Fast and Space-Efficient Virtual Machine Checkpointing *

Eunbyung Park, Bernhard Egger, and Jaejin Lee
School of Computer Science and Engineering, Seoul National University, Seoul 151-744, Korea
eunbyung@aces.snu.ac.kr, bernhard@aces.snu.ac.kr, jlee@cse.snu.ac.kr
http://aces.snu.ac.kr

Abstract

Checkpointing, i.e., recording the volatile state of a virtual machine (VM) running as a guest in a virtual machine monitor (VMM) for later restoration, includes storing the memory available to the VM. Typically, a full image of the VM's memory along with processor and device states are recorded. With guest memory sizes of up to several gigabytes, the size of the checkpoint images becomes more and more of a concern.

In this work we present a technique for fast and space-efficient checkpointing of virtual machines. In contrast to existing methods, our technique eliminates redundant data and stores only a subset of the VM's memory pages. Our technique transparently tracks I/O operations of the guest to external storage and maintains a list of memory pages whose contents are duplicated on non-volatile storage. At a checkpoint, these pages are excluded from the checkpoint image.

We have implemented the proposed technique for paravirtualized as well as fully-virtualized guests in the Xen VMM. Our experiments with a paravirtualized guest (Linux) and two fully-virtualized guests (Linux, Windows) show a significant reduction in the size of the checkpoint image as well as the time required to complete the checkpoint. Compared to the current Xen implementation, we achieve, on average, an 81% reduction in the stored data and a 74% reduction in the time required to take a checkpoint for the paravirtualized Linux guest. In a fully-virtualized environment running Windows and Linux guests, we achieve a 64% reduction of the image size along with a 62% reduction in checkpointing time.

Categories and Subject Descriptors D.4 [*Operating Systems*]: Organization and Design; D.4.7 [*Reliability*]: Checkpoint/Restart; D.4.2 [*Storage Management*]: Virtual Memory

General Terms Design, Measurement, Performance, Reliability

Keywords Virtual Machine, Checkpointing

* This work was supported by grant 2009-0081569 (Creative Research Initiatives: Center for Manycore Programming) and the BK21 project from the National Research Foundation of Korea funded by the Korean government (Ministry of Education, Science and Technology). ICT at Seoul National University provided research facilities for this study.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

VEE'11, March 9–11, 2011, Newport Beach, California, USA.
Copyright © 2011 ACM 978-1-4503-0501-3/11/03...$10.00

1. Introduction

Virtualization technology continues to gain importance in server, desktop, and embedded systems. Its applications are numerous. Data center and cluster computing providers use virtualization technology to run many OS instances concurrently on a single, physical machine with high performance, thereby providing custom-tailored environments, better use of physical resources, higher availability through live migration and security [5, 8, 11, 13, 31, 32, 34].

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 useful in many situations: 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.

To completely record the state of a virtual machine, the virtual machine monitor (VMM) must store the virtual CPU's state, the current state of all emulated devices, and the contents of the virtual machine's memory to non-volatile memory. An identical copy of the VM's memory contents stored to disk, therefore, the size of the snapshot is typically dominated by the amount of the guest's memory.

With the rapid growth of memory in virtual machines, the size of snapshots is becoming more and more of a concern, especially in situations where a multitude of snapshots is taken over a certain period of time, such as, for example, in intrusion detection. The prevalent technique to alleviate this problem is incremental checkpointing [1, 23, 26, 27]. In incremental checkpointing, the first snapshot contains a complete image of the VM's memory. Subsequent snapshots only store pages that have been modified since the previous snapshot. Initially all memory pages are marked read-only. Whenever the guest modifies a page, that page is remapped read-write. When the subsequent snapshots are taken, only pages that are mapped read-write are included in the image.

In this paper, we propose a fast and space-efficient virtual machine checkpointing method for memory. With this technique, the size of the snapshot image is reduced considerably. Furthermore, compared to existing methods, significantly less time is needed to take a snapshot. The basic observation is that it is unnecessary for the VMM to store the full contents of the VM's memory because a large part of the data is already present on external storage. Modern operating systems allocate most of the memory that is not currently in use by either the kernel or the running processes to a cache commonly known as *page cache* [7]. The page cache contains data that has been recently read from (or is to be written to) block devices in order to reduce the long access latency incurred when accessing external devices. In situations where it is unnecessary to restore the contents of memory not currently allocated by the guest, such pages do not need to be included in the snapshot.

In a VM environment, all I/O operations to devices go through the VMM. In the proposed technique, the VMM keeps track of 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 both in Xen's paravirtualized and fully-virtualized VMM. In both cases, the VMM keeps transparently track of I/O operations. In a paravirtualized environment, the guest VM informs the VMM about the contents of its page cache and free memory pages when it is about to be checkpointed. The VMM uses this information to filter out pages that have been modified after their contents have been read from/written to disk. Additionally, the VMM also discards free memory pages from the memory image. In a fully-virtualized environment, the unmodified guests do not provide any information to the VMM. To maintain consistency, the VMM must intercept all possible actions that can invalidate the contents of a memory page that is known to contain data from external storage. This is accomplished by marking the guest's memory pages as read-only in the shadow page tables of the VMM and updating the internal data structures on a write fault.

The contributions of this paper are as follows:

- We introduce a fast and space-efficient checkpointing technique for virtual machines. By ignoring memory pages whose contents are available on non-volatile storage, we are able to reduce the time and the space required to take a checkpoint.
- We have implemented the proposed technique in Xen's paravirtualized environment and modified the Linux guest running in it. We identify and correctly handle all scenarios that could lead to a corruption of the VMM's disk-to-memory mapping data structures which would result in a corrupted memory image.
- We have implemented the proposed technique in Xen's fully-virtualized VMM and ran the benchmarks on Microsoft Windows and Linux guest operating systems.
- We compare the performance of the unmodified Xen VMM with the paravirtualized and fully-virtualized approach, both for individual checkpoints as well as incremental checkpoints. Compared to the unmodified VMM, we achieve an 81% and 64% reduction in disk space plus a 74% and 62% reduction in time required to complete a snapshot in paravirtualized and fully-virtualized environments, respectively. The proposed method also outperforms incremental checkpointing with a 57% reduction in disk space and a 26% reduction in time needed to take a checkpoint.

