Tesseract: Reconciling Guest I/O and Hypervisor Swapping in a VM

Kapil Arya [*]
Northeastern University
kapil@ccs.neu.edu

Yury Baskakov
VMware, Inc.
ybaskako@vmware.com

Alex Garthwaite [*]
agarthwaite@acm.org

Abstract

Double-paging is an often-cited, if unsubstantiated, problem in multi-level scheduling of memory between virtual machines (VMs) and the hypervisor. This problem occurs when both a virtualized guest and the hypervisor overcommit their respective physical address-spaces. When the guest pages out memory previously swapped out by the hypervisor, it initiates an expensive sequence of steps causing the contents to be read in from the hypervisor swapfile only to be written out again, significantly lengthening the time to complete the guest I/O request. As a result, performance rapidly drops.

We present Tesseract, a system that directly and transparently addresses the double-paging problem. Tesseract tracks when guest and hypervisor I/O operations are redundant and modifies these I/Os to create indirections to existing disk blocks containing the page contents. Although our focus is on reconciling I/Os between the guest disks and hypervisor swap, our technique is general and can reconcile, or deduplicate, I/Os for guest pages read or written by the VM.

Deduplication of disk blocks for file contents accessed in a common manner is well-understood. One challenge that our approach faces is that the locality of guest I/Os (reflecting the guest's notion of disk layout) often differs from that of the blocks in the hypervisor swap. This loss of locality through indirection results in significant performance loss on subsequent guest reads. We propose two alternatives to recovering this lost locality, each based on the idea of asynchronously reorganizing the indirected blocks in persistent storage.

We evaluate our system and show that it can significantly reduce the costs of double-paging. We focus our experiments on a synthetic benchmark designed to highlight its effects. In our experiments we observe Tesseract can improve our benchmark's throughput by as much as 200% when using traditional disks and by as much as 30% when using SSD. At the same time worst case application responsiveness can be improved by a factor of 5.

Categories and Subject Descriptors D.4.2 [*Storage Management*]: Virtual memory, Main memory, Allocation/deallocation strategies, Storage hierarchies

Keywords Virtualization; memory overcommitment; swapping; paging; virtual machines; hypervisor

1. Introduction

Guests running in virtual machines read and write state between their memory and virtualized disks. Hypervisors such as VMware ESXi [1] likewise may page guest memory to and from a hypervisor-level swap file to reclaim memory. To distinguish these two cases, we refer to the activity within the guest OS as *paging* and that within the hypervisor as *swapping*. In overcommitted situations, these two sets of operations can result in a two-level scheduling anomaly known as "double paging". Double-paging occurs when the guest attempts to page out memory that has previously been swapped out by the hypervisor and leads to long delays for the guest as the contents are read back into machine memory only to be written out again.

While the double-paging anomaly is well known [6–8, 13, 17], its impact on real workloads is not established. In addition, recent studies show that other factors significantly impact the performance of guests in the presence of uncooperative hypervisor swapping activity [5]. This paper also does not address the question of how often the double-paging anomaly occurs in real workloads. Nonetheless, we do show its effects can be mitigated in a virtual environment.

Our approach addresses the double-paging problem directly in a manner transparent to the guest. First, the virtual machine is extended to track associations between guest memory and either blocks in guest virtual disks or in the hypervisor swap file. Second, the virtual disks are extended to support a mechanism to redirect virtual block requests

[*] Work done while all authors were at VMware.

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. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

VEE '14, March 1–2, 2014, Salt Lake City, Utah, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2764-0 /14/03…$15.00.
http://dx.doi.org/10.1145/2576195.2576198

to blocks in other virtual disks or the hypervisor swap file. Third, the hypervisor swap file is extended to track references to its blocks. Using these components to restructure guest I/O requests, we eliminate the main effects of double-paging by replacing the original guest operations with indirections between the guest and swap stores. An important benefit of this approach is that where hypervisors typically attempt to avoid swapping pages likely to be paged out by the guest, the two levels may now cooperate in selecting pages since the work is complementary.

We have prototyped our approach on the VMware Workstation [3] platform enhanced to explicitly swap memory in and out. While the current implementation focuses on deduplicating guest I/Os for contents stored in the hypervisor swap file, it is general enough to also deduplicate redundant contents between guest I/Os themselves or between the hypervisor swap file and guest disks.

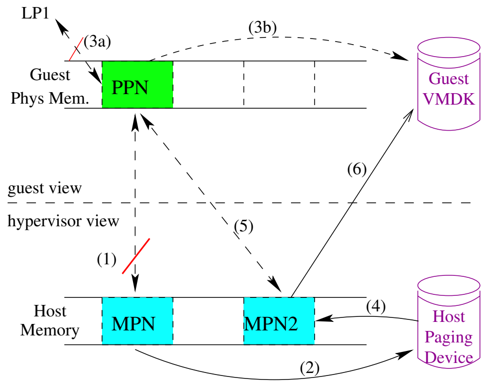
We present results using a synthetic benchmark that show, for the first time, the cost of the double-paging problem. We also show the impact of an unexpected side-effect of our solution: loss of locality caused by indirections to the hypervisor swap file which can substantially slow down subsequent guest I/Os. Finally, we describe techniques to detect this loss of locality and to recover it. These techniques isolate the expensive costs of the double-paging effect and making them asynchronous with respect to the guest.

We begin, in Section 2, with an exploration of the problems we are solving. In Section 3, we offer a high-level overview of our solution and the challenges it addresses. In Section 4, we describe the implementation of our basic prototype, in Section 5 we consider extensions to recover guest locality through defragmentation, and in Section 6, we offer some initial results. In Section 7, we turn to related work. Finally, in Sections 8 and 9, we outline possible future directions and conclude.

2. Motivation: The Doubly-Paging Anomaly

Tesseract has four objectives. First, to extend VMware's host platforms to explicitly manage how the hypervisor pages out memory so that its swap subsystem can employ many of the optimizations used by the ESX platform. Second, to prototype the mechanisms needed to identify redundant I/Os originating from the guest and virtual machine monitor (VMM) and eliminate these. Third, to use this prototype to justify restructuring the underlying virtual disks of VMs to support this optimization. Finally, to simplify the hypervisor's memory scheduler so that it need not avoid paging out memory that guest may decide to page. To address these, the project initially focused on the double-paging anomaly.

One of the tasks of the hypervisor is to allocate and map host (or machine) memory to the VMs it is managing. Likewise, one of the tasks of the guest operating system in a VM is to manage the guest physical address space, allocating and mapping it to the processes running in the guest. In both cases, either the set of machine memory pages or the set of guest physical pages may be oversubscribed.



(1), (2)	:	Swap out
(3a,3b)	:	Guest block write request
(4)	:	Memory allocation and swap in
(5)	:	Establish PPN to MPN mapping
(6)	:	Write block to guest disk
(7)	:	Zero the new MPN for reuse

Figure 1: An example of double-paging.

In overcommitted situations, the appropriate memory scheduler must repurpose some memory pages. For example, the hypervisor may reclaim memory from a VM by swapping out guest pages to the hypervisor-level swap file. Having preserved the contents of those pages, the underlying machine memory may be used for a new purpose. The guest OS may reclaim memory within a VM too to allow a guest physical page to be used by a new virtual mapping.

As hypervisor-level memory reclamation is transparent to the guest OS, the latter may choose to page out to a virtualized disk pages that were already swapped by the hypervisor. In such cases, hypervisor must synchronously allocate machine pages to hold the contents and read the already swapped contents back into that memory so they can be saved, in turn, to the guest OS's swap device. This multi-level scheduling conflict is called double-paging.

Figure 1 illustrates the double-paging problem. Suppose the hypervisor decides to reclaim a machine page (MPN) that is backing a guest physical page (PPN). In step 1, the mapping between the PPN and MPN is invalidated and, in step 2, the contents of MPN is saved to the hypervisor's swap file. Suppose the guest OS later decides to reallocate PPN for a new guest virtual mapping. It, in turn, in step 3a invalidates the guest-level mappings to that PPN and initiates an I/O to preserve its contents in a guest virtual disk (or guest VMDK). In handling the guest I/O request, the hypervisor must ensure that the contents to be written are available in memory. So, in step 4, the hypervisor faults the contents into a newly allocated page (MPN2) and, in step 5, establishes a mapping from PPN to MPN2. This sequence puts extra pressure on the hypervisor memory system and may further cause additional hypervisor-level swapping as a result of allocating MPN2. In step 6, the guest OS completes the I/O by writing the contents of MPN2 to the guest VMDK. Finally, the guest OS is able to zero the contents of the new MPN so that the PPN that now maps to it can be used for a new virtual mapping in step 7.

