Karlsruher Institut für Technologie

Efficient Main Memory Deduplication Through Cross Layer Integration

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)
genehmigte

Dissertation
von

Dipl.-Inform. Konrad Miller

aus Kiel

Tag der mündlichen Prüfung:	17. Juli 2014
Hauptreferent:	Prof. Dr. Frank Bellosa Karlsruher Institut für Technologie
Korreferent:	Prof. Dr. Wolfgang Schröder-Preikschat Friedrich-Alexander Universität Erlangen-Nürnberg

Abstract

An operating system with more random access memory available can use this additional storage capacity to improve the overall system performance. This improvement mainly stems from additional caching and the increased degree of multiprogramming. A prime example is cloud computing. In cloud computing, virtual machines (VMs) permit the flexible allocation and migration of services as well as the consolidation of systems onto fewer physical machines, while preserving strong service isolation. The main memory size limits how many VMs can be co-located on a physical host and also greatly influences their I/O-speed.

Memory sharing was already a hot topic in the 60s, when time sharing systems were first used. In consequence, the copy-on-write mechanism and simple sharing policies were invented to reduce memory duplication from equal source objects (e.g., shared libraries). Today, memory sharing is an important research topic again as previous studies have shown that the memory footprint of VMs often contains a significant amount of pages with equal content while at the same time traditional memory sharing approaches are not applicable in such workloads. The main problem in deduplicating virtual machine memory is a semantic gap caused by the isolation of VMs that makes the identification of duplicate memory pages difficult.

Memory deduplication scanners disregard the sources of and reasons for duplicated memory pages and base the detection of such sharing opportunities purely on their content. Memory scanners continuously create and update an index of memory contents at a certain rate. If memory is to be inserted into the index that already contains the respective page-data, both memory pages are merged and shared using copy-on-write. One of the two memory pages can then be released and reused fully transparently to the affected processes (VMs).

Memory scanners directly trade computational overhead and memory bandwidth with deduplication success and latency. Especially the merge latency, the time between establishing certain content in a page and merging it with a duplicate, is high in such systems. Current scanners, for example VMware's ESX, need a long time (in the range of 5–30 min) to detect new sharing opportunities and therefore can only find static sharing opportunities instead of exploiting the full sharing potential.

This thesis makes the following, novel contributions:

1 Analysis of Duplication Characteristics

Prior work has focused on the mechanisms of memory deduplication. In consequence, the analyses that have been done in this area have targeted the amount of memory that can be saved using the respective approaches.

This thesis can confirm the results that were previously published. However, it goes further and also analyzes the reasons for duplication and takes a more detailed look into the temporal and spatial characteristics of duplication in different workloads.

In virtualized environments, we have observed that all types of memory contents can contribute to memory redundancy, however that many sharable pages in the host originate in accesses to background storage. Those pages are in addition not likely to change their contents in the near future due to the way I/O-caching algorithms function.

2 Cross Layer Integration for Memory Scanners

Following our analyses we have developed XLH [61, 62]. Our approach extends main memory deduplication scanners through Cross Layer I/O-based Hints (XLH). XLH focuses the deduplication effort on memory parts with a higher prospect to yielding good sharing candidates. In the case of virtualized environments those are recently modified memory areas that belong to the I/O-caches of virtual machine guests.

3 Deduplication and Sharing in Virtual Environments

We were able to show that XLH helps finding and exploiting sharing opportunities earlier than regular memory scanners. XLH's early detection of sharing opportunities saves more memory by deduplicating otherwise missed short-lived pages and by increasing the time long-lived duplicates remain shared.

In our benchmarks, XLH can merge duplicates that stem from virtual disk images earlier than the KSM memory scanner by minutes. XLH can save up to eight times as much memory at the same scan-rate settings [62].

4 Performance Considerations

A thorough analysis of overheads, performance benefits and performance penalties is an important part of this thesis. Deduplication has two sides to it: The overhead caused by the deduplication process itself on the one hand. The overhead that applications experience, for example due to additional copy-on-write page-faults, on the other hand.

