Karlsruher Institut für Technologie

# Using I/O-based Hints to Make Memory-Deduplication Scanners More Efficient

Diploma Thesis
by

cand. inform. Fabian Franz

at the Department of Computer Science

| | |
|---|---|
| Supervisor: | Prof. Dr. Frank Bellosa |
| Supervising Research Assistant: | Dipl.-Inform. Konrad Miller |

Day of completion: July 29th, 2012

I hereby declare that this thesis is my own original work which I created without illegitimate help by others, that I have not used any other sources or resources than the ones indicated and that due acknowledgment is given where reference is made to the work of others.

Karlsruhe, July 29th, 2012

# Deutsche Zusammenfassung

Der primäre Engpass für die Konsolidierung von mehreren Virtuellen Maschinen (VMs) auf einem physischen Computer ist die begrenzte Größe der Hauptspeichers. Speicher kann nicht beliebig aufgerüstet werden und ist im Vergleich zu mehr CPU Geschwindigkeit immer noch eine teure Resource. Moderne Betriebssysteme benutzen *Paging* um Applikationen mehr virtuellen Speicher zuzuteilen als physisch vorhanden ist.

Um Speicher zu sparen werden physische Speicherseiten mit identischem Inhalt unter Nutzung von *copy-on-write* dedupliziert, d.h. schreibgeschützte virtuelle Seiten zeigen auf die gleiche physische Seite. Dies ist ein übliches Verfahren und wird meist in Kombination mit `mmap` und `fork` benutzt. Zum Beispiel werden die Seiten einer dynamischen Bibliothek dedupliziert, so dass diese nur einmal Speicher im System belegt.

Eine virtuelle Maschine ist eine "Black Box"für den Host auf dem sie läuft. Der Host hat keine Information über den Inhalt des Speichers der VM und kann diesen demnach nicht deduplizieren.

Lösungsansätze für dieses Problem sind Paravirtualisierung und *Memory Scanner* (Speicherscanner). In der Paravirtualisierung gibt das Gast-Betriebssystem in der VM dem Host Informationen über die Speicherseiten und ermöglicht so dem Host den Speicher zu deduplizieren, freizugeben oder anderweitig zu nutzen. Memory Scanner dagegen katalogisieren die Speicherseiten des Gastes basierend auf ihrem Inhalt und finden so Duplikate.

Memory Scanner sind begrenzt durch eine gewisse *Scan Rate* mit der sie den Speicher auf Duplikate untersuchen, z.B. 100 Seiten alle 100 ms. Mit dieser Scan Rate kann ein Memory Scanner aber keine kurz-lebigen Deduplikationsmöglichkeiten finden, da die Seiten sich wieder geändert haben bevor der Scanner sie findet. Eine höhere Scan Rate, wo der Scanner z.B. eine ganze CPU voll auslastet, ist nicht gerechtfertigt, da der Aufwand in keinem Verhältnis zum Nutzen steht.

Benchmarks, die als Teil dieser Arbeit durchgeführt wurden, zeigen dass 80%

aller Deduplikationsmöglichkeiten zwischen 30 s und 5 min existieren. Es existiert also ein großes Deduplikationspotential, dass von Memory Scannern mit der Standard Scan Rate nicht gefunden wird. Zwischen 64% und 93% der gemeinsam nutzbaren Seiten zwischen VMs stammen aus dem *Page cache* der Gastbetriebssysteme [13].

Diese Arbeit führt einen neuen Ansatz ein, der *I/O Requests* im VFS des Host Betriebssystems abfängt und die Adressen der zugehörigen Speicherbereiche in der Form von Hints an Memory Scanner weiter gibt. Dieser Ansatz wird im folgenden KSM++ genannt. Mit dieser Information kann KSM++ Speicher früher scannen als traditionelle Memory Scanner und findet so Deduplikationsmöglichkeiten die ihren Ursprung im Page Cache der Gastbetriebssysteme haben.

KSM++ wurde als Erweiterung von KSM (Kernel Samepage Merging) [1] im Linux Kernel als Prototyp implementiert und evaluiert. Im Gegensatz zur Paravirtualisierung muss das Gastbetriebssystem nicht geändert werden.

Die Evaluationsergebnisse zeigen, dass KSM++ doppelt soviel Speicher deduplizieren kann wie KSM und Deduplizierungsmöglichkeiten, bei einer Scan Rate von 1000 Seiten pro Sekunde, bis zu 8 Minuten früher findet als KSM.

Als Fazit ist KSM++ eine nützliche Erweiterung um Memory Scanner effektiver und effizienter zu machen.

# Acknowledgments

This work is dedicated to my wife Prisca.

Her love, patience and support during this period have been invaluable.

I also want to thank my parents for their support.

# Contents

# Chapter 1

# Introduction

One of the major developments in the current IT landscape is cloud computing. Virtualization of Server and Desktop machines have made it possible to scale server farms easily and add new machines into the "cloud". Servers are no longer bound to one single physical machine, but can be moved around from one physical machine to another, duplicated and distributed to other regions of the world while maintaining strong service isolation.

The limited main memory size of the hosting server (host) is the primary bottleneck for consolidating more virtual machines (VMs) on one physical machine. While the speed of the CPU (Central Processing Unit) and the number of CPUs on a server have increased enormously, memory is still limited and hence an expensive resource. A modern operating system (OS) provides more virtual memory to the applications than is physically available. In the context of virtual machines this is called overcommitting memory and enables the host to run more VMs. However, when overcommitting memory in virtualized environments, the host cannot decide which memory pages are currently not used by the guests. Therefore, it needs to fall back to swapping in situations of memory pressure.

One solution to decrease memory pressure on the host other than swapping, is to consolidate memory by sharing pages having the same content using a copy-on-write (COW) mechanism. This kind of memory sharing is a common procedure in many operating systems. For example the contents of dynamic libraries are shared so that these only exist once in the system memory. A virtual machine is a black box to the host. So even though there may be plenty of redundant data between VMs, the host does not know where pages with the same content reside in memory. This is the so-called semantic gap [6]. One solution to bridge this gap and to find sharing opportunities is to use memory scanners. A memory scanner

scans the memory of the VMs periodically to find all pages that have the same content in order to share them.

However, memory scanners are limited by the scan rate at which they are running and they trade computational overhead with deduplication success. In order to find short-lived sharing opportunities, the scan rate needs to be set really high – i.e. by having the memory scanner occupy an entire CPU. In consequence memory scanners consume too many CPU cycles to justify the need for finding short-lived sharing opportunities. Benchmarks conducted in this thesis show that up to 80% of all sharing opportunities are short-lived (see Section 5.4) and exist between 30 sec and 5 min. A classic memory scanner will not find those opportunities using the default scan rate. Kloster et al. have reported that on average between 64% and 93% of the shareable pages between VMs originate from the page cache [13].

In this thesis it is proposed to intercept read and write requests (I/O) in the host and provide the location of the associated memory areas as hints to memory scanners. This enables the memory scanner to scan areas of I/O activity – pages stemming from the page cache of the VMs – earlier than if the memory was scanned in a linear way.

Benchmarks conducted in this thesis show that by using this approach memory scanners are able to find six times more sharing opportunities that stem from the page cache of virtual machines. Further, in this work it is claimed that memory scanners using this extension both need to scan less pages before finding a merge and can share up to twice as much memory – all using the same scan-rate.

The approach was implemented in Linux' Kernel Samepage Merging (KSM) with QEMU [3] as VMM (Virtual Machine Monitor) to verify the claims of this work. KSM is a popular memory scanner included in the Linux mainline kernel. In this work KSM was altered to provide I/O hints and process these hints interleaved with the linear scan. This approach and the prototype implementation are called KSM++. KSM++ issues hints for all pages that are target of a read or write operation in the Virtual File System (VFS) layer of the host. When a guest makes a read or write request to its Virtual Disk the Virtual DMA (Direct Memory Access) Controller translates this request into a corresponding VFS read or write request. Such the pages KSM++ intercepts in the host are stemming from the guests page cache.

KSM++ intercepts I/O requests inside the VFS layer. The requests stem from virtual machines or applications and they can target physical disks or network file systems (NFS). To use KSM++ only the host OS needs to be modified. Therefore, the solution described in this thesis is independent of the underlying guest OS and will work as long as there is sharing potential between VMs. Both the effective-

ness and efficiency of the approach have been evaluated using a standard set of benchmarks and the result is that KSM++ is able to merge hinted pages minutes earlier than KSM and can reach close to the optimum sharing possibilities.

## Organization of the thesis

The remainder of the thesis is organized in the following way:

Chapter 2 introduces the background necessary for understanding this work and presents related work from the "paravirtualization" research field. The proposed approach is discussed in detail as an abstract concept in Chapter 3. KSM++, the prototype based on this design, and some of the implementation challenges are presented in Chapter 4. In Chapter 5 the prototype implementation is evaluated to verify the claims of this thesis. Chapter 6 concludes this work with a summary of the benchmark results and presents possibilities for future research work.

# Chapter 2

# Background

This chapter introduces the basics considered facts in this work. Further, the question "How can memory be saved (in virtualized environments)?" is answered by investigating research provided by related work.

## 2.1 Page Sharing

This section discusses the concept of virtual memory and how paging can be used by modern operating systems to share memory in traditional and virtualized environments.

### Virtual Memory

Modern operating systems virtualize memory to map physical memory on a page by page basis. The Memory Management Unit (MMU) provides hardware support to translate mapped virtual addresses to physical ones. This allows non-linear physical regions to be mapped to one linear virtual region. Not all virtual pages need to be mapped to physical pages and several different virtual pages can be mapped to one physical page. This enables the OS to share one physical page across several virtual pages.

The MMU also supports the operating system with the protection of physical pages. This makes it possible to map pages copy-on-write. In case the MMU finds a violation of the page protection (a write request to a read-only page or a read request to a not mapped page) the MMU triggers a fault in the OS. The OS

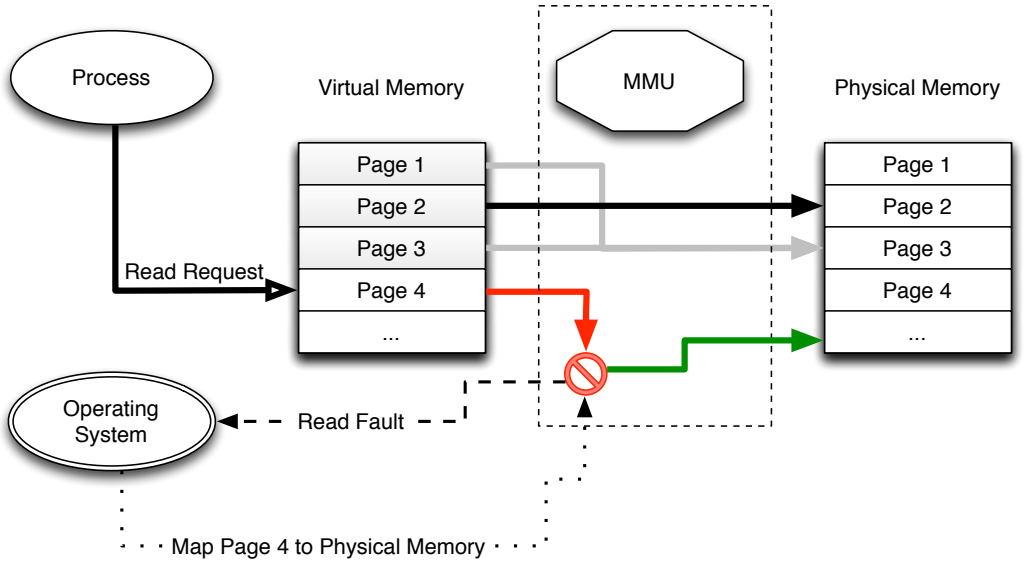


Figure 2.1: Both virtual and physical memory is divided into pages with a fixed page size. The MMU translates accesses to virtual addresses into physical pages. Virtual pages 1 and 3 both map to physical page 3. The MMU cannot map virtual page 4, so it triggers a fault in the OS. The OS maps virtual page 4 to a physical page and the MMU can now resolve the access.

then takes appropriate measures to resolve the fault (see Figure 2.1). If a page is marked copy-on-write, the operating system resolves the fault by allocating a new page, copying the content of the old page to the new page, marking the new page read-write and mapping the virtual address, which led to the fault, to this new page. To resolve a read fault, the OS could for example read from a file to populate the page or if it cannot map the page kill the process (as a last resort).

Virtual memory makes it possible for a host to overcommit memory to applications – i.e. to provide them with more memory than physically exists. In situations of high memory pressure – for example when an application tries to use all of its allocated memory – the OS will swap out currently unused pages to disk.

In conclusion the concept of virtual memory allows saving memory by sharing virtual pages, i.e. by mapping several different virtual pages to one physical page and mapping them copy-on-write.

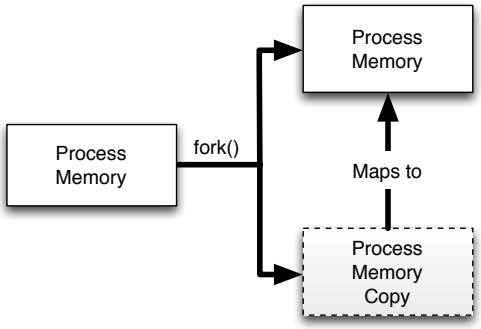


Figure 2.2: The memory of a duplicated process is shared with the original and both are mapped copy-on-write.

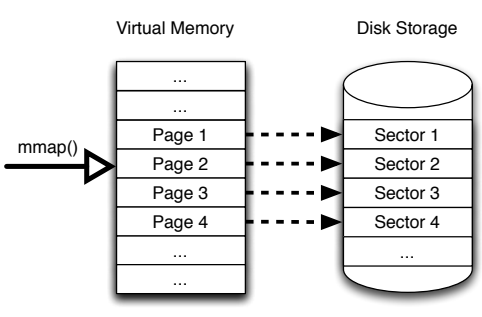


Figure 2.3: The mmap'ed memory area is backed by a file on the disk storage. A page access triggers a read on the file.

## Traditional Page Sharing

Two ways are traditionally used to save memory in state of the art UNIX based operating systems: First, the address space of a duplicated process is shared copy-on-write across all "clones". Second, shared libraries for example are mapped from disk via the `mmap` system call and are shared read-only across all processes, so that each library only exists once in the system memory.

Using copy-on-write semantics for process duplication makes this a very cheap operation: When a process calls the `fork` system call in order to duplicate the process the address space of the process is marked read-only. Then a new virtual address space is created for the clone and mapped (read-only) to the same physical pages as the original address space (see Figure 2.2). As soon as one of the two newly created processes writes to a shared page, a fault is triggered, the OS then copies the page and the process receives its own private writable copy of the page.

Pages can be deliberately mapped as being shared by utilizing the `mmap` system call. A common usage is to map files into memory in order to be able to access them randomly without having to load the whole file into memory. A read on a non-mapped page causes the OS to fault the page in from the backing file (see Figure 2.3). This is most commonly utilized with shared libraries and executable files, because those have the highest sharing potential. However, mmap has one important limitation: It can only share the memory if the pages originate from the same file (with the same inode). As soon as a file is copied and the inode changes, mmap on those files will create duplicate pages in the buffer cache of the OS. The terms buffer cache and page cache are used interchangably in this work.

## Page Sharing in Virtualized Environments

Unfortunately the aforementioned methods cannot be applied directly to virtualized environments. In a virtual environment the host has no "knowledge" of the inner state of its guests. This is by design and one of the characteristics of virtualization, the so-called semantic gap [6]. Therefore the host can for example not share a library between guest A and guest B (see Figure 2.4). And if the same process is running on two guests at the same time, the host won't know – even if both address spaces are marked read-only.

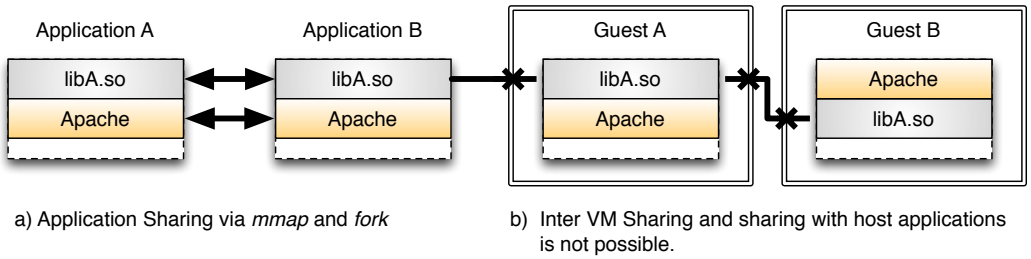


Figure 2.4: a) Memory is shared between host applications. b) Memory cannot be shared across VMs or from a VM application with a host application due to the `semantic gap`. The host has no knowledge of the internal workings of its guests.

Despite these shortcomings there exist approaches to make use of traditional page sharing techniques for virtual machines: Android's Cygote [21] shares the Dalvik VM and the core libraries across all processes by using COW semantics. Other approaches use copy-on-write to share the initial image of whole guests [15, 25]. The drawback of these other approaches is that only the initial pages after the `fork` are shared. While normal applications usually retain a large sharing potential, the same is not true for virtual machines. Virtual machines might start with a large similarity, then have no similarity at all and then again have a large similarity.

As soon as there is activity on fully shared virtual machines, the number of shared pages will gradually decrease. This is especially true for I/O centric workloads, where the page cache of a virtual machine can grow to up to 90% of the free memory. Even if the same I/O was read on the host, it would still create duplicate pages in the page caches of the different guests.

Pages stemming from a read or write operation on disk are usually stored in the buffer cache of the OS. In the case of virtualized environments, there exist a cache hierarchy: The host has a buffer cache and each guests has its own private buffer

cache as well. Hence the same file read by two VMs will lead to duplicate pages in the buffer caches of the host and both guests (see Figure 2.5). Therefore the buffer caches may have high redundancy and such big potential for saving memory.

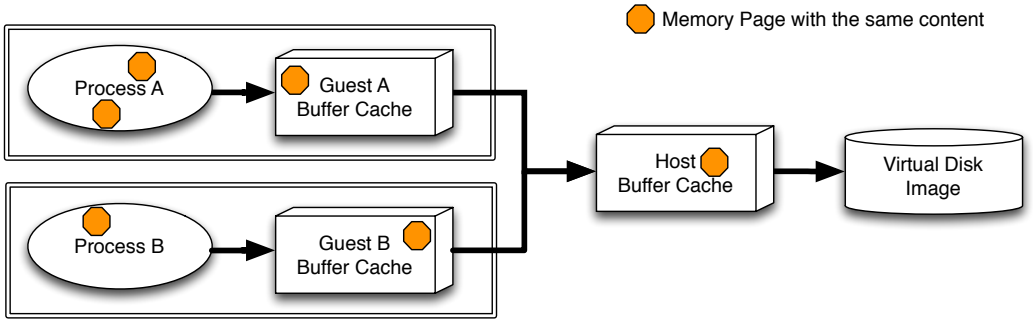


Figure 2.5: The cache hierarchy creates duplicate entries in the buffer caches of the host and its guests.

To mitigate the problem of duplicate cache entries in host and guest, guest and/or host buffer caches can be turned off. Prateek Sharma et al. [24] reported that turning off buffer caches leads to decreased I/O performance. They suggest to remove pages cached in the guest periodically from the host cache instead.

Another source of duplicate memory is the host buffer cache itself. Virtual Machines usually use a seperate VDI (Virtual Disk Image). Even though there might be lots of redundant data in between these images, a read on these images will lead to duplicate memory pages in the host buffer cache.

To solve this problem special copy-on-write images can be used in combination with mmap, where the VDI is split into an original read-only part and a private read-write part. A write on the original leads to a copy of the block to the private part. The principle is the same as for copy-on-write for memory pages.

Z. Zhang, H. Chen, and H. Lei [29] were successful in saving memory via mmap by turning host page caches on only for the original image and guest page caches only for the private part without reduced performance.

In conclusion saving memory in virtualized environments is possible using the same techniques applied in traditional operating systems, but the full sharing potential cannot be exploited due to the semantic gap.

## 2.2 Paravirtualization

One solution to close the semantic gap in order to save memory is to use paravirtualization. In paravirtualized environments the guests are modified to cooperate with the host. The modified guests will provide the host with information about the inner state, so that the host can take appropriate measures to share memory pages between VMs and between guest and host. The semantic gap and using a paravirtualized guest is depicted in Figure 2.6.

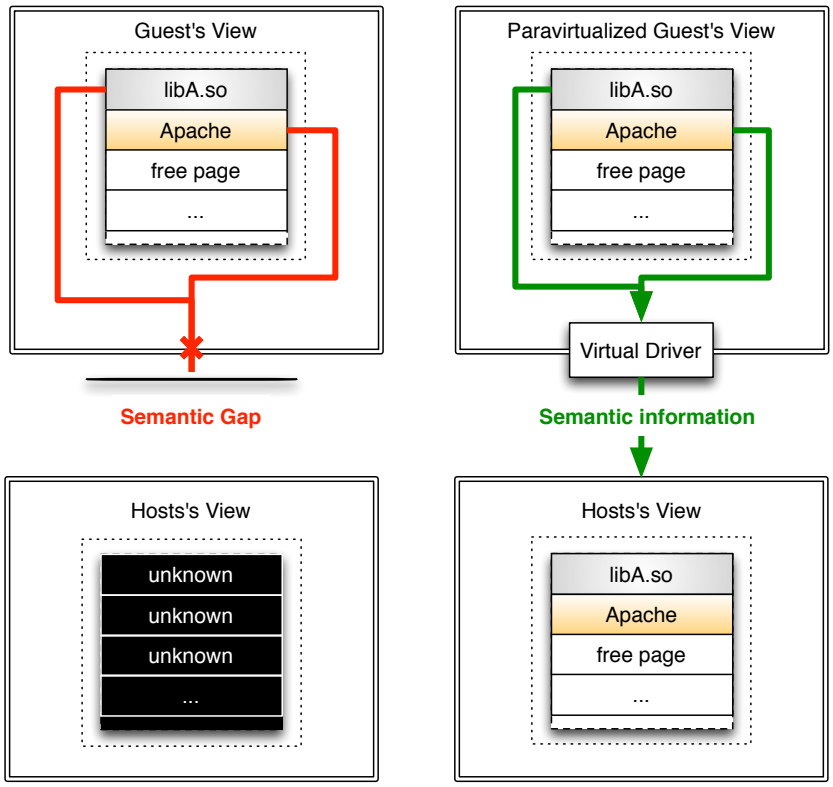


Figure 2.6: *Left*: The non-paravirtualized guest is a black box to the host. The host has no semantic information about the inner state of the guest. *Right*: The paravirtualized guest provides semantic information via a virtual driver to the host. The guest OS is modified.

Two main approaches used in paravirtualization can be distinguished to solve the problem of saving memory:

- ... based on knowledge of memory content
- ... based on knowledge of disk content

## Memory Ballooning

Both hosts and guests can have high memory pressure. Guests are overprovisioned in order to move memory pressure from guests to the host, which also increases sharing opportunities between guests. In a high memory pressure situation the host cannot decrease the memory of guests as it does not know if the guest needs the memory or not. Even if a certain amount of memory was reserved, it is not feasible to increase or decrease the amount of memory frequently on a page-granular basis [10].

To solve this problem a so called "balloon" driver [26] can be used. A VM gets supplied with a particular amount of overcommitted memory, but the balloon driver immediately claims the overcommitted part and gives it back to the host. According to policy the host can inflate or deflate the balloon of a particular guest. This effectively moves the memory pressure back from the host to the guest. When the guest has high memory pressure it can prune its caches or swap pages to disk.