The rest of this work is organized as follows: Section 2 discusses related work; Section 3 contains an overview of VM checkpointing; and Section 4 outlines our method. In Section 5, we discuss the implementation of the proposed technique. The experimental setup is described in Section 6 and Section 7 presents the results. Finally, Section 8 concludes the paper.

2. Related Work

Since the emergence of virtualization technology, physical memory allocation to VMs and related functionality such as checkpointing, live migration, fault tolerance, and logging with replay have received a lot of attention in the research community.

While the available CPUs and I/O bandwidth on a physical machine can be fairly time-shared or multiplexed, allocating or dynamically reclaiming memory is not easy due to the fact that memory has an inherent weakness as a shared resource. Reclaiming memory from a running VM, without seriously affecting its performance, is not easy because the VMM has limited knowledge over the contents memory and thus its importance to the VM. A common technique is Ballooning [32]: the VMM communicates with the ballooning driver running inside the VM. This driver requests non-shared memory from the guest operating system. Since that memory cannot be used by any guest application, the VMM can reuse this memory and allocate it to another VM. Most major VMMs, such as KVM [14], VMware [4], VirtualBox [3], and Xen [6], make use of the ballooning technique to reclaim memory from a running VM. Transcendent memory [2, 22] 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. Paravirtualized guests can access that memory indirectly through an API. Based on paravirtualization, several approaches have been proposed such as efficient memory provisioning [29], memory sharing between multiple VMs [9, 24], and hot spot migration [33].

VM live migration [10, 25] builds on checkpointing techniques to migrate a running VM from one physical machine to another without stopping the execution of the VM being migrated. Both studies take a pre-copy approach in which memory pages are iteratively copied. In the first iteration, the VM's memory as a whole is transferred to the target machine while the guest keeps running on the source machine. In the following iterations, only modified pages are copied repeatedly until either a fixed number of iterations is reached or the number of dirty pages drops below a certain threshold. The VM is then stopped on the source machine and the VM's state, as well as with the remaining dirty pages, is sent to the target machine where the VM is restarted. Post-copy approaches have also been studied [15].

Remus [11] and Kemari [30] achieve high availability of VMs based on continuous live migration. A backup of the running VM on a physically different host is maintained by repeatedly transferring the state of the running VM. In case of a hardware failure, the backup starts running on behalf of the original VM. Frequent checkpointing is used to keep the amount of modified pages low. In case of a write-intensive VM, even frequent checkpoints cannot guarantee a sufficiently low number of modified pages. In such situations, a copy-on-write (CoW) approach [15] can be taken. All memory pages are marked read-only and transferred immediately to the backup on a write fault.

The studies most relevant to the proposed page cache tracking mechanism are Geiger [16] and VM memory access tracing with an exclusive cache in the VMM [21]. 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 [35] used to estimate the WSS is based on the reuse distance [16]. 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.

Both operating system debugging, through the use of a VM [18], and intrusion analysis, via logging and replay [12], benefit from efficient VM checkpoints. The Potemkin virtual honeyfarm [31], used to gather intelligence on network honeypots, and Snowflock [19], a system which enables rapid construction of VM cluster computing systems, also profit from efficient checkpointing.

3. 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 (also referred to as *snapshots*) 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 the 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), and 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) should not be 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 the cache.

VMMs provide a paravirtualized or a fully-virtualized execution environment. In a paravirtualized 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, paravirtualization is a pervasive virtualization technology supported by most major VMMs due to its benefits, such as improved I/O performance [20, 28], better memory management [9, 24, 29], and load balancing [33]. 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 [6] offer a paravirtualized environment as well as full virtualization. The following paragraphs describe the differences in context of checkpointing for both the paravirtualized and the fully-virtualized Xen implementation.

Paravirtualized environment. When the VMM checkpoints a VM, the paravirtualized guest is first notified through the para-API. The guest itself then carries out several tasks including disconnecting devices, disabling interrupts, or marking page tables. After finishing the preparation work, the guest transfers control to the VMM. The VMM uses the privileged *dom0* domain 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, such as network adapters, the keyboard, the display and the number and context of the processors (*vcpus*) are stored as well. Special care needs to be taken when saving the page tables of the guest as they contain physical machine addresses of memory allocated to the guest at checkpoint time. The physical addresses are changed to pseudo-physical addresses that simplify restoration as well as migration to another host.

Restoring a guest running in a VM is the exact opposite process. The *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. The pseudo-physical addresses in the guest's page tables are modified to match the actual physical addresses of the memory allocated to the VM. 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.

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

Table 1. Data duplication between memory and disk.

VMM	Snapshot Size (MB)	Remarks
VMware	1057	
VirtualBox	450	compressed
Xen	1025	

Table 2. Size of a snapshot of a VM with 1 GB of memory.

Fully-virtualized environment. In a fully-virtualized environment, the guest is not aware that it is being checkpointed. 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* to fully virtualize memory accesses. See Section 5.3). Restoring a fully-virtualized guest is similar to restoring a paravirtualized one. In a fully-virtualized environment, however, the VMM has to re-initialize the devices itself since the guest expects the devices to operate as if it had never been checkpointed.

Live migration [10, 25] and VM replication systems, such as Remus [11], build on top of checkpointing/restoration functionality. In both cases, frequent and continuous checkpointing functionality is required. Under such circumstances, *incremental checkpointing*, i.e., saving only the differences since the last checkpoint, is more efficient than saving the whole state of the VM. Incremental checkpointing can also be beneficial for guests that are not write-intensive.

4. Motivation

Memory sizes of several gigabytes are not uncommon these days, even for virtual machines. 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. Depending on the usage scenario and the size of the memory, the memory allocated to this *page cache* may easily exceed 90% of the physically available memory.

Table 1 shows the extent to which memory contents are duplicated on disk. To extract this data, we have saved the guest's memory after booting-up the system and copying some large files. We then compared the memory contents to the disk on a 4096-byte page level. While copying around large files provokes a big page cache and may not be a very realistic 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 [3], VMware [4], and Xen [6]. The snapshots for VMware and Xen are both a bit larger than the VM's memory size. VirtualBox compresses snapshots, but nevertheless stores all of the memory allocated to the VM.

