PV, PVHVM run the GNU/Linux kernel version 3.9.4). For the HVM Windows experiments Microsoft Windows 7 (64-bit) was used.

Lacking a standard benchmark suite for checkpointing virtualized environments, we have chosen several general benchmark scenarios that are similar to what has been used in related work [6], [26], [27]. Table 3 lists the benchmarks. `Postmark` runs the PostMark benchmark and represents an I/O intensive workload. `Make` compiles the current Linux kernel tree by executing the commands *make bzImage; make modules; make modules_install*; this represents both an I/O- and CPU-intensive workload. `Copy` copies a large files and `Gzip` compresses the same file; `Copy` is I/O-intensive while `Gzip` is more CPU-intensive. `Download` downloads a file from a remote server. The `Movie`, `Software Installation`, and the two `Desktop` benchmarks represent user sessions with several applications running at the same time. `Movie` uses the system video player to play a movie. In `Software Installation`, LibreOffice, Mozilla Thunderbird, and OS-dependent security updates are installed. For `Desktop - Internet`, we run two Firefox web browsers with three open tabs each browsing and streaming videos from YouTube. In `Desktop - Office`, LibreWriter is used to open, edit, and save a number of documents. `Idle` and `Bootup`, finally, represent an idle desktop and a system that has just been booted up. All checkpoints were taken while the benchmark in the guest was running (when applicable).

The results for each benchmark and each virtual machine were obtained by taking the average of three independent runs.

9 Results

9.1 Single Checkpoints

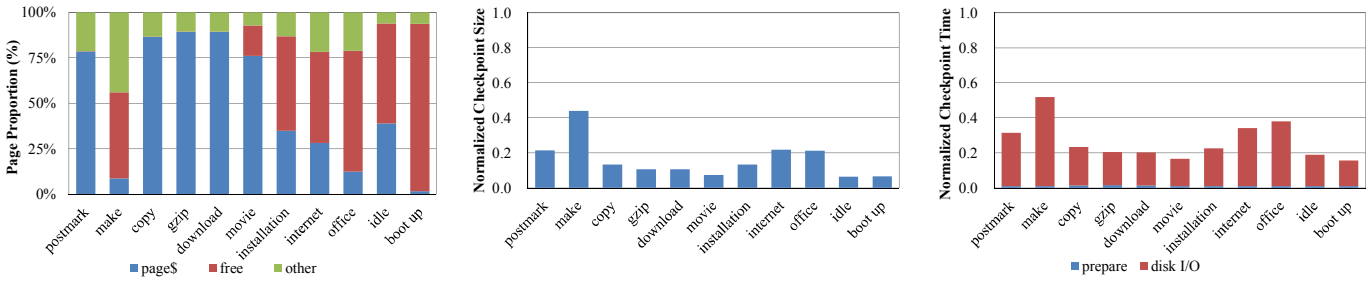
The results of taking a checkpoint for PV, PVHVM, HVM - Linux, and HVM - Windows guests are shown in Figures 5, 6, 7, and 8.

Memory Contents. Subfigure (a) shows a breakup of the VM's memory into the three categories `page$`, `free`, and `other`. `Page$` represents the amount of pages in the *page_to_block* map of the VMM. `Free` represents pages that are reported to be free at the time of the checkpoint. Information about free pages is only available in PV and PVHVM guests (Figures 5 (a) and 6 (a), respectively). `Other` represents memory pages whose contents are unknown, that is, the page is neither on in the *page_to_block* map nor on the kernel's free list. These pages hold the actual volatile data of the benchmarks such as stack and heap pages. For HVM guests (Figures 7 (a) and 8 (a)) free pages cannot be detected and are included in `other`.

The breakup of the memory contents shows that I/O-intensive benchmarks such as `postmark`, `copy`, `gzip`, and `download` exhibit a very high amount of duplication between the VM's memory and the disk independent of the virtualization technology. Windows seems to be a bit less aggressive at using the available memory as a page cache than Linux (Figure 7 (a) vs. Figure 8 (a)). Interactive benchmarks such as `software installation`, `desktop - internet`, `desktop - office`, and `idle` show a rather low occupancy of the page cache, but instead a large amount of free memory. Since our technique cannot eliminate free pages from HVM guests, a combination of the proposed approach with ballooning [4] will yield better results.