Memory balloons do not directly save memory, but redistribute it so that the memory is used more efficiently. In the end result more guests can be run on the same host.

## Collaborative Memory Management

According to IBM's CMM, [23] there are 2 commonly used approaches for overcommitting memory:

- Dynamic Partitioning - The memory of the guest is dynamic and is (de)allocated according to a global memory strategy.
- Memory Virtualization - The host swaps guest memory in the same way memory is overcommitted usually to applications.

Dynamic Partitioning is often implemented via Ballooning, while Memory Virtualization is commonly implemented via host paging.

The problem of both approaches is according to CMM that in highly overcommitted scenarios the paging or ballooning leads to a high overhead and longer response times for guests that need free memory pages.

CMM solves these problems by using collaborative memory management to find pages that are currently unused in the guest. To implement this CMM uses 4x3 states for each guest page (stable, unused, volatile, potential volatile in the guest

and resident, preserved, zero in the host). Using this CMM guests willingly mark memory, so that the host can make a decision which pages to drop or swap out. Those pages can then be used by other guests. With CMM unused or zero pages can be reclaimed faster and with less overhead by the host than with host paging or memory balloons. CMM was only implemented for Linux on System Z.

## Disco

As already described a huge sharing potential exist in the buffer caches of the VMs. (Cellular) Disco [5, 8] shares memory across several Virtual Machines by utilizing a shared COW disk for all Virtual Machines. If a virtual machine reads a file block, which is already mapped to another VM, the VM gets the same pages mapped in their address space. All memory is mapped read-only and uses normal COW semantics on write attempts. Disco does this by providing the VMs with both virtual networking and virtual disk devices and can such intercept every disk request that DMAs data into memory. They additionally intercept bcopy calls to support sharing via NFS while copying in- and outside of the network buffer.

Disco shows that sharing memory is possible by sharing the buffer cache of the virtual machines. It however is dependent that the Virtual Machines share their disk.

## Satori

Satori [19] extends the work done by Disco by implementing sharing-aware block devices, which additionally to the sharing of the same blocks also allow the sharing of pages that have the same content. This allows Satori to be independent of using COW disks and to share blocks that are identical within the same disk. For content-aware sharing Satori uses a hash table to map disk block contents to memory mappings.

Satori makes use of memory balloons to distribute the memory savings across several VMs. In this model sharings are distributed proportionally to the amount a VM is already sharing.

Satori shows that the majority of the sharing opportunities are short-lived and that many of the short-lived sharing opportunities stem from the buffer cache.

### Other Paravirtualization Approaches

XHive [12] swaps out pages not to a disk but instead to the host, which can then decide to swap them out to a disk or keep them in memory – depending on how much memory pressure the host has. This solves the problem of double swapping, where a page in the worst case needs to be swapped in on the host, to be then swapped out by the guest. This is happening when both host and guest have high memory pressure. It also gives guests a higher chance that this page is kept in memory while reducing memory consumption of the guest.

Transcendent Memory [16, 17] provides guests with a key/value storage system, which the guest can use to cache I/O requests in the host. This prevents the problem of double caching I/O both in the host and in the guest.

XenFS [27] is a prototype of a file system that is used across guests. XenFS makes it possible to share pages and page mappings by using COW on the file system layer.

### Limitations

All techniques seen in this Section need changes to the guest and cooperation of the guests to work.

## 2.3 Memory Scanners

The host does not know where the duplicated pages in the VMs are as it has no information about the inner life of the guests. To bridge this semantic gap, one strategy is to scan the memory of the guests for page sharing opportunities. A memory scanner periodically scans the memory of all VMs for pages with duplicate content. With this information the host is able to share these pages using usual copy-on-write semantics. The functionality of a generic memory scanner is depicted in Figure 2.7.

The are two memory scanners that have been explored in this work; namely KSM and VMWare ESX Server.

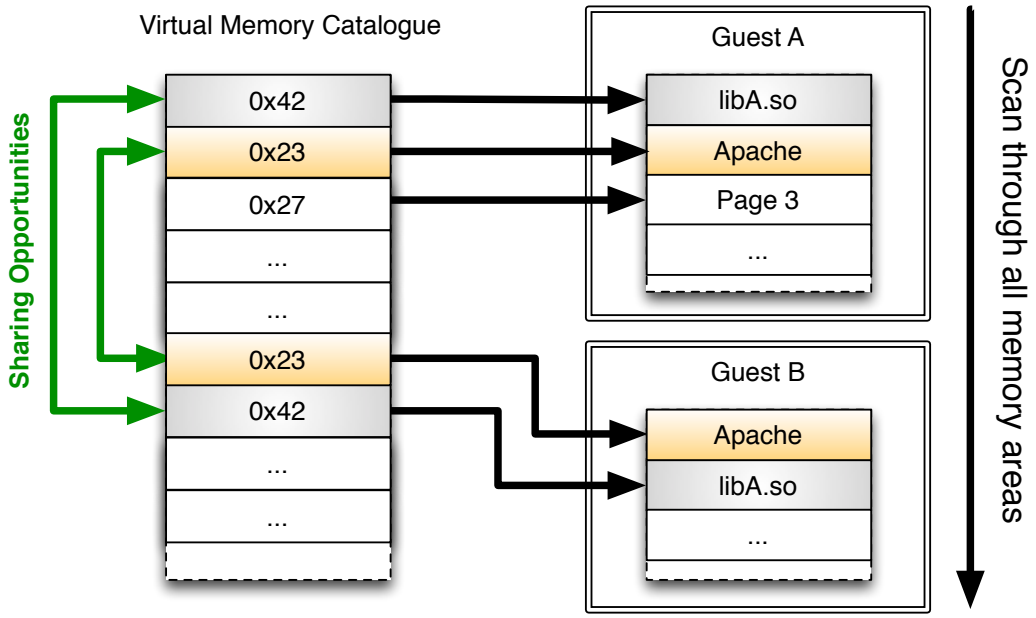


Figure 2.7: *Generic Memory Scanner* – The memory scanner scans all guest pages and catalogues them in a virtual memory catalogue storing the checksum of the page. By looking up information from the virtual memory catalogue, sharing opportunities can be found.

## KSM

KSM [1] is a generic memory scanner and same memory merger that operates on anonymous pages. Memory that is scanned by KSM first needs to be advised via the `madvise` system call. The advised memory areas are scanned periodically in a linear way on a page by page basis. One complete scan of all advised pages is defined as a *scan round*. For each page scanned a checksum is calculated. If the checksum is still the same since the last run, then page is marked as a potential sharing candidate. Those sharing candidates are checked if the content matches another page in the so called stable tree, which holds all the currently shared pages. If this is not the case the potential sharing candidate is checked against the unstable tree, which contains all potential sharing candidates. If no match is found the new page is inserted into the unstable tree. If a match is found, the sharing partner is removed from the unstable tree and both pages are shared and then inserted into the stable tree.

KSM has a constant scanning overhead, which means that a certain amount of CPU is always used by the KSM kernel process. KSM is woken up periodically, scans a fixed number of pages and the process is suspended then again until it is

woken up again. With this technique the amount of CPU used is not 100% all the time, but can be tweaked to use less CPU power (for example 20%). The KSM scan thread is single threaded.

A memory scanner always trades saved memory against this constant usage of CPU power. The KSM scan process will be discussed in more detail in Section 3.1).

### VMWare ESX Server

VMWare ESX Server [26] also periodically scans the memory. However ESX scans the memory in a random fashion. It uses a hash table for storing potential sharing candidates. For each page that has the same checksum (fingerprint) as another page, the pages are compared and if they are identical shared and marked read-only.

Additionally to supporting host paging, ESX employs a memory ballooning strategy for distributing free pages to the guests.

### Limitations

Memory scanners are limited by their scan rate and are always a trade-off between saved memory and used CPU power.

## 2.4 Summary

This chapter has introduced the background needed for understanding the design and implementation of KSM++. Traditional methods, memory scanning and paravirtualization solutions to save memory have been presented. The approach to use information about I/O requests to save memory is known from paravirtualization, but paravirtualization has the disadvantage that the guest needs to be modified, which is not suitable for many environments.

The approach of using information about I/O activity in combination with memory scanning – without using paravirtualization – is detailed in the next chapter.

# Chapter 3

# Design

This chapter first explains the KSM scan process in more detail. Then it presents KSM++, a novel approach for using I/O based hints with the goal to make memory deduplication scanners more efficient. Last it outlines how KSM++ intercepts I/O requests, stores I/O based hints and interleaves these hints into the scan process of KSM.

## 3.1 KSM

KSM is a memory scanner and same page merger. Applications advise memory as mergeable to KSM, which in turn scans these advised memory areas for pages with duplicate content and shares them. KSM is called periodically on a fixed interval. The KSM process can be divided into 4 parts: Scan, Compare, Merge and Unmerge. An overview of the functionality of KSM is depicted in Figure 3.1, while the four parts and how they work together can be found in Figure 3.2.

KSM was not specifically designed for VMs, it can be used with any application as long as the memory is mergeable. In its implementation KSM is able to only merge anonymous memory, but not named memory pages. Named memory pages have an associated backing file, i.e. a file mapped by `mmap`. As the memory of a VM is a black box to the host, all memory pages of the guest are anonymous memory to the host. Therefore, those pages can be merged by KSM. This is only an implementation issue, since in theory KSM can merge any memory areas.

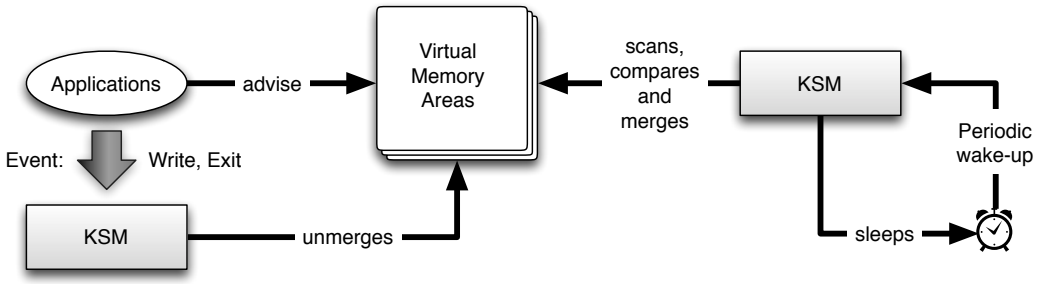


Figure 3.1: Applications advise virtual memory areas as mergeable to KSM. KSM periodically scans, compares and merges pages with duplicate content in these areas. When a process writes to a shared page or exits KSM unmerges the affected memory areas.

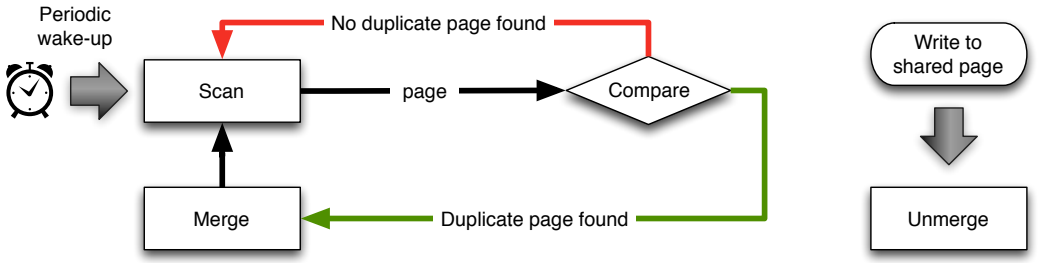


Figure 3.2: For each page scanned, KSM compares it to known sharing candidates and merges the pages on a match. On a write request to a shared page, KSM unmerges the page.

## Scan

KSM scans the advised virtual memory areas linearly from start to end. To throttle the CPU consumption, KSM scans `pages_to_scan` number of pages every `sleep_millisecs` ms. A full scan of all advised memory areas is called *scan round*.

For every scanned virtual page, a *reverse map item* (rmap_item) is created and used. A rmap_item stores state information about its virtual page – the checksum of the page content and if the page is shared or not. Each virtual page is represented by exactly one rmap_item in the system. The KSM scan process needs to re-use rmap_items when it scans a virtual page again.

Each scanned virtual page is compared with existing sharing candidates to find pages with the same content via KSM's compare routine.

## Compare

The compare routine checks if a page with the same content already exists in the system. In order to that KSM needs to use lookup data structures.

KSM utilizes two red-black trees as lookup structures: The *stable tree* contains pointers to all merged pages, while the *unstable tree* contains all potential sharing candidates. A potential sharing candidate is defined by KSM as a page that has not changed since the last time it was compared. As a virtual page has no representation as an object, the corresponding rmap_item is used for insertion in the tree instead. The unstable red-black tree is depicted in Figure 3.4. It is sorted by the content of the contained pages. On average only a few bytes of each page need to be scanned to make a decision where to insert the corresponding rmap_item in the tree.

The whole process is depicted in Figure 3.3.

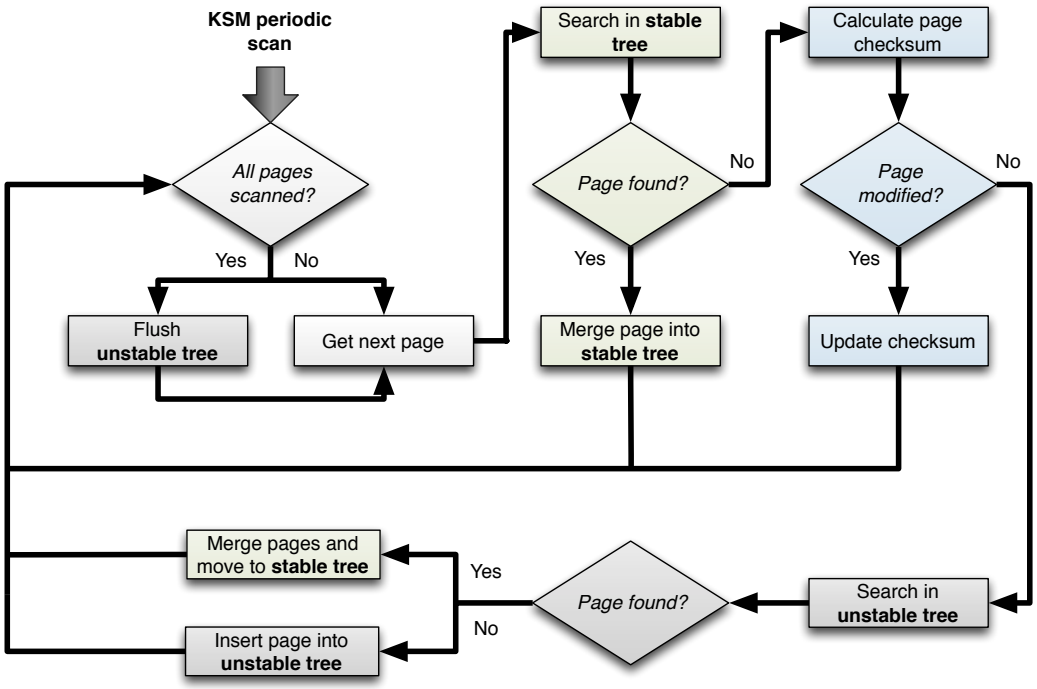


Figure 3.3: KSM's Compare Routine – *Get next page* refers to the scan routine that provides the next page and rmap_item to scan. KSM flushes the unstable tree of all potential sharing candidates after it has inspected all advised pages. [18]

At the beginning of the compare routine, the rmap_item can be in one of three states (shared, unstable and volatile). If the rmap_item is currently shared, no

further work is needed and the process ends. If the rmap_item is unstable it was marked as a potential sharing candidate and had been inserted into the unstable tree. If it is volatile it might be a future sharing candidate.

At the beginning of the process the rmap_item is removed from the unstable tree. If the rmap_item had not been in the unstable tree, nothing happens.

Then it is checked against a match in the stable tree. If a match is found the page is merged into the stable tree and the compare routine finishes.

KSM has a heuristic to only inspect pages for potential sharing partners that have not changed since the last scan round. For each page a jhash2 checksum is calculated and stored in the rmap_item. Only if the page checksum still matches the rmap_item checksum is the page considered a potential sharing candidate. Else the checksum is updated in the rmap_item and the compare routine finishes.

Potential sharing candidates are looked up in the unstable tree. If a match is found, the two pages are merged into the stable tree and the item is removed from the unstable tree. If no match was found the sharing candidate is inserted into the unstable tree instead.

After a full scan round the unstable tree is flushed and such effectively cleared of all potential sharing candidates.

## Merge

The merge routine maps the virtual pages to one physical page with the same content and maps the physical page copy-on-write. It inserts the rmap_items into the stable tree. The unmapped duplicate physical pages are free'd. As the nodes in the stable tree are indexed by page content, several rmap_items can now represent one tree node. To solve this problem, rmap_items are organized into a single linked list per physical page. The organization of the stable tree is depicted in Figure 3.5.

## Unmerge

If an application writes to a page that is currently shared within KSM, the OS gets an exception from the MMU, the exception handler breaks the page sharing and the process gets it own private read-write copy of the page. Additionally, the corresponding page is marked as not shared. It is removed from the stable tree during the next search/insert operation in the tree, which also sets the state of the

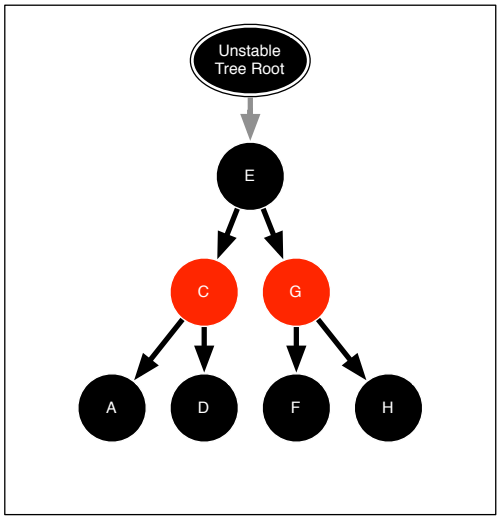


Figure 3.4: *rmap_items* in the unstable red-black tree. For clarification the connections between the rmap_items were not drawn in this figure. A red-black tree is always balanced.

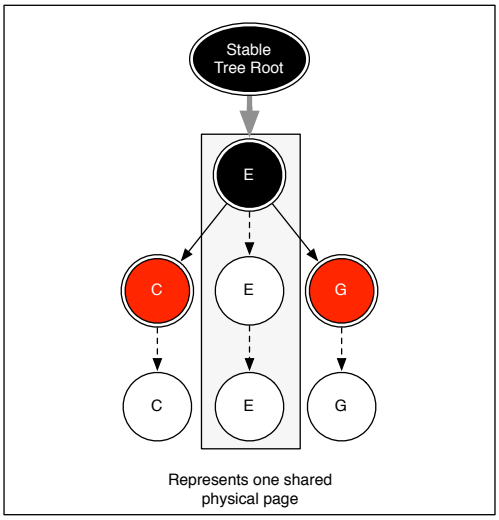


Figure 3.5: *rmap_items* in the stable red-black tree. Each node (marked by a double-outline) contains a list of rmap_items participating in the sharing of one physical page.

corresponding rmap_item to volatile. If it was the last sharing partner, the stable node is removed from the tree.

## KSM Performance Characteristics

Memory scanners like KSM have the limitation that the scan-rate needs to be very high so that they can find short-lived sharing opportunities, i.e. by having the memory scanner occupy an entire CPU. The formula for the run-time (in seconds) of one complete scan of all advised memory areas is:

$$t_{complete-scan} = \frac{advised\_pages * sleep\_millisecs}{1000 * pages\_to\_scan} \tag{3.1}$$

KSM scans *pages_to_scan* pages every *sleep_millisecs* ms. *advised_pages* is the advised memory size divided by the PAGE_SIZE.

Given a sharing opportunity that exists for $t_{sharing-opportunity}$ seconds and $t_{search}$ seconds, which KSM needs to select the page for scanning, the time that KSM can actively share the page is:

$$t_{active-sharing} = t_{sharing-opportunity} - t_{search} - t_{complete-scan} \quad (3.2)$$

From the time point the sharing opportunity exists, KSM needs $t_{search}$ seconds to select the affected page for scanning out of all advised memory areas. This page must then not change for one complete scan round ($t_{complete-scan}$) as only pages that fullfill the checksum heuristic are considered for sharing. Then KSM starts sharing the page for $t_{active-sharing}$ seconds until the sharing opportunity ends.

The timeline of sharing opportunities using KSM is depicted in Figure 3.6.

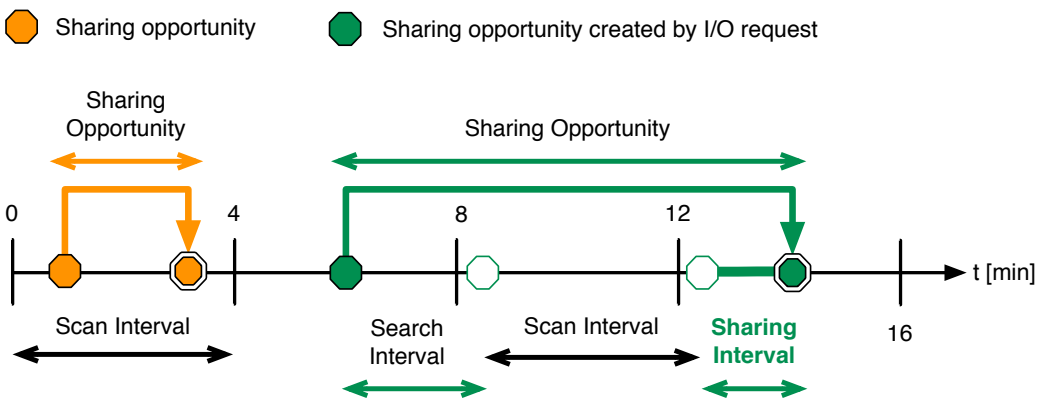


Figure 3.6: KSM cannot find sharing opportunities that exist shorter than the scan interval. The sharing interval is decreased by the time KSM needs to find the page plus the time to wait for one full scan round. (see Formula 3.2)

To put this into perspective: To scan 2 VMs with 512 MB each, KSM needs around 4 min at the standard scan rate. This work defines short-lived sharing opportunities to exist between 30 sec and 5 min.

As can be seen in the timeline, KSM cannot find sharing opportunities that exist shorter than the scan interval ($t_{complete-scan}$). Therefore KSM misses around 60-80% of the short-lived sharing opportunities (see Section 5.4). In the worst case KSM needs to scan the full memory twice before it finds a sharing opportunity.

## 3.2 KSM++

Bugnion et al. 1997; Milos et al. 2009; Kloster, Kristensen, and Mejlholm 2007 [5, 13, 19] have shown that the buffer caches of VMs have a large sharing potential. The sharing potential can exist between VMs or within the buffer cache of one VM.

Pages in the buffer cache of VMs stem from read or write requests from/to the Virtual Disk Image (VDI). This thesis proposes an extension to memory scanners to find short-lived sharing opportunities that stem from the page cache of VMs by inspecting I/O requests in the host. The location of the I/O requests is passed in the form of hints to a buffer in the memory scanner. The memory scanner interleaves the inspection of hinted pages with the normal linear scan. In the following this new approach is called KSM++.

The timeline of sharing opportunities using KSM++ is depicted in Figure 3.7.