From Tables 1 and 2 it follows that there exists a big potential to reduce the size of VM snapshots. Instead of storing the redundant data in the VM snapshot, a simple reference to the external storage medium, along with an offset, suffices. This will not only reduce

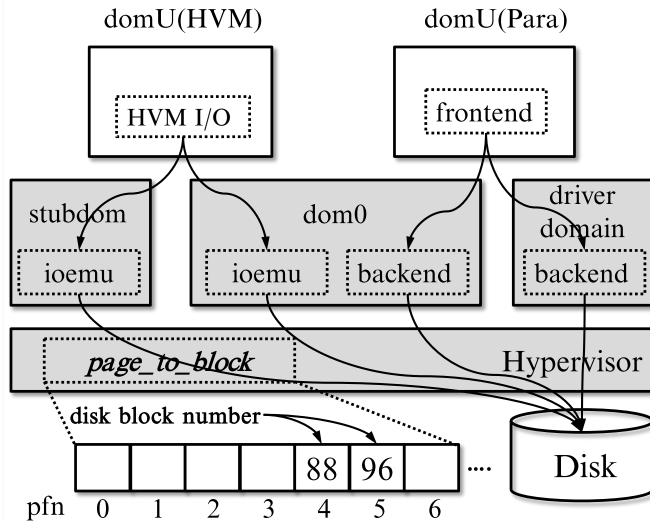


Figure 1. Disk I/O flow in the Xen VMM and the page-to-block map.

the size, but also shorten the time required to take a snapshot. This reduction comes at the expense of slightly increasing the time to restore a snapshot since reading a contiguous file into memory is faster than fetching single blocks from a disk image.

There exist other opportunities to further reduce the size of snapshots, such as free memory pages and memory that is not mapped into the guest's address space. The former can only be exploited in a paravirtualized environment since VMM has no knowledge of which mapped memory pages the guest is currently using and which represent free pages. The latter can be excluded from the snapshot both in paravirtualized and fully-virtualized environments because the VMM can inspect the guest's page tables to check what pages are actually mapped.

5. Implementation

To achieve space as well as time efficient checkpointing, the proposed method minimizes duplication of pages in the checkpoint file that are also available on the disk 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.

5.1 Transparent I/O Interception

In virtualized environments, the VMM (or *hypervisor*) 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 or the hardware, bus arbitration is not visible to the software and simultaneous requests to the physical disk or NIC without mediation of software can cause unknown behavior or even device crashes. For this reason, I/O operations to secondary storage without hardware virtualization support are always under control by the VMM. We can thus intercept all I/O requests transparently.

Figure 1 shows the flow of disk I/O requests in the Xen VMM for both a paravirtualized and a fully-virtualized guest. An I/O request from a paravirtualized guest running in the user domain, *domU* (Figure 1 on the right), is either forwarded directly to the VMM's domain zero, *dom0*, or handled by a specialized driver domain. Any I/O request to the physical disk can therefore be intercepted in the driver backend either in the VMM's privileged domain zero or the driver domain for the disk. For fully-virtualized guests (Figure 1 on the left), I/O requests go through *ioemu* located either in *dom0* or a stub domain *stubdom*. Here, we can intercept all I/O activity in *ioemu*.

Mapping page frames to disk blocks. We associate the VM's memory contents with physical disk blocks by transparently tracking all read and write requests from the guest to external storage. The gathered information is stored in a memory page-to-disk block map ($page_to_block$ map) that is maintained 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* (*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 1 GB of allocated memory, the $page_to_block$ map requires 2 MB of memory. I/O operations are costly operations by themselves; the table update does not noticeably increase the I/O latency (Section 7).

Figure 1 shows the contents of the $page_to_block$ after the guest reads the disk blocks number 88 and 96 into its memory pages at index 4 and 5, respectively.

The VMM can track all disk I/O operations of the guest transparently and maintain a mapping of memory pages to disk blocks. Once in the guest's memory, however, modifications to cached pages through memory writes are hidden from the VMM. One could compare the contents of both the memory page and the associated disk blocks when the checkpoint is taken. This, however, would impose an unacceptable computational overhead. In order to efficiently store only those memory pages whose contents are not duplicated on the guest's disk, the memory-to-disk mapping represented by the $page_to_block$ map must be up-to-date at checkpoint time. To maintain an up-to-date $page_to_block$ map, modifications to memory pages in the map must be tracked. This is where the paravirtualized and fully-virtualized implementation diverge. The following sections describe the implementation issues for the paravirtualized environment (Section 5.2) and the fully-virtualized approach (Section 5.3).

5.2 Paravirtualized Environment

In a paravirtualized environment, there exist two type of pages that do not need to be included in a snapshot: (a) unmodified pages in the page cache, and (b) free pages.

5.2.1 The Page Cache

The page cache, also referred to as disk cache or buffer cache, is designed to offset the performance gap between the main memory (RAM) and block devices by keeping the most frequently used data of block devices in main memory. In modern operating systems such as Microsoft Windows, Linux, Solaris, and many more, the biggest part of the available memory (i.e., memory available to the operating system and not in use by the kernel or any active application) is allocated to the page cache.

The page cache caches pages for an indefinite period of time. As long as the system has sufficient memory to satisfy all currently running processes and no additional memory requests occur, pages are not dropped from the page cache. That is, data in the page cache is not expelled unless it is replaced by more recent data or by the kernel reclaiming pages from the page cache to serve memory allocation requests.

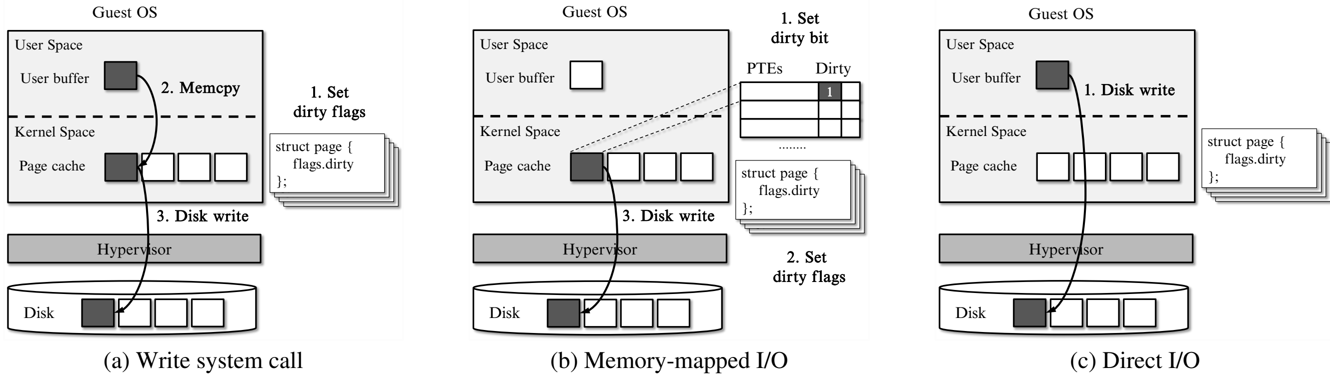


Figure 2. Write operations : write system call, memory-mapped I/O, and direct I/O.