A hypervisor has no control over when a virtualized guest may page memory out to disk, and may even employ reclamation techniques like ballooning [17] in addition to hypervisor-level swapping. Ballooning is a technique that co-opts the guest into choosing pages to release back to the platform. It employs a guest driver or agent to allocate, and often pin, pages in the guest's physical address-space. Ballooning is not a reliable solution in overcommitted situations since it requires guest execution to choose pages and release memory and the guest is unaware of which pages are backed by MPNs. Hypervisors that do not also page risk running out of memory. While preferring ballooning, VMware uses hypervisor swapping to guarantee progress. Because levels of overcommitment vary over time, hypervisor swapping may interleave with the guest, under pressure from ballooning, also paging. This can lead to double paging.

The double-paging problem also impacts hypervisor design. Citing the potential effects of double-paging, some [13] have advocated avoiding the use of hypervisor-level swapping completely. Others have attempted to mitigate the likelihood through techniques such as employing random page selection for hypervisor-level swapping [17] or employing some form of paging-aware paravirtualized interface [7, 8]. For example, VMware's scheduler uses heuristics to find "warm" pages to avoid paging out what the guest may also choose to page out. These heuristics have extended effects, for example, on the ability to provide large (2MB) mappings to the guest. Our goals are to address the double-paging problem in a manner that is transparent to the guest running in the VM and identifies and elides the unnecessary intermediate steps such as steps 4, 5 and 6 in Figure 1 and to simplify hypervisor scheduling policies. Although we do not demonstrate that double-paging is a problem in real workloads, we do show how its effects can be mitigated.

3. Design

We now describe our prototype's design. First, we describe how we extended the hosted platform to behave more like VMware's server platform, ESX. Next, we outline how we identify and eliminate redundant I/Os. Finally, we describe the design of the hypervisor swap subsystem and the extensions to the virtual disks to support indirections.

3.1 Extending The Hosted Platform To Be Like ESX

VMware supports two kinds of hypervisors: the hosted platform in which the hypervisor cooperatively runs on top of an unmodified host operating system such as Windows or Linux, and ESX where the hypervisor runs as the platform kernel, the *vmkernel*. Two key differences between these two platforms are how memory is allocated and mapped to a VM, and where the network and storage stacks execute.

In the existing hosted platform, each VM's device support is managed in the *vmx*, a user-level process running on the host operating system. Privileged services are mediated by the *vmmon* device driver loaded into the host kernel, and control is passed between the vmx and the VMM and its guest via vmmon. An advantage of the hosted approach is that the virtualization of I/O devices is handled by libraries in the vmx and these benefit from the device support of the underlying host OS. Guest memory is mmapped into the address space of the vmx. Memory pages exposed to the VMM and guest by using the vmmon device driver to pin the pages in the host kernel and return the MPNs to the VMM. By backing the mmapped region for guest memory with a file, hypervisor swapping is a simple matter of invalidating all mappings for the pages to be released in the VMM, marking, if necessary, those pages as dirty in the vmx's address space, and unpinning the pages on the host.

In ESX, network and storage virtual devices are managed in the vmkernel. Likewise, the hypervisor manages per-VM pools of memory for backing guest memory. To page memory out to the VM's swap file, the VMM and vmkernel simply invalidate any guest mappings and schedule the pages' contents to be written out. Because ESX explicitly manages the swap state for a VM including its swap file, it is able to employ a number of optimizations unavailable on the current hosted platform. These optimizations include the capturing of writes to entire pages of memory [4], and the cancellation of swap-ins for swapped-out guest PPNs that are targets for disk read requests.

The first optimization is triggered when the guest accesses an unmapped or write-protected page and faults into the VMM. At this point, the guest's instruction stream is analyzed. If the page is shared [17] and the effect of the write does not change the content of the page, page-sharing is not broken. Instead, the guest's program counter is advanced past the write and it is allowed to continue execution. If the guest's write is overwriting an entire page, one or both of two actions are taken. If the written pattern is a known value, such as repeated 0x00, the guest may be mapped a shared page. This technique is used, for example, on Windows guests because Windows zeroes physical pages as they are placed on the freelist. Linux, which zeroes on allocation of a physical page, is simply mapped a writeable zeroed MPN. Separately, any pending swap-in for that PPN is cancelled. Since the most common case is the mapping of a shared zeroed-page to the guest, this optimization is referred to as the PShareZero optimization.

The second optimization is triggered by interposition on guest disk read requests. If a read request will overwrite whole PPNs, any pending swap-ins associated with those PPNs are deferred during wrie-preparation, the pages are pinned for the I/O, and the swap-ins are cancelled on successful I/O completion.

We have extended Tesseract so that its guest-memory and swap mechanisms behave more like those of ESX. Instead of mmapping a pagefile to provide memory for the guest, Tesseract's vmx process mmaps an anonymously-backed region of its address space, uses madvise to mark the range as `NOTNEEDED`, and explicitly pins pages as they are accessed by either the vmx or by the VMM. Paging by the hypervisor becomes an explicit operation, reading from or writing to an explicit swap file. In this way, we are able to also employ the above optimizations on the hosted platform. We consider these as part of our baseline implementation.

3.2 Reconciling Redundant I/Os

Tesseract addresses the double-paging problem *transparently* to the guest allowing our solution to be applied to unmodified guests. To achieve this goal, we employ two forms of interposition. The first tracks writes to PPNs by the guest and is extended to include a mechanism to track valid relationships between guest memory pages and disk blocks that contain the same state. The second exploits the fact that the hypervisor interposes on guest I/O requests in order to transform the requests' scatter-gather lists. In addition, we modify the structure of the guest VMDKs and the hypervisor swap file, extending the former to support indirections from the VMDKs into the hypervisor swap disk. Finally, when the guest reallocates the PPN and zeroes its contents, we apply the PShareZero optimization in step 7 in Figure 1.

In order to track which pages have writable mappings in the guest, MPNs are initially mapped into the guest read-only. When written by the guest, the resulting page-fault allows the hypervisor to track that the guest page has been modified. We extend this same tracking mechanism to also track when guest writes invalidate associations between guest pages in memory and blocks on disk. The task is simpler when the hypervisor, itself, modifies guest memory since it can remove any associations for the modified guest pages. Likewise, virtual device operations into guest pages can create associations between the source blocks and pages. In addition, the device operations may remove prior associations when the underlying disk blocks are written. This approach, employed for example to speed the live migration of VMs from one host to another [14], can efficiently track which guest pages in memory have corresponding valid copies of their contents on disks.

The second form of interposition occurs in the handling of virtualized guest I/O operations. The basic I/O path can be broken down into three stages. The basic data structure describing an I/O request is the scatter-gather list, a structure that maps one or more possibly discontiguous memory extents to a contiguous range of disk sectors. In the *preparation* stage, the guest's scatter-gather list is examined and a new request is constructed that will be sent to the underlying physical device. It is here that the unmodified hypervisor handles the faulting in of swapped out pages as shown in steps 4 and 5 of Figure 1. Once the new request has been constructed, it is *issued asynchronously* and some time later there is an *I/O completion* event.

To support the elimination of I/Os to and from virtual disks and the hypervisor block-swap store (or BSST), each guest VMDK has been extended to maintain a mapping structure allowing its virtual block identifiers to refer to blocks in other VMDKs. Likewise, the hypervisor BSST has been extended with per-block reference counts to track whether blocks in the swap file are accessible from other VMDKs or from guest memory.

The tracking of associations and interposition on guest I/Os allows four kinds of I/O elisions:

swap - guest-I/O a guest I/O follows the hypervisor swapping out a page's contents

swap - swap a page is repeatedly swapped out to the BSST with no intervening modification

guest-I/O - swap the case in which the hypervisor can take advantage of prior guest reads or writes to avoid writing redundant contents to the BSST

guest-I/O - guest-I/O the case in which guest I/Os can avoid redundant operations based on prior guest operations where the results known reside in memory (for reads) or in a guest VMDK (for writes)

For simplicity, Tesseract focuses on the first two cases since these capture the case of double-paging. Because Tesseract does not introspect on the guest, it cannot distinguish guest I/Os related to memory paging from other kinds of guest I/O. But the technique is general enough to support a wider set of optimizations such as disk deduplication for content streamed through a guest. It also complements techniques that eliminate redundant read I/Os across VMs [13].

3.3 Tesseract's Virtual Disk and Swap Subsystems

Figure 2 shows our approach embodied in Tesseract. The hypervisor swaps guest memory to a block-swap store (BSST) VMDK, which manages a map from guest PPNs to blocks in the BSST, a per-block reference-counting mechanism to track indirections from guest virtual disks, and a pool of 4KB disk blocks. When the guest OS writes out a memory page that happens to be swapped out by the hypervisor, the disk subsystem detects this condition while preparing to issue the write request. Rather than bringing memory contents for the swapped out page back to memory, the hypervisor updates the appropriate reference counts in the BSST, issues the I/O, and updates metadata in guest VMDK and adds a reference to the corresponding disk block in BSST.