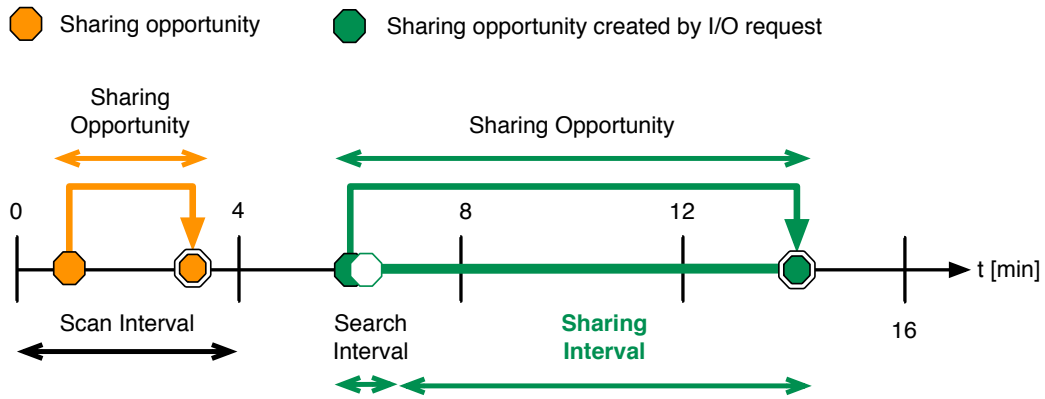


Figure 3.7: KSM++ can compare pages stemming from I/O requests earlier than the linear scan. The search interval to find the sharing opportunity is smaller and the heuristic to wait for one complete scan round is not used.

## Scan Areas of I/O Activity

The flow of I/O requests from guest application to host physcial disk is depicted in Figure 3.8.

When an application reads from a file inside of a VM, the read request is first sent to the guest operating system. The guest OS conducts a read request on the virtual disk. The Virtual Machine Monitor translates the read request to the virtual disk to a read request on the Virtual Disk Image – the backing file of the Virtual Disk in the host. The host OS sends the read request to the physical disk. Such a read request by an application in the VM always (with caches excluded) triggers a read request on the physical disk.

As a VFS read request to the Virtual Disk Image in the host corresponds to a read request in the guest OS on the Virtual Disk, the buffers used on the host side stem

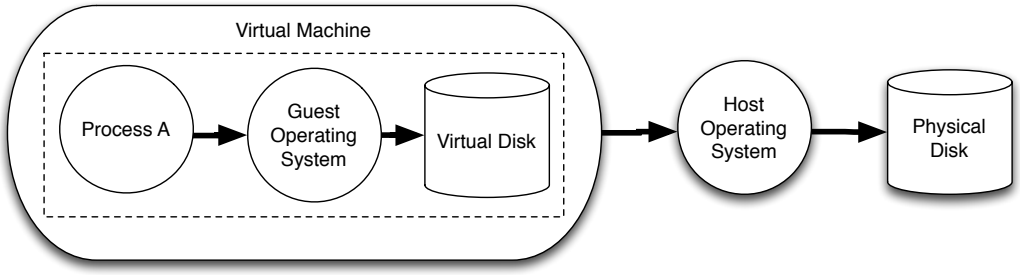


Figure 3.8: A process reading from a file in the guest, triggers an I/O read operation in the host OS by the Virtual Machine.

from the page cache of the guest. In consequence I/O requests in the host contain information about the location of buffer cache pages in the guest.

KSM++ makes use of this information and stores it in the form of hints to a buffer in the memory scanner. A hint contains a pair of virtual address start/length of the I/O request, which represents the location of the buffer cache pages in the guest.

The whole process of intercepting I/O requests is depicted in Figure 3.9.

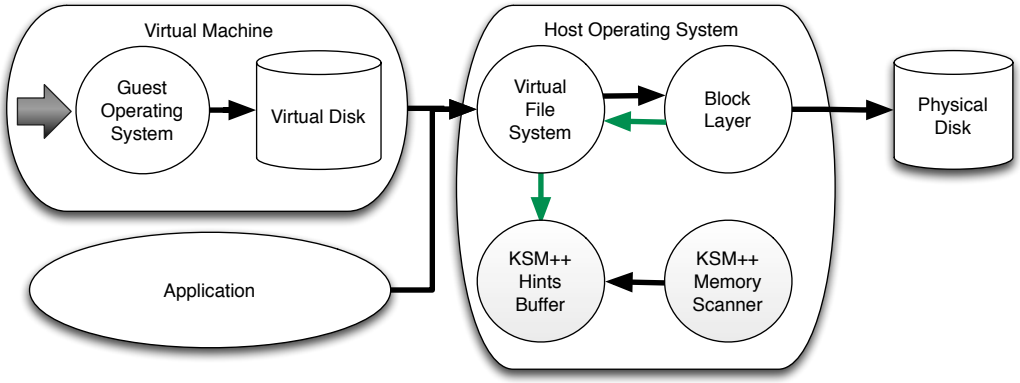


Figure 3.9: Applications or VMs do a read or write request. KSM++ stores semantic information about the request to a hints buffer in the VFS read and write functions before they return.

The I/O requests are intercepted inside of the Virtual File System (VFS) layer of the host OS. The VFS is an abstraction layer through which all read or write requests to files in the system go. It does not matter if the file resides on a physical disk, on a network file system or in the memory of the OS. Hence the VFS is a suitable place for KSM++ to generate hints.

KSM++ does not generate the hints in the Virtual Machine Monitor, but in the

VFS in order to issue hints for deduplication of normal application memory (i.e. not VMs). Hence KSM++ generates hints for all pages that are mergeable.

I/O requests can be intercepted before the actual read operation takes place or after it has returned and the data has been read/written. KSM++ generates hints after the data has been read/written in order to make sure that the affected I/O area has changed before the memory scanner processes the hint.

### Storing of Hints and dealing with I/O Bursts

I/O is generally bursty and can produce hundreds of megabytes of data per second. In consequence KSM++ could produce millions of hints per second. This is much more than the memory scanner can process. For example with the standard scan rate KSM can scan 4 MB/s and with a setting, which occupies one CPU core to 68%, around 20 MB/s. To prevent I/O requests from piling up, KSM++ stores hints in a space limited ring buffer. In this hints buffer the oldest entries are overwritten automatically. With this design I/O bursts cannot overload the memory scanner.

### Interleave processing of Hints with regular Scan

Buffer cache pages are not the only pages with sharing potential. Barker et al. found around 15% sharing potential stemming from heap segments of applications [2]. To find both short-lived sharing opportunities stemming from the buffer caches and long-lived sharing opportunities from other sources, KSM++ does not replace the linear KSM scan with processing of hints (hinting), but instead processes hints mixed with the scan process. To ensure that both pages found by hints and pages found by the linear scan are processed, KSM++ interleaves processing of hints with the regular scan process.

To make interleaving of hints possible, the KSM scan process needs to be changed in two places: First, the scan process gets the next page either from the linear scan cursor or from the hinted area. Second, the checksum heuristic, which is used by KSM to only process long-lived sharing opportunities, is not used for processing hints, because hints should be processed as soon as possible. An overview of this is depicted in Figure 3.10.

KSM++'s full compare routine is depicted in Figure 3.11.

Scanning the memory pages of a hint is similar to the linear scan, but it starts at the beginning of the hinted area and ends once the area has been completely scanned.

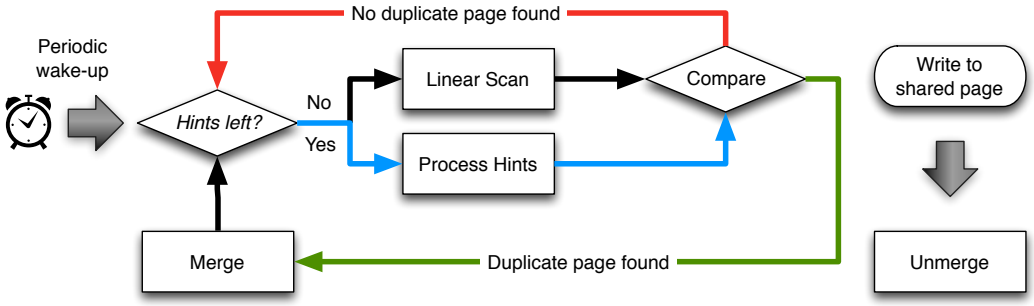


Figure 3.10: For each page scanned or hinted, KSM++ compares it to known sharing candidates and merges the pages on a match. On a write request to a shared page, KSM++ unmerges the page.

KSM++ generates and uses the same rmap_items as the linear scan process. If the area of the hint has been completely scanned and all rmap_item / page tuples have been compared against potential sharing candidates, the next hint is processed. If there are no more hints to process, KSM++ uses the remaining "quota" of pages to continue the linear scanning process.

#### Interleaving Ratio

KSM++ processes a fixed number of pages stemming from hints, then a fixed number of pages stemming from the linear scan, etc. Any number of hint spurts can be interleaved with any number of scan spurts. To keep the CPU consumption constant KSM++ falls back to linear scan if there are no more hints to process in a hint spurt. Pages that had been already a target of a hint in this scan round are skipped and not compared in order to not scan pages twice.

## 3.3 Summary

This chapter outlined the design of KSM++. The approach proposed in this thesis intercepts I/O requests and pushes the location information of the requests as hints to a hints buffer. KSM++ interleaves the usual linear memory scan with the processing of hints.

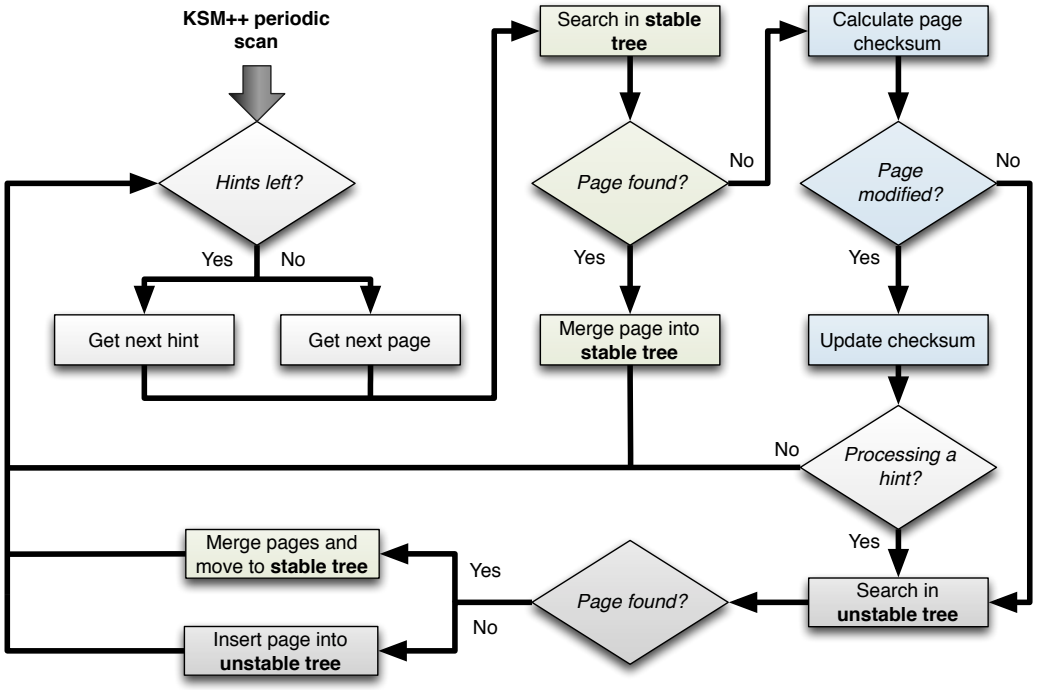


Figure 3.11: KSM++'s Compare Routine – KSM++ processes hints as long as there are hints left. A match is first checked for in the stable tree. If a sharing partner was found, the page is merged into the stable tree. If KSM++ is processing hints or the page had not changed since the last scan round – the unstable tree is checked. [18]

# Chapter 4

# Implementation

This chapter covers how the design that was outlined in the previous chapter has been implemented, what challenges were met and overcome and which decisions have been taken. The prototype introduced in this work, namely KSM++, extends KSM under Linux 3.4-rc3 by only around 600 SLOC (source lines of code) to implement I/O based hints. A high level overview of the KSM++ prototype components is depicted in Figure 4.1.

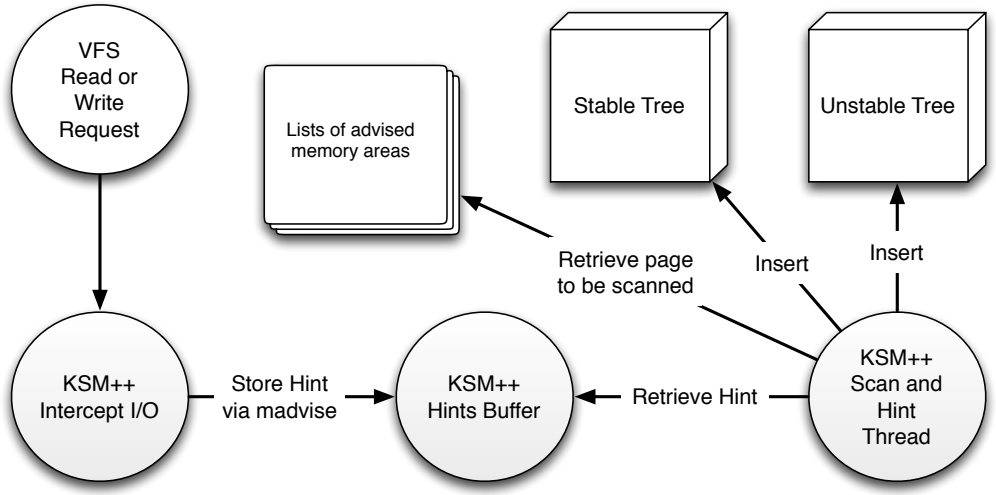


Figure 4.1: High Level Overview of KSM++ prototype components. KSM++ intercepts and stores I/O in the Hints Buffer and interleaves these hints with the linear scan of the advised memory areas. The Scan and Hint Thread inserts sharing opportunities and actively shared pages into the tree structures.

Using KSM and Linux has several advantages. First, the source code is freely available, which makes it easy to extend the kernel to integrate I/O-based hints. Second, since KSM already provides data structures, mechanisms and basic policies for memory deduplication, the KSM++ prototype only needs to implement the new parts of the approach described in this work. Third, KSM can be used as the baseline for evaluating KSM++s performance. Last, KSM already exposes statistics about the number of shared pages, the number of items in the unstable tree and the number of pages that need to be scanned and therefore provides insights about the applicability of such a system.

## 4.1 KSM

KSM is a memory deduplication scanner implemented in the Linux kernel. Applications advise memory as mergeable to KSM. KSM scans, compares and merges these advised memory areas on a fixed wake-up interva. In other words, KSM scans `pages_to_scan` pages every `sleep_time` ms.

### Data Structures

KSM uses a global scan cursor *ksm_scan* to store the state of the scan process while the KSM process is sleeping.

In order to be able to scan each advised virtual memory area linearly, KSM uses rmap_item lists for representing the state of virtual pages per virtual address space. A rmap_item contains the virtual address, the checksum and the current sharing state (volatile, unstable, stable) of the respective page. The list of rmap_items is not created when the memory is advised to KSM. Rather, as the scan progresses, it is build up on the fly. The location of the last scanned rmap_item is stored in the scan cursor structure. Beginning from this location the list is scanned for a rmap_item matching the address. If none can be found a new rmap_item with this address is inserted at the tail of the list.

Rmap_items are organized in one list per virtual address space. The rmap_item data structure for one address space is depicted in Figure 4.2.

KSM is able to dynamically grow the advised memory areas. The scan cursor structure contains a pointer to the currently scanned address space (mm_slot), the virtual address and the currently scanned rmap_item. rmap_item lists are created per address space and stored in the mm_slot. The ksm_scan data structure

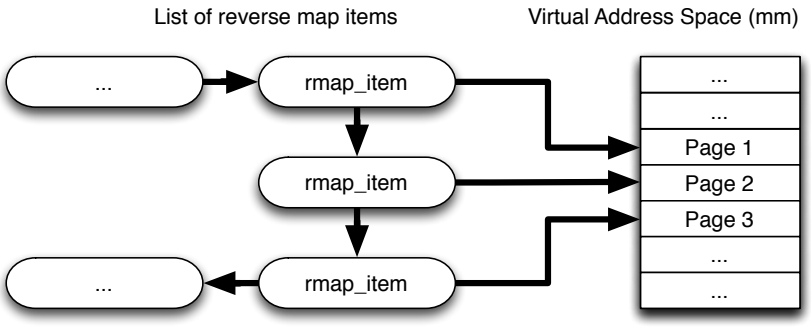


Figure 4.2: *rmap_items* are ordered into a singly linked list per address space and reference virtual memory addresses. They additionally store a checksum and the sharing state of the page.

also contains information about the number of scan rounds as a sequence number, which is used to decide when an rmap_item was last compared. All aforementioned data structures are depicted in Figure 4.3.

KSM cleans up virtual memory areas, which belong to processes that have exited. This takes place concurrently to the linear scan and KSM uses the scan_cursor to avoid a race condition: Rmap_items that belong to the exited address space are removed only when they are not part of the currently scanned address space.

## 4.2 KSM++

KSM++ generates and stores I/O hints and processes these hints interleaved with the linear scan in order to find sharing opportunities that stem from the page cache. This section discusses the prototype implementation of KSM++.

### Intercepting I/O

KSM++ intercepts I/O in the Virtual File System (VFS) layer. The location of the target buffer of a read request or the location of the source buffer of a write request are stored in the KSM++ Hints Buffer. In order to do that, KSM++ first needs to resolve the buffers address space and the location within the virtual region.

The `madvise` system call allows a process to tell the kernel how it expects to use its memory areas. It translates buffers pointing into process space into globally valid virtual addresses within a virtual region. Processes already use the madvise

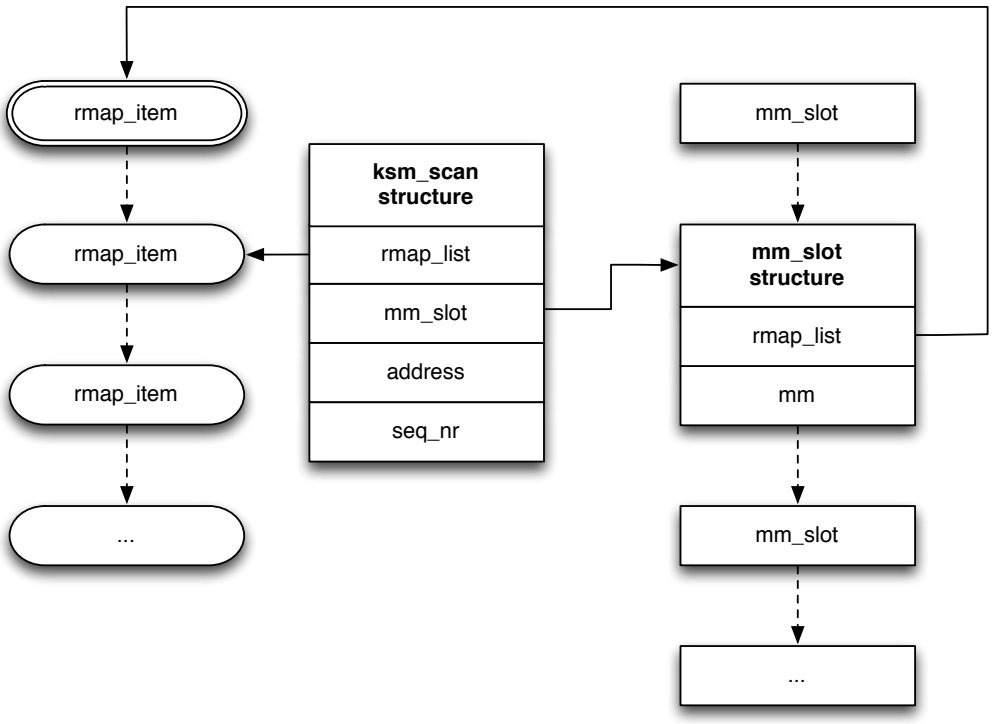


Figure 4.3: KSM does a periodic scan of advised memory areas. The ksm_scan cursor structure points to the currently scanned virtual address (represented by mm_slot, address and rmap_item). Each mm_slot includes the list of rmap_items representing the virtual address space.

system call to advise memory areas as mergeable to KSM/KSM++. KSM++ extends the madvise system call with a new flag in order to advise a memory area as a hint. The prototype implementation enables KSM++'s hinting mechanism to call madvise from within the kernel in the VFS Layer. However, hints will be only provided for memory areas that had been previously advised as being mergeable.

## Storing Hints

The VFS is capable of producing much more hints than KSM++ is able to handle with its limited scan rate. Using a queue structure as Hints Buffer has the disadvantage that KSM++ processes potentially outdated hints, while using a stack has the drawback that the stack can grow very large and hence needs purging and expire mechanism that clean it periodically from hints that are no longer relevant.

In this work Hints are buffered in a Bounded Circular Stack. A Bounded Circular stack is a LIFO (Last-In, First-Out) structure with a fixed number of items. The oldest entries are overwritten, when new hints come in. The advantages of using a Bounded Circular stack are that the stack has a fixed size and that it does not need to allocate/deallocate memory for new entries. Furthermore, the stack does not need to implement a purging mechanism based on the "age" of the hint, since the oldest hints are automatically expired from the stack. An example of how the bounded circular stack works can be found in Figure 4.4.

The `stack_size` is configurable via the sysfs interface and limits the rate of I/O traffic that KSM++ needs to process.

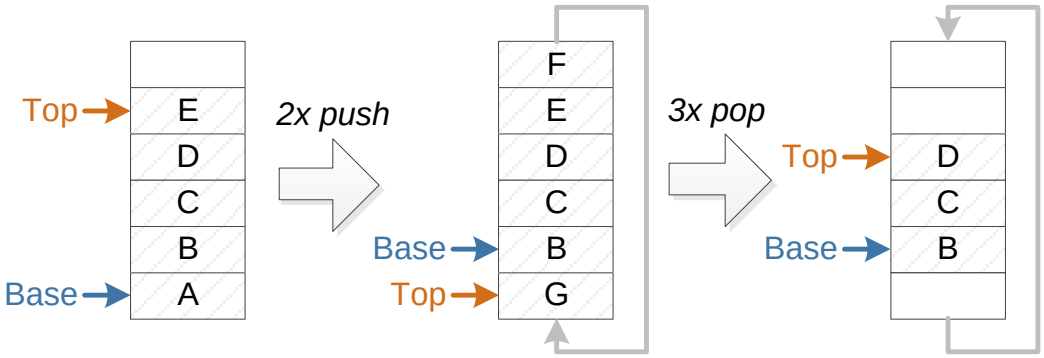


Figure 4.4: Storage of hints in the Bounded Circular Stack. The oldest entries are overwritten when new entries are pushed. [18]

## Interleaving Hints

To interleave processing hints and KSM's scan process the prototype implementation alters KSM to support two types of spurts: A scan spurt and a hint spurt.

`max_hint_runs` hint processing spurts are interleaved with `max_scan_runs` scan processing spurts and the interleaving ratio is configurable via the sysfs interface. The ksm_scan data structure is not well suited for scanning random memory areas, but was designed for a linear scan. Using an additional cursor does not work, because the ksm_scan cursor is used globally within KSM and further components depend on it.

To solve this problem KSM++ saves or restores the state of the scan cursor when transitioning from linear scan to processing hints or vice versa. The same is done for the state of the hinting process.

Methods that were used unchanged in KSM do not know about the scan cursor backup. The state of the scan_cursor might for example point to a no longer existing memory area when KSM++ the scan cursor from the backup. Therefore, KSM++ needs to double check that the memory area is still valid and if not the scan is restarted from the beginning. The state of the data structures during a hint spurt is depicted in Figure 4.5.