Write operations to the disk are also cached by the page cache. The page cache typically operates in *write-back* mode, i.e., dirty pages are not written to disk immediately, but rather marked dirty and flushed periodically by a background thread.

Note that a reverse mapping from disk blocks to memory pages is not necessary because the OS guarantees that a disk block is not cached more than once by the page cache.

5.2.2 Maintaining Consistency

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. However, at the moment of the checkpoint dirty pages may exist 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 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 (Figure 2a). If a page cache hit occurs, the kernel marks the affected pages as 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 pages to the page cache, marks them as dirty, and then performs the memory copy operation. The interpretation of a page's dirty flag is thus that the contents of the page differ from the data on disk. For the VMM, it is not possible to detect changes to the internal data structures of a guest, hence it is not aware of the fact that some pages in the page cache are dirty.

In the paravirtualized Xen environment, the guest performs 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 adds them to a list of pages to be saved by the VMM.

Memory mapped I/O. The memory pages involved in memory mapped I/O are part of the kernel's page cache, that is, there is no user space buffer involved (Figure 2b). As a consequence, the 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.

It is possible that some dirty bits in the PTEs have not yet been propagated to the corresponding dirty flags in the page cache at the moment a guest is checkpointed. The VMM knows which memory pages contain the guest's page tables and scans the user address space for *dirty* bits. The corresponding pages in the $page_to_block$ map are included in the memory checkpoint. It would be possible to let the guest scan the PTEs as part of the preparatory work, however, we chose to perform this task in the VMM in order to keep the necessary changes to the guest OS to a minimum.

Note that the kernel often maps memory pages into its own address space. The Linux kernel maps the whole 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 2c). When a self-caching application, such as Oracle's DBMS, opens a file with the O_DIRECT flag, 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, these user mode buffers are not part of the page cache and have thus not been added to the file-backed page list. We conservatively assume that all pages used for direct I/O are dirty and include them in the snapshot 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 could also track these types of I/O operations, however, because of the small number of such blocks and for the sake of simplicity, we do not consider such pages for exclusion from the memory checkpoint.

In the Linux kernel, subpages are managed with an additional data structure in the form of a linked buffer list pointed to by the $buffer_head$ on top of the page cache data structure. For block-based I/O requests, the kernel adds a new element to the buffer list and links it to a page in the page cache. Every element in the buffer list thus points to a specific part of a page that is logically split into several blocks. For compatibility reasons, some elements of the buffer list may point to an entire page, however, such elements are easily recognized by their size being equal to the size of the page. We scan the $buffer_head$ data structure and remove all logically split pages from the $page_to_block$ map.

5.2.3 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 on the contents of free pages can be made. For security reasons or when allocating the *BSS* (i.e., zero-filled) segment of an executable image, memory pages may be requested to be *zero-filled-on-demand* (ZFOD). However, ZFOD pages are not handled by the low-level memory manager that is in charge of managing the free pages, hence we can safely ignore this issue.

In the Linux kernel, a system-wide memory manager, the *buddy allocator*, is responsible for satisfying memory requests and managing free pages at the lowest level. The buddy allocator manages free pages in chunks. A chunk consists of physically continuous free pages. The size of a chunk (i.e., the number of consecutive free pages) is a power of 2, and the buddy allocator maintains several free lists for the different chunk sizes. We have modified the guest OS to inform the VMM of all physical frame numbers of free pages in the context of the preparatory work carried out by the guest before a checkpoint. These pages are then excluded from the memory checkpoint.

5.3 Fully-virtualized Environments

In a fully-virtualized environment, the VMM cannot obtain any information about pages in the page cache directly from the guest. Instead, the VMM itself must track all operations that might lead to an inconsistency between the memory page and the VMM's $page_to_block$ map.

5.3.1 Double Mappings and Unaligned I/O

Unlike in paravirtualized environments, a reverse mapping from disk blocks to memory pages is required with full virtualization. Assuming that the guest OS has issued two read operations $read(pfn : 1, block : 16, size : 4K)$ and $read(pfn : 3, block : 16, size : 4K)$, the $page_to_block$ map contains two references to block $\#16$ in indices 1 and 3. A subsequent write operation to block $\#16$, i.e., $write(pfn : 2, block : 16, size : 4K)$ must invalidate the entries at positions 1 and 3, plus 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 that provides a reverse mapping from disk blocks to memory pages. The hash key is the disk block number, and the data is a linked list containing the indices of all associated memory pages.

The reverse page-to-block map is also used to keep track of unaligned I/O requests. In the paravirtualized environment, page cache-related I/O operations are aligned at 8-block boundaries (assuming 512 byte disk blocks, this is equal to the size of a memory page; 4 KB). However, a fully-virtualized guest may issue I/O requests to arbitrary disk block numbers. The offset to the next lower 8-block boundary is thus also recorded in the reverse page-to-block map. This allows us to track I/O requests that span several entries in the $page_to_block$ map and correctly invalidate them if necessary.

5.3.2 Maintaining Consistency

Similarly to the paravirtualized scenario, the VMM has to follow modifications to memory pages that are currently tracked in the $page_to_block$ map. In a fully-virtualized environment, the VMM maintains shadow page tables to provide the guest with the illusion of virtual memory. Our technique builds on top of this mechanism to detect modifications to tracked memory pages.