Figure 3 shows timelines for the scenario when guest OS is paging out an already swapped page with and without Tesseract. With Tesseract we are able to eliminate the overheads of a new page allocation and a disk read.

To achieve this, Tesseract modifies the I/O preparation and I/O completion steps. For write requests, the memory pages in the scatter-gather list are checked for valid associations to

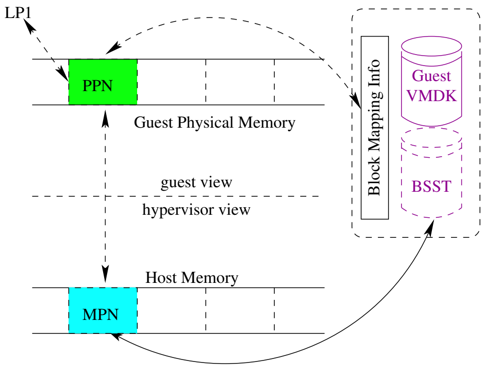
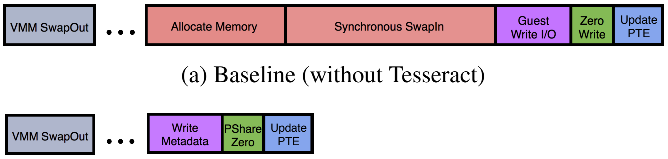


Figure 2: Double-paging with Tesseract.



(a) Baseline (without Tesseract)

(b) With Tesseract

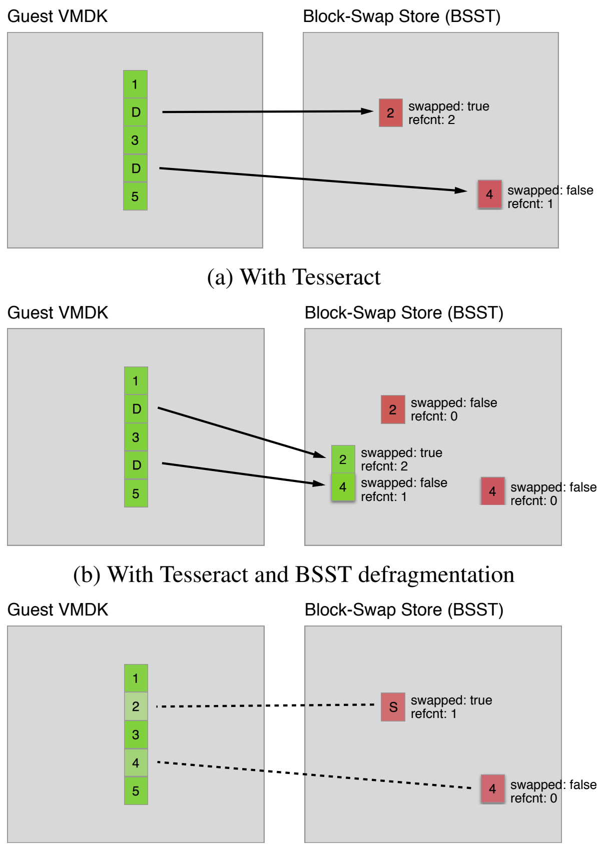
Figure 3: Write I/O and hypervisor swapping.

blocks in the BSST. If these are found, the target VMDK's mapping structure is updated for those pages' corresponding virtual disk blocks to reference the appropriate blocks in the BSST and the reference counts of these referenced blocks in the BSST are incremented. For read requests, the guest I/O request may be split into multiple I/O requests depending on where the source disk blocks reside.

Consider the state of a guest VMDK and the BSST as shown in Figure 4a. Here, a guest write operation wrote five disk blocks in which two were previously swapped to the BSST. In this example, block 2 still contains the swapped contents of some PPN and has a reference count reflecting this fact and the guest write. Hence, its state has "swapped" as true and a reference count of 2. Similarly, block 4 only has a nonzero reference count because the PPN whose swapped contents originally created the disk block has since been accessed and its contents paged back in. Hence, its state has "swapped" as false and a reference count of 1. To read these blocks from the guest VMDK now requires three read operations: one against the guest VMDK and two against the BSST. The results of these read operations must then be coalesced in the read completion path.

One can view the primary cost of double-paging in an unmodified hypervisor as impacting the write-preparation time for guest I/Os. Likewise, one can view the primary cost of these cases in Tesseract as impacting the read-completion time. To mitigate these effects, we consider two forms of defragmentation. Both strategies make two assumptions:

- the original guest write I/O request (represented in blue) captures the guest's notion of expected locality, and



(a) With Tesseract

(b) With Tesseract and BSST defragmentation

(c) With Tesseract and guest VMDK defragmentation

Figure 4: Examples of Tesseract and of defragmentation.

- the guest is unlikely to immediately read the same disk blocks back into memory

Based on these assumptions, we extended Tesseract to asynchronously reorganize the referenced state in the BSST. In Figure 4b, we copy the referenced blocks into a contiguous sequence in the BSST and update the guest VMDK indirections to refer to the new sequence. This approach reduces the number of split read operations. In Figure 4c, we copy the references blocks back to the locations in the original guest VMDK where the guest expects them. With this approach, the typical read operation need not be split. In effect, Tesseract asynchronously performs the expensive work that occurred in steps 4, 5, and 6 of Figure 1 eliminating its cost to the guest.

4. Implementation

Our prototype extends VMware Workstation as described in section 3.1. Here, we provide more detail.

4.1 Explicit Management of Hypervisor Swapping

VMware Workstation relies on the host OS to handle much of the work associated with swapping guest memory. A pagefile is mapped into the vmx's address space and calls to the vmmon driver are used to lock MPNs backing this memory as needed by the guest. When memory is released through hypervisor swapping, the pages are dirtied, if necessary, in the vmx's address space and unlocked by vmmon. Should the host OS need to reclaim the backing memory, it does so as if the vmx were any other process: it writes out the state to the backing pagefiles and repurposes the MPN.

For Tesseract, we modified Workstation to support explicit swapping of guest memory. First, we eliminated the pagefile and replaced it with a special VMDK, the block swap store (BSST) into which swapped-out contents are written. The BSST maintains a partial mapping from PPNs to disk blocks tracking the contents of currently swapped-out PPNs. In addition, BSST maintains a table of reference counts on the blocks in the BSST referenced by other guest VDMKs.

Second, we split the process for selecting pages for swapping from the process for actually writing out contents to the BSST and unlocking the backing memory. This split is motivated by the fact that having eliminated duplicate I/Os between hypervisor swapping and guest paging, the system should benefit by both levels of scheduling choosing the same set of pages. The selected swap candidates are placed in a victim cache to "cool down". Only the coldest pages are eventually written out to disk. This victim cache is maintained as a percentage of locked memory by the guest—for our study, 10%. Should the guest access a page in the pool, it is removed from the pool without being unlocked.

When the guest pages out memory, it does so to repurpose a given guest physical page for a new linear mapping. Since this new use will access that guest physical page, one may be concerned that this access will force the page to be swapped in from the BSST first. However, because the guest will either zero the contents of that page or read into it from disk and because the VMM can detect that the whole page will be overwritten before it is visible to the guest, the vmx is able to cancel the swap-in and complete the page locking operation.

4.2 Tracking Memory Pages and Disk Blocks

There are two steps to maintaining a mapping between disk blocks and pages in memory. The first is recognizing the pages read and written in guest and hypervisor I/O operations. By examining scatter-gather lists of each I/O, one can identify when the contents in memory and on disk match. While we plan to maintain this mapping for all associations between guest disks and guest memory, we currently only track the associations between blocks in the BSST and main memory.

The second step is to track when these associations are broken. For guest memory, this event happens when the guest modifies a page of memory. The VMM tracks when this happens by trapping the fact that a writable mapping is required and this information is communicated to the vmx. For device accesses, on the other hand, this event is tracked either through explicit checks in the module which provides devices the access to guest memory, or by examining page-lists for I/O operations that read contents into memory pages.

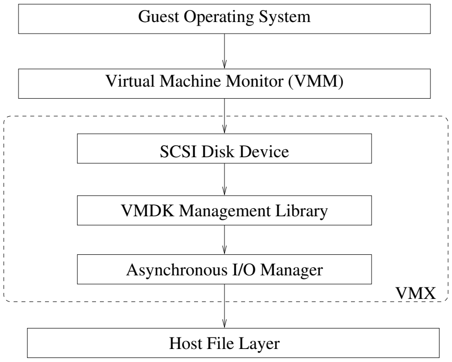


Figure 5: VMware Workstation I/O Stack

4.3 I/O Paths

When the guest OS is running inside a virtual machine, guest I/O requests are intercepted by the VMM, which is responsible for storage adaptor virtualization, and then passed to the hypervisor, where further I/O virtualization occurs.