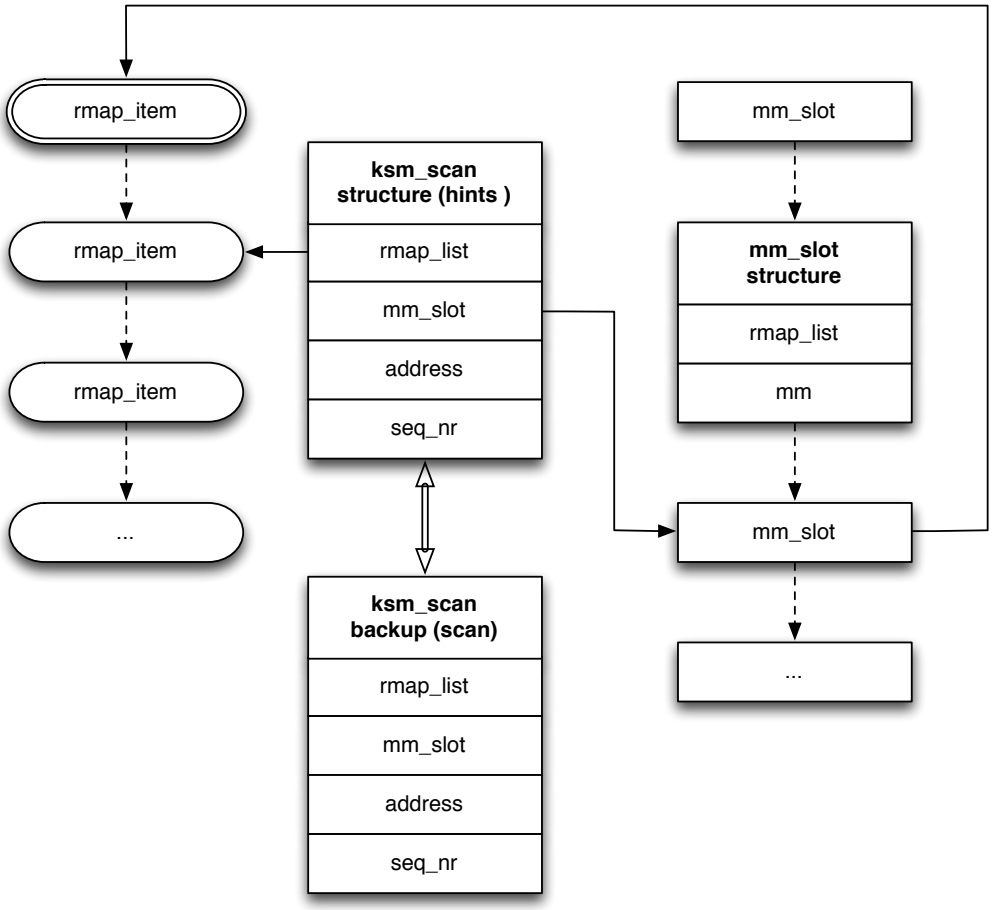


Figure 4.5: While doing a hint spurt, the scan cursor state is backed up in a seperate data structure.

## 4.3 Degeneration of the Unstable Tree

KSM/KSM++ is using the actual page contents as index into the tree instead of hash keys. The advantage of this lies in the fact that KSM/KSM++ on average

only needs to compare a few bytes of page data to decide where this page needs to be inserted into the unstable tree. Further, the tree is able to grow dynamically as more pages get advised to KSM/KSM++. However KSM/KSM++ does not know when or how an underlying page changes. Pages are also *not* marked read-only when inserted into the unstable tree, but remain writable for the VM. Modification of these pages can corrupt the unstable tree.

As depicted in Figure 4.6, when a page changes, the nodes in a subtree can become unreachable. Consequently breaking the root of the tree can make 50% or more of the nodes unreachable. Therefore, the unstable tree is prone to degeneration – in the kernel build benchmark up to 90% of pages became unreachable in the tree (see Section 5.4).

The stable tree is not affected by this problem as all pages contained in the stable tree are write-protected and marked as KSM shared page. On a write attempt, the sharing is broken and the page is marked as a normal anonymous page. The page is removed from the stable tree during the next search or insert operation.

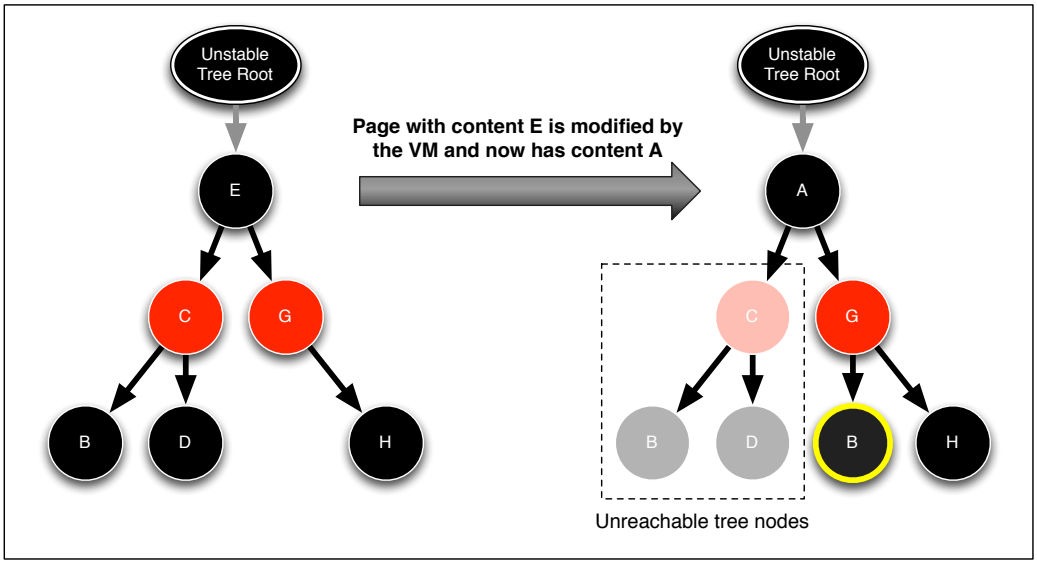


Figure 4.6: Nodes in the unstable tree can become unreachable due to modification of the contained pages. In this example the content of a page has changed from 'E' to 'A'. The pages with content C, B and D are unreachable. Therefore, a page with content B is inserted a second time and the sharing opportunity is not detected.

KSM implements two strategies – one to avoid insertion of frequently changing pages and another one to remove corrupted nodes from the tree – to mitigate the degeneration of the tree: First, only pages that have not changed since the last

scan are considered for insertion into the unstable tree (checksum heuristic). KSM determines if a page has changed or not by storing a checksum in the respective rmap_item. As pages will change eventually, the tree is still degenerating – albeit slower. Second, KSM takes a drastic measure to remove degenerated pages: It flushes the unstable tree after one full scan of all advised memory areas (scan round) and starts over with an empty tree.

The problem gets worse when KSM++ starts inserting pages that stem from hints into the unstable tree as those:

- are not protected via the checksum heuristic
- reside in the guests page cache and those pages are likely to change more often than other pages contained in the unstable tree – depending on the I/O traffic

KSM++ does not have "scan round" semantics and hinted pages should be merged as quickly as possible. In consequence, KSM++ inserts hinted pages into the unstable tree immediately.

Buffer cache pages are modified when applications running in the VM write to files or the guest OS replaces pages in the guests buffer cache. Especially working sets that exceed the size of the guests page cache (for example in I/O centric workloads), will lead to frequent replacement of page cache pages. In the kernel build benchmark around 260 pages per second were modified in the unstable tree (see Section 5.4). When the guests buffer cache changes more often, the degeneration of the tree gets worse.

## Marking pages read-only

To solve the problem of the degenerating tree, KSM was modified to map either hinted pages or all pages that get inserted into the unstable tree read-only. Pages in the Linux Kernel have a special mapping property that usually points to the backing store (inode) of a file mapped page. In the case that the page is anonymous – and for the host all VM pages are anonymous – this mapping is set to PAGE_MAPPING_ANON. KSM already makes use of the mapping property in order to mark pages as being actively shared in the stable tree and to store the corresponding tree node. KSM++ extends this functionality and enables it to work also for unstable tree nodes.

For each page that is inserted into the unstable tree, KSM++ marks and maps the page as KSM read-only and stores the active rmap_item in the mapping property. As soon as the VM attempts to write to a page, KSM++ is notified. KSM++ clears

the page mapping, marks the page as a normal anonymous page and removes the rmap_item from the unstable tree. Then the page is mapped read-write and the write is permitted. Note that this is different from the copy-on-write case used for stable pages as the page does not need to be copied. This process is depicted in Figure 4.7.

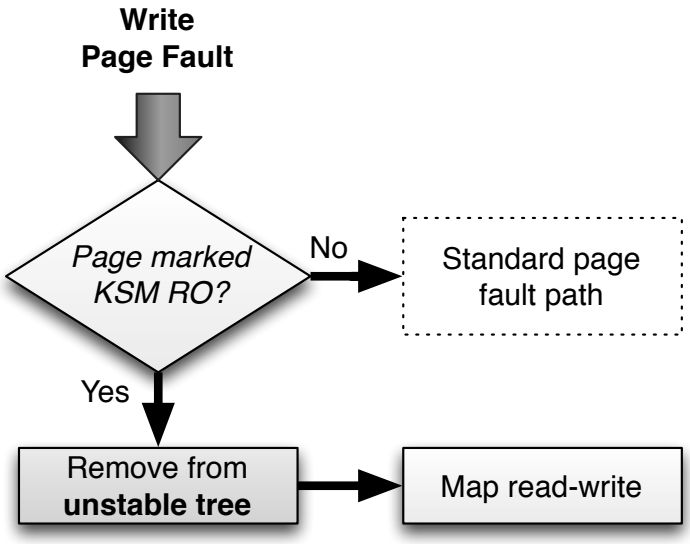


Figure 4.7: Pages marked with the KSM read-only flag are removed from the unstable tree and mapped read-write.

The disadvantage of mapping pages read-only for the purpose of being notified lies in the fact that remapping pages is considered "bad practice" and increases the overhead substantially [19]. The degeneration of the tree will be discussed in more detail as part of the evaluation in Section 5.4.

## Hash Table

Another possibility to avoid the degeneration of the tree is to replace the unstable tree with a hash table with a fixed number of entries. A size of 256 kB was chosen for the number of items in the hash table. As the hash key KSM++ uses the checksum stored in the rmap_item modulo the number of items. KSM already calculates the checksum of all pages for each rmap_item as part of the checksum heuristic, so there is no computational overhead for calculating the hash keys.

Hash collision conflicts – i.e. a page with the same hash key, but different content – can be resolved by overwriting the respective slot with the new value or by

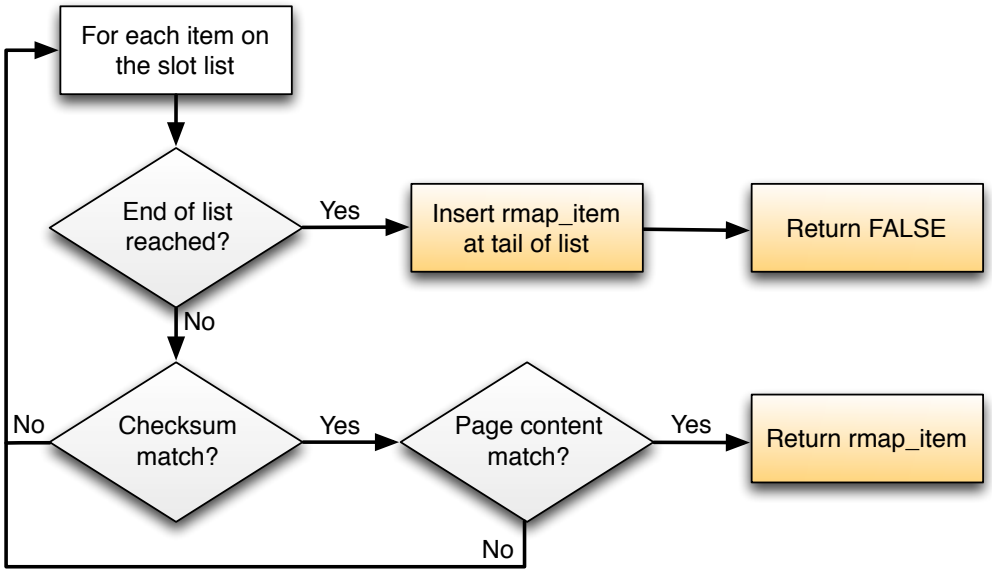


Figure 4.8: KSM++ retrieves pages from the hash table via its checksum from the respective slot utilizing chaining. To find a match the checksum and the actual page contents are compared to the sharing candidate. If no match can be found, the rmap_item is inserted into the slot.

inserting the entries with duplicated hash keys into a singly linked list. KSM++ uses a standard hash table with a singly linked list at the end for chaining.

For retrieving an item from the hash table by checksum, the hash key is calculated in order to find the correct slot. For each item in the slot list first the checksum is compared. Only if the checksum matches, does KSM++ compare the actual page contents. If the page had not changed since it was inserted, the rmap_item is returned. Else it is inserted at the end of the list. The process for finding a sharing partner by checksum is depicted in Figure 4.8.

Pages are not mapped read-only when the hash table is active. Hence KSM++ will again not know when and how pages change. However, since KSM++ only stores the hash keys in the hash table and not the page contents, the hash table is not affected by degeneration. The worst thing that can happen is that a potential sharing partner with matching checksum turns out to be a false positive – because the page content had changed since the item was inserted into the hash table.

## 4.4 Summary

This chapter covered the prototype implementation of the approach described in this work on top of KSM running on Linux. This prototype, named KSM++, extends KSM with I/O based hints and needs only around 600 SLOC of changes to the Linux kernel. KSM++ intercepts, stores and processes hints interleaved with the regular scan. The degeneration of the unstable tree was solved by either marking hinted pages read-only or by replacing the unstable tree with a hash table.

# Chapter 5

# Evaluation

This chapter evaluates the thesis, that I/O-based hints can improve the deduplication characteristics of memory scanners. First, the general benchmark setup is described. Second, metrics are defined which can be used to compare deduplication policies quantitatively. Third, the benchmarks that are used to measure those metrics are described and interpreted in detail. The chapter concludes with a summary of the evaluation results.

## 5.1 Benchmark Setup

In this section the goals of the evaluation, the server setup and the different configuration parameters are described.

### Goals

The thesis of this work is that I/O-based hints can improve memory scanners enabling them to find short-lived sharing opportunities that stem from the buffer caches of VMs. KSM can already find short-lived sharing opportunities by setting the scan rate to a high value – i.e., a setting in which KSM occupies an entire CPU. KSM++, the prototype introduced in this work, uses I/O-based hints to detect sharing opportunities that stem from the buffer caches of VMs.

In the following sections answers to the following questions are explored:

- Can KSM++ find sharing opportunities that stem from the page cache via the hinting mechanism even with low scan rates?

- How many sharing opportunities can KSM++ find in comparison to KSM?
- How many sharing opportunities can KSM++ find in comparison to the theoretical maximum?
- How many sharing opportunities can KSM++ find if I/O requests with the same content are not coming in at the same time?
- How many sharing opportunities can KSM++ find in a mixed workload, in which not many sharing opportunities stem from the page cache?
- How much CPU consumes KSM++ in comparison to KSM?
- What is the I/O and run-time overhead of running KSM++ instead of KSM?
- How many pages does KSM++ need to visit to find a sharing opportunity compared to KSM?
- What is the longevity of sharing opportunities?
- What influence has the stability of the unstable tree on the detected sharing opportunities?
- What is the impact of the configuration parameters (stack size and interleaving ratio)?

## Server Setup

All benchmarks have been conducted on a server with the following components:

- Quad core Intel i7-2600K processor
- 24GB DDR3 RAM
- Intel X25-E SSD.

The Virtual Machines, that were used in the benchmarks have been set up to run Ubuntu Linux 11.04 in its default configuration. The host was also running under Ubuntu Linux 11.04 with a custom kernel that includes KSM++, – the prototype implementation of this work. Each of the two VMs were running on its own VCPU (Virtual Central Processing Unit), the third VCPU was used to conduct the measurements and controlling of the VMs and KSM/KSM++ was usually taking the fourth VCPU - as the implementation is single threaded (see also Table 5.1).

Table 5.2 shows the general benchmark parameters that were used as the foundation for all of the following benchmarks.

| Assignment | Process |
|---|---|
| CPU 1 | Virtual Machine 1 |
| CPU 2 | Virtual Machine 2 |
| CPU 3 | Measurements & Controlling |
| CPU 4 | KSM or KSM++ |

Table 5.1: CPU Assigments for 2 VMs, measuring and KSM

| Parameter | Value | Description |
|---|---|---|
| scan_runs | 1 ǀ 0 | Interleave each scan-spurt... |
| hint_runs | 1 ǀ 0 | ... with one hint-spurt |
| pages_to_scan | 100 | # of pages to scan on wake-up |
| sleep_time | 100 ǀ 20 ǀ 200 | Sleep-time between spurts $[ms]$ |
| stack_size | 8192 | Size of the hints stack |
| hash_table_size | 256 KiB | Size of the hash table (KSM++) |
| RAM size | 512 MiB | Size of virtual main memory |

Table 5.2: General benchmark parameters. Syntax for values is: default_value ǀ value 2 ǀ value 3 ǀ ...

## Initial Benchmark State

Two methods were used for creating a stable, reproducible state at the beginning of the benchmark: Either the benchmark starts with no shared pages or with the maximum possible sharings. Starting the benchmark with maximum sharings is more realistic for simulating long-lived VMs while starting from a clean state shows the advantages when spawning VMs. In this the warmed up state has been chosen as the initial state for all benchmarks going forward. There is a certain variance in the initial warmed up state – between 84k and 91k pages are shared at the beginning.

## Benchmark Sensors

KSM/KSM++ have been instrumented to provide statistics through the sysfs interface. The following parameters are available and have been measured every second (see Tables 5.3, 5.4 and 5.5).

| Sensor | Description |
|---|---|
| full_scans | The number of full scans of the whole memory KSM has conducted |
| pages_shared | The number of pages shared on the system |
| pages_sharing | The number of pages that take part in sharing |
| pages_unshared | The number of pages that are no longer shared |
| pages_volatile | The number of possible sharing candidates (rmap_items) |

Table 5.3: KSM default sensors

| Sensor | Description |
|---|---|
| consumed_hints | The number of hints KSM++ consumed |
| currently_pending_hints | The number of hints currently pending on the stack |
| dropped_hints | The number of hints dropped, because the stack ran out of space |
| drops_per_wakeup | ECMA value of the average that is dropped per wakeup |
| produced_hints | The number of hints produced by I/O requests |

Table 5.4: KSM++ stack sensors; stack works as a producer-consumer model

## KSM vs. KSM RO

As shown in Section 4.3 the unstable tree can degenerate: KSM/KSM++ uses the page content as index into the tree and such a write to a page can make tree nodes unreachable. In consequence KSM/KSM++ can find less sharing opportunities than if all nodes in the tree were reachable.

KSM employs two strategies to mitigate this problem: First, only sharing candidates are inserted into the tree that have not changed for one full scan of all advised memory areas by using a checksum heuristic. Second, the tree is flushed after each such scan round and build up from the start again.

However, the protection of KSM's checksum heuristic is reduced by the fact that pages change eventually and then degenerate the tree. As scan rounds can be long (e.g. 4 min for 100 ms wake-up time), the tree can be in a degenerated state for a long time and decrease the amount of sharing opportunities that KSM can find.

As KSM++ does not use the checksum heuristic, the tree degeneration is in KSM++ a greater problem than in KSM. The percentage of reachable nodes in the unstable tree after one scan round can be found in Table 5.6.

KSM++ solves this problem by mapping the pages in the tree read-only (KSM++ RO) or by replacing the unstable tree with a hash table (KSM++ HT). KSM++

| Sensor | Description; # pages... |
|---|---|
| num_mapped_hints_ro | ... coming from hints mapped read-only |
| num_mapped_scans_ro | ... coming from scans mapped read-only |
| num_remapped_rw | ... remapped read-write after fault |
| pages_hinted | ... examined based on a hint |
| pages_hint_merges | ... merged based on a hint |
| pages_scan_merges | ... merged based on the scanning |
| pages_scanned | ... examined based on the scanning |
| pages_skipped | ... skipped and not compared to the |
| scanned_in_hint_run | ... scanned after hint run (# hint pages < pages_to_scan) |
| unstable_tree_pages_counted | ... total in the unstable tree |
| unstable_tree_pages_reachable | ... reachable in the unstable tree |
| time_used | The time used by the KSM process |

Table 5.5: KSM++ map and merge sensors

| Vanilla KSM | KSM++ |
|---|---|
| 33.6% - 53.0% | 10.0% |

Table 5.6: Percentage of the unstable tree that is reachable at the end of a scan round. Processing pages without KSM's checksum heuristic degenerates the tree more. The protection of the checksum heuristic is limited and the unstable tree still degenerates.

also removes the flushing of the tree after each scan round. In order to compare KSM++ RO/HT to KSM and remove the effect of "degeneration", KSM's unstable tree pages are mapped also read-only and this approach is called KSM RO.

Zero pages are pages that contain only zeroes and those pages are often used to denote free memory. Therefore sharing zero pages gives a false sense of free memory in the host as those pages could be used by the guest at any time. While Vanilla KSM merges zero pages, the modified version of KSM used in this work, does not merge zero pages. Whenever the term *KSM* is used in the following benchmarks, it is not Vanilla KSM, but Vanilla KSM with zero page merging removed.

The most important parameter differences for the five approaches (Vanilla KSM, KSM RW/RO, KSM++ RO/HT) are shown in Table 5.7.

| Parameter | Vanilla KSM | KSM | KSM RO | KSM++ RO | KSM++ HT |
|---|---|---|---|---|---|
| hinting | - | 0 | 0 | 1 | 1 |
| vfs_hinting | - | 0 | 0 | 1 | 1 |
| vfs_hinting_writes | - | 0 | 0 | 1 | 1 |
| hashtable | - | 0 | 0 | 0 | 1 |
| map_scans_ro | - | 1 | 1 | 1 | 0 |
| map_hints_ro | - | - | - | 1 | 0 |
| merge_zero_pages | 1 | 0 | 0 | 0 | 0 |
| skip_reset_tree | 0 | 0 | 1 | 1 | 1* |

Table 5.7: Parameters and their differences in the different approaches; *the hash table is never reset

## 5.2 Effectiveness

This work defines *Effectiveness* as the number of pages that can be shared at any given point in time. In the following it will be demonstrated that KSM can find most short-lived sharing opportunities that stem from the buffer cache of VMs if it is extended by I/O hints.

First, the maximum possible sharing opportunities are determined without using pages. This optimum is used to evaluate how much of the sharing potential can be found when I/O-based hints are used. The vm_snapshot kernel module [9] was used to measure the amount of sharing opportunities when running different benchmarks. Vm_snapshot – a tool similar to Exmap [4] – records information about page tables and page contents at a high frequency (on average once per second). It delivers information such as the number of sharing opportunities, the number of zero pages and the achievable memory savings.

The results of different benchmarks, namely Kernel Build, Apache and Mixed, are discussed seperately in the following Sections. In the following graphs a steeper rise means that sharing opportunities are found and merged at a higher rate. A higher curve is an indicator that more pages are shared in the system. Both symbolize a better merge effectiveness.