Contents

Chapter 1

Introduction

An operating system (OS) with more random access memory (RAM) available can use this additional storage capacity to improve the overall system performance through caching and to increase the degree of multi-programming:

Applications are slowed down when they wait for I/O, whether they are I/O-bound or not. Thus, applications directly benefit from file- and meta-data caching in main memory. First, a RAM based cache is an order of magnitude faster than background storage. Second, this cache can also be used to buffer writes. The OS can then return to the application directly after the write operation has been buffered and before it has been persisted to the background store. Current consumer grade random access memories[1] outperform even the fastest, available enterprise class SSDs[2] by a factor of almost 300x regarding random access throughput, by a factor of 35x regarding linear access throughput, and by a factor of 20x in access latency. The latency gap is much higher when writing. The throughput of cached disk accesses using those devices[3] consequently shows a speed-up of more than 250x compared to uncached random access and a speed-up of more than 26x compared to uncached linear access.

Moreover, the amount of available RAM is a determining factor for how many applications can be run in parallel and thus how well all system components can be utilized. Application performance decreases drastically [25] when the system cannot keep the current working set [26] in memory. Today, the amount of available main memory limits the number of virtual machines (VMs) a physical machine can host [39]. Hypervisors reach the upper bound for the number of hosted VMs when the sum of all working sets approaches the amount of available memory. When this point is passed, the hypervisor begins swapping memory from and to the background store at vast performance degradation for all VMs [35, 84].

[1] e.g., DDR3-10667U: throughput 10.67 GB/s, latency: 10 ns
[2] e.g., Samsung 840 Pro: 4kB random read throughput 0.036 GB/s, latency 200 ns [34]
[3] measured with hdparm

Memory Sharing

Memory is often in short supply [39]. In common virtualization environments, such as cloud computing data centers, the leading companies employ consumer-grade PC hardware. This type of hardware currently supports up to 32 GiB of DDR3 RAM per CPU [19], because of limitations of the chipset and mainboard design. Hardware that supports more memory is currently much more expensive.

Virtual memory management techniques such as segmentation [75] or paging [31] give applications an own address space to store data. Every application can only access memory in their own address space, so using segmentation or paging gives an application the illusion of exclusively using a computer's main memory. When using virtual memory, multiple virtual addresses can be mapped to the same physical address with the effect that different programs share physical memory.

There are three different possibilities to share memory between applications in operating systems employing virtual memory management. First, memory can be shared *read-write*. In this case, both applications see each other's writes to the memory region. Second, memory regions can be shared *read-only*. This, for example, enables OSes to robustly share libraries among processes while keeping the address spaces isolated. Writes to such memory regions are forbidden. Third, memory can be shared *copy-on-write* (COW). COW makes it possible to share memory between different applications in a way that allows modification. The involved applications never see mutual modification, however. Using COW main memory is shared between address spaces while maintaining the illusion of exclusive access when writing. We focus on analyzing and increasing copy-on-write sharing in this thesis.

Benefits from Sharing

Sharing main memory using virtual memory management has benefits on different levels in the hard- and software stack:

- Sharing frees main memory that would otherwise be occupied by duplicates. The gained free space can then be used to provide larger caches for lower levels in the storage hierarchy, or to run more applications.
- Sharing main memory frees space in preceding physically indexed levels in the storage hierarchy. Shared memory regions are transported only once to CPU caches resulting in more space for caching other address space areas in those faster, smaller memories.
- Sharing can achieve copy semantics without actually copying data in memory, thus effectively speeding up copy operations.

1.1. Sharing and Deduplication Techniques

When running multiple applications [46] and specifically in *virtual environments* in which a physical machine is executing multiple virtual machines (VMs), a considerable amount of main memory can be shared and thus freed through employing sharing- and deduplication techniques [5, 13, 39, 61–63, 84]. Operating systems have been using techniques to initially share address spaces among applications and shared background storage caches for decades. Recently, those techniques have been complemented with approaches to share memory regions even after modification.

Sharing Address Spaces

A new process is created in UNIX based OSes by `fork`ing a thread of execution off of an existing process. When `fork`ing, the OS creates an address space (AS) for the new process and (semantically) copies the `fork`ing processes AS content to the new AS. Actually, the OS shares the parent address space with its child using copy-on-write instead of copying memory in RAM.

Note that creating a new process is not the same as starting a program. In order to start a new program, an application first forks itself and then overwrites the new processes AS, with the program to be started, using `exec`. At this point shared memory between the two processes is lost. Starting a program twice, in consequence, results in two processes that do not share any part of their address space except of the data shared through the file system cache.

Sharing the File System Cache

The contents of previously loaded files are typically cached in an operating system's file cache (e.g., the page-cache in Linux).