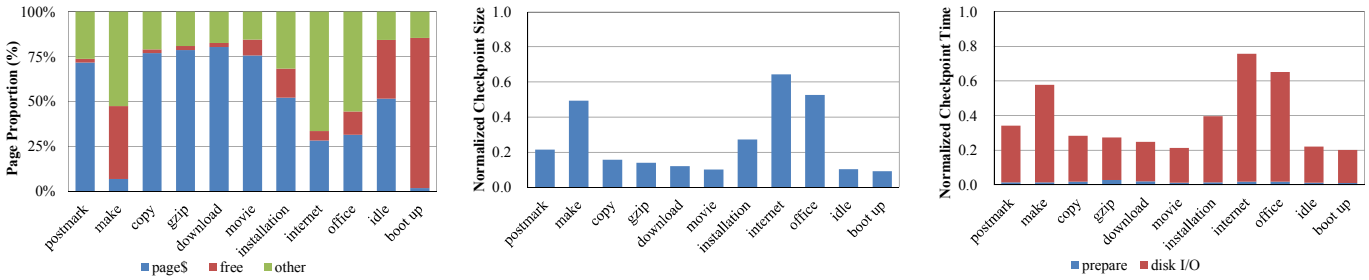
Checkpoint Image Size. Subfigure (b) in Figures 5 to 8 compares the size of the checkpoint image to the original (unmodified) Xen VMM. Since the proposed technique is able to eliminate all but the memory pages labeled `other` in the corresponding subfigures (a), the amount of `other` pages correlates directly to the size of the checkpoint image. For PV and PVHVM guests, the reduction in the size of the checkpoint image is significant: for PV guests, the average size of the checkpoint image is only 660 MB or 16% of the 4100 MB obtained from the unmodified Xen VMM. The size of the checkpoint images for PVHVM guests is slightly larger than that of PV guests. This difference is caused because the I/O performance of PVHVM guests is significantly better than that of PV guests. Since we checkpoint the guests after a fixed amount of time, the PVHVM guest will have progressed further in the benchmark than the PV guest which typically leads to less free pages in the guest's memory.

For fully-virtualized guests the proposed technique is not able to eliminate free pages from the checkpoint image. Free pages are classified as `other` which consequently leads to bigger checkpoint images. The worst



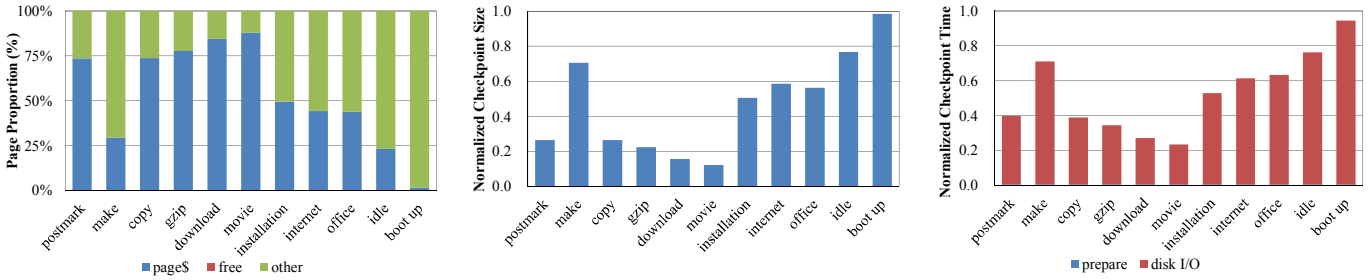
(a) Memory Occupancy (b) Checkpoint Image Size (c) Checkpoint Time

Fig. 5. Results for PV guests running Linux.