This work distinguishes between four different approaches, namely *KSM++ HT, KSM++ RO, KSM and KSM RO*. This work defines these terms as shown in Table 5.8. *HT* denotes that the unstable tree was replaced with a hash table, while *RO* means that all pages that are inserted in the unstable tree are mapped read-only.

| Term | Definition |
|---|---|
| KSM | Vanilla KSM with zero page merging removed;<br>Pages in the unstable tree are mapped read-write |
| KSM RO | Like KSM;<br>Pages in the unstable tree are mapped read-only |
| KSM++ RO | KSM extended by I/O hinting mechanism;<br>Pages in the unstable tree are mapped read-only |
| KSM++ HT | KSM extended by I/O hinting mechanism;<br>The unstable tree is replaced with a hash table |

Table 5.8: The term definitions of the four approaches used in this work.

## Kernel Build

The first benchmark, namely KBUILD-512, shows that KSM++ can find short-lived sharing opportunities that stem from the buffer cache of the VMs. In this benchmark two VMs simultaneously build an identical Linux kernel version. This workload opens files, compiles them and writes the compiled code as objects to disk. The objects are read from disk again, linked and written back as shared objects or executables. The workload is both CPU and I/O bound, i.e. it needs to use both the CPU and the disk. Sharing opportunities can be found in the I/O cache of the guest OS, e.g. when the same files are requested, and in the heap memory of the compiler, e.g. when the compiler creates the parse tree of the source file before writing the compiled object to disk. Both VMs are configured to use 512 MB memory. KBUILD-512 has been already used by the Satori [19] paper.

Both VMs are fully booted and warmed up before the benchmark is started as explained in Section 5.1. Both VMs start an identical kernel build at the same time. The expectation is to find the majority of the sharing opportunities that stem from the buffer cache of the VMs as long as KSM++ can keep up with the amount of hints coming in from the VFS layer.

The parameters that differ from the defaults (Table 5.2) are listed in Table 5.9.

Four benchmark runs were conducted for wake-up times of 20 ms, 100 ms and 200 ms (4x3 benchmarks). Scanning 100 pages at a wake-up interval of 100 ms is the default for KSM in Ubuntu.

| Parameter (KSM++) | 20 ms | 100 ms | 200 ms |
|---|---|---|---|
| stack_size | 40960 | 8192 | 4096 |

Table 5.9: Parameters used in the Kernel build and Apache benchmarks based on the wake-up times

The results of the benchmark can be found in Figure 5.2. Figure 5.1 shows that at a wake-up interval of 20 ms, KSM++ can merge the majority of the sharing opportunities found by vm_snapshot [9]. It also shows that KSM++ merges more pages at a 100 ms wake-up interval, than KSM with 20 ms. This means that in this benchmark KSM++ merges sharing opportunities about 5 times more effectively than KSM. The curve tracesthe optimum sharing opportunities closely. Important is that the benchmark is started with 86k - 91k shared pages (due to the warmup phase). This explains why the curves of the different approaches not start at the exact same point even though the state of the VMs *should* be the same at this point in time. All four approaches first loose around 25% of the active sharings, before the curve starts to climb again. The reason is that previously shared pages are broken up when the benchmark starts and the kernel starts to re-claim buffered and cached pages. This is similar to the development of the optimum sharing opportunities.

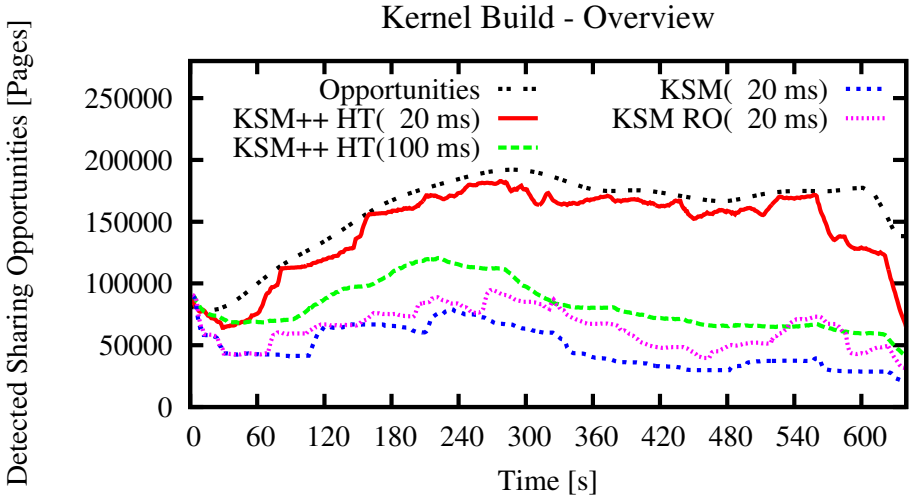


Figure 5.1: Kernel Build merge performance with varying wake-up times showing both KSM++ and KSM (RO). KSM++ (20 ms) is very close to the optimum sharing opportunities. KSM++ (100 ms) is still better than KSM (20 ms) and most times better than KSM RO (20 ms).

Table 5.10 shows the number of pages KSM++ has generated a hint for compared to the number of successful merges. On average the findings show a ratio of 25-45%, which means that out of 100 hinted pages, 25 to 45 lead to a successful merge. Given that 2 hints are needed to produce one merge, hints are correct 50-90% of the time in this benchmark. KSM on the other hand has an average ratio of 9-11% for pages scanned to pages merged (from scanning).

Additionally to the pages that are found during hint spurts, KSM++ merges 2-3% of the pages through scanning. These pages are likely to be non buffer cache pages, such as buffers of both the applications (heaps) and the operating system running in the VM (slabs).

| Scan Rate (KSM++) | Pages Hinted | Pages Merged | Dropped Hints | Ratio |
|---|---|---|---|---|
| 20 | 423266 | 188454 | 451017 | 45% |
| 100 | 236459 | 74602 | 610884 | 32% |
| 200 | 140700 | 35237 | 1066879 | 25% |

Table 5.10: Merge performance of the KSM++ hinting mechanism for the Kernel build configuration with varying wake-up times

The curves for 100 ms and 200 ms for KSM++ are significantly lower and not even close to the maximum. The reason for this is that KSM++ cannot process the hints fast enough as they come in. This leads to dropped hints, which in the worst case could mean that KSM++ can find no sharing opportunities at all from hinting. For example if KSM++ drops one of the sharing partners, it needs to rely on the memory scanner to find the other sharing partner in time. The number of dropped hints is a good indicator of the lost sharing potential – provided that the I/O hints have sharing potential like in this benchmark, e.g. when identical files are requested.

In this benchmark KSM is only able to find a significant amount of sharing opportunities using a wake-up time of 20 ms. Then, the scan rate is high enough to find sharing opportunities stemming from the page cache in time.

With 20 ms and 100 pages scanned per spurt, KSM is able to scan 5,000 pages per second (pps), while with 200 ms it can only scan 500 pps. Once both VMs utilize the full amount of their assigned memory (429 MB) the advised memory areas that are scanned contain 220,000 pages, which means that a full scan of the memory takes 44/220/440 s with wake-up times of 20/100/200 ms. The Kernel build benchmark has a run time of around 660 s. This means that with higher wake-up intervals (>=100 ms) KSM starts finding buffer cache pages only after 1/3 or 2/3 of the benchmark time has passed. The buffer cache size is around

94,000 pages and at the time point of 220/440 seconds there has been a total traffic of 65,000/136,000 pages. Consequently for a wake-up time of 100 ms most pages have been already evicted from the guests buffer cache before KSM considers them as a sharing candidate. For the case of 200 ms wake-up interval KSM cannot find buffer pages at all – the time to scan the advised memory areas is too long.

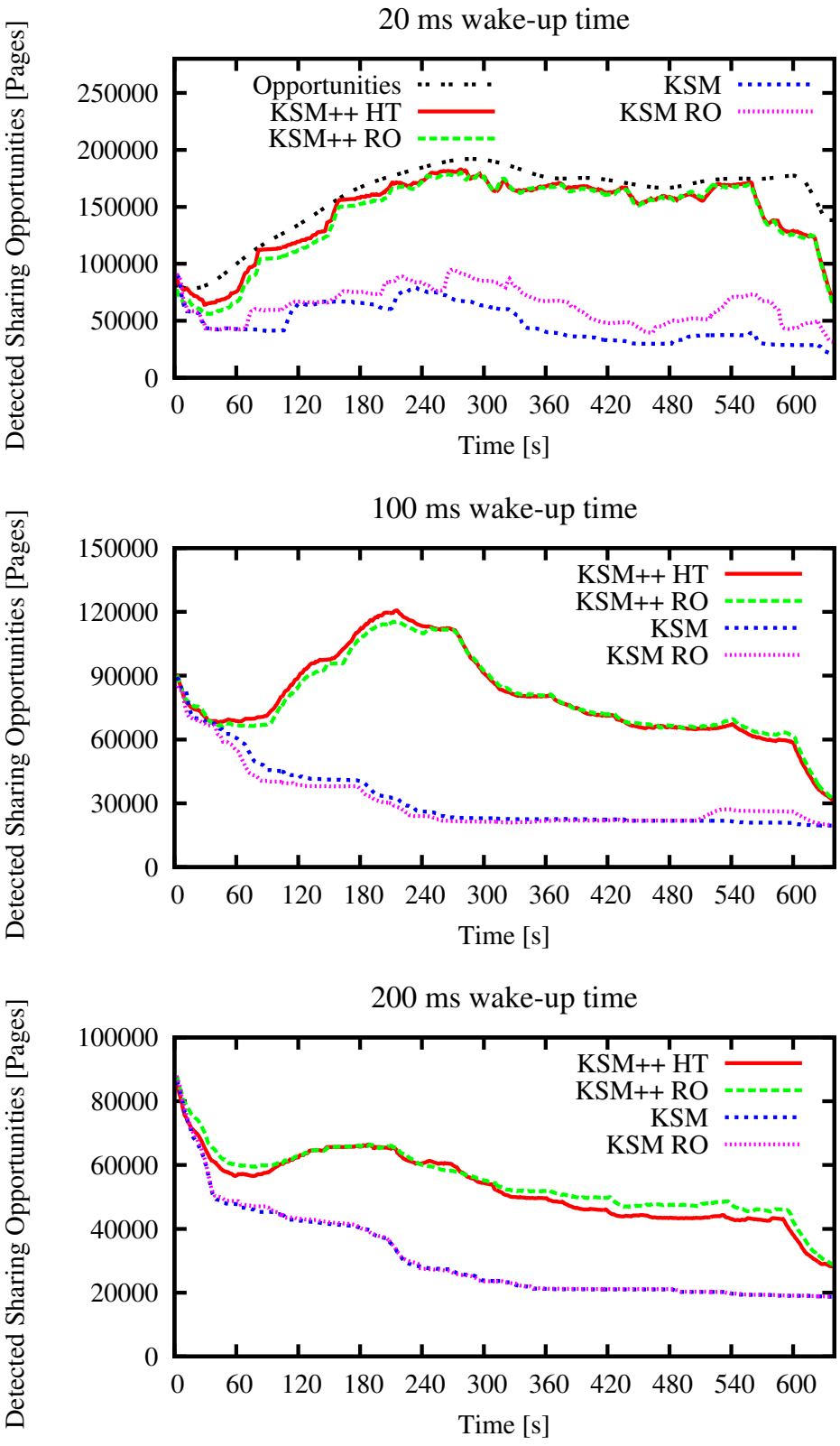


Figure 5.2: Kernel build merge performance at 100 pages per wake-up with varying wake-up times. Top to bottom: 20 ms, 100 ms, 200 ms. With a wake-up time >= 100 ms KSM does not find new sharing opportunities stemming from the page cache.

## Apache

Another common scenario is the virtualization of webserver farms. In this scenario a good sharing ratio is predicted if similar I/O is requested. Again a benchmark utilized already in the Satori [19] paper was reproduced. As in the kernel build benchmark the VMs are fully booted, before the benchmark process is started. The httperf benchmarking tool runs on another physical machine, which is connected to the benchmark machine via gigabit ethernet. Httperf requests files serially as unary tree like 0/0/0/0.html, 0/0/0/1.html, etc. To create a random distribution two disk images are attached to the VMs with the same files, but in a different randomized order. The result set has a total size of 512 MB, which exceeds the buffer cache of the VMs, which is sized dynamically but can grow up to 368 MB.

HTTPerf was set to a request rate so that KSM++ can process all incoming I/O requests and does not need to drop hints. In order to not achieve superior performance at a wake-up interval of 20 ms, but find no sharing opportunities with 200 ms, the request rate was adjusted based on the scan-rate. The parameters used for httperf can be found in table 5.11.

| HTTPerf Option | 20 ms | 100 ms | 200 ms |
|---|---|---|---|
| Request Rate [req/s] | 20 | 10 | 5 |
| Num Connections | 24000 | 12000 | 6000 |

Table 5.11: Varying Parameters for the httperf benchmarking tool per wake-up time. Request rate is how many requests httperf sends per second. The number of connections is how many connections httperf creates totally. The benchmark runs always 20 min.

This benchmark shows that KSM++ also finds sharing opportunities when the I/O is received at different times. Again, the sharing performance for 20, 100 and 200 ms of all four approaches has been measured. The results can be found in Figure 5.4 where the maximum sharing opportunities are also depicted.

The results show that KSM++ is effective in finding sharing opportunities stemming from the guests page cache even if the page cache contents are varying over time. At a wake-up interval of 20 ms KSM++ is able to achieve almost the optimum of sharing opportunities. At a wake-up interval of 20 ms the curve is rising slower, but steady. KSM on the other hand can find only few sharing opportunities that stem from the page cache – even with a wake-up interval of 20 ms.

The overview in Figure 5.3 shows that at 540 s KSM++ HT finds more sharing

opportunities with 100 ms than with 20 ms. Such KSM++ HT (100 ms) seems to be able to share more pages than KSM++ HT (20 ms). The reason is that the httperf parameters are adjusted per wake-up interval. Therefore a comparision between 20 ms and 100 ms will need to take into account that the incoming I/O rate for 100 ms is half the I/O rate of 20 ms. To compensate for this effect and make the two scan rates comparable, the x-axis has been scaled by a factor of 0.5.

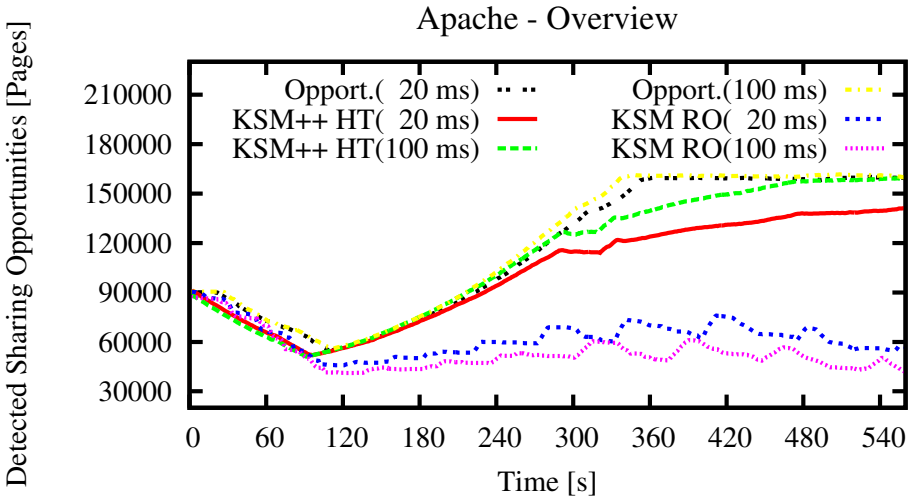


Figure 5.3: Apache merge performance with varying wake-up times showing both KSM++ and KSM (RO). KSM++ with 20 ms is very close to the optimum sharing opportunities. the 100 ms curves have been scaled by 0.5 on the x-axis, because the 100 ms I/O request rate is half the 20 ms request rate.

In Figure 5.4 it can be seen that KSM++ HT/RO (20 ms) starts to drop hints and looses sharing opportunities, while KSM++ (100 ms) can keep up with the load.

While KSM++ (100 ms) can merge more sharing opportunities than KSM++ (20 ms), the opposite is true for KSM. KSM (100 ms) cannot find the same amount of sharing opportunities, because the scan rate is too low to find identical pages before the content of the buffer caches change.

Table 5.12 shows the number of pages KSM++ has generated a hint for compared to the number of successful merges. On average the findings show a ratio of 25-39%, which means that out of 100 hinted pages, 25 to 39 lead to a successful merge. Given that 2 hints are needed to produce one merge, hints are correct 50-78% of the time in this benchmark. KSM on the other hand has an average ratio of 4-8% for pages scanned to pages merged (from scanning).

KSM++ (200 ms) has a lower merge ratio of 25% for hinted pages, because the

| Scan Rate (KSM++) | Pages Hinted | Pages Merged | Dropped Hints | Ratio |
|---|---|---|---|---|
| 20 | 579833 | 226535 | 23214 | 39% |
| 100 | 306292 | 116855 | 975 | 38% |
| 200 | 161731 | 41095 | 0 | 25% |

Table 5.12: Merge performance of the KSM++ hinting mechanism for the Apache benchmark with varying wake-up times

benchmark in this interval does not have as many sharing opportunities than in the 20 and 100 ms cases – i.e. the page caches are still being filled and due to the random order not many pages are similar. By running the benchmark longer this work expects to find a merge ratio of around 38% for 200 ms as well.

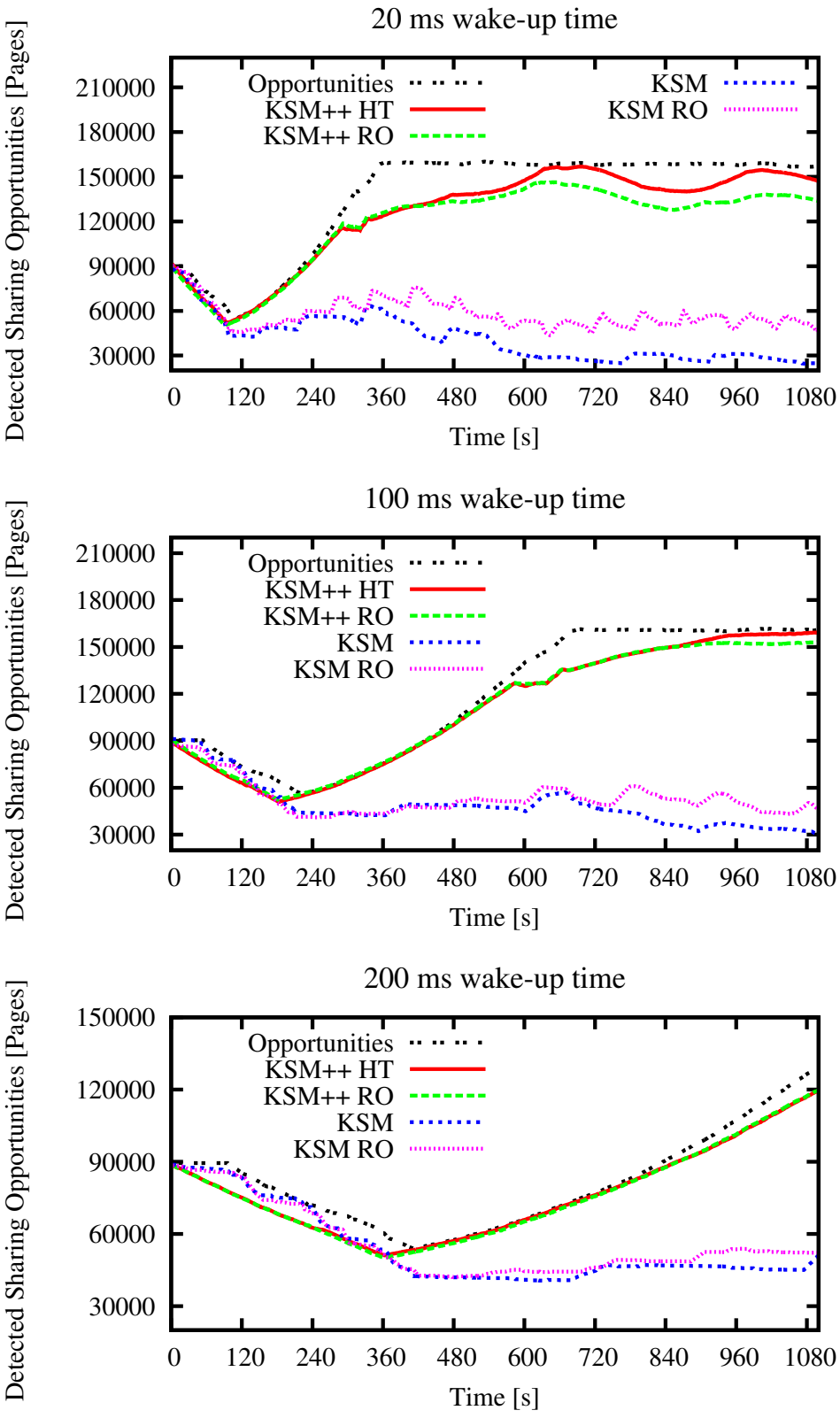


Figure 5.4: Apache merge performance at 100 pages per wake-up with varying wake-up times. Top to bottom: 20 ms, 100 ms, 200 ms. With a wake-up time >= 100 ms KSM finds only few sharing opportunities stemming from the page cache.

## Mixed - Worst-Case scenario

So far only benchmarks have been shown where KSM++ has a clear advantage compared to KSM as the I/O had been suited for many sharing opportunities that come from the buffer caches of the VMs. The Mixed benchmark mixes the Apache and kernel build scenarios to simulate two VMs with completely different workloads. VM #1 is building the kernel, while VM #2 is conducting the Apache benchmark. It can be expected that no sharing opportunities can be found besides the application and operating system pages and similar pages per VM. All results for the four approaches (KSM HT, KSM++ RO, KSM RW and KSM RO) can be found in Figure 5.6.

Figure 5.5 shows an overview of the best configurations for this workload.

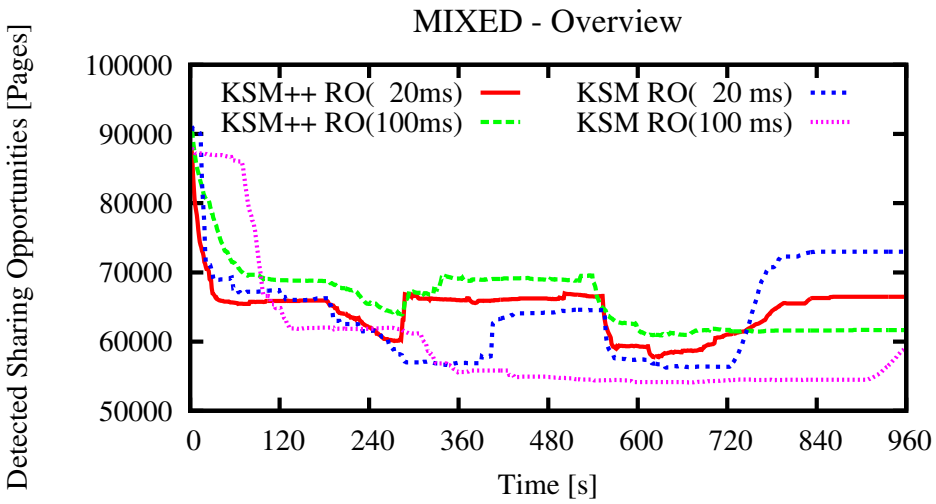


Figure 5.5: Mixed Overview

It is important to note that the I/O generation rate of httperf was scaled to the scan rate as described in the Apache benchmark, but the kernel build is the same for every scan rate. This effect can be seen for KSM RO (100 ms) for which the curve descends slower than for KSM RO (20 ms) in the interval between the start and 120 s.