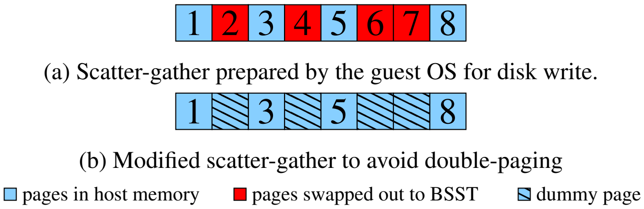
Figure 5 identifies the primary modules in VMware Workstation's I/O stack. Tesseract inspects scatter-gather lists of incoming guest I/O requests in the SCSI Disk Device layer, where a request to the guest VMDK may be updated and extra I/O requests to the BSST may be issued as shown in table 2. Waiting for the completion of all the I/O requests needed to service the original guest I/O request is isolated to the SCSI Disk Device layer as well. When running with defragmentation enabled (see Section 5), Tesseract allocates a pool of worker threads for handling defragmentation requests.

4.3.1 Guest Write I/Os

Guest I/O requests have PPNs in scatter-gather lists. The vmx rewrites the scatter-gather list, replacing guest PPNs with virtual pages from its address space before passing it further to the physical device. Normally, for write I/O requests, if a page was previously swapped, so that PPN does not have a backing MPN, the hypervisor allocates a new MPN and brings page's contents from disk.

With Tesseract, we check if the PPNs are already swapped out to BSST blocks by querying the PPN BSST-block mapping. We then use a virtual address of a special dummy page in the scatter-gather list for each page that resides in the BSST. On completion of the I/O, metadata associated with the guest VMDK is updated to reflect the fact that the contents of guest disk blocks for BSST-resident pages are in the BSST. This sequence allows the guest to page out memory without inducing double-paging.

Figure 6 illustrates how write I/O requests to the guest VMDK are handled by Tesseract. Tesseract recognizes that contents for pages 2, 4, 6 and 7 in the scatter-gather list provided by the guest OS reside in the BSST (Figure 6a). When a new scatter-gather list to be passed to the physical device is formed, a dummy page is used for each BSST resident (Figure 6b).



(a) Scatter-gather prepared by the guest OS for disk write.

(b) Modified scatter-gather to avoid double-paging

Figure 6: The pages swapped out to BSST are replaced with a dummy page to avoid double-paging. Indirections are created for the corresponding guest disk blocks.

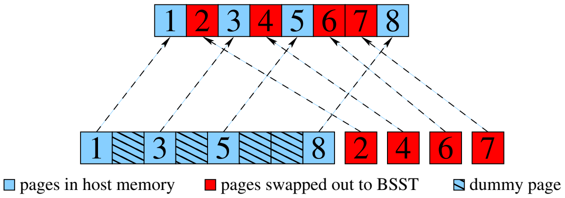


Figure 7: Original guest read request split into multiple reads requests due to holes in the guest VMDK.

4.3.2 Guest Reads I/Os and Guest Disk Fragmentation

Recognizing that data may reside in both the guest VMDK and the BSST is a double-edged sword. On the guest write path it allows us to dismiss pages that are already present in the BSST and thus avoid swapping them in just to be written out to the guest VMDK. However, when it comes to guest reads, the otherwise single I/O request might have to be split into multiple I/Os. This happens when some of the data needed by the I/O is located in the BSST.

Since data that has to be read from the BSST may not be contiguous on disk, the number of extra I/O requests to the BSST may be as high as the number of data pages in the original I/O request that reside in the BSST. We refer to a collection of pages in the original I/O request for which a separate I/O request to the BSST must be issued as a *hole*. Read I/O requests to the guest VMDK which have holes are called *fragmented*.

We modify a fragmented request so that all pages that should be filled in with the data from the BSST are replaced with a dummy page which will serve as a placeholder and will get random data read from the guest VMDK. So in the end for each fragmented read request we issue one modified I/O request to the guest VMDK and N requests to the BSST, where N is the number of holes. After all the issued I/Os are completed, we signal the completion of the originally issued guest read I/O request.

In Figure 7, the guest read I/O request finds disk blocks for pages 2, 4, 6 and 7 located on the BSST, where they are taking non-contiguous space. Tesseract issues one read request to the guest VMDK to get data for pages 1, 3, 5 and 8. In the scatter-gather list sent to the physical device, a dummy page is used as a read target for pages 2, 4, 6 and 7. Together with that one read I/O request to the guest VMDK, four read I/O requests are issued to the BSST. Each of those four requests reads data from one of the four disk blocks in the BSST.

4.3.3 Optimization of Repeated Swaps

In addition to addressing the double-paging anomaly by tracking guest I/Os whose contents exist in the BSST, we also implemented an optimization for back-to-back swap-out requests for a memory page whose contents remain clean. If a page's contents are written out to the BSST, and later swapped back in, we continue to track the old block in the BSST as a form of victim cache. If the same page is chosen to be swapped out again and there has been no intervening write, we simply adjust the reference count for the block copy that is already in the BSST.

4.4 Managing Block Indirection Metadata

Tesseract keeps in-memory metadata for tracking PPN-to-BSST block mappings and for recording block indirections between guest and BSST VMDKs. The PPN-to-BSST block mapping is stored as key-value pair using a hash table. Indirection between guest and BSST VMDKs are tracked in a similar manner.

Tesseract also keeps reference counts for the BSST blocks. When a new PPN-to-BSST mapping is created, the reference count for the corresponding BSST block is set to 1. The reference count is incremented in the write prepare stage for PPNs found to have PPN-to-BSST block mappings. This ensures that such BSST blocks are not repurposed while the guest write is still in progress. Later, on the write completion path, the guest-VMDK-to-BSST indirection is created. The reference count of the BSST blocks is decremented during hypervisor swap in operation. It is also decremented when the guest VMDK block is overwritten by new contents and the previous guest block indirection is invalidated. Blocks with zero reference counts are considered free and reclaimable.

4.4.1 Metadata Consistency

While updating metadata in memory is faster than updating it on the disk, it poses consistency issues. What if the system crashes before the metadata is synced back to persistent storage? To reduce the likelihood of such problems, Tesseract periodically synchronizes the metadata to disk on the same schedule used by the VMDK management library for virtual disk state. However, because reference counts in the BSST and block-indirections in VMDKs are written at different stages in an I/O request, crashes must be detected and a `fsck`-like repair process run.

4.4.2 Entanglement of guest VMDKs and BSST

Once indirections are created between guest and BSST VMDK, it becomes impossible to move just the guest VMDK. To disentangle the guest VMDK, we must copy each block from the BSST to its guest VMDK for which there is an indirection. This can be done both online and offline. More details about the online process are in Section 5.2.

5. Guest Disk Fragmentation

As mentioned in Section 4.3.2, when running with Tesseract, guest read I/O requests might be fragmented in the sense that some of the data the guest is asking for in a single request may reside in both the BSST and the guest VMDK.

The fragmentation level depends on the nature of the workload, the guest OS, and swap activity at the guest and the hypervisor level. Our experiments with SPECjbb2005 [16] showed that even for moderate level of memory pressure as much as 48% of all read I/O requests had at least one hole.

By solving double-paging problem Tesseract significantly reduced write-prepare time of the guest I/O requests since synchronous swap-in requests no longer cause delays. However, a non-trivial overhead was added to read-completion. Indeed, instead of waiting for a single read I/O request to the guest VMDK, the hypervisor may now have to wait for several extra read I/O requests to the BSST to complete before reporting the completion to the guest.

To address these overheads, Tesseract was extended with a *defragmentation* mechanism that improves read I/O access locality and thus reduces read-completion time. We investigated two approaches to implementing defragmentation - *BSST defragmentation* and *guest VMDK defragmentation*. While defragmentation is intended to help reduce read-completion time, it has its own cost. Defragmentation requests are asynchronous and reduce time to complete affected guest I/Os, but, at the same time, they contribute to a higher disk load and in the extreme cases may have an impact on read-prepare times. The defragmentation activity can be throttled on detecting performance bottlenecks due to higher disk load. ESX, for example, provides a mechanism, SIOC, that measures latencies to detect overload and enforce proportional-share fairness [9, 12]. The defragmentation mechanism could participate in this protocol.

5.1 BSST Defragmentation

The idea behind BSST defragmentation is to take guest write I/O requests and use them as a hint of what BSST blocks might be accessed together in a single I/O read request in the future. Given that information we then group together the identified blocks in the BSST.