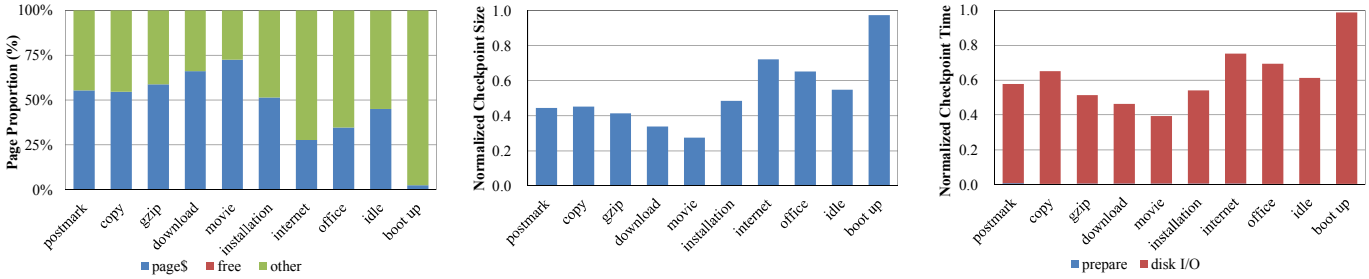
(a) Memory Occupancy (b) Checkpoint Image Size (c) Checkpoint Time

Fig. 6. Results for PVHVM guests running Linux.



(a) Memory Occupancy (b) Checkpoint Image Size (c) Checkpoint Time

Fig. 7. Results for HVM guests running Linux.



(a) Memory Occupancy (b) Checkpoint Image Size (c) Checkpoint Time

Fig. 8. Results for HVM guests running Windows.

case is demonstrated by the `bootup` scenario: even though most of the memory is free, the lack of knowing so prevents our technique from excluding the pages from the snapshot (as mentioned in Section 2, combining ballooning [4] with our technique solves this problem). Nevertheless, the average size of the checkpoint images is reduced to 47% and 53% for HVM Linux and Windows guests, respectively. Table 5 lists the detailed results for each benchmark and virtualization technology.

Checkpoint Time. Figures 5 to 8 (c) show the breakdown of the checkpointing time into `prepare` and `disk I/O`. The former represents the time required to prepare the VM for the checkpoint. In original Xen, this phase merely consists of disconnecting devices whereas in the PV and PVHVM implementation of the proposed technique information about dirty pages in the page cache and free pages needs to be compiled and delivered to the VMM. This preparation time takes up to 35-times longer in comparison to original Xen; however, even so the preparation time only takes about 10% of the total time required to save a checkpoint to disk (PVHVM - gzip exhibits the longest preparation time of all benchmarks: 0.7 seconds of the total 7 seconds). Compared to original Xen with a checkpoint time of 25 seconds this overhead is not significant. Table 6 shows the detailed results for each benchmark and virtualization technology.

The required time to generate a checkpoint is dominated by the amount of data which needs to be written to disk. Comparing the checkpoint time with the size of the checkpoint images in subfigures (b) and (c) very clearly illustrates this correlation.

Restoration time. Figure 9 compares the restoration time of unmodified Xen to the naïve restoration (denoted `p2b order`) and the optimized restoration (denoted `sorted`) implementation as elaborated in Section 4.3. The excessive overhead caused by the frequent repositionings of the rotating disk's read head is reflected by the significantly longer restoration times - restarting the HVM Linux VM running `make` from a space-optimized checkpoint, for example, suffers from a 20-fold slowdown compared to original Xen. In absolute numbers, the restoration of the VM takes 600 seconds while original Xen only takes about 28 seconds. The optimized implementation outperforms original Xen in the case of PV and PVHVM guests by 18% and 2%, on average. This result is possible because free pages are not part of the checkpoint. For HVM guests, the optimized implementation still suffers from a slight slowdown compared to original Xen: the HVM Linux guests require 15% or 7 seconds more than original Xen, for HVM Windows guests the slowdown is more severe with 30% or 13 seconds compared to original Xen. Detailed results for each benchmark are listed in Table 7.

Sorting Overhead. To optimize the restoration time when loading the snapshot image from rotating disks, the I/O requests to disk are first sorted in ascending order of physical disk blocks. The sorting is done in the VMM when the restore operation is initiated. The time overhead of sorting the *page_to_block* map is around 200ms on our test machine for a VM with 4 GB of memory. Compared to the average restoration time of 20 to 40 seconds, this overhead is negligible.

9.2 Incremental Checkpointing

Exploiting free pages and unmodified pages in the page cache allow us to save a significant amount of space and time. In a scenario where incremental checkpoints are necessary, the proposed method can be used to further increase the efficiency of incremental checkpointing. To understand the effects of incremental checkpointing, we have run four benchmark in a VM with 1 GB of RAM while taking periodic checkpoints. For each of the benchmarks, ten checkpoints were taken over the course of the entire benchmark. The checkpointing interval differs for each benchmark as their execution times differ as well: for *Make*, the checkpointing interval was set to 360 seconds, for *Postmark* to 25, for *Software update* to 10, and for *Gzip* to 5 seconds.