In the middle of the benchmark (in the interval between 300 s and 540 s) KSM++ HT (100 ms) finds the most sharing opportunities, followed closely by KSM++ HT (20 ms). At 960 s KSM RO (20 ms) is better then KSM++ HT (20 ms). The reason is that the kernel build benchmark ends at around 660 s and the remaining sharing opportunities are coming just from the Apache benchmark. The curves

for KSM++ (20 ms) and KSM++ (100 ms) are very close together. The reason for this is that the sharing opportunities stem from self sharing in the VM running the Kernel build.

Table 5.13 shows the number of pages KSM++ has generated a hint for compared to the number of successful merges. On average the findings show a ratio of 2-5%, which means that out of 100 hinted pages, 2 to 5 lead to a successful merge. Given that 2 hints are needed to produce one merge, hints are correct 4-10% of the time in this benchmark. KSM on the other hand has an average ratio of 1% for pages scanned to pages merged (from scanning).

| Scan Rate (KSM++) | Pages Hinted | Pages Merged | Dropped Hints | Ratio |
|---|---|---|---|---|
| 20 | 366815 | 16685 | 190928 | 4.5% |
| 100 | 226599 | 11450 | 244703 | 4.7% |
| 200 | 185402 | 3558 | 314996 | 1.9% |

Table 5.13: Merge performance of the KSM++ hinting mechanism for the Mixed benchmark with varying wake-up times

Additionally to the pages that are found during hint spurts, KSM++ merges 0.1-0.3% of the pages through scanning.

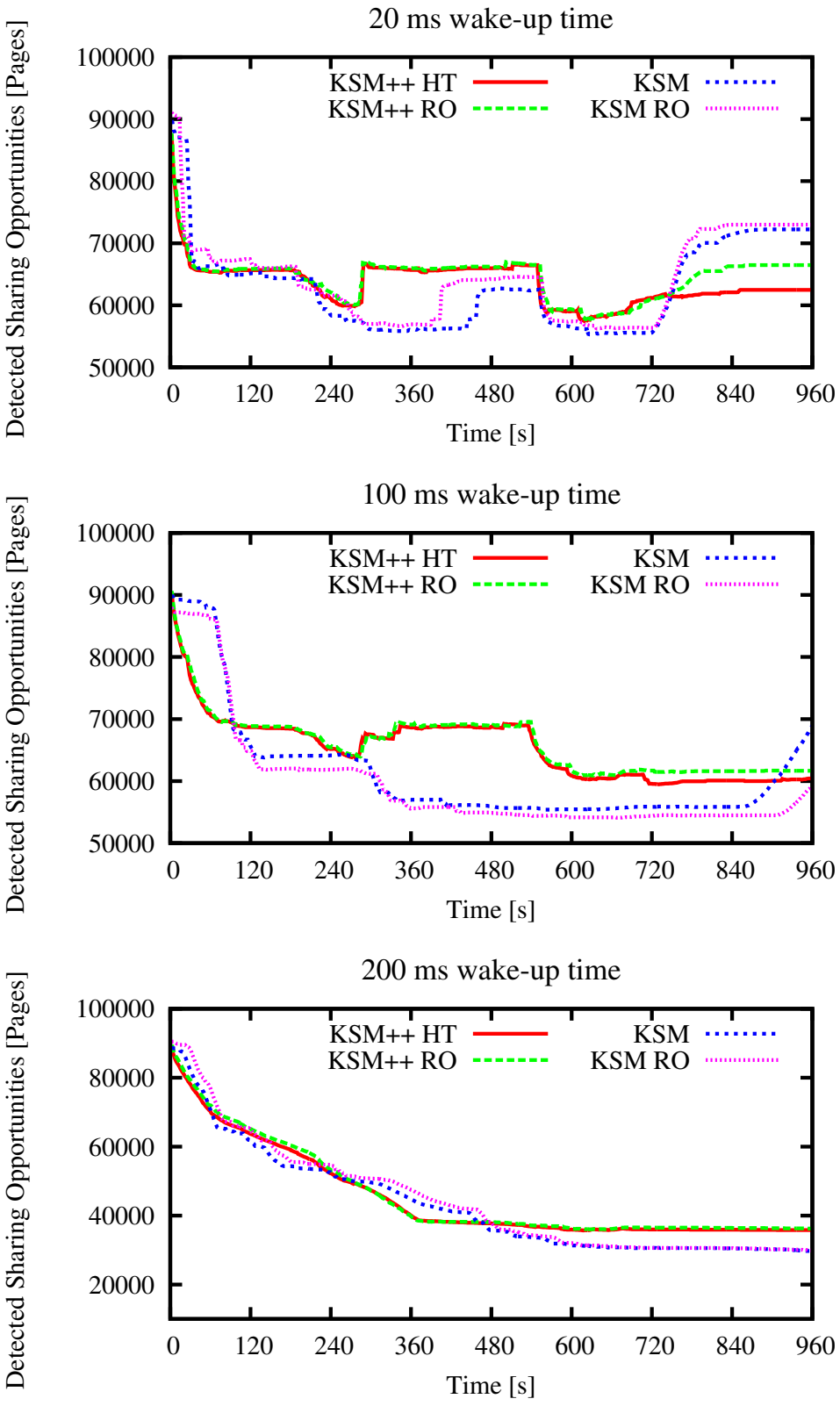


Figure 5.6: Mixed merge performance at 100 pages per wake-up with varying wake-up times. Top to bottom: 20 ms, 100 ms, 200 ms.

## 5.3 Efficiency

*Efficiency* is defined in this thesis as the work that needs to be done until a successful merge operation. The work that has to be done consists of visiting a certain amount of pages, and in case of KSM++ additionally of the overhead of the hinting mechanism which can be further divided into the overhead for creating, storing, and processing hints. Another important factor is the influence on the applications' run-time performance and the I/O throughput. Useful metrics to cover this are the number of pages that need to be scanned for each page that is actively shared in the system and the ratio of pages visited that were successfully merged. These metrics are defined in this thesis as the sharing and merge performance of a benchmark.

In this section the following benchmark results are presented and discussed:

- TOP – CPU Consumption of the KSM Process
- BONNIE++ – I/O Overhead and Runtime
- Kernel Build – Sharing and Merge performance
- Apache – Sharing and Merge performance
- Mixed – Sharing and Merge performance

The objective is to show how efficient KSM/KSM++ is and that KSM++ can find more sharing opportunities with using roughly the same CPU consumption without having to adjust the scan rate – i.e. that it finds pages faster and earlier than KSM also with the standard scan rate of KSM (100 ms wakeup time).

### CPU Consumption

The CPU consumption has been measured with the Linux commandline tool top for the kernel build benchmark every second. This was done to measure what a wake-up time of e.g. 20 ms means in terms of system load. The results can be found in Table 5.14.

On average a wake-up interval of 20 ms means that KSM/KSM++ uses one CPU core to around 68%, with 100 ms to 31% and with 200 ms to 17%. The read-only solution needs more CPU cycles in most cases – as expected. With 100 ms KSM++ uses around 5-6% more CPU than KSM, though nowhere near the 68% that standard KSM would need to find short-lived sharing opportunities. The reason is that KSM++ with 100 ms merges a lot more pages than KSM, because KSM

| Approach | 20 ms | 100 ms | 200 ms |
|---|---|---|---|
| KSM++ HT | 67.05% | 33.61% | 16.94% |
| KSM++ RO | 66.19% | 34.13% | 16.17% |
| KSM | 68.75% | 27.47% | 16.32% |
| KSM RO | 68.92% | 28.12% | 17.52% |
| Average | 67.72% | 30.83% | 16.74% |

Table 5.14: CPU consumption measured with `top` for the four different approaches.

will skip many of the pages in the page cache as they don't fullfill the checksum heuristic (i.e. KSM is too slow to find these sharing opportunities). Therefore the CPU cycles that KSM++ uses more, have been used for finding and merging pages.

## BONNIE++

KSM++ is not only effective at finding sharing opportunities, but that it is also more efficient in doing so. Efficiency can be divided into the measured results and the effort needed (overhead) to receive this results. The overhead of KSM++ compared to KSM is mainly the handling of I/O requests and the removal of the checksum mechanism for pages stemming from I/O hints. While the bigger overhead in the memory scan process can easily be justified by finding more sharing opportunities, the overhead of inspecting I/O requests has direct influence on the performance of applications. Another less visible overhead is present both for KSM and KSM++ compared to a system not running a memory scanner in that KSM locks pages into memory while merging them. This work only briefly inspects the latter overhead and mainly focuses on the overhead induced by the introspection and handling of I/O hints.

In this it is postulated that the overhead for handling I/O requests consists of the overhead to translate the buffer into a VM region (via `madvise`) and for pushing the memory region to the hints buffer. The hint buffer uses locking, which might indicate that there is a correlation between scan rate and I/O performance.

The Bonnie++ I/O benchmark suite has been used to measure the throughput and the latency of the four approaches. Bonnie++ was setup to run for exactly 30 seconds. Due to differences in scheduling and the overhead of KSM/KSM++ the runtime can vary slightly and has been measured. Note that the throughput and latency are displaying the read performance for block-based reading, but the

runtime does also include the effects of writing.

- runtime is a good indicator of overhead
- reading is a good indicator for I/O performance
- reading is a good indicator for I/O latency
- reading is a good indicator for the impact of hinting
- runtime is a good indicator of page locking by KSM

The benchmark has been run without merging any pages to remove the overhead of marking pages copy-on-write and breaking them again. The objective is to only measure the overhead of the added I/O hinting mechanism and the scanning of pages. In order to do that the pages are still scanned, locked in memory (i.e. PAGE_GET), but not inserted in the unstable tree or checksummed. However, they are compared to a zero page before they are being put back.

The results can be found in Figure 5.7. Each benchmark was run 30 times and the results have been stripped to the 0.05/0.95 quantile.

As the latency had some very high values (10000+), those results have been removed from the result set. As the runtime had some very high values (33000+), those results have also been removed from the result set.

The figures show both the arithmetic mean and the .05 and 0.95 quantiles for each benchmark.

For the throughput it can be seen that KSM++ HT is always slightly better than KSM++ RO.

In the latency with a wake-up time of 200 ms KSM++ is worse than KSM by around 1000 us. A possible reason for this is that KSM never takes a look at the pages affected by bonnie in the buffer cache while the hinting mechanism processes those pages.

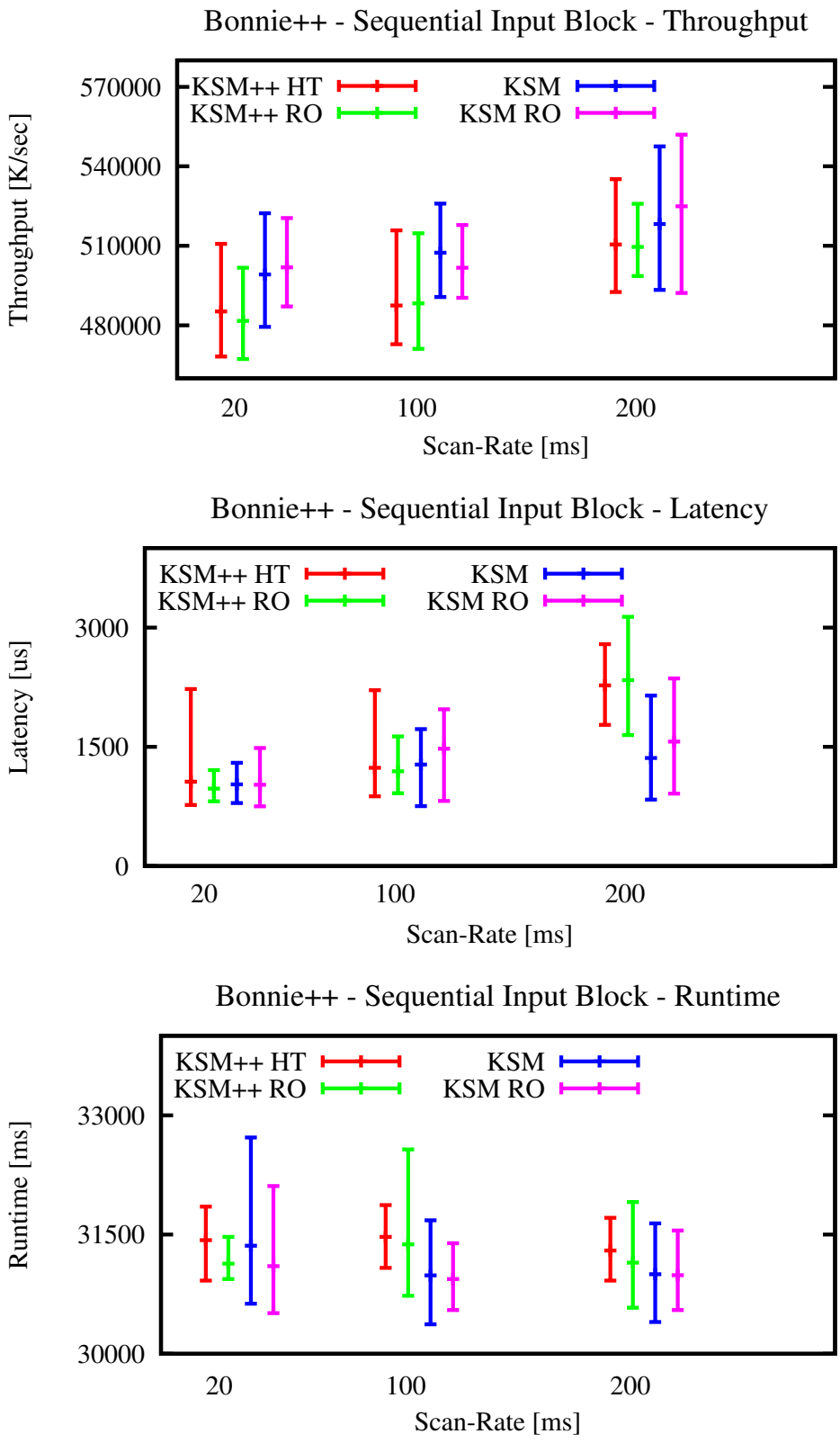


Figure 5.7: Bonnie++ Benchmark - Sequential Input Block – *Top*: Throughput *Middle*: Latency *Bottom*: Run time of the whole bonnie benchmark

## Kernel Build

The Kernel build benchmark explained in Section 5.2 has already shown that KSM++ can in this benchmark share on average more pages than KSM, e.g. with a wake-up time of 100 ms KSM++ is able to share 2-4 times more pages than KSM. With 100 ms KSM++ needs to consume slightly more CPU (around 6%) to reach that goal. The question remains how much work needs to be done to find and merge a sharing opportunity. It is the objective here to show, that KSM++ does not need to scan more pages, but makes smarter choices of which pages to scan.

Two metrics have been used for the sharing and merge performance and will be presented for the following benchmarks:

- Visited Pages per Sharing – How many pages does KSM/KSM++ need to visit for each shared page in the system?
- Visit to Merge Ratio – How many pages does KSM/KSM++ need to visit to find a merge partner?

The overview results for the kernel build benchmark are depicted in Figure 5.8. The full results can be found in the appendix in Figure A.1.

In the Kernel build benchmark two VMs build the Linux kernel simultaneously and such produce the same I/O requests at roughly the same time. The goal is to show how good KSM/KSM++ can exploit sharing opportunities that come in at the same time.

For the sharing performance (top part of Figure 5.8) it can be seen that one the one hand KSM++ needs to visit only around 5-10 pages for each shared page and is quite constant in this interval over the run-time of the benchmark. KSM on the other hand has a very varying number of pages (10-25) to visit per share. In general KSM++ is more efficient as it needs to scan around 50% less pages than KSM. As expected with a wake-up time of 20 ms KSM++ needs to visit the lowest number of pages (5-7), but this does not show how many of the visited pages lead to a successful merge.

In the bottom part of Figure 5.8 the merge performance is depicted. As in most cases 2 merge partners are needed to find a match, 50% is an upper bound for the visit to merge ratio.

KSM++ using a wake-up interval of 100 ms has the highest percentage of successful merges (20%), while KSM reaches a maximum of 10% and only with a wake-up interval of 20 ms. While KSM++ had been the clear winner in terms

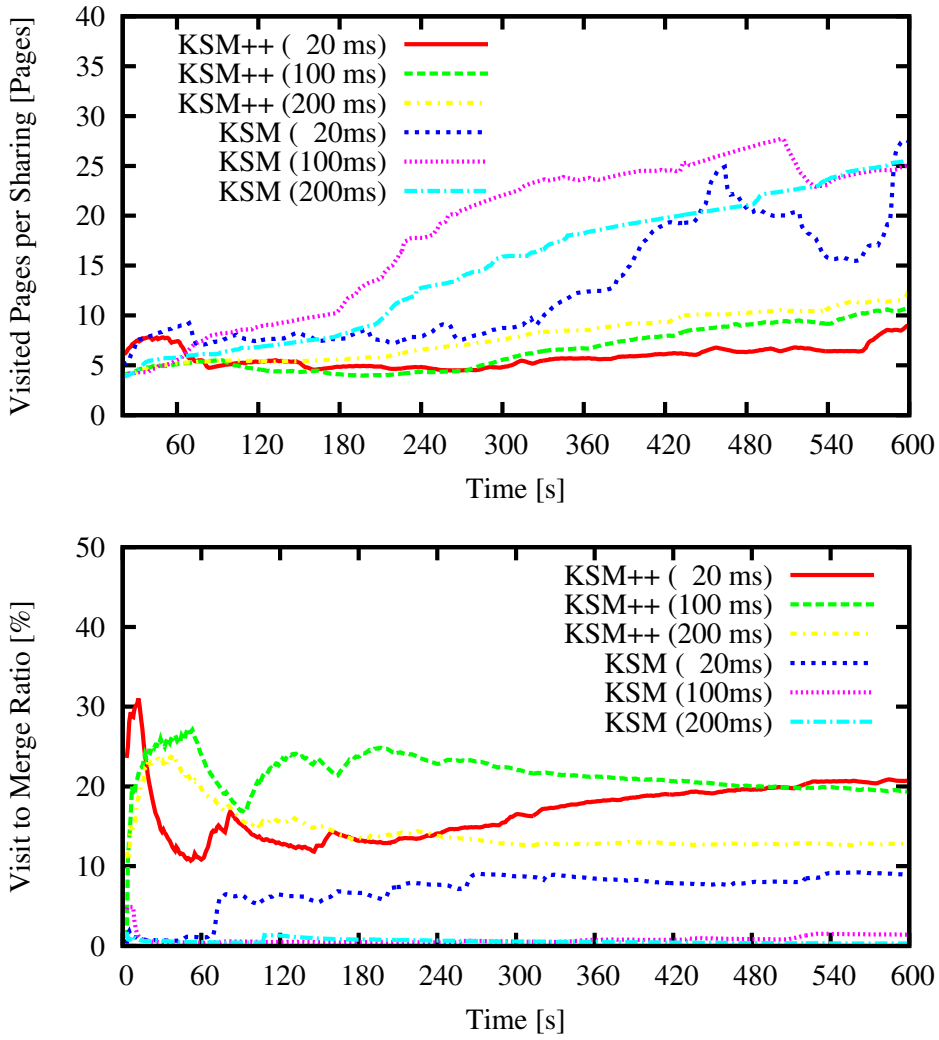


Figure 5.8: Efficiency - Kernel Build – *Top*: Visited pages to find sharing opportunities – Overview for different wake-up times*Bottom*: Visit to Merge Ratio – Percentage of visited pages leading to a successful merge

of pages to visit per sharing, a better merge performance can be reached with a slower scan rate.

The reason is that KSM++ (20 ms) can easily cope with the amount of incoming I/O hints. This means that a lot of time is used for scanning pages in a hint spurt (around 100k pages), which in this benchmark does not lead to more page merges (only 2-3% of all the merges stem from scanning).

KSM using a wake-up time of >=100 ms can find almost no merges: The scan rate

is too slow to find pages stemming from the page cache. This matches the results from Section 5.2.

In conclusion KSM++ is suitable for this workload, though both KSM++ (20 ms) and KSM++ (100 ms) have different advantages. By carefully tweaking the scan rate to the workload used, good share and merge ratios can be reached.

## Apache

The share and merge performance of KSM/KSM++ in the Apache benchmark is depicted in Figure 5.9. The full results for all wake-up times can be found in the appendix in Figure A.2.

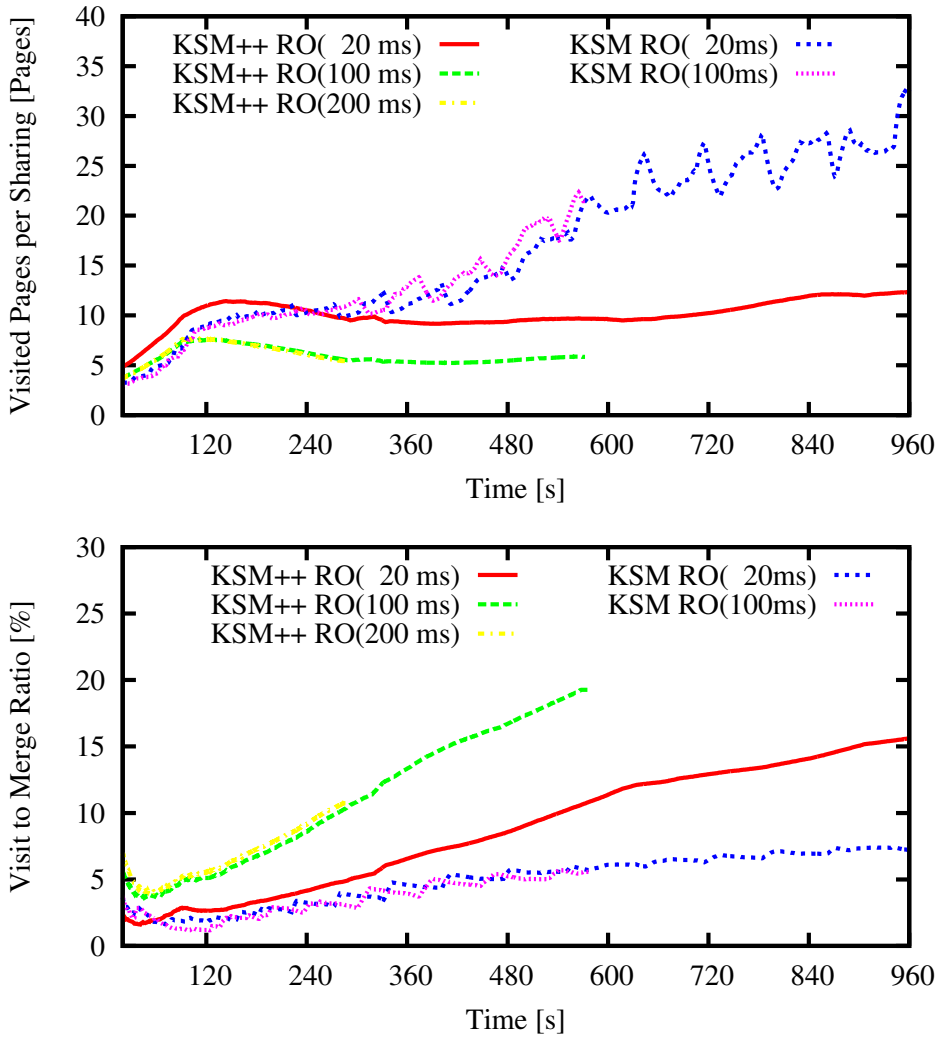


Figure 5.9: Efficiency - Apache – *Top*: Visited pages to find sharing opportunities – Overview for different wake-up times*Bottom*: Visit to Merge Ratio – Percentage of visited pages leading to a successful merge

The Apache benchmark simulates a webserver farm. Apache is serving the same random files at different times. The goal is to find out how good KSM/KSM++ can cope with I/O requests that have sharing potential, but are not coming in at the

same time (like in the Kernel build benchmark). The I/O generation rate has been adjusted to the scan rate, so that KSM++ is able to process most hints.