Figure 8 shows a scatter-gather list of the write I/O request that goes to the guest VMDK. In that request, the contents of pages 2, 4, 6 and 7 is already present in the BSST. As soon as these blocks are identified, a worker thread picks up a reallocation job that will allocate a new block in BSST and will copy the contents of BSST blocks for pages 2, 4, 6 and 7 into that new block.

BSST defragmentation is not perfect. Since blocks are still present in both the guest VMDK and the BSST, extra I/O requests to the BSST can not be entirely eliminated. In addition, BSST defragmentation tries to predict read access locality from write access locality and obviously the boundaries of read requests will not match with the boundaries of the write requests. So each read I/O request that without defragmentation would have required reads from both the guest VMDK and the BSST will still be split into the one which goes to the guest VMDK and one or more requests to the BSST. All this contributes to longer read completion times as shown in table 4.

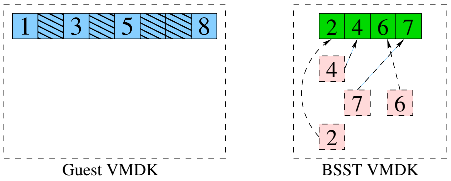


Figure 8: Defragmenting the BSST

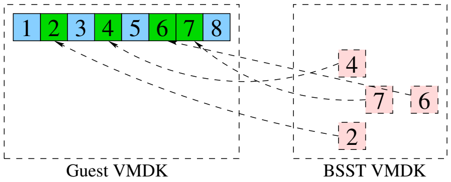


Figure 9: Defragmenting the guest VMDK

However, it is relatively easy to implement BSST defragmentation without worriying too much about data races with the I/O going to the guest VMDK. It can significantly reduce the number of extra I/Os that have to be issued to the BSST to service the guest I/O request as shown in Table 3.

If a guest read I/O request preserves the locality observed at the time of guest writes, we need more than one read I/O request from the BSST only when it hits more than one group of blocks created during BSST defragmentation. Although this is entirely dependent on a workload, one can expect read requests to typically be smaller than write requests, and, so, the number of extra I/O requests to BSST being reduced to one (fits into one defragmented area) or two (crosses the boundary of two defragmented areas) in many cases.

5.2 Guest VMDK Defragmentation

Like BSST defragmentation, guest VMDK defragmentation uses the scatter-gather lists of write I/O requests to identify BSST blocks that must be copied. But unlike BSST defragmentation, these blocks are copied to the guest VMDK. The goal is to restore the guest VMDK to the state it would have had without Tesseract. Tesseract with guest VMDK defragmentation replaces swap-in operations with asynchronous copying from the BSST to the guest VMDK. For example, in Figure 9, blocks 2, 4, 6 and 7 are copied to the relevant locations on the guest VMDK by a worker thread.

We enqueue a defragmentation request as soon as the scatter-gather list of the guest write I/O request is processed and blocks to be asynchronously fetched to the guest VMDK are identified. The defragmentation requests are organized as a priority queue. If a guest read I/O request needs to read data from the block that has not been copied from the BSST, the priority of the defragmentation request that refers to the

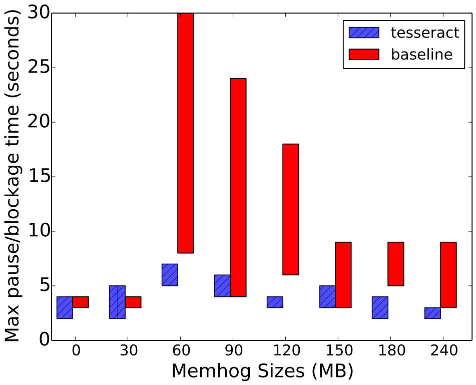


Figure 11: Maximum single pauses observed in SPECjbb instantaneous scoring with various levels of guest memory pressure. Host memory overcommitment is 10%.

block is raised to highest and the guest read I/O request is blocked until copying of all the missing blocks finishes.

While Tesseract with guest defragmentation can have an edge over Tesseract without defragmentation, it is not always a win. With guest defragmentation, before a guest I/O read request has a chance to be issued to the guest VMDK, it may become blocked waiting for a defragmentation request to complete. This may end up being slower than issuing requests to the BSST and the guest VMDK in parallel.

Disentanglement of Guest and BSST VMDKs. Guest defragmentation has an added benefit of removing the entanglement between guest and BSST VMDK. Once there are no block indirections between guest and BSST VMDK, the guest VMDK can be moved easily. This also allows us to disable Tesseract double-paging optimization on-the-fly.

6. Evaluation

We ran our experiments on an AMD Opteron 6168 (Magny-Cours) with 12 1.9 GHz cores, 1.5 GB of memory and a 1 TB 7200rpm Seagate SATA drive, a 1 TB 7200rpm Western Digital SATA drive, and a 128 GB Samsung SSD drive. We used OpenSUSE 11.4 as the host OS and a 6 VCPU 700 MB VM running Ubuntu 11.04. We used Jenkins [2] to monitor and manage execution of the test cases.

To ensure same test conditions for all test runs, we created a fresh copy of the guest virtual disk from backup before each run. For the evaluation we ran SPECjbb2005 [16] that was modified to emit instantaneous scores every second. It was run with 6 warehouses for 120 seconds. The heap size was set to 450 MB. The SPECjbb benchmark creates several warehouses and processes transactions for each of them.

We induced hypervisor-level swapping by setting a max limit on the pages the VM can lock. The BSST VMDK was preallocated. Swap-out victim cache size was chosen to be 10% of the VM's memory size.

Figure 10 and Figures 12–14, represent results from five trial runs. Figure 17 represents results from three trial runs.

6.1 Inducing Double-Paging Activity

To control hypervisor swapping, we set a hypervisor-imposed limit on the machine memory available for the VM. Guest paging was induced by running the SPECjbb benchmark with a working set larger than the available guest memory.

To induce double-paging, the guest must page out the pages that were already swapped by the hypervisor. Since, the hypervisor would choose only the cold pages from the guest memory, we employed a custom memhog that would lock some pages in the guest memory for a predetermined amount of time inside the guest. While the pages were locked by this memhog, a different memhog would repeatedly touch the rest of available guest pages making them "hot". At this point the pages locked by the first memhog are considered "cold" and swapped out by the hypervisor.

Next, memhog unlocks all its memory and the SPECjbb benchmark is started inside the guest. Once the warehouses have been created by SPECjbb, the memory pressure increases inside the guest. The guest is forced to find and page out "cold pages". The pages unlocked by memhog are good candidates as they have not been touched in the recent past.

We used memhog and memory locking in our setup to make the experiments more repeatable. In real world the conditions we were simulating could have been observed, for example, when execution phase shift of an application occurs, or when an application that caches a lot of data in memory and not actively uses is descheduled and another memory intensive application is woken up by the guest.

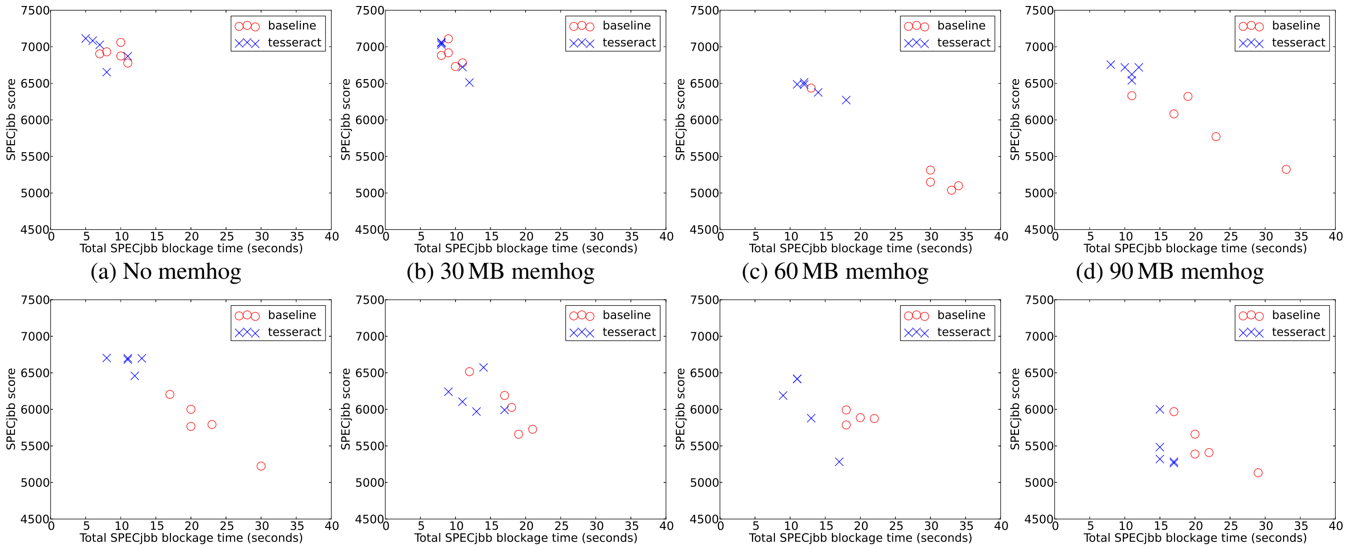
As a baseline we ran with Tesseract disabled. This effectively disabled analysis and rewriting of guest I/O commands so that all pages affected by an I/O command that happened to be swapped out by the hypervisor had to be swapped back in before the command could be issued to disk.