Files can be mapped into main memory to be easily and efficiently accessed non-linearly, for example via the `mmap` system call in UNIX. Multiple such mappings to the same file are shared within the file cache. Program binary images and dynamically shared libraries are generally implemented through this mechanism [30].

Using this mechanism has the limitation that it solely shares memory regions that come from the same source object; a memory region from the same inode. When one copies a file, both files will not be shared in memory as the "copy" semantic is not known to the OS.

Sharing Virtual Machine Memory

When introducing virtual machines (VMs), the hypervisor and VM encompass the same kind of semantic gap regarding "copies" that the file system cache has.

VMs generally ship with their own virtual disk image (VDI) that contains copies of shared libraries, programs and configuration files or even data that are also used in other VMs. In Figure 1.1, `libA` is used in both VMs, but the hypervisor potentially cannot even interpret/read the file systems used by the VMs. Even if it could read the file systems it could not infer the equity of files solely based on the path within the file system image.

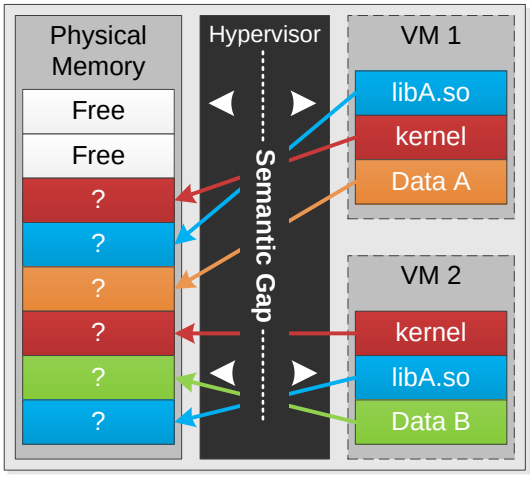

Figure 1.1.: All semantic knowledge known to the guest OS is lost when using virtualization.

The hypervisor does not know the "same object" semantic between VMs as the objects' semantic is hidden behind the virtualization layer. Thus, traditional sharing policies cannot be used for deduplicating VMs as those mechanisms are based on the source of objects instead of being based on the contents of those objects in memory. OS researchers have derived the following two solutions.

Paravirtualization-based Deduplication

The approaches that fall into this category aim to circumvent the semantic gap by introspecting operations within the VMs and communicating semantic information though an interface between the VMs and the hypervisor (Figure 1.2).

In some workloads, many duplicates come from copying data within memory or are different copies of the same file. Disco [12] and Satori [63] operate under the assumption that one can observe the creation of duplicates and then instantly deduplicate them. Both implement a form of content addressable disk and are

capable of deduplicating memory contents that come from this disk at the costs of I/O-overhead and the burden of changing the guest OS. Disco also hooked calls such as `bcopy` to cope with direct copies being created in memory.

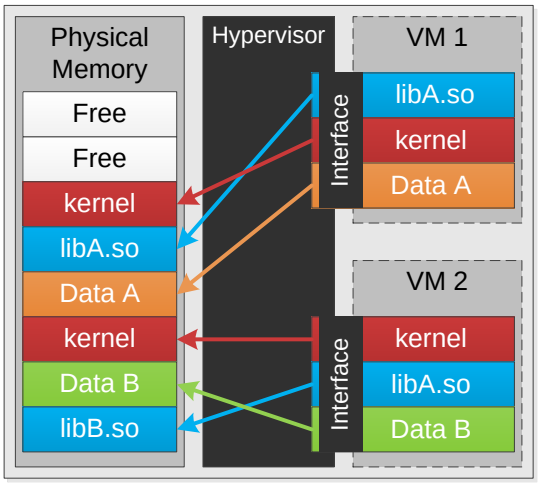

Figure 1.2.: Paravirtualized systems interface with the guest virtual machines to close (parts of) the semantic gap.

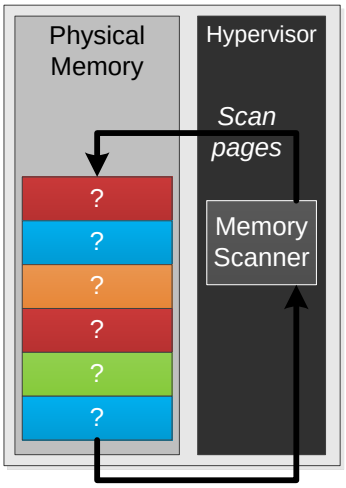

Figure 1.3.: Memory scanners create an index of main memory contents without regard to their semantic origin.