The curves for 100 ms and 200 ms are shorter, as the results have been scaled on the x-axis (explained in Section 5.2). KSM++ needs to visit between 5-10 pages for each shared page, while KSM needs to visit 10-25 pages.

It can be seen that KSM++ (100 ms/200 ms) needs to visit the least pages here. Both can cope with the (adjusted) I/O generation rate, but are scanning less pages than KSM++ (20 ms).

In terms of merge performance KSM++ has a ratio between 2% and 20%, while KSM has a visit to merge ratio between 2% and 7.6%. Again KSM++ (100 ms/200 ms) can merge more pages per visited page and is such using less CPU cycles than KSM++ (20 ms), i.e. KSM++ (20 ms) is visiting pages with no sharing potential.

In conclusion, KSM++ with lower scan rates and a wake-up interval of 100 ms/200 ms is the most efficient solution for this workload.

## Mixed

The share and merge performance of KSM/KSM++ in the Mixed benchmark is depicted in Figure 5.10. The full results for all wake-up times can be found in the appendix in Figure A.3.

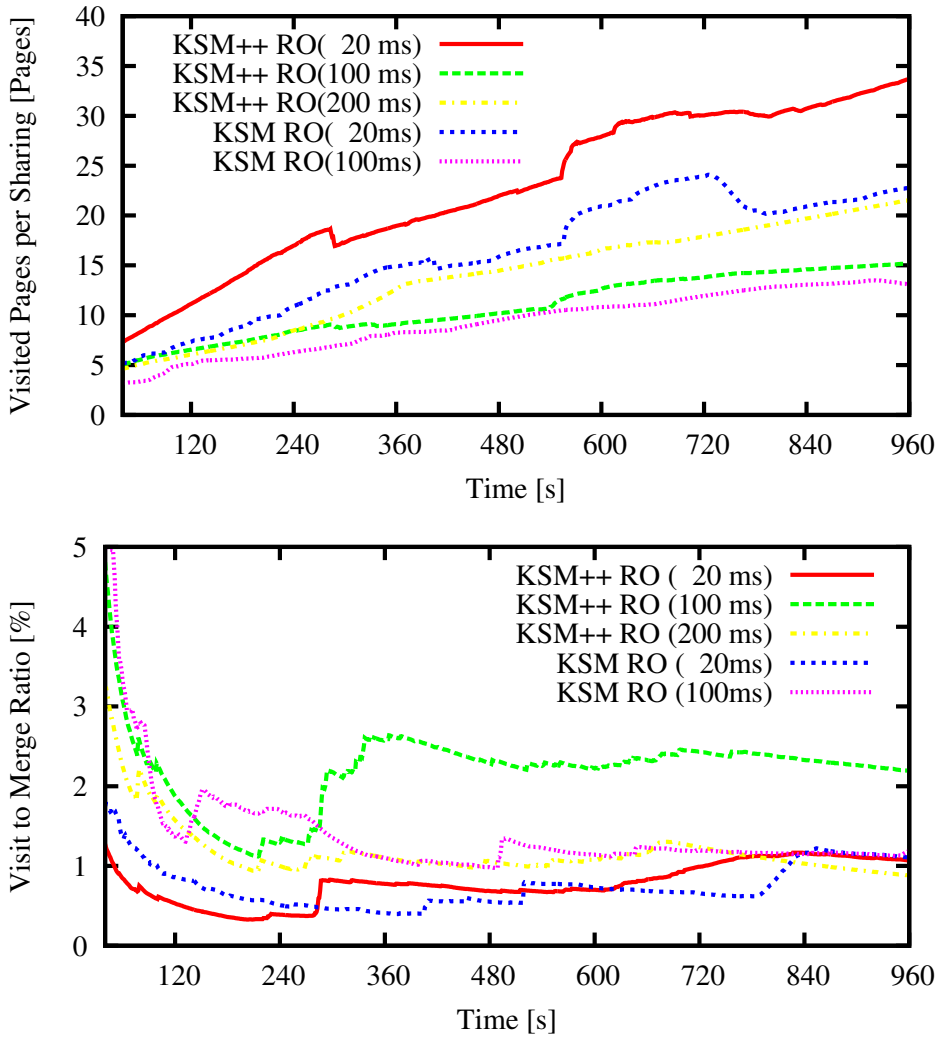


Figure 5.10: Efficiency - Mixed – *Top*: Visited pages to find sharing opportunities – Overview for different wake-up times*Bottom*: Visit to Merge Ratio – Percentage of visited pages leading to a successful merge

The Mixed benchmark is a mix of the Kernel build and Apache benchmarks, running each in their own VM. The goal is to find out how good KSM/KSM++ can

cope with a worst case scenario, when there is not much sharing potential that stems from the page cache.

In the top graph it can be seen that KSM++ needs to visit 5 to 25 pages per shared page, while KSM needs to visit between 5 and 15 pages. While the share performance of KSM++ (20 ms) is worse than KSM (20 ms), KSM++ (100 ms) reaches around the same amount of pages to visit (4-5) as KSM (100 ms).

In terms of merge performance KSM++ has a visit to merge ratio between 0.3% and 2.5%, while KSM has a ratio between 0.8% and 2% (This ignores the breaking of already shared pages by the warmup at the beginning).

While KSM++ (20 ms) is slightly worse than KSM (20 ms), KSM++ (100 ms) is partially better than KSM (100 ms) and reaches overall a better merge performance.

In conclusion KSM++ (100 ms) is more efficient, than KSM++ (20 ms) for the Mixed workload. KSM (100 ms) may be better suited for a mixed workload than KSM++ and tweaking the interleaving ratio of scan spurts to hint spurts might gain a larger efficiency for KSM++.

## 5.4 Additional Research

In addition to the research on the effectiveness and efficiency of the approach described in this thesis, further research has been conducted on the following characteristics:

- The longevity of the sharing opportunities in one benchmark
- The effect of the degeneration of the unstable tree on the number of found sharing opportunities
- The effect of the introduced configuration-parameters, stack-size and interleaving ratio, on the number of found sharing opportunities

A wake-up interval of 100 ms has been used for all benchmarks in this section.

### Lifetime of Sharing Opportunities

To motivate the work in this thesis a pre-benchmark has been conducted to find the longevity of the sharing opportunities. Vm_snapshot [9] has been used while running the Kernel build benchmark. The results are shown in Figure 5.11.

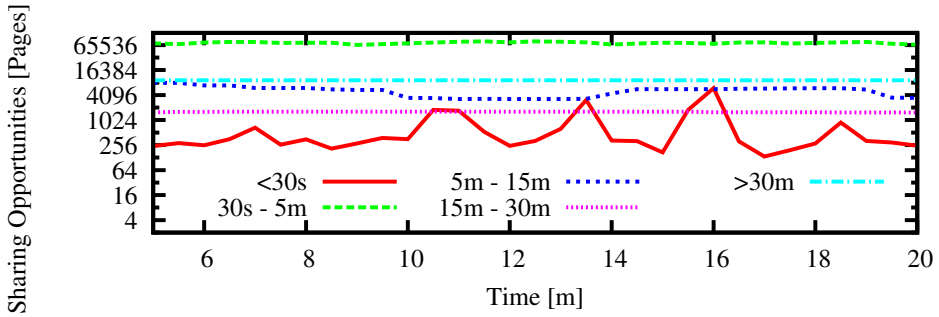


Figure 5.11: The longevity of the sharing possibilities between two VMs that compile the Linux kernel at the same time. Note the logarithmic scale. Sharing opportunities that last between 30 sec and 5 minutes are by far the most common [18].

It can be seen that 80% of the short-lived sharing opportunities come from the interval between 30 sec and 5 minutes. To put this interval into perspective: One scan round of KSM (100 ms) takes 4 min and to find one sharing opportunity with checksum heuristic takes between 4 and 8 min.

## Stability of the Unstable Tree

KSM uses a red-black tree as lookup data structure, which contains potential sharing candidates. This tree gets corrupted when applications write to pages that are contained in the tree, because KSM uses the page contents as index into the tree. Therefore, KSM calls this the unstable tree. Section 4.3 discusses the problem of degenerating tree nodes more thoroughly.

The effect gets worse when hinted pages are inserted into the unstable tree and those are not protected by the checksum heuristic. The following benchmark investigates the effect that scanned pages and normal pages have on the tree in all four different approaches.

The benchmark has been conducted warmed up (i.e. the benchmark starts with 100% sharing between VMs) and cold (i.e. the benchmark starts with no shared pages).

The cold state has been used for investigation, while the warmed up state has been used to control that the parameters work for the warmed up state as well.

The results are depicted in Figure 5.12.

In the training graph (figure at the top) it can be seen that KSM RW can find significantly less sharing opportunities than KSM RO. If hints are utilized the degeneration is less severe. The reason is due to the increased merge performance of KSM++. The majority of the I/O pages that have sharing potential get inserted quickly into the stable tree, which is protected against write operations. Mapping only the hinted pages read only improves the number of detected sharing opportunities significantly.

In the control graph (figure at the bottom) there is not much difference between KSM RW and KSM RO. The reason is that KSM does not find new sharing opportunities, but for existing shared pages the sharing is broken. Only at the end can KSM RO make use of the read only unstable tree. For hinted pages the situation is different. KSM++ RW looses sharing potential, but by mapping only the hinted pages read-only the effect vanishes mostly.

A good metric for the stability of a tree is the number of nodes contained in the tree versus the number of nodes that are reachable. The stability has been measured after each scan round before the unstable tree was flushed. The results are shown in Table 5.15.

These results match with the insights gathered from the graphs. Even the unstable tree of Vanilla KSM (RW) degenerates in an I/O centric benchmark. By inserting

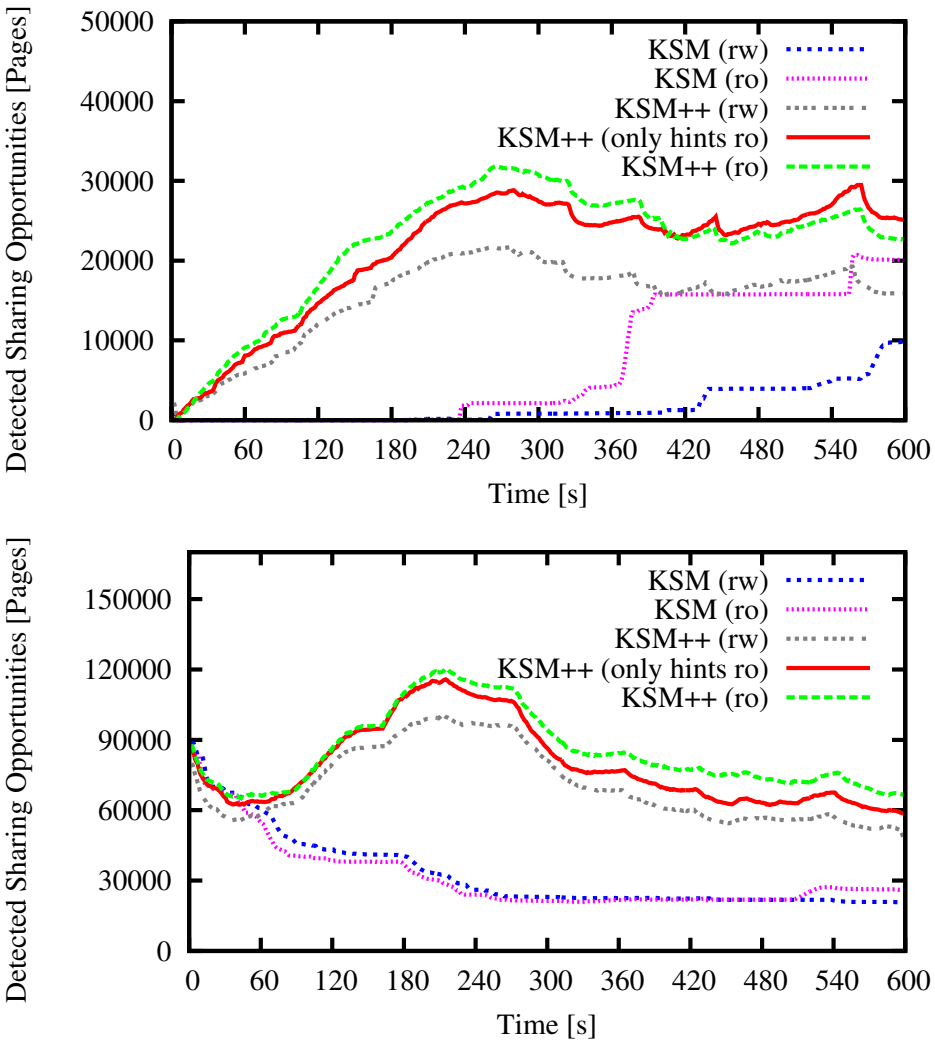


Figure 5.12: Merge performances depend heavily on the stability of the unstable tree and the temporal locality of unstable tree accesses. *Top*: Training starting with no shared pages. *Bottom*: Control with maximum shared pages at start.

hinted pages into the unstable tree, only 10% of the nodes are reachable. The effect is reversed if hinted pages are mapped only. 98.9% of the nodes are now reachable in the tree.

Therefore, in conclusion, in this benchmark the I/O pages are responsible for the degeneration of the tree.

| Vanilla KSM | Hints (all rw) | Hints (hints ro) |
|---|---|---|
| 33.6% - 53.0% | 10.0% | 98.9% |

Table 5.15: Percentage of the unstable tree that is reachable at the end of a scan round. Processing hinted pages without KSM's checksum heuristic degenerates the tree. Mapping the hinted pages read-only but leaving the scanned pages read-write improves the tree's stability significantly. The I/O-pages are responsible for degenerating the tree.

## Configuration Parameters

### Stack Size

In KSM++ a bounded circular stack is used to store the hints received and automatically cap the number of incoming hints. The stack size is configurable via the sysfs interface. The question that remained was what a suitable stack-size could be. This question was evaluated by running the Kernel build benchmark (see Section 5.2) with varying stack sizes and the results are shown in Figure 5.13.

The benchmarks have been conducted both in warmed up and cold states – cold for training, warmed up for control.

In the training graph (top) it can be seen that a stack size of 8 kB finds the most sharing opportunities. Contrary to the intuitive idea, a larger stack size is not necessarily better. The reason is that a flood of hints at one point in time might also include hints that are outdated at the time they are processed. As the stack is overwriting the oldest entries a smaller stack size ensures that hints are always "fresh".

To put a stack size of 8 kB perspective:

By investigating benchmark data, it turned out that the average I/O request spans a size of around 1.5 pages. This means a full stack has on average 12288 pages to process and KSM/KSM++ consumes 100 pages per scan spurt. Therefore using a wake-up time of 100 ms a full stack has been read after around 13 seconds.

If the stack has too less entries, useful hints might be removed. If the stack has too many entries, useless hints might be processed. Time that is not used for processing hinted pages, is used for scanning. And this is what gives a smaller stack its advantage.

This effect can be seen in the control graph (bottom). As the benchmark starts with the maximum detected sharing opportunities, the scan part is less important

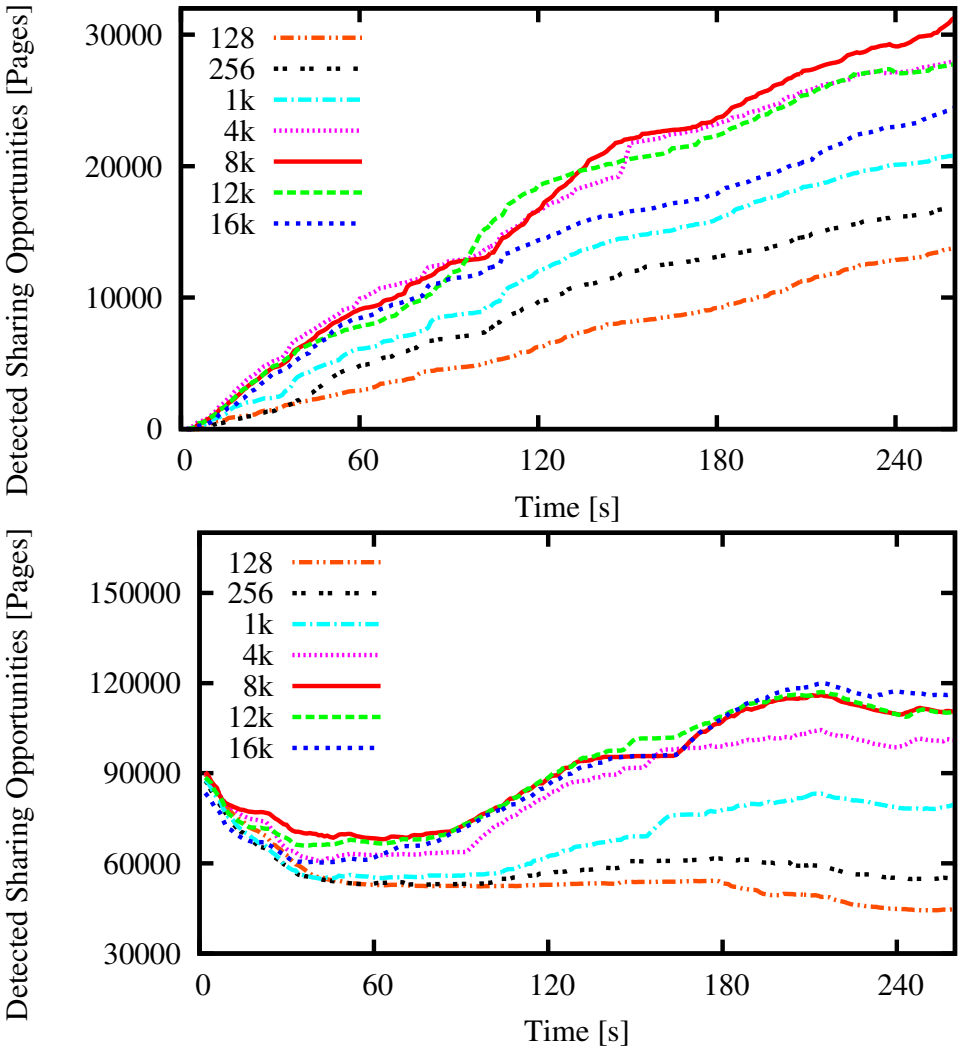


Figure 5.13: Kernel Build deduplication effectiveness with varying stack sizes (KSM++). *Top*: Training starting with no shared pages. *Bottom*: Control with maximum shared pages at start.

to find more sharing opportunities. The 8 kB stack is only the best candidate at the beginning of the benchmark and then finds less sharing opportunities than the 12 kB and 16 kB stacks. For the 16 kB stack the effect is reversed: At the beginning – when the scan part is more important – the 16 kB stack finds less sharing opportunities than the 8 kB stack. When the processing of hints gets more important the 16 kB stack is better.

While a warmed up state is a realistic scenario, the linear scan still finds pages to merge that would otherwise remain undetected. By limiting the stack size the

effect of useless hints is also mitigated. Therefore, based on the training and control graphs a stack size of 8 kB has been chosen for all benchmarks with a wake-up time of 100 ms.

### Interleaving Ratio

KSM++ can be configured to use different interleaving ratios of hint spurts to scan spurts, e.g. a ratio of 6:2 means that KSM processes hints for six spurts, then scans for two spurts, processes hints for six spurts. The interleaving ratio has an effect on how many hints can be processed and how many hints are dropped. The goal of this benchmark is to find a suitable interleaving ratio and investigate how important the linear scan is compared to processing hints. As a benchmark base the Kernel build scenario has been chosen and run with different interleaving ratios.

The benchmark has been conducted both in warmed up and cold states – cold for training, warmed up for control. All results are depicted in Figure 5.14.

In the training graph (top) it can be seen that the configuration that detects the most sharing opportunities has a ratio of 1:1 for hint spurts to scan spurts. Processing more hints slows the scan process down and in consequence KSM++ can find less sharing opportunities.

In the first control graph (middle) it can be seen that a ratio of 1:1 from hinting to scanning does not find most sharing opportunities. Removing the scan completely and just processing hints is the most effective solution in this case. However with ratios of 2:1 and 3:1 the linear scan is still important for finding sharing opportunities. Adding more hint spurts decreases the amount of shared pages. Just with a ratio of 100:1 processes KSM++ hints fast enough to mitigate the dependency on the linear scan. However a ratio of 1:1 finds most sharing opportunities and is an useful choice.

In the second control graph (bottom) it can be seen that using ratios of 1:2 or 1:3 for hint runs to scan runs, KSM++ is still able to detect more sharing opportunities than without using hints at all. A ratio of 1:100 however will not achieve useful results. The hinted pages have almost no impact on the amount of shared pages.

In conclusion a ratio of 1:1 was chosen, because using more hint runs does not achieve results when the sharing opportunities have not yet been found. Setting the ratio in favor of scanning could be useful for workloads, where the I/O generation rate is low (e.g. 0.5 MB/s) or where buffer cache pages have little sharing potential (like in the Mixed benchmark).

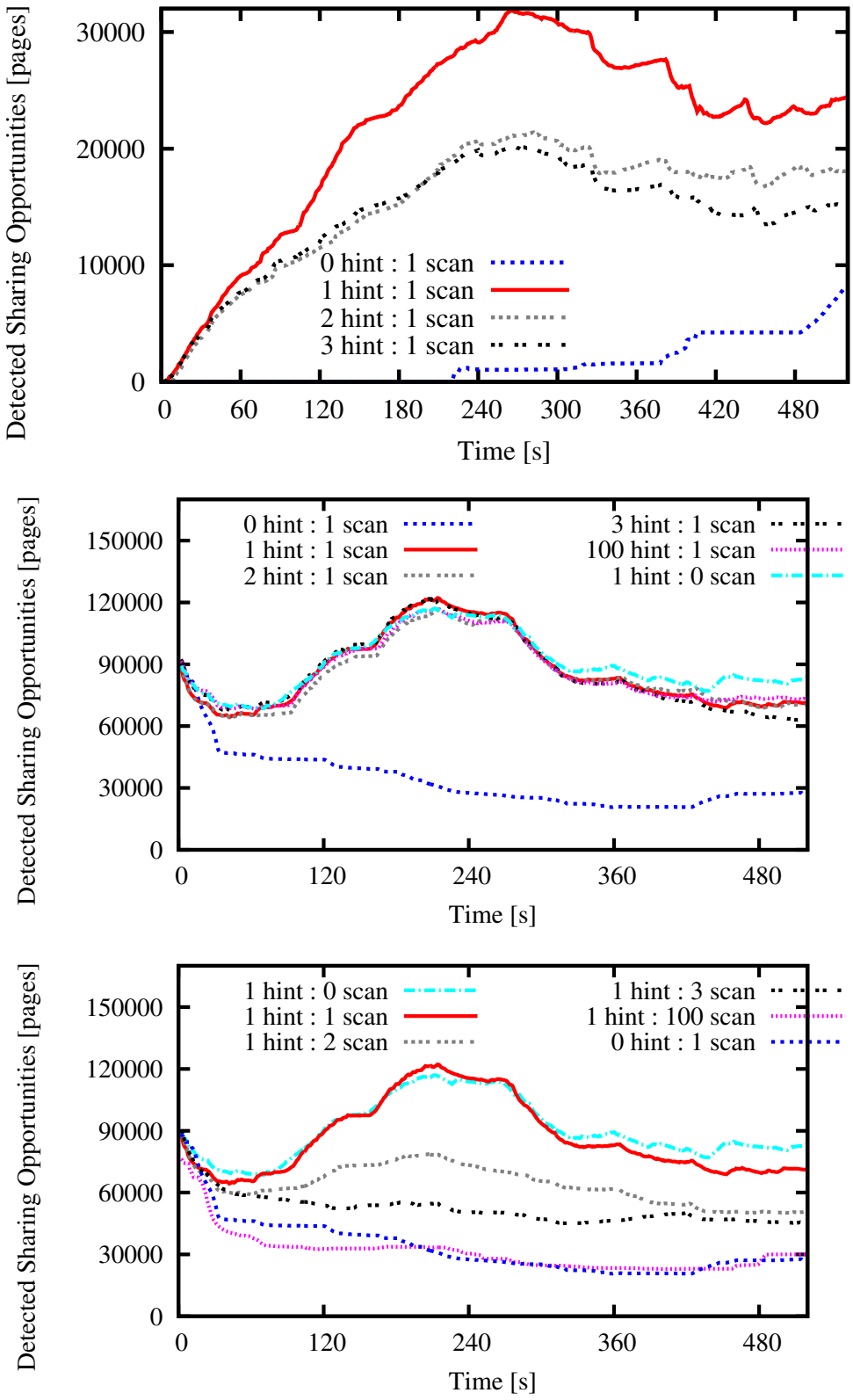


Figure 5.14: Deduplication performance with different hint-spurt vs. scan-spurt ratios. The 0:1 configuration corresponds to KSM.