Shadow page tables. In a fully-virtualized environment, the guest does not control the hardware page tables; instead it maintains guest page tables, and the VMM manages the hardware page tables. The VMM reflects changes made by the guest to its page tables into the hardware page tables. These tables are commonly referred to as *shadow page tables* (SPTs). Initially, VMM identifies the guest's page tables by following the (guest's) page table base register and walking the page tables. It then marks these pages read-only to provoke a write fault exception whenever the guest modifies one of its page tables. When running on real hardware, page tables contain physical addresses. For the guest, the VMM uses pseudo-physical addresses (PFNs) in its page tables, and the VMM uses machine physical addresses (MFNs) in the shadow page tables. Keeping the shadow page tables in sync with the guest's tables and validating and translating PFNs to MFNs are important, performance-critical operations in a fully-virtualized environment. There are many subtleties associated with shadowing 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 that point to such pages as read-only. This happens whenever we update the $page_to_block$ map due to an I/O operation. 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 write to a write-protected page, a write 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 finally restarts the aborted write operation.

Memory pages can be mapped into the virtual address space more than once. Since it would be too time-consuming to scan all shadow page tables for possible double mappings, we maintain a reverse map that maps MFNs to shadow page table entries. The reverse map does not need to track all pages, only pages in the $page_to_block$ map that have been re-mapped read-write need to be stored in the reverse shadow page table.

On modern hardware, shadow page tables are no longer necessary thanks to hardware-assisted virtualized memory translation (extended page tables (EPT, Intel) and nested page tables (NPT, AMD), respectively). With EPT/NPT, a global physical-to-machine table is available per domain (and not per process), so flipping one entry in that table is sufficient to mark a page read-only, thereby eliminating the need for a reverse map.

5.4 Incremental Checkpointing

With incremental checkpointing, only the changes since the previous checkpoint are recorded. Several studies on VM live migration [10, 25] 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. At 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 5.3) and a dirty map to track pages that have been changed. As 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

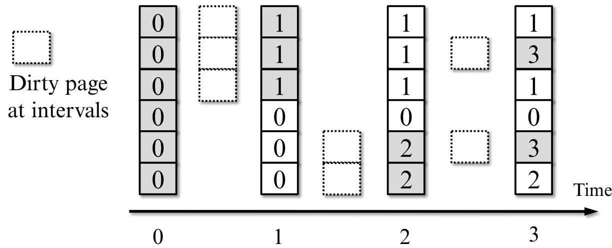


Figure 3. Incremental checkpointing

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 clearing the dirty bits in the SPTs. Then, SPT mode is enabled to track modifications to memory pages ending at 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 3 shows the history for three incremental checkpoints.

Incremental checkpoints are especially effective for guests that are not write intensive or in situations where frequent checkpoints are required, otherwise the advantage is lost. As shown in the next section, compared to the technique presented in this paper, incremental checkpointing outperforms our basic method only if the number of modified pages between checkpoint internals is sufficiently small.

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 since the previous checkpoint and do not need to be saved. The combined method outperforms incremental checkpointing in all situations (see Section 7).

6. Experimental Setup

We have conducted a series of experiments with several benchmark applications representing various usage patterns. The following measures are of interest to us: (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, with and without incremental checkpointing, perform in terms of disk space and runtime? (3) how big is the runtime overhead incurred by transparently tracking I/O accesses and the additional bookkeeping? and finally, (4) what is the effect on checkpoint restoration time?

We have implemented the proposed technique for fast and space-efficient checkpointing for paravirtualized and fully-virtualized environments in the 3.4.2 Xen VMM, the 2.6.31.6 dom0 pvops Linux kernel, and the 2.6.32.10 domU Linux kernel; the official release versions at the time development started. The experiments have been conducted on a host machine with an Intel Core Duo T2300 running at 1.66 GHz with 2 GB of RAM. The guest operating systems ran in a domU environment with one virtual CPU and 1 GB of RAM.

Lacking a standard benchmark suite for virtualized environments, we have chosen several general benchmark scenarios that are similar to what has been used in related work [11, 18]. Table 3 lists the benchmarks. *Make* compiles the current Linux kernel tree by executing the commands *make bzImage; make modules; make modules_install*. *Gzip* compresses a 1.4 GB log file of an Apache web server. *Make* and *Gzip* represent CPU-intensive benchmarks with moderate memory and disk I/O bandwidth. The *postmark* benchmark [17] is designed to measure file system performance. We have chosen Postmark as a representative for an I/O intensive application. Postmark was configured to run 1000 transactions on 128 files ranging in size from 4 KB to 10 MB. The *Desktop* benchmark represents a user session with several applications running at the same time. In the guest VM, we run two Firefox web browsers with three open tabs each, as well as OpenOffice Writer and OpenOffice Presentation. Checkpointing is often used before and after performing software updates in order to revert to the last known working and clean configuration, respectively. This scenario is represented by *Software installation* in which the installation of several widely-used packages are performed.

Benchmark	Description
Postmark	file system benchmark (file size 4K-10MB; 128 files; 1000 transactions)
Desktop	Firefox with two windows and 3 tabs each; OpenOffice Writer and Presentation
Software Installation	installation of OpenOffice and Firefox
Gzip	compress an Apache access log file of 1.4 GB
Make	compilation of the Linux kernel tree (make bzImage; make modules; make modules_install)

Table 3. Benchmark Scenarios

7. Results

We have conducted two sets of experiments: in the first, for both paravirtualized and fully-virtualized guests, a single checkpoint was taken after each benchmark application was run to completion. In the second, only for a paravirtualized guest, a series of checkpoints were taken at fixed interval while the benchmark applications were still running.

7.1 Single Checkpointing in Paravirtualized Environments

Table 4 presents results for a single checkpoint taken after each benchmark ran to completion using the paravirtualized technique presented in this paper. Columns two to five contain the break down of the memory contents into different memory categories recognized by our technique. Column two, *Free pages*, shows the amount of free memory at the time of checkpointing. *Page cache*, in column three, displays the size of the clean pages in the page cache that do not need to be included in the snapshot image. The fourth column, *Dirty pages*, shows the total size, in megabytes, of the dirty pages in the page cache, i.e., pages that need to be saved because they had been modified but have not yet been written to disk. Column five, labeled *Others*, shows the amount of memory in use for other purposes, such as the heap or the stack. With the proposed method, only the amounts listed in columns four and five, *Dirty pages* and *Others*, need to be saved to disk; not the entire 1 GB of memory. The sum of columns two and three, *Free pages* and *Page cache*, represents the amount of memory that does not need to be saved in the checkpoint file and is equal to the improvement over the default Xen checkpointing method. Column 5, *Space*, shows the total amount of memory, in megabytes and as a percentage, that was saved to disk. The last column, *Time*, shows both the total time required to take the checkpoint and the speedup compared to the unmodified Xen checkpointing.