Figure 10 shows the results of periodic checkpointing. For each benchmark, the upper-hand graph shows the breakdown of the memory contents at each checkpoint. Here, *free* represents the amount of free memory. *Page cache* shows the amount of unmodified pages in the page cache. *Unchanged* stands for the amount of unmodified pages. None of these three classes of pages need to be saved to disk. *Changed*, finally, shows the amount of pages that have been modified since the last checkpoint and therefore need to be included in the incremental checkpoint image. To show the correlation between the page cache and the free pages more precisely, *changed* includes dirty pages from the page cache, heap, and stack, whereas *unchanged* only contains pages from the heap and the stack. Unmodified pages from the page cache are shown in *page cache*. The lower-hand graph shows the total amount of memory saved at each checkpoint for *unmodified*, the unmodified Xen checkpointing method, *page cache + Free*, the proposed method without incremental checkpointing, *inc*, incremental checkpointing implemented on top of *unmodified*, and, finally, *inc + page cache + free* representing the proposed method with incremental checkpointing.

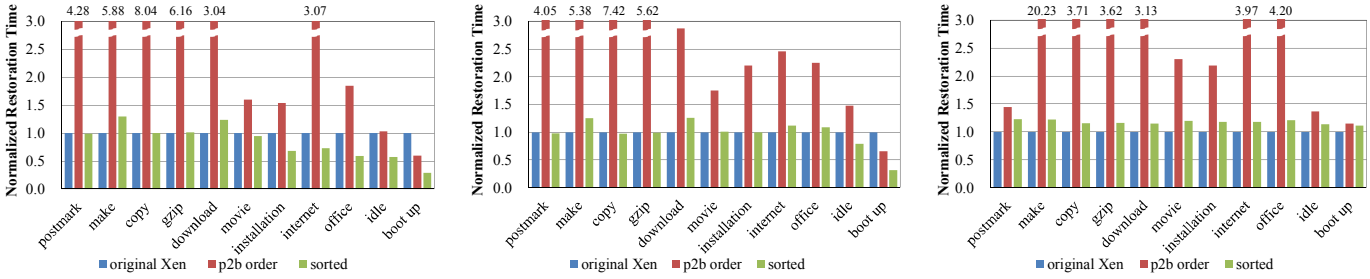
For the benchmarks *Make*, *Postmark*, and *Gzip*, the proposed method without incremental checkpointing (*page cache + free*) outperforms incremental checkpointing (*inc*), whereas for *Desktop*, simple incremental checkpointing *inc* performs better than *page cache + free*. This shows the inherent weakness of incremental checkpointing in write-intensive situations. In addition good performance can be achieved by not saving pages from the page cache that are consistent with the data on disk. Another observation that can be made from the graphs in Figure 10 is that the proposed method performs well and does not fluctuate regardless of the checkpointing interval or the I/O intensity of the application. Applying the proposed technique to incremental checkpointing further improves its effectiveness. The results show that *inc + page cache + free* outperforms all other methods independent of the I/O activity or the checkpointing interval.

TABLE 5
Checkpoint images sizes compared to unmodified Xen.

Benchmark	PV - Linux		PVHVM - Linux		HVM - Linux		HVM - Windows	
	Size[MB]	Size[%]	Size[MB]	Size[%]	Size[MB]	Size[%]	Size[MB]	Size[%]
Postmark	880	21	826	21	1088	27	1827	45
Make	1805	44	1899	49	2894	71	-	-
Copy	551	13	605	16	1082	26	1859	45
Gzip	437	11	535	14	916	22	1694	41
Download	439	11	465	12	643	16	1391	34
Movie Player	303	7	385	10	504	12	1128	28
Software Installation	549	13	1047	27	2070	50	1993	49
Desktop - Internet	897	22	2475	64	2407	59	2961	72
Desktop - Office	870	21	2025	53	2306	56	2678	65
Idle	260	6	396	10	3149	77	2250	55
Bootup	268	7	348	9	4037	98	3996	97
Average	660	16	1000	26	1918	47	2178	53

TABLE 6
Checkpoint times compared to unmodified Xen.