6.2 Application Performance

While it is hard to control and measure the direct impact of individual double-paging events, we use the pauses observed in the instantaneous score of SPECjbb to characterize the application behavior. Depending upon the amount of double-paging activity, the pauses can be as big as 60 seconds in a 120 second run and negatively affect the final score. Often the pauses are associated with garbage collection activity.

6.2.1 Varying Levels Of Guest Memory Pressure

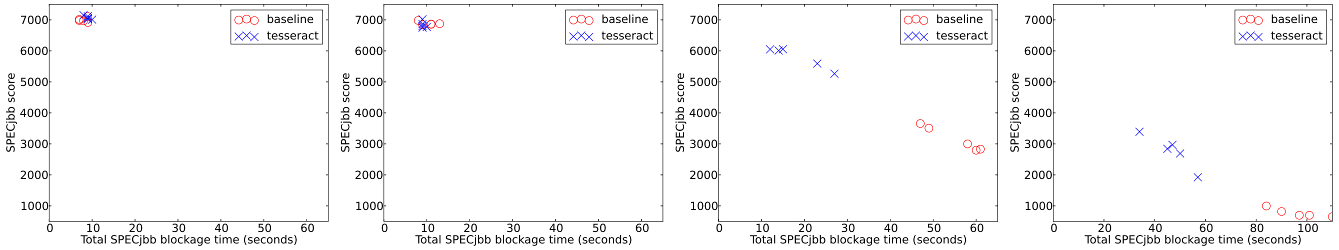
Figure 10 shows scores and pause times for different sizes of memhog inside the guest with 10% host overcommitment. When the guest is trying to page out pages which are swapped by the hypervisor, the latter is swapping them back in and is forced to swap out some other pages. This cascade effect is responsible for increased pause period for the baseline. With Tesseract, however, the pause periods grow at a lower rate. This growth can be explained by longer wait times due to increased disk activity. It should be noted that although the scores are about the same for higher guest memory



(a) No memhog (b) 30 MB memhog (c) 60 MB memhog (d) 90 MB memhog

(e) 120 MB memhog (f) 150 MB memhog (g) 180 MB memhog (h) 240 MB memhog

Figure 10: Trends for the score and pauses in SPECjbb runs with various levels of guest memory pressure. Host overcommitment is 10%.



(a) 0% host overcommitment (b) 5% host overcommitment (c) 15% host overcommitment (d) 20% host overcommitment

Figure 12: Score corresponding to length of max pause in SPECjbb runs with various levels of host overcommitment and 60 MB memhog.

pressure, the total pauses for Tesseract are less than that for the baseline.

Figure 11 shows the effect of increased memory pressure on the length of the biggest application pause. The bars represent the range of max pauses for individual sets of runs. There are five runs in each set. Notice that Tesseract clearly outperforms the baseline. The highest max pause time with Tesseract is 7 seconds, whereas for the baseline it is 30 seconds. This shows that with Tesseract the application is more responsive.

6.2.2 Varying Levels Of Host Memory Pressure

To study the effect of increasing memory pressure by the hypervisor, we ran the application with various levels of host overcommitment with 60 MB memhog inside the guest.

Figure 12 shows the effect of increasing the host memory pressure on the application pauses. For lower host pressure (0% and 5%), the score and pause times for the baseline and Tesseract are about the same. However, for higher memory pressure there is a significant difference in the performance. For example, in the 20% case, the baseline observes pauses in the range of 80–110 seconds. Tesseract on the other hand observes pauses in a much lower range of 30–60 seconds.

Figure 16 shows the max pauses observed by the application as the host memory pressure grows. As before, the max pause is insignificant at lower memory pressure, but with a higher pressure Tesseract clearly outperforms the baseline.

Host (%)	Guest I/Os Issued (#)	I/Os with holes (#)	I/Os 1 – 20 holes (#)	I/Os 21 – 50 holes (#)	I/Os > 50 holes (#)	Double-paging candi-dates (#)
0	1,030	0	0	0	0	0
5	981	537	343	106	88	11,254
10	1,042	661	358	132	171	19,381
15	1,292	766	377	237	152	22,584
20	1,366	981	524	177	280	32,547

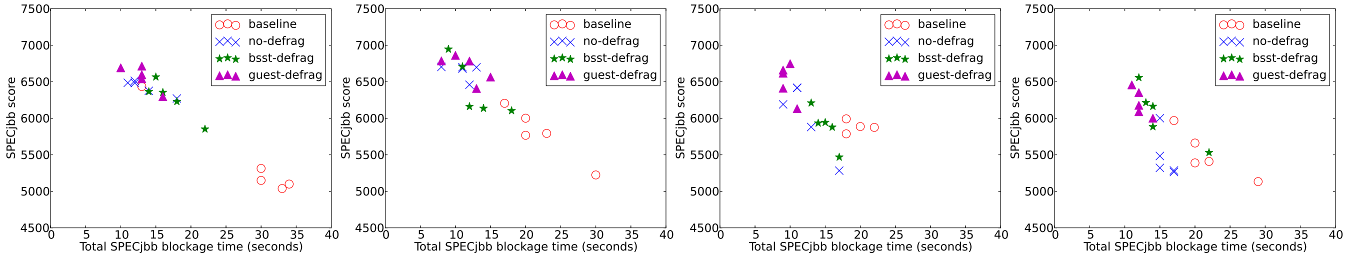
Table 1: Holes in write I/O requests for various levels of host overcommitment. The memhog inside the guest is 60 MB.

6.3 Double-Paging and Guest Write I/O Requests

Table 1 shows why double-paging is affecting guest write I/O performance. As expected, if the host is not experiencing memory pressure, none of the 1,030 guest write I/O requests refer to pages swapped by the hypervisor.

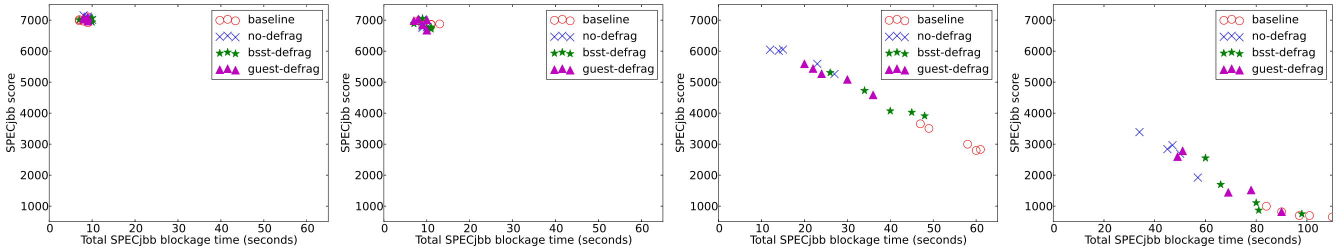
As memory pressure builds up, more and more guest write I/O requests require one or more pages to be swapped in before a write can be issued to the physical disk. All of this contributes to a longer write-prepare time for such a requests.

Consider a setup with 20% host memory is overcommitment. Of 1,366 guest write I/O requests 981 had at least one page that had to be swapped in. Then, 524 guest write I/O requests needed between 1 and 20 swap-in requests completed



(a) 60 MB memhog (b) 120 MB memhog (c) 180 MB memhog (d) 240 MB memhog

Figure 13: Score and pauses in SPECjbb runs under various defragmentation schemes with 10% host overcommitment.



(a) 0% host overcommitment (b) 5% host overcommitment (c) 15% host overcommitment (d) 20% host overcommitment

Figure 14: Score and pauses in SPECjbb under various defragmentation schemes with varying host overcommitment and 60 MB memhog

Host (%)	Guest I/Os Issued (#)	I/Os w/ Holes (#)	Total Holes (#)	Total I/Os Issued (#)	Score
0	5,152	0	0	5,152	7,010
5	5,230	708	1,675	6,197	6,801
10	5,206	2,161	5,820	8,865	6,271
15	4,517	2,084	6,990	9,423	6,048
20	5,698	2,739	11,854	14,813	2,841

Table 2: Holes in read I/O requests for Tesseract without defragmentation for various levels of host overcommitment. The memhog inside the guest is 60 MB.