The results show that the ratio of dirty pages to clean pages in the page cache is very low. This is not surprising because with the default settings for the page cache, Linux flushes dirty pages back to disk every 5 seconds and allows, at most, 10% of dirty pages. Except for *Desktop*, the majority of available memory was

Benchmark	Free pages [MB]	Page cache [MB]	Dirty pages [MB]	Others [MB]	Space [MB]	Ratio [%]	Time [sec]	Speedup
Postmark	49.2	704.6	2.6	267.6	270.2	26.4	12.86	3.0
Desktop	489.5	257.1	6.4	271.0	277.4	27.0	12.47	3.1
Software update	151.0	739.5	16.6	116.7	133.3	13.0	9.72	4.0
Gzip	19.5	883.6	3.5	117.4	120.9	11.8	8.87	4.3
Make	141.1	713.7	5.0	164.2	169.2	16.5	8.53	4.5
Average	170.6	659.7	6.8	247.4	194.2	18.9	10.49	3.8

Table 4. Results of checkpointing paravirtualized guests.

OS	Benchmark	Space [MB]	Ratio [%]	Time [sec]	Speedup	Reverse Map Average
Linux	Postmark	363.7	35.3	16.17	2.4	0.440
	Desktop	809.1	78.7	30.77	1.3	0.364
	Software Installation	263.5	25.6	13.95	2.8	0.349
	Gzip	157.8	15.3	10.48	3.7	0.810
	Make	178.9	17.4	10.55	3.6	0.472
Windows	Postmark	231.1	22.5	14.19	2.7	0.202
	Desktop	759.5	73.8	32.00	1.2	0.315
	Software Installation	357.5	34.7	15.80	2.4	0.203
	Gzip	237.9	23.1	12.70	3.0	0.220
	Average	373.2	36.3	17.40	2.6	0.375

Table 5. Results of checkpointing fully-virtualized guests.

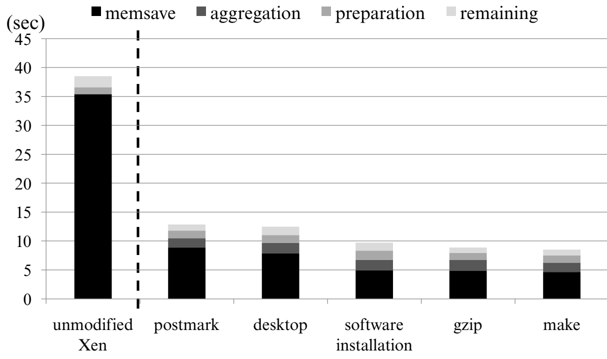


Figure 4. Breakdown of the checkpoint time.

allocated to the page cache and did not have to be saved to disk. *Desktop* and *Postmark* display a relatively large amount of memory in the *Others* column. *Desktop* runs several big applications that use large amounts of heap memory, and *Postmark* consumes a lot of memory for user-level buffers due to frequent *write* system calls. *Desktop* is not a very I/O intensive benchmark, hence the amount of memory committed to the page cache is relatively small, and, as a consequence, the amount of free memory is rather large.

Compared to the default Xen implementation that saves the entire 1 GB of memory, our method achieves a reduction of 89% in the amount of disk space consumed in a paravirtualized environment, at a 3.5-fold speedup in runtime. The time required to write a checkpoint image to external storage is dominated by the I/O latency, hence the reduction of disk I/O is directly reflected in the speedup.

Runtime overhead. To analyze the runtime overhead of the proposed technique, we have divided the time required to save a checkpoint into different phases: *Memsave* indicates the time required to write the memory contents to disk. *Aggregation* represents the time needed to combine the *page_to_block* map with the information obtained from the guest and perform the scan for dirty bits in the page tables. *Preparation* shows the time consumed by the guest to execute the preparation work and includes the time required to transfer information about free and dirty pages to the VMM. *Remaining* is the time consumed to record the device states and finalize the checkpoint.

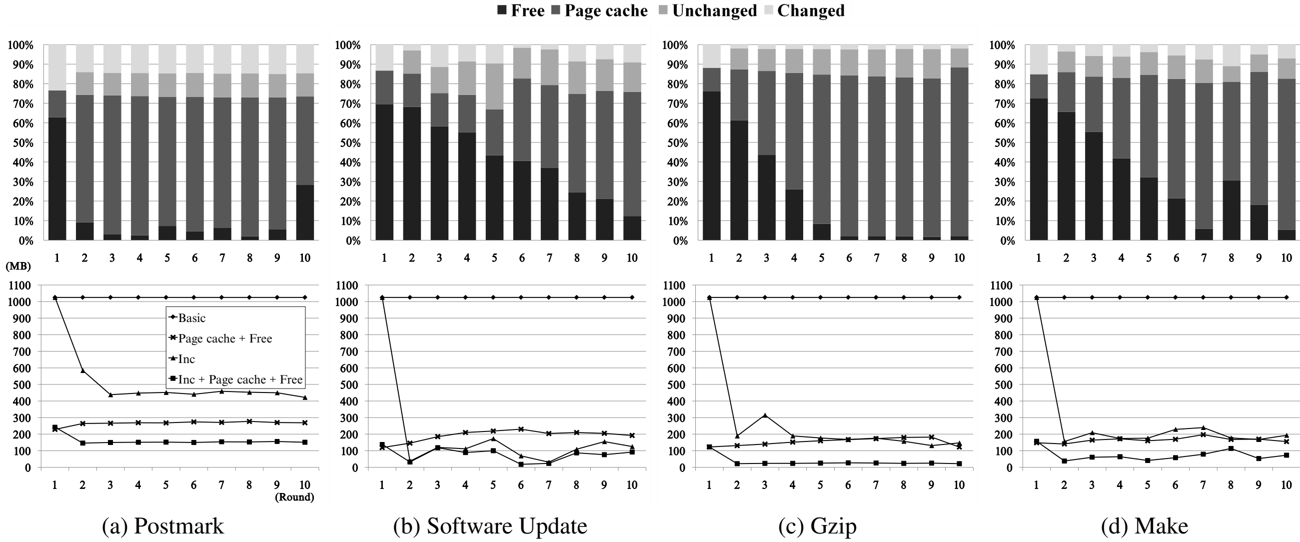
Figure 4 shows the breakdown of the checkpoint time for the different benchmarks using the proposed technique and the standard Xen implementation. Including the additional work required for our technique, *preparation* takes only 0.2s on average and does not significantly contribute to the total checkpointing time. The *aggregation* phase does not appear in the standard Xen implementation. Using the proposed technique, it consumes an average of 1.7 seconds. This is not insignificant, however, the benefits of reduced I/O time, as shown by the *memsave* phase, by far outweigh the cost introduced by the *aggregation* phase.