## 5.5 Summary

This chapter showed that I/O based hints are an effective solution and can make memory-deduplication scanners more efficient.

KSM++, the prototype developed as part of this work, is 2-5 times more efficient than KSM/KSM RO in the Kernel Build benchmark. On average it only needs to visit 5-10 pages per shared page in the system, while KSM RO needs to visit 10-25 pages. In this benchmark I/O requests with the same content are generated at roughly the same time. In the Apache benchmark the sharing opportunities are not generated at the same time. Here KSM++ is 2-4 times more efficient than KSM/KSM RO and needs to visit on average 5-10 pages, while KSM RO needs to visit 10-25 pages. In the Mixed benchmark using a wake-up interval of 20 ms KSM++ is 25-50% less efficient than KSM, while with a wake-up interval of 100 ms, KSM++ is only slightly worse than KSM and on average both need to visit 5-10 pages per shared page. However overall with 100 ms wake-up time KSM++ has a 50% higher merge ratio than KSM.

The intercepting of I/O requests has an impact on the I/O performance, I/O latency and run-time of applications. The impact was measured using the Bonnie++ benchmark suite. KSM++ reaches 96-97% of the I/O throughput that KSM is able to achieve. Using a wake-up time of 20 ms KSM++ has a 4% worse latency than KSM, while with a wake-up time of 200 ms the latency is 60% worse. In terms of run-time KSM++ takes on average 0.2%-1.5% longer to complete the benchmark. To put this number in perspective, for the Kernel build benchmark, which has a run-time of around 11 minutes, this makes a difference between 1 and 9 seconds.

In Conclusion, KSM++ can find both short-lived and long-lived sharing opportunities while consuming around the same amount of CPU than KSM. KSM++ can share more pages than KSM, because it is making smarter choices of sharing candidates.

# Chapter 6

# Conclusion

This chapter gives a summary of the research presented in this thesis and provides insights into future research possibilities.

This thesis investigated the effect of I/O based hints on memory deduplication scanners. KSM++ (the prototype developed as part of this work) enhances the KSM memory scanner by giving strategic hints about potential sharing candidates. Hinted pages are prioritised in the merge process.

## Limitations

This work has uncovered the following limitations of this approach:

KSM++ cannot decide if an I/O hint is useful for finding sharing opportunities. There are at least two scenarios in which hints are not useful: First, the hinted page has no sharing partner at all or no sharing partner in the current working set. Second, the hinted page is short-lived and modified before sharing amortizes the associated overhead.

In the case that the sharing potential is very short-lived, processing the hint can hurt the performance. For a very short-lived sharing opportunity, KSM++ needs to merge the page and map it read-only and then copy it and map it again read-write, which is a time consuming operation. If the hint has no sharing potential, CPU cycles are wasted and the regular scan is slowed down.

KSM++ should also make smarter choices about dropping hints instead of always overwriting the oldest entries. Currently an I/O burst larger than the configured stack_size can remove all useful hints from the stack. This especially affects sce-

narios where the I/O workload is "mixed": Given one VM that has two processes, where process one produces I/O requests at a low rate that are suitable for sharing (for example an Apache webserver) and process two produces I/O bursts at a high rate (for example streaming a movie), then the I/O bursts will remove most of the useful hints by the Apache process.

## Benefits

This work has shown that:

- KSM++ merges the theoretical maximum of sharing opportunities under certain conditions
- KSM++ finds both short-lived and long-lived sharing opportunities by interleaving hinting and scanning mechanisms
- KSM++ needs to scan less pages to find a sharing candidate (without using more CPU consumption). It makes smarter choices to decide which pages are potential sharing candidates.

KSM++ is 2-5 times more efficient than KSM/KSM RO in CPU and I/O bound benchmarks. On average it only needs to visit 5-10 pages per shared page in the system, while KSM RO needs to visit 10-25 pages.

In a mixed workload, in which not many sharing opportunities stem from the page cache, KSM++ is 2-50% less efficient than KSM, even though it is on average able to deduplicate more pages.

Intercepting I/O requests has an impact on I/O performance, latency and run-time of applications. KSM++ reaches 96-97% of the I/O throughput that KSM is able to achieve. KSM++ has a 4%-60% worse latency than KSM and the run-time of benchmarks is on average 0.2%-1.5% longer than with KSM. Overall KSM++ can find both short-lived and long-lived sharing opportunities while consuming around the same amount of CPU than KSM. KSM++ can share more pages than KSM, because it is making smarter choices of sharing candidates.

In conclusion utilizing I/O-based hints is a useful extension for page sharing based on memory scanners provided that the workload includes sharing opportunities that stem from the buffer caches of VMs.

## 6.1 Future Work

Future work could concentrate on setting up criteria of what usable/unusable hints are and use these criteria to make smarter choices of which hints to process. For example I/O bursts could be detected and not provided as hints. This could mitigate the effect of losing sharing potential as soon as KSM++ starts to drop hints. Dynamically adjusting interleaving ratio

In the Mixed benchmark, using a wake-up interval of 20 ms, KSM++ is less efficient than KSM. The reason is that the hints KSM++ processes in this benchmark are not leading to sharing opportunities (only 2% of the hinted pages lead to a successful merge). KSM++ slows the linear scan down and such can find less pages to share than KSM does.

Future work could dynamically adjust the interleaving ratio of processed hints to scanned pages based on the hint quality.

### Inclusion of Deduplicated I/O to improve Hint Quality

Koller et al. increased the performance of I/O operations by utilizing content similarity on disk [14]. There exist a variety of deduplicating file systems and deduplicating block storage systems [7,11,20,22,28]. Paravirtualized approaches [5, 19] have successfully used the information about block deduplication to make smarter choices of sharing candidates.

KSM++ could be further extended to make use of the block deduplication information to improve the hint quality.

### Sharing of the host page cache

The buffer caches of guest and host have high sharing potential due to the cache hierarchy, i.e. the same I/O Requests are cached both in the host and in the guest. Therefore it would be useful to share the host buffer cache with the guest buffer caches. Future work should find solutions to remove the implementation limitation of KSM/KSM++ to share only anonymous pages. This would allow the hosts buffer cache to grow larger without using more memory as long as those pages exist in the buffer caches of the guests.

# Appendix A

# Appendix

## A.1 Efficiency

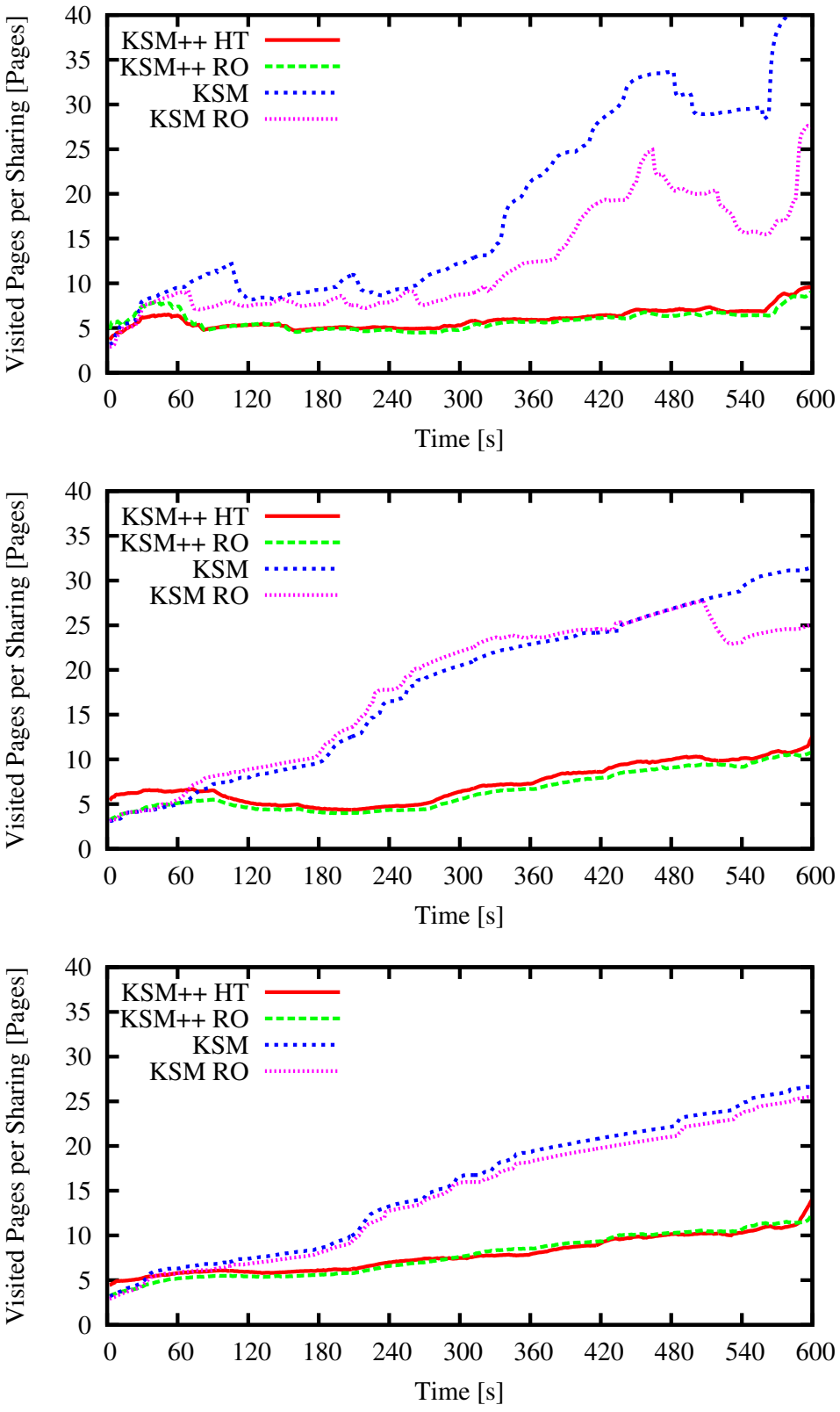


Figure A.1: Efficiency per 20,100,200ms (top to bottom) - Kernel Build

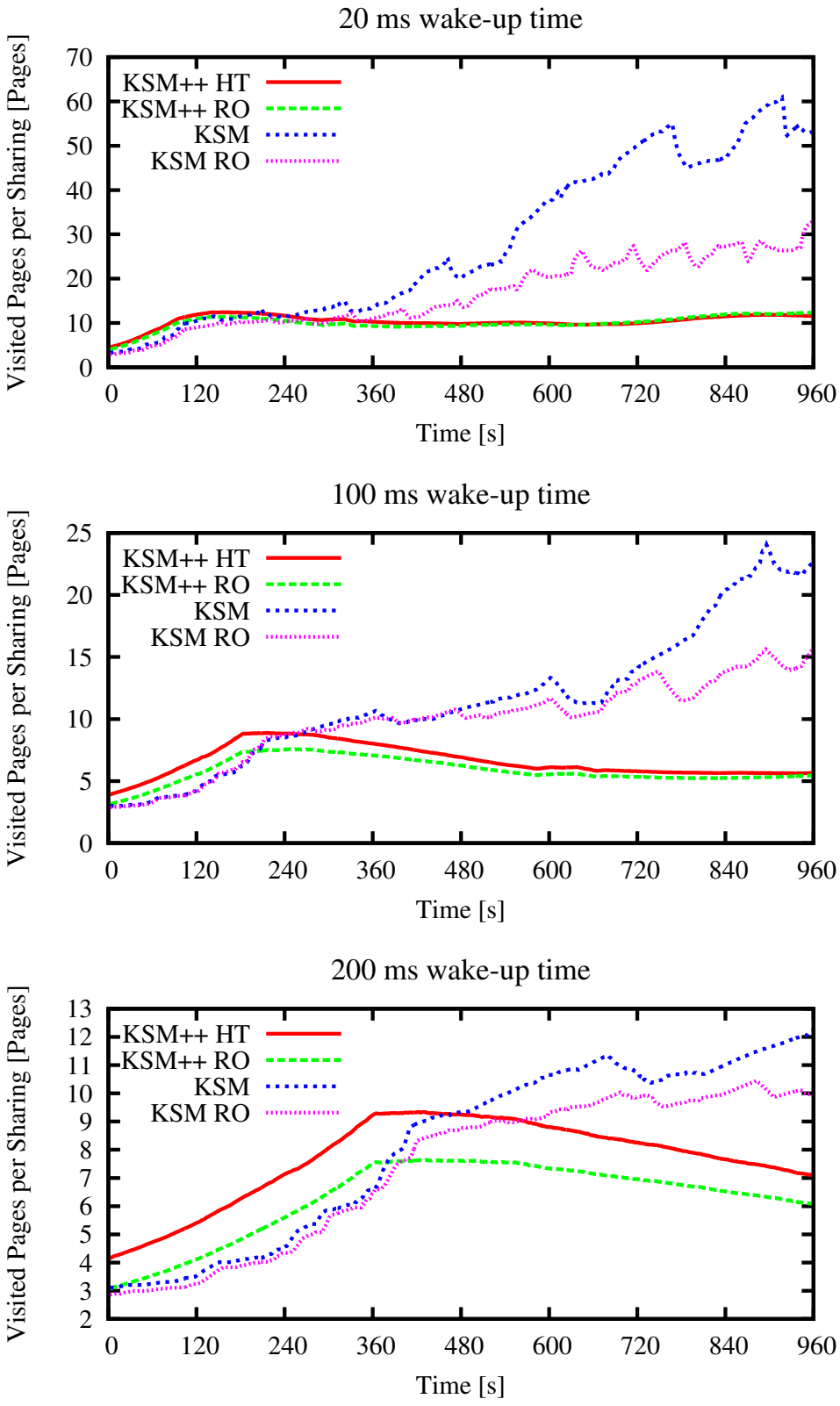


Figure A.2: Efficiency per 20,100,200ms (top to bottom) - Apache

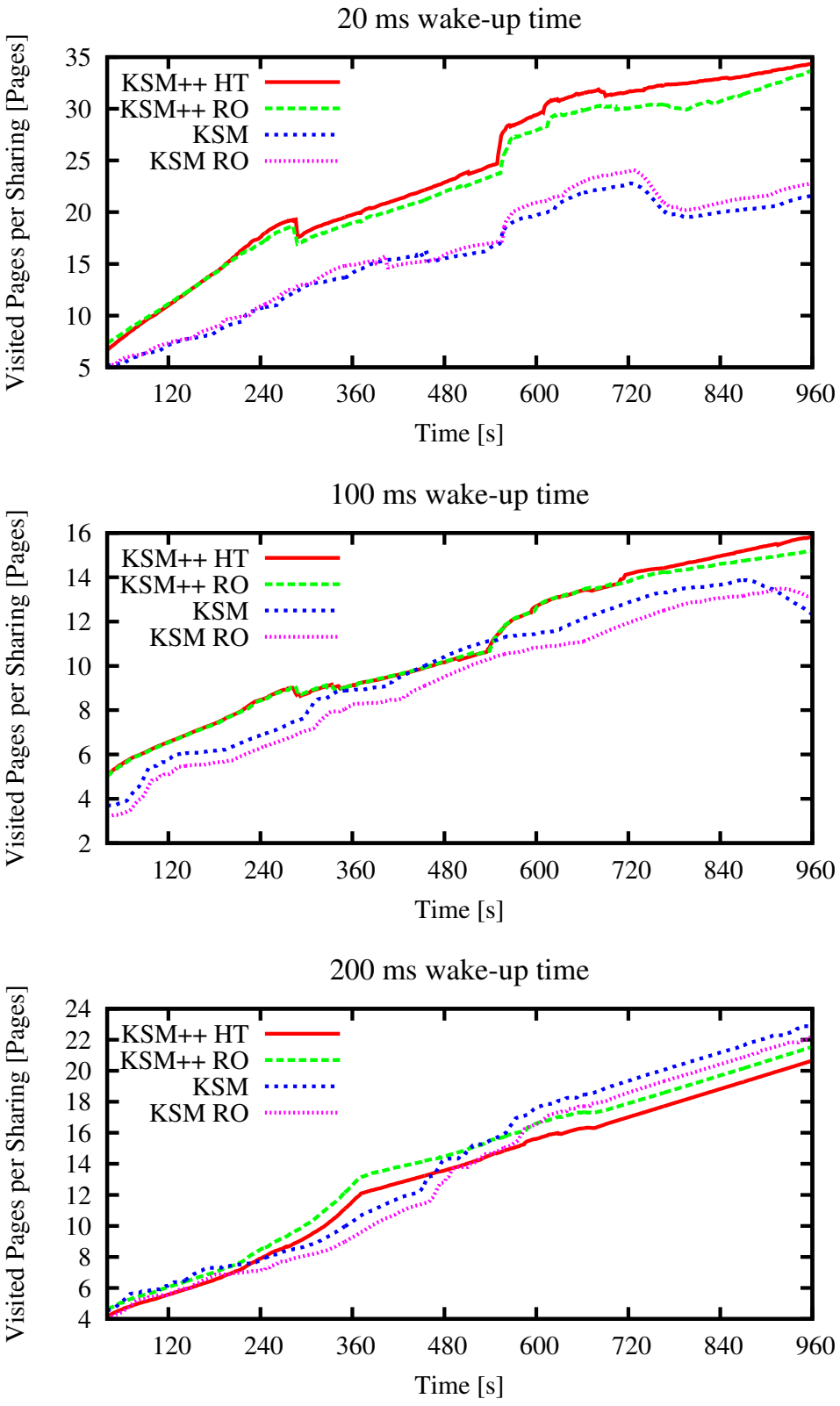


Figure A.3: Efficiency per 20,100,200ms (top to bottom) - MIXED

# Bibliography


[1] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. In *OLS '09: Proceedings of the Linux Symposium*, July 2009.

[2] Sean Barker, Timothy Wood, Prashant Shenoy, and Ramesh Sitaraman. An empirical study of memory sharing in virtual machines. In *Proceedings of the USENIX ATC*, Berkeley, CA, 2012. USENIX Association.

[3] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the USENIX ATC*, ATEC '05, Berkeley, CA, USA, 2005. USENIX Association.

[4] John Berthels. Exmap memory analysis tool. `http://www.berthels.co.uk/exmap/`, 2006.

[5] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM Trans. Comput. Syst.*, 15, November 1997.

[6] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, Washington, DC, USA, 2001. IEEE Computer Society.

[7] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki. Hydrastor: a scalable secondary storage. In *Proccedings of the 7th conference on File and storage technologies*, FAST '09, pages 197–210, Berkeley, CA, USA, 2009. USENIX Association.

[8] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, SOSP '99, New York, NY, USA, 1999. ACM.

[9] Thorsten Groeninger, Konrad Miller, and Frank Bellosa. Analyzing Shared Memory Opportunities in Different Workloads. Study Thesis, 2011. System Architecture Group, KIT, Germany.

[10] K. Fraser J. H. Schopp and M. J. Silbermann. Resizing memory with balloons and hotplug. In *Proceedings of the 2006 Ottawa Linux Symposium*, 2006.

[11] K. Iijima N.A. Quynh K. Suzaki, T. Yagi and Y. Watanabe. Effect of readahead and file system block reallocation for lbcas (loopback content addressable storage). In *Linux Symposium 2009*.

[12] Hwanju Kim, Heeseung Jo, and Joonwon Lee. Xhive: Efficient cooperative caching for virtual machines. *IEEE Trans. Comput.*, 60, January 2011.

[13] J. F. Kloster, J. Kristensen, and A. Mejlholm. Determining the use of Interdomain Shareable Pages using Kernel Introspection. Technical report, Aalborg University, 2007.

[14] Ricardo Koller and Raju Rangaswami. I/o deduplication: Utilizing content similarity to improve i/o performance. *Trans. Storage*, 6, September 2010.

[15] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, and et al. Snowflock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, New York, NY, USA, 2009. ACM.

[16] Dan Magenheimer, Chris Mason, Dave McCracken, and Kurt Hackel. Paravirtualized paging. In *Proceedings of the First conference on I/O virtualization*, WIOV'08, Berkeley, CA, USA, 2008. USENIX Association.

[17] Dan Magenheimer, Chris Mason, Dave McCracken, and Kurt Hackel. Transcendent Memory and Linux. In *OLS '09: Proceedings of the Linux Symposium*, July 2009.

[18] Konrad Miller, Fabian Franz, Thorsten Groeninger, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. Ksm++: Using i/o-based hints to make memory-deduplication scanners more efficient. In *Proceedings of the ASPLOS Workshop on Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE'12)*, London, UK, March 3 2012.

[19] Grzegorz Miłós, Derek G. Murray, Steven Hand, and Michael A. Fetterman. Satori: enlightened page sharing. In *Proceedings of the USENIX ATC*, Berkeley, CA, USA, 2009. USENIX Association.

[20] M. Satyanarayanan B. Karp A. Perrig N. Tolia, M. Kozuch and T. Bressoud. Opportunistic use of content addressable storage for distributed file systems. In *USENIX Annual Tech Conf 2003*.

[21] Patrick Brady. Anatomy & Physiology of an Android. In *Google I/O Developer Conference*, 2008.

[22] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *Proceedings of the Conference on File and Storage Technologies*, FAST '02, pages 89–101, Berkeley, CA, USA, 2002. USENIX Association.

[23] Martin Scwidefsky, Huburtus Franke, Ray Mansell, Himanshu Raj, Damian Osisek, and JonHyuk Choi. Collaborative memory management in hosted linux environments. In *Proceedings of the Linux Symposium, Volume 2*, 2006.

[24] Prateek Sharma and Purushottam Kulkarni. Singleton: system-wide page deduplication in virtual environments. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 15–26, New York, NY, USA, 2012. ACM.

[25] Michael Vrable, Justin Ma, Jay Chen, and et al. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, New York, NY, USA, 2005. ACM.

[26] Carl A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36, December 2002.

[27] Williamson, Mark. Xen Wiki: XenFS. `http://wiki.xensource.com/xenwiki/XenFS`, 2007.

[28] Lawrence L. You, Kristal T. Pollack, and Darrell D. E. Long. Deep store: An archival storage system architecture. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 804–8015, Washington, DC, USA, 2005. IEEE Computer Society.

[29] H. Chen Z. Zhang and H. Lei. Small is big: Functionally partitioned file caching in virtualized environments. In *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 2012), Boston, MA, USA, June 2012*.