Defrag Strategy	Reads w/o Holes (#)	Reads w/ Holes (#)	Total Holes (#)	BSST Reads Issued (#)	Total Reads Issued (#)	Defrag I/Os	
						Reads Issued (#)	Writes Issued (#)
No-Defrag	3,025	1,203	2,456	2,456	6,684	0	0
BSST	2,946	1,235	2,889	1,235	5,416	12,674	616
Guest	3,909	0	0	0	3,909	11,538	11,538

Table 3: Total I/Os with BSST and guest defragmentation.

by the hypervisor in order to proceed, 177 needed between 21 and 50 swap-in requests completed, and, finally, 280 guest write I/O requests needed more than 50 swap-in requests.

6.4 Fragmentation in Guest Read I/O Requests

Table 2 quantifies the amount of extra read I/O requests that has to be issued to the BSST if defragmentation is not used.

If the host is not under memory pressure there is no hypervisor level swapping activity and all 5,152 guest read I/O requests can be satisfied without going to the BSST.

At higher levels of memory pressure, the hypervisor starts swapping pages to disk. Tesseract detects pages in guest write I/O requests that are already in the BSST to avoid swap-in requests for such pages. The amount of work saved by Tesseract on the write I/O path is quantified in Table 1.

When host memory is 20% overcommitted we can see that out of 5,698 guest read I/O requests 2,739 will require extra read I/Os to be issued to read data from the BSST. The total number of such an extra I/Os to the BSST was 11,854, which made the total number of read I/O requests issued to both the guest VMDK and the BSST equal 14,813.

6.5 Evaluating Defragmentation Schemes

Figures 13 and 14 show the impact of using BSST and guest VMDK defragmentation on SPECjbb throughput, while Figures 15 and 16 give an insight into SPECjbb responsiveness.

Guest defragmentation performs better than the baseline in all situations and is as good or better than BSST defragmentation. With low levels of host memory overcommitment Tesseract with guest VMDK defragmentation secures better SPECjbb scores than Tesseract without defragmentation and performs on par in responsiveness metrics.

With increasing host memory overcommitment, Tesseract without defragmentation starts outperforming Tesseract with either of the defragmentation schemes in both the application throughput and responsiveness as the total and maximum pause times grow slower for the no-defragmentation case. This is due to the fact that at a higher level of hypervisor level swapping, guest read I/O becomes more and more fragmented and pending defragmentation requests become a bottleneck leading to longer read completion times.

Table 3 shows the I/O overheads of the two defragmentation schemes compared to Tesseract without them. For this table, 3 runs with similar scores and similar number of guest read I/O requests were selected. With BSST VMDK defragmentation enabled, Tesseract was able to reduce the number of synchronous I/O requests to BSST VMDK from 2,889 (2.23 reads per I/O with holes on average) to 1,235 (1 read per I/O with holes). To do BSST VMDK defragmentation, 12,674 asynchronous reads from BSST VMDK and 616 asynchronous writes to BSST VMDK had to be issued. This num-

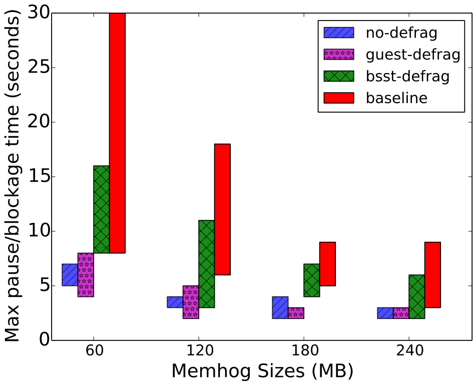


Figure 15: Comparing max single pause for SPECjbb under various defragmentation schemes with 10% host memory overcommitment.

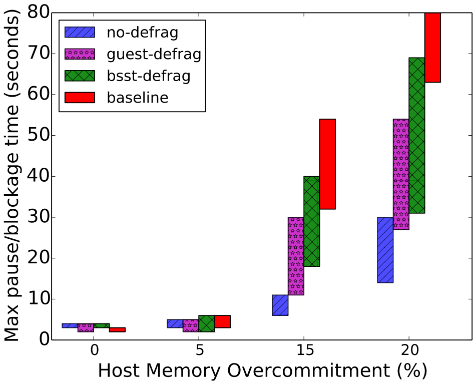


Figure 16: Comparing max single pause for SPECjbb under various defragmentation schemes with various levels of host memory overcommitment. Memhog was sized at 60 MB.

ber of writes equals the number of guest write I/O requests with holes. Guest VMDK defragmentation eliminated holes in guest read I/O requests entirely, so there were no guest-related reads from BSST VMDK. To achieve this, 11,538 asynchronous reads from BSST VMDK and the same number of asynchronous writes to the guest VMDK were issued.

6.6 Using SSD For Storing BSST VMDK

SSDs have drastically better performance over magnetic disk in terms of lower seek times for random reads. However, their relatively higher cost keeps them from getting into mainstream server market. They are used in smaller units for boosting performance. One potential application for SSDs in servers is as a hypervisor swap device allowing for higher memory overcommitment as the cost of swapping is reduced.

In our experiment, we placed the BSST VMDK on a SATA SSD. Figure 17 shows the performance of the baseline and Tesseract. At lower memory pressure, there is no difference in the performance, but as the memory pressure increases, at both guest and hypervisor level, Tesseract starts to show benefits over the baseline.

6.7 Overheads

I/O Path	Baseline	No-defrag	BSST defrag	Guest defrag
Read prepare	0	37	30	109
Read completion	0	232	247	55
Write prepare	24,262	220	256	265
Write completion	0	49	91	101

Table 4: Average read and write prepare/completion times for baseline and for tesseract with and without defragmentation (in microseconds). The host overcommitment was 10% while the memhog size was 60 MB.

I/O Path Overhead Table 4 presents Tesseract overheads on I/O paths. The average overhead per I/O is on the order of microseconds. Read prepare time for guest defragmentation is higher than the others due to the contention on guest VMDK during defragmentation. At the same time, the read completion time for guest defragmentation case is much lower than the other two cases as there are no extra reads going to the BSST. On the write I/O path, the defragmentation schemes have larger overhead. This is due to the background defragmentation of the disks which is kicked off as soon as the write I/O is scheduled.

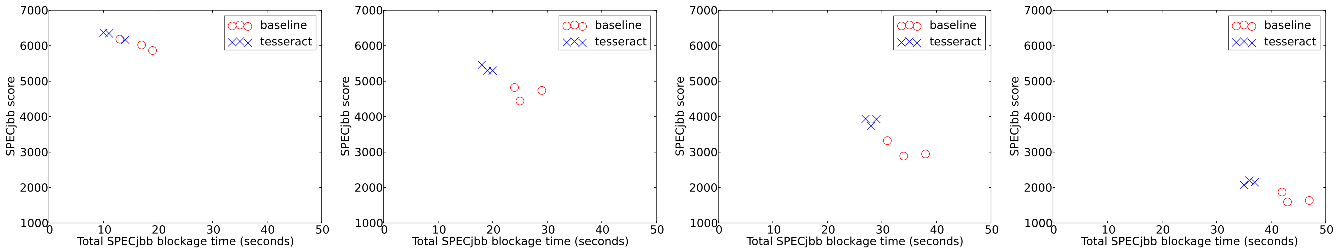
Memory Overhead Per Section 4.4, Tesseract maintains in-memory metadata for three purposes: tracking (a) associations between PPN and BSST blocks; (b) reference counts for BSST blocks; and (c) indirections between guest VMDK and BSST VMDK. We use 64 bits to store a (4 KB) block number. To track associations between PPN and BSST blocks we reuse MPN field in page frames maintained by the hypervisor so there is no extra memory overhead here. In general case where associations between PPN and blocks in guest VMDK have to be tracked we will need a separate memory structure with a maximum overhead of 0.2% of VM's memory size. Each BSST block's reference count requires 4 bytes per disk block. To optimize the lookup for free/available BSST blocks, a bitmap is also maintained with one bit for each block. The guest VMDK to BSST VMDK indirection metadata requires 24 bytes for each guest VMDK block for which there is a valid indirection to BSST. A bitmap similar to that for BSST is maintained for guest VMDK blocks to determine if an indirection to BSST exists for a given guest VMDK block.

7. Related Work

Our project intersects three areas. The first is that of uncooperative hypervisor swapping and the double-paging problem. The second concerns the tracking of associations between guest memory and disk state. The third concerns memory and I/O deduplication.