Transparently tracking I/O requests and keeping the *page_to_block* map up-to-date does incur a small runtime overhead. To measure that overhead, we ran the I/O-intensive *Postmark* benchmark in an unmodified Xen environment and under our modified conditions. The results show no statistically significant difference. In other words, the overhead incurred by tracking I/O requests is not noticeable.

Restoration Time. With the presented technique, we have measured restoration times of up to 50% slower than those obtained in an unmodified Xen environment. This degradation is due to the increased seek activity caused by reading a small number of disk blocks spread all over the guest's virtual disk. Currently, we read the disk blocks one-by-one in no particular order which may lead to an excessive seek overhead. This overhead can be reduced by sorting the disk blocks in ascending order (assuming that the disk image itself is not heavily fragmented on the physical disk). We are saving this optimization for future work.

7.2 Single Checkpointing in Fully-virtualized Environments

Table 5 presents the results of our method in a fully-virtualized environment. Except for *Make*, we ran all benchmarks on both Ubuntu Linux and Microsoft Windows XP. Column three shows the amount of data, in megabytes and as a percentage, that was saved to disk. Column four shows the total time that was required to take the checkpoint and the speedup compared to an unmodified Xen environment. The last column, labeled *Reverse Map Average*, shows the average degree of reverse mapping per page. A value of 1 indicates that all guest pages have been mapped one time.



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

Figure 5. Variations of space overhead and memory usage.

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	-	-	12.48	269.8	17.59	460.8	8.41	151.4
Software update	-	-	9.80	200.1	5.26	103.4	6.32	70.7
Gzip	-	-	8.53	156.3	8.12	183.2	5.18	24.3
Make	-	-	8.67	166.2	7.87	191.3	5.91	64.6
Average	38.50	1025	9.9	198.1	9.71	234.7	6.5	77.6

Table 6. Results of periodic checkpointing.

In a fully-virtualized environment, our approach saves 64% of disk space at a 2.6-fold speedup in runtime compared to the default Xen implementation. A comparison of the results obtained for a fully-virtualized guest with the paravirtualized environment shows that the latter performs significantly better. This is because in a fully-virtualized environment, free pages cannot be detected, however, in the paravirtualized case free pages are excluded from the snapshot. *Desktop* shows that in the paravirtualized environment, a total of 490 MB of free pages and 257 MB of page cache data were excluded, resulting in a snapshot image of only 277 MB. A snapshot in the fully-virtualized environment, however, is a sizable 809 MB. This corresponds almost exactly to the savings that can be achieved from the page cache in the paravirtualized environment.

The numbers for *Reverse Map Average* deserve some more analysis. Since we maintain the reverse map only for read-write SPTEs (and ignore read-only mapped entries), the average reverse mapping degree is very low. Most modern OSes use a copy-on-write approach for multi-mappings, hence mappings are typically read-only. In other words, the space overhead of the reverse map is moderate even if the guest OS re-maps the same physical memory page several times.

Runtime overhead. The runtime overhead caused by transparently tracking I/O requests, maintaining the reverse map and handling the extra write fault exception is most pronounced for benchmarks with a high write activity such as *Postmark*. Compared to the unmodified Xen VMM, the experiments show a moderate 1.5% runtime performance degradation for the *Postmark* benchmark.

7.3 Periodic checkpointing

Exploiting free pages and unmodified pages in the page cache allow us to save a significant amount of space and time, however, we can further optimize the proposed method for periodic checkpoints by combining it with incremental checkpointing. To understand the effects of incremental checkpointing, we run the benchmark applications (Table 3) while taking periodic checkpoints. The *Desktop* benchmark is excluded from these experiments since it is not a good candidate for periodic checkpoints because the system is idle once all applications have been started. For each of the remaining four benchmarks, we have taken ten checkpoints over the course of each benchmark. The checkpointing interval differs for each benchmark as their execution times differ as well: for *Make*, the checkpointing interval was 360 seconds, for *Postmark* 25, *Software update* 10, and *Gzip* 5 seconds.

Figure 5 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*, which represents 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 5 is that the proposed method does perform well and does not fluctuate regardless of the checkpointing interval or the I/O intensity of the application. Our technique directly benefits from the observation that the guest OS itself maintains consistency between dirty pages in the page cache and the disk while keeping the ratio of dirty pages at a minimum. In addition to that, the proposed technique benefits from incremental checkpointing as well. Our results show that *inc + page cache + free* outperforms all other methods independent of the I/O activity or the checkpointing interval.

Note that pure incremental checkpointing, as represented by *inc*, still needs to save the full contents of the guest's memory at the first checkpoint, whereas the proposed method, with or without incremental checkpointing, is able to reduce the size of the first checkpoint to a small fraction of the guest's memory.

The upper-hand graph of the *Make* benchmark (Figure 5 (d)) shows the correlation between the page cache and free pages. As more and more files get compiled, the number of pages allocated to the page cache increases and consequently the number of free pages decreases. At the 7th checkpoint, there are almost no free pages left, and the OS starts reclaiming pages from the page cache to satisfy memory requests. As a result, the amount of pages in the page cache has shrunk by the amount of pages reclaimed as free pages. This can be observed in the memory breakdown graph for the 8th checkpoint. This is a typical phenomenon in modern operating systems and can be observed in the other benchmarks as well.