Benchmark	PV - Linux		PVHVM - Linux		HVM - Linux		HVM - Windows	
	Time[sec]	Speedup	Time[sec]	Speedup	Time[sec]	Speedup	Time[sec]	Speedup
Postmark	8.6	3.1	8.6	2.9	10.9	2.7	16.3	1.7
Make	14.2	1.8	14.5	1.7	19.3	1.3	-	-
Copy	6.4	4.1	7.1	3.5	10.6	2.6	18.1	1.7
Gzip	5.6	4.8	6.9	3.5	9.4	3.0	14.2	2.0
Download	5.6	4.8	6.2	3.9	7.4	3.8	12.9	2.2
Movie Player	4.6	5.7	5.4	4.7	6.4	4.3	10.9	2.6
Software Installation	6.2	4.4	10.0	2.5	14.4	1.8	15.0	1.8
Desktop - Internet	9.4	3.1	19.0	1.4	16.7	1.7	20.8	1.3
Desktop - Office	10.4	2.9	16.3	1.6	17.2	1.6	19.2	1.4
Idle	5.2	4.9	5.5	4.4	20.7	1.2	17.0	1.6
Bootup	4.3	6.5	5.1	4.9	25.7	1.0	27.4	1.0
Average	7.3	4.2	9.5	3.2	14.4	2.3	17.2	1.7

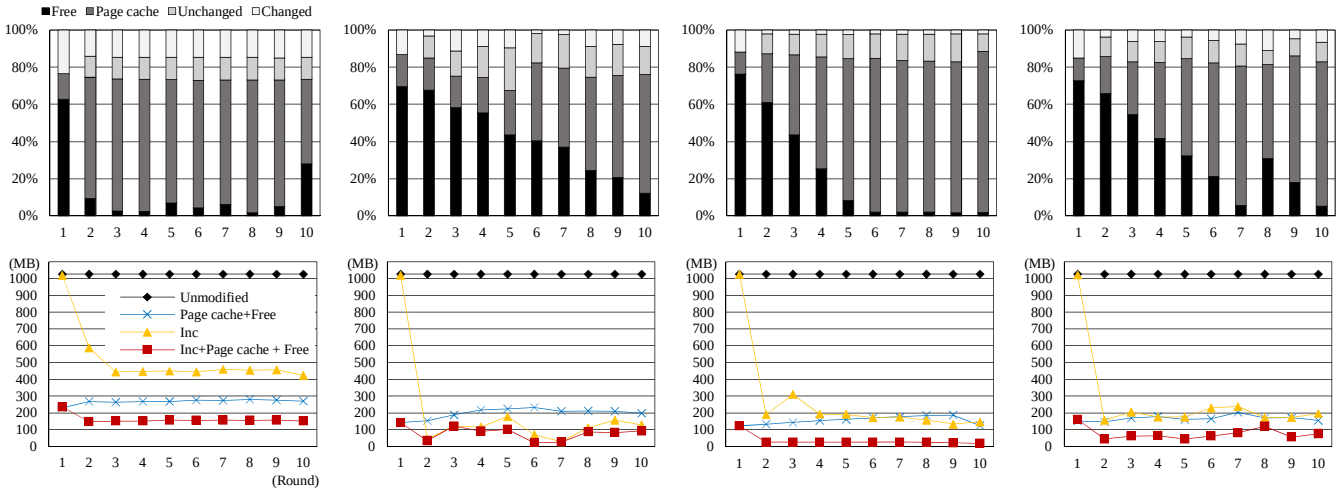


(a) PV Linux (b) PVHVM Linux (c) HVM Linux

Fig. 9. Restoration of VMs from space-optimized checkpoints.

TABLE 7
Restoration times compared to unmodified Xen.