7.1 Hypervisor Swapping and Double Paging

Recent, concurrent work by Amit, Tsafrir, and Schuster [5] systematically explores the behavior of uncooperative hypervisor swapping and implement an improved swap subsystem for KVM called VSwapper. The main components of their implementation are the Swap Mapper and the False Reader Preventer. The paper identifies five primary causes for performance degradation, studies each, and offers solutions to address them. The first, "silent swap writes", corresponds to



(a) 15% host overcommitment (b) 20% host overcommitment (c) 25% host overcommitment (d) 30% host overcommitment

Figure 17: Tesseract performances with BSST placed on an SSD disk. The memhog size was 60 MB.

our notion of guest-I/O–swap optimization which we do not yet support because we do not support reference-counting on blocks in guest VMDKs. The second and third, "stale swap reads" and "false swap reads", and their solutions are similar to the existing ESX optimizations that cancel swap-ins for memory pages that are either overwritten by disk I/O or by the guest. For "silent swap writes" and "stale swap reads", the Swap Mapper uses the same techniques Tesseract does to track valid associations between pages in guest memory and blocks on disk. Their solution to "false swap reads", the False Reader Preventer, is more general, however, because it supports the accumulation of successive guest writes in a temporary buffer to identify if a page is entirely overwritten before next read. The last two, "decayed swap sequentiality" and "false page anonymity", are not issues we consider. In their investigation, they did not observe double-paging to have much impact on performance. This is likely due to the fact that they followed guidelines from VMware and provisioned guests with enough VRAM that guest paging was uncommon and most of the experiments were run with a persistent level of overcommitment. We view their effort as complementary to ours.

The double-paging problem was first identified in the context of virtual machines running on VM/370 [6, 15]. Goldberg and Hassinger [6] discuss the impact of increased paging when the virtual machine's address exceeds that with which it is backed. Seawright and MacKinnon [15] mention the use of *handshaking* between the VMM and operating system to address the issue but do not offer details.

The Cellular Disco project at Stanford describes the problem of paging in the guest and swapping in the hypervisor [7, 8]. They address this double-paging or redundant paging problem by introducing a virtual paging device in the guest. The paging device allows the hypervisor to track the paging activity of the guest and reconcile it with its own. Like our approach, the guest paging device identified already swapped-out blocks and creates indirections to these blocks that are already persistent on disk. There is no mention of the fact that these indirections destroy expected locality and may impact subsequent guest read I/Os.

Subsequent papers on scheduling memory for virtual machines also refer in passing to the general problem. Waldspurger [17], for example, mentions the impact of double-paging and advocates random selection of pages by the hypervisor as a simple way to minimize overlap with page-selection by the guest. Others projects, such as the Satori project [13], use double-paging to advocate against any mechanism to swap guest pages from the hypervisor.

Our approach differs from these efforts in several ways. First, we have a system in which we can—for the first time—measure the extent to which double-paging occurs. Second, we have an approach that directly addresses the problem of double-paging in a manner *transparent* to the guest. Finally, our techniques change the relationship between the two levels of scheduling: by reconciling and eliding redundant I/Os, Tesseract encourages the two schedulers to choose the *same* pages to be paged out.

7.2 Associations Between Memory and Disk State

Tracking the associations between guest memory and guest disks has been used to improve memory management and working-set estimation for virtual machines. The Geiger project [10], for example, uses paravirtualization and intimate knowledge of the guest disks to implement a secondary cache for guest buffer-cache pages. Lu et al. [11] implement a similar form of victim cache for the Xen hypervisor.

Park et al. [14] describe a set of techniques to speed live-migration of VMs. One of these techniques is to track associations between pages in memory and blocks on disks whose contents are shared between the source and destination machines. In cases where the contents are known to be resident on disk, the block information is sent to the destination in place of the memory contents. In the paper, the authors describe techniques for maintaining this mapping both through paravirtualization and through the use of read-only mappings for fully virtualized guests.

7.3 I/O and Memory Deduplication

The Satori project [13] also tracks the association between disk blocks and pages in memory. It extends the Xen hypervisor to exploit these associations, allowing it to elide repeated I/Os that read the same blocks from disk across VMs immediately sharing these pages of memory across those guests.

Originally inspired by the Cellular Disco and Geiger projects, Tesseract shares much in common with these approaches. Like many of them, it tracks valid associations between memory pages and disk blocks that contain identical content. Like Park et al., it employs techniques that are fully transparent to the guest allowing it to be applied in a wider set of contexts. Unlike the Satori projects which focused on eliminating redundant read operations across VMs, Tesseract uses that mapping information to deduplicate I/Os *from* a specific guest *and* its hypervisor. As such, our approach complements and extends these others.

8. Future Work

We plan to extend our prototype to support reconciliation and elimination of all redundant I/Os whether from guest operations or the hypervisor. For this, we need to do a cleaner job of restructuring the VMDK structure to support indirections and reference-counting of blocks. Following this redesign to its logical conclusion, one would structure a set of guest VMDKs as (thinly-provisioned) maps from guest block identifiers to a general sea-of-blocks in some datastore.

Second, we plan to investigate the interaction of ballooning and hypervisor-level swapping. Ballooning is often used first and more frequently than swapping. Used in this order, one typically does not see much double paging because the balloon has already applied pressure within the guest before swapping commences. However, over a longer period of oscillation, one can imagine double paging occurring because of later inflations of ballooning in the guest.

Finally, our experience in this project has led us to question the existing interface for issuing I/O requests with scatter-gather lists. Given that the underlying physical organization of the disk blocks can differ significantly from the virtual disk structure, it makes little sense for a scatter-gather list to require that the target blocks on disk be contiguous. Having a more flexible structure may allow I/Os to be expressed more succinctly and to be more effective at communicating expected relationships or locality among those disk blocks.

9. Conclusion

We present Tesseract, a system that directly and transparently addresses the double-paging problem. We have described how this issue may arise in the context of guests and hypervisors as each attempts to overcommit its memory resources. We have outlined the design and implementation of Tesseract describing how it reconciles and eliminates redundant I/O activity between the guest's virtual disks and the hypervisor swap subsystem by tracking associations between the contents of pages in guest memory and those on disk. We have identified how, implemented naively, Tesseract interferes with the guest's notion of locality and we have offered two approaches to recover that locality through defragmentation. We have presented the first empirical data on the cost of the double-paging problem. Finally, we have outlined a number of directions we plan to investigate.

10. Acknowledgements

We thank Maxime Austruy for his help with the hosted I/O path and discussions about the BSST. We owe much to Joyce Spencer and Jerri-Ann Meyer for continued support of the project, and to Ron Mann who got telemetry on overcommitment from shipped hosted products. The paper has been improved much by thoughtful comments and feedback from Gene Cooperman, from anonymous reviewers, and especially from our shepherds, Dan Tsafrir and Nadav Amit.

References

[1] VMware vSphere hypervisor. http://www.vmware.com/go/ESXiInfoCenter.

[2] Jenkins. http://jenkins-ci.org.

[3] VMware workstation. http://www.vmware.com/products/workstation.

[4] O. Agesen. US patent 8380939: System/method for maintaining memory page sharing in a virtual environment, 2011.

[5] N. Amit, D. Tsafrir, and A. Schuster. Vswapper: A memory swapper for virtualized environments. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-XIX, 2014.

[6] R. P. Goldberg and R. Hassinger. The double paging anomaly. In *Proceedings of the May 6-10, 1974, national computer conference and exposition*, AFIPS '74, pages 195–199, 1974.

[7] K. Govil. *Virtual clusters: resource management on large shared-memory multiprocessors*. PhD thesis, Stanford University, 2001.

[8] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 18:229–262, August 2000.

[9] A. Gulati, I. Ahmad, and C. A. Waldspurger. Parda: Proportional allocation of resources for distributed storage access. In *Proccedings of the 7th Conference on File and Storage Technologies*, FAST '09, pages 85–98, 2009.

[10] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: monitoring the buffer cache in a virtual machine environment. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-XII, 2006.

[11] P. Lu and K. Shen. Virtual machine memory access tracing with hypervisor exclusive cache. In *Proceedings of the 2007 USENIX Annual Technical Conference*, 2007.

[12] P. Manning and J. Dieckhans. Storage i/o control technical overview and considerations for deployment. 2010. URL http://www.vmware.com/files/pdf/techpaper/VMW-vSphere41-SIOC.pdf.

[13] G. Miłós, D. G. Murray, S. Hand, and M. A. Fetterman. Satori: enlightened page sharing. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, 2009.

[14] E. Park, B. Egger, and J. Lee. Fast and space-efficient virtual machine checkpointing. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '11, 2011.

[15] L. Seawright and R. MacKinnon. VM/370 - a study of multiplicity and usefulness. *IBM Sys. Jrnl*, 18(1):4–17, 1979.

[16] Standard Performance Evaluation Corporation. SPECjbb2005. http://www.spec.org/jbb2005.

[17] C. A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36:181–194, Dec. 2002.