Main Memory Scanners

To get around the semantic gap without modifying the guest OS, main memory deduplication scanners were introduced in VMware's ESX [84] and later also adopted in the Linux kernel under the name Kernel Samepage Merging (KSM) [2].

Memory deduplication scanners disregard the sources of and reasons for duplicated memory pages and base the detection of such *sharing opportunities* purely on their contents (Figure 1.3). Memory scanners continuously create and update an index of memory contents at a certain rate. If memory is to be inserted into the index that already contains the respective page-data, both memory pages are merged and shared using copy-on-write. One of the two memory pages can then be released and reused fully transparently to the affected processes (VMs).

Memory scanners directly trade computational overhead and memory bandwidth with deduplication success and latency. Especially the merge latency, the time between establishing certain content in a page and merging it with a duplicate, is high in such systems. Current scanners need a long time (in the range of 5–30 min) to detect new sharing opportunities and therefore can only find static sharing opportunities instead of exploiting the full sharing potential.

1.2. Contributions

This thesis makes the following main contributions:

Analysis of Duplication Prior work has focused on the mechanisms of memory deduplication. In consequence the analyses that have been done in this area have targeted the amount of memory that can be saved using the respective approaches.

This thesis can confirm the results that were previously published. However, it goes further and also analyzes the reasons for duplication and takes a more detailed look into the temporal and spatial characteristics of duplication in different workloads.

In virtualized environments, we have observed that all types of memory contents can contribute to memory redundancy, however that many sharable pages in the host originate from accesses to background storage. Those pages are in addition not likely to change their contents in the near future due to the way disk-caching algorithms function.

Cross-Layer Integration for Memory Scanners Following our analyses we have developed XLH [61, 62]. Our approach extends main memory deduplication scanners through Cross Layer I/O-based Hints (XLH). XLH focuses the deduplication effort on memory parts with a higher prospect to yielding good sharing candidates. In the case of virtualized environments those are recently modified memory areas that belong to the I/O-caches of virtual machine guests.

Deduplication and Sharing in Virtual Environments We were able to show that XLH can help to find and exploit sharing opportunities earlier than regular memory scanners. XLH's early detection of sharing opportunities can save more memory by deduplicating otherwise missed short-lived pages and by increasing the time long-lived duplicates remain shared.

In our benchmarks, XLH can merge duplicates that stem from virtual disk images earlier than the KSM memory scanner by minutes. In total, XLH can save up to eight times as much memory at the same scan-rate settings [62].

Performance Considerations when using Deduplication A thorough analysis of overheads, performance benefits and performance penalties is an important part of this thesis. Deduplication has two sides to it. The overhead caused by the deduplication process itself on the one hand. The overhead that applications experience, for example due to additional copy-on-write page-faults, on the other hand.

1.3. Underlying Publications and Theses

This thesis would not exist without the lively cooperation with my colleagues and students and is in consequence based on many previously published documents. The presented results were evaluated more extensively and thoroughly than previously possible. Moreover, I have interpreted these results in the new, extended context, thus the interpretation and the inferred conclusions may differ from previous publications.

I advised the following students when they wrote their study- and diploma theses at IBDS. They have contributed to this thesis, in chronological order:

- **Thorsten Gröninger** has written a kernel module to extract content based hashes from all memory pages of a specified process during the course of his study thesis [37]. He has used the module to conduct some initial experiments about memory duplication quantities. We have later used this kernel module to generate the "sharing opportunities" lines throughout the benchmarks of the published papers and this thesis.

- **Marc Rittinghaus** has implemented memory tracing in the Simics full system simulator for his diploma thesis [69]. Parts of the data he has gathered to evaluate his approach have been incorporated in Chapter 3.

- **Fabian Franz** has analyzed memory deduplication hints preliminary in his diploma thesis [32]. The first implementation of KSM++ that later evolved to XLH was joined work of Fabian Franz and myself. His work is visible in Chapter 4 and Chapter 5.

- **Thorsten Gröninger** improved the tracing speed of Marc Rittinghaus' analysis platform by exchanging Simics with the faster QEMU as a part of his diploma thesis [38]. QEMU is much faster due to binary translation. QEMU is however not meant to be used for simulation which introduced a whole set of new challenges such as a missing precise cycle counter. The resulting implementation allowed for detailed analyses of much longer-running workloads than previously possible. Thorsten Gröninger's results have flown into Chapter 3.