Table 6 shows the average time required to save a checkpoint and the average size of the memory checkpoint files for an unmodified Xen implementation (*unmodified*, column two), the proposed method without incremental checkpointing (*page cache + free*, column three), the standard Xen implementation with incremental checkpointing (*incremental*, column five), and finally, the proposed method with incremental checkpointing (*incremental + page cache + free*, last column). Each of the result columns is broken down into the time required to take a checkpoint and the amount of disk space consumed. In the case of *unmodified*, the full memory contents are saved at every checkpoint independent of the benchmark. When computing the average, we do not consider the first checkpoint since incremental checkpointing (*incremental*) needs to save the full memory contents at the first iteration. Comparing our method with incremental checkpointing (*incremental + page cache + free*) to the *unmodified* approach, we achieve a 92% reduction in disk space and an 83% reduction in the time consumed to save the checkpoint. It also outperforms incremental checkpointing (*incremental*) by 67% for disk space and 33% for checkpointing time. Finally, we observe that our method, without incremental checkpointing (*page cache + free*), requires more time on average (9.9 sec compared to 9.71 sec) than the incremental approach (*incremental*) despite a smaller amount of saved data (198.7 MB compared to 234.7 MB). The reason for this is the aggregation time (Section 7.1) that is not present in *incremental*.

8. Conclusion and Future Work

In this work, we have presented a technique for fast and space-efficient checkpointing of virtual machines. Modern operating systems (OS) use the better part of the available memory for a page cache that caches data recently read from or written to disk. Through transparent I/O interception at the virtual machine monitor (VMM) level, we track I/O requests and maintain an up-to-date mapping of memory pages to disk blocks in our *page_to_block* map. At checkpoint time, we exclude those pages from the memory images written to disk, thereby saving a considerable amount of disk space and time. Several operations in the OS, 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. We have identified all these operations and notify the VMM in order to keep the mapping consistent.

We have implemented the proposed technique into the current Xen VMM, both for paravirtualized and fully-virtualized guests. Our experiments show that, on average, we achieve an 81% reduction in the stored data and a 74% reduction in the time required to take a single checkpoint in a paravirtualized environment.

For future work, we plan to integrate the presented technique into a framework for live migration of virtual machines with the goal of improving both the migration time as well as the downtime of the running guest.

References

[1] Red Hat, Inc. LVM architectural overview. `http://www.redhat.com/docs/manuals/enterprise/RHEL-5-manual/Cluster_Logical_Volume_Manager/LVM_definition.html`.

[2] Transcendent Memory Project. `http://oss.oracle.com/projects/tmem`.

[3] VirtualBox. `http://www.virtualbox.org`.

[4] VMware Workstation. `http://www.vmware.com/products/workstation`.

[5] 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. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html`.

[6] 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*, pages 164–177, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5. doi: http://doi.acm.org/10.1145/945445.945462.

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

[8] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Trans. Comput. Syst.*, 14(1):80–107, 1996. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/225535.225538.

[9] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM Trans. Comput. Syst.*, 15(4):412–447, 1997. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/265924.265930.

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

[11] 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*, pages 161–

174, Berkeley, CA, USA, 2008. USENIX Association. ISBN 111-999-5555-22-1.

[12] 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*, pages 211–224, New York, NY, USA, 2002. ACM. ISBN 978-1-4503-0111-4. doi: http://doi.acm.org/10.1145/1060289.1060309.

[13] 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.

[14] I. Habib. Virtualization with kvm. *Linux J.*, 2008, February 2008. ISSN 1075-3583. URL `http://portal.acm.org/citation.cfm?id=1344209.1344217`.

[15] M. R. Hines and K. Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 51–60, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-375-4. doi: http://doi.acm.org/10.1145/1508293.1508301.

[16] 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*, pages 14–24, New York, NY, USA, 2006. ACM. ISBN 1-59593-451-0. doi: http://doi.acm.org/10.1145/1168857.1168861.

[17] J. Katcher. PostMark: A New File System Benchmark. Technical Report Technical Report TR3022, Network Appliance, October 1997.

[18] 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*, pages 1–1, Berkeley, CA, USA, 2005. USENIX Association.

[19] 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*, pages 1–12, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-482-9. doi: http://doi.acm.org/10.1145/1519065.1519067.

[20] 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*, pages 3–3, Berkeley, CA, USA, 2006. USENIX Association.

[21] 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*, pages 1–15, Berkeley, CA, USA, 2007. USENIX Association. ISBN 999-8888-77-6.

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

[23] D. T. Meyer, G. Aggarwal, B. Cully, G. Lefebvre, M. J. Feeley, N. C. Hutchinson, and A. Warfield. Parallax: virtual disks for virtual machines. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 41–54, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-013-5. doi: http://doi.acm.org/10.1145/1352592.1352598.

[24] 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, 2009. USENIX Association.

[25] 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*, pages 25–25, Berkeley, CA, USA, 2005. USENIX Association.

[26] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: transparent checkpointing under unix. In *TCON'95: Proceedings of the USENIX 1995 Technical Conference Proceedings on USENIX 1995 Technical Conference Proceedings*, pages 18–18, Berkeley, CA, USA, 1995. USENIX Association.

[27] J. S. Plank, Y. Chen, K. Li, M. Beck, and G. Kingsley. Memory exclusion: optimizing the performance of checkpointing systems. *Softw. Pract. Exper.*, 29(2):125–142, 1999. ISSN 0038-0644. doi: http://dx.doi.org/10.1002/(SICI)1097-024X(199902)29:2⟨125::AID-SPE224⟩3.0.CO;2-7.

[28] 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*, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[29] 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*, pages 313–328, Ottawa, Ontario, Canada, 2006.

[30] Y. Tamura. Kemari: Virtual machine synchronization for fault tolerance using domt. In *Xen Summit*, 2008.

[31] 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*, pages 148–162, New York, NY, USA, 2005. ACM. ISBN 1-59593-079-5. doi: http://doi.acm.org/10.1145/1095810.1095825.

[32] C. A. Waldspurger. Memory resource management in vmware esx server. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, pages 181–194, New York, NY, USA, 2002. ACM. ISBN 978-1-4503-0111-4. doi: http://doi.acm.org/10.1145/1060289.1060307.

[33] 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, 2007. USENIX Association.

[34] 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*, pages 21–30, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-375-4. doi: http://doi.acm.org/10.1145/1508293.1508297.

[35] 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*, pages 177–188, New York, NY, USA, 2004. ACM. ISBN 1-58113-804-0. doi: http://doi.acm.org/10.1145/1024393.1024415.