Benchmark	PV - Linux		PVHVM - Linux		HVM - Linux		HVM - Windows	
	Time[sec]	Speedup	Time[sec]	Speedup	Time[sec]	Speedup	Time[sec]	Speedup
Postmark	26.7	1.0	25.4	1.0	34.1	0.8	39.9	0.7
Make	34.4	0.8	32.3	0.8	33.8	0.8	-	-
Copy	26.7	1.0	25.3	1.0	32.0	0.9	33.8	0.8
Gzip	27.0	1.0	25.6	1.0	32.1	0.9	42.3	0.7
Download	32.9	0.8	32.4	0.8	31.8	0.9	35.2	0.8
Movie Player	25.1	1.1	26.2	1.0	33.1	0.8	34.1	0.8
Software Installation	18.2	1.5	25.6	1.0	32.6	0.9	45.3	0.6
Desktop - Internet	20.0	1.4	28.6	0.9	33.2	0.9	43.6	0.6
Desktop - Office	16.2	1.7	27.8	0.9	33.6	0.8	48.6	0.6
Idle	15.8	1.8	20.4	1.3	31.4	0.9	46.4	0.6
Bootup	7.6	3.5	8.1	3.2	30.7	0.9	30.6	0.9
Average	22.8	1.2	25.2	1.0	32.6	0.9	40.0	0.7



(a) Postmark (b) Software Update (c) Gzip (d) Make

Fig. 10. Variations of space overhead and memory usages

TABLE 8
Periodic checkpointing compared to unmodified Xen.

Benchmark	Unmodified		Page cache + Free		Incremental		Incremental + Page cache + Free	
	Time[sec]	Space[MB]	Time[sec]	Space[MB]	Time[sec]	Space[MB]	Time[sec]	Space[MB]
Postmark	38.6	1025.0	12.5	269.8	17.6	460.8	8.4	151.4
Software update	38.6	1025.0	9.8	200.1	5.3	103.4	6.3	70.7
Gzip	38.6	1025.0	8.5	156.3	8.1	183.2	5.2	24.3
Make	38.6	1025.0	8.7	166.2	7.9	191.3	5.9	64.6
Average	38.6	1025.0	9.9	198.1	9.7	234.7	6.5	77.6

10 Conclusion and Future Work

In this work, we have presented a technique for fast and space-efficient checkpointing of virtual machines for virtual desktop cloud environments.

The technique is based on the observation that modern operating systems use the better part of the available memory to cache data recently read from or written to external storage. Through transparent I/O interception at the VMM level, the technique tracks I/O requests and maintains an up-to-date mapping of memory pages to disk blocks in the *page_to_block* map. At checkpoint time, pages known to contain data duplicated on disk are excluded from the checkpoint image, thereby saving a considerable amount of disk space and time. Several operations in OSes, such as removing pages from the page cache or writing to pages without flushing them to disk, are invisible to the VMM and would invalidate the *page_to_block* map. Pages known to hold duplicated data are mapped read-only by the VMM in order to detect in-memory updates.

We have implemented the proposed technique into the Xen VMM, for PV, PVHVM, and HVM guests. In comparison with an unmodified Xen hypervisor, experiments with Linux and Windows guests, on average, achieve a 86%, 76%, 53%, and 47% reduction in the stored data and a 73%, 62%, 47%, and 38% shorter time required to take a checkpoint for PV, PVHVM, HVM Linux, and HVM Windows guests, respectively. Restoration of guests is up to 1.2 times faster for PV and PVHVM guests; for fully-virtualized guests we observe an average slowdown of about 20% compared to unmodified Xen. In addition, combining the technique with incremental checkpointing was shown to further reduce the size of Xen's incremental checkpoints by factor of three.

Overall, the proposed techniques shows a broad applicability and good performance for a wide variety of benchmarks and virtualization techniques. The technique is not limited to the Xen VMM and independent of most other optimizations and can be combined in order to obtain even better results.

Acknowledgments

This work was supported in part by the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (No. 2012R1A1A1042938 and 2013R1A3A2003664). ICT at Seoul National University provided research facilities for this study. A preliminary version of this paper appeared in the proceedings of the 7^{th} ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE '11) [26].

REFERENCES

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A berkeley view of cloud computing," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28, Feb 2009. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html

[2] W. Zhao and Z. Wang, "Dynamic memory balancing for virtual machines," in *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. New York, NY, USA: ACM, 2009, pp. 21–30.

[3] 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 *OSDI '08: Proceedings of the 8th symposium on Operating systems design and implementation*, 2008.

[4] C. A. Waldspurger, "Memory resource management in vmware esx server," in *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*. New York, NY, USA: ACM, 2002, pp. 181–194.

[5] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault tolerance," *ACM Trans. Comput. Syst.*, vol. 14, no. 1, pp. 80–107, 1996.

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

[7] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage, "Scalability, fidelity, and containment in the potemkin virtual honeyfarm," in *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 2005, pp. 148–162.