- **Marco Kroll** has analyzed the memory scanning overhead as well as on the effects that the memory deduplication process has on other co-scheduled applications. Some insights gained during the course of his diploma thesis [49] are presented in Chapter 6.

The following papers have been published before the submission of this thesis:

- Preliminary results of the application of memory deduplication hints for making memory deduplication more efficient have been published at the ASPLOS workshop RESoLVE 2012 [61].

- The publication was later extended and published as a full paper at USENIX ATC 2013 [62]. Fabian was of great help. He did not only participate in the implementation of KSM++ and XLH, but also supervised the benchmarking process for the evaluation of both papers. He is consequently the co-author of both publications. My colleagues Marc Rittinghaus, Marius Hillenbrand and my advisor Frank Bellosa have helped revise the papers and were available for discussion throughout my work. They are also co-authors of the publications.

1.4. Organization

The remainder of this thesis is organized as follows:

Chapter 2 – Background and Literature Review introduces terms and principles that the thesis is based on. It also introduces an overview of related work in the fields of virtual memory management, memory sharing for applications, and memory deduplication for virtual machines. Previous work specifically related to single topics is presented in the beginning of the respective chapters.

Chapter 3 – Analysis of Main Memory Duplication and Sharing first discusses different possibilities to measure memory duplication. It then motivates the use of memory saving techniques by quantifying the amount of savable memory as well as the sources and characteristics of sharable pages obtained through various formerly discussed techniques.

Chapter 4 – Cross-Layer Integration through Deduplication Hints starts with a brief introduction to x86 paging and the Linux virtual memory system. It then thoroughly describes the design and implementation of cross-layer hints (XLH), the main contribution of this thesis.

Chapter 5 – Deduplicating Virtualized Environments first introduces related work and background information related to virtual machine memory allocation and memory deduplication. It then analyzes the benefits and drawbacks that XLH has on virtualized environments by first introducing metrics to compare different deduplication strategies and by then defining benchmarks and discussing their results.

Chapter 6 – Performance Considerations analyzes the work that memory scanners put into the deduplication process. It moreover analyzes runtime properties of memory deduplication.

Chapter 7 – Conclusion first recapitulates the main points of the thesis before summarizing the main contributions as well as the limitations of the presented work. This chapter closes with an outlook on possible future research directions.

Chapter 2

Background and Literature Review

This chapter describes terms (Section 2.1) and operating system (OS) principles that this thesis is based upon. It moreover gives a general overview of related work that tangents the thesis. Related work which is specific to single chapters is discussed within those respective chapters.

Paged virtual memory systems can safely map the same physical pages into multiple virtual address spaces while keeping modifications private to each address space (Section 2.2). The OS can reduce the memory footprint of processes if it knows in advance which memory regions contain equal data contents (Section 2.3).

Recently, workloads have become popular that make it hard for the underlying OS to decide in advance which pages it can share. A prime example is a hypervisor that executes multiple virtual machines (VMs). This led to the development of mechanisms specifically tailored to deduplicate memory within and across VMs. Solutions for deduplicating virtual machine memory, as well as the limitations thereof, are introduced in Section 2.4. The chapter concludes by summarizing the limitations of the current state of the art in memory deduplication (Section 2.5).

2.1. Terms

Throughout this work, instead of defining pages to be **equal** if their source objects are the same, I use a weaker definition of equity for memory pages: I call pages equal if their data contents match. Memory pages that are exclusively filled with 0-bytes are denoted by **zero-pages**.

Equal pages mapping to different page frames can be merged to a single page frame which is then referenced by both pages sharing the frame. I refer to pages that are modified to become equal to at least one other page in the system as **sharing opportunity**. The total amount of sharing opportunities is equivalent to the total amount of memory that can be freed and thus reused.

After the merging has taken place, I call the remaining page frame to be **sharing** while I refer to all freed pages referencing this page frame as **shared** pages. The sum of shared pages represents the total amount of memory that is currently **saved** by deduplication on page granularity.

The term **host** denotes the layer beneath a guest virtual machine if virtualization is used or the OS that provides the runtime environment of native applications. The host can thus represent the virtual machine monitor, hypervisor