[8] C. Jo, E. Gustafsson, J. Son, and B. Egger, "Efficient live migration of virtual machines using shared storage," in *Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, ser. VEE '13. New York, NY, USA: ACM, 2013, pp. 41–50. [Online]. Available: http://doi.acm.org/10.1145/2451512.2451524

[9] "VirtualBox," http://www.virtualbox.org.

[10] D. Bovet and M. Cesati, *Understanding the Linux Kernel, 3rd Edition*. Oreilly & Associates, 2005.

[11] "Xen 4.5.0," http://www.xenproject.org/downloads/xen-archives/xen-45-series/xen-450.html.

[12] I. Habib, "Virtualization with kvm," *Linux J.*, vol. 2008, February 2008. [Online]. Available: http://portal.acm.org/citation.cfm?id=1344209.1344217

[13] "VMware Workstation," http://www.vmware.com/products/workstation.

[14] 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 *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 2003, pp. 164–177.

[15] "Transcendent Memory Project," http://oss.oracle.com/projects/tmem.

[16] D. Magenheimer, C. Mason, D. McCracken, and K. Hackel, "Transcendent memory and linux," in *Proceedings of the Linux Symposium*, Montreal, Quebec Canada, 2009, pp. 191–200.

[17] M. Schwidefsky, H. Franke, R. Mansell, H. Raj, D. Osisek, and J. Choi, "Collaborative memory management in hosted linux environments," in *Proceedings of the Linux Symposium*, Ottawa, Ontario, Canada, 2006, pp. 313–328.

[18] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum, "Disco: running commodity operating systems on scalable multiprocessors," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 412–447, 1997.

[19] G. Milos, D. G. Murray, S. Hand, and M. A. Fetterman, "Satori: Enlightened page sharing," in *ATC'09: 2009 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2009.

[20] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, "Black-box and gray-box strategies for virtual machine migration," in *NSDI'07: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2007.

[21] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Geiger: monitoring the buffer cache in a virtual machine environment," in *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 2006, pp. 14–24.

[22] P. Lu and K. Shen, "Virtual machine memory access tracing with hypervisor exclusive cache," in *ATC'07: 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2007, pp. 1–15.

[23] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar, "Dynamic tracking of page miss ratio curve for memory management," in *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 2004, pp. 177–188.

[24] I. Zhang, A. Garthwaite, Y. Baskakov, and K. C. Barr, "Fast restore of checkpointed memory using working set estimation," in *ACM SIGPLAN Notices*, vol. 46, no. 7. ACM, 2011, pp. 87–98.

[25] B. Egger, E. Gustafsson, C. Jo, and J. Son, "Efficiently restoring virtual machines," in *NPC'13: Proceedings of the 10th IFIP international conference on Network and parallel computing*. Springer, 2013.

[26] E. Park, B. Egger, and J. Lee, "Fast and space-efficient virtual machine checkpointing," in *VEE'11: Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. New York, NY, USA: ACM, 2011, pp. 75–86. [Online]. Available: http://doi.acm.org/10.1145/1952682.1952694

[27] S. T. King, G. W. Dunlap, and P. M. Chen, "Debugging operating systems with time-traveling virtual machines," in *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2005, pp. 1–1.

[28] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "Revirt: enabling intrusion analysis through virtual-machine logging and replay," in *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*. New York, NY, USA: ACM, 2002, pp. 211–224.

[29] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan, "Snowflock: rapid virtual machine cloning for cloud computing," in *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*. New York, NY, USA: ACM, 2009, pp. 1–12.

[30] J. Liu, W. Huang, B. Abali, and D. K. Panda, "High performance vmm-bypass i/o in virtual machines," in *ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2006, pp. 3–3.

[31] J. R. Santos, Y. Turner, G. Janakiraman, and I. Pratt, "Bridging the gap between software and hardware techniques for i/o virtualization," in *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2008, pp. 29–42.

[32] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating two-dimensional page walks for virtualized systems," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS XIII. New York, NY, USA: ACM, 2008, pp. 26–35.

[33] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*. Berkeley, CA, USA: USENIX Association, 2005, pp. 273–286.

[34] M. Nelson, B.-H. Lim, and G. Hutchins, "Fast transparent migration for virtual machines," in *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2005, pp. 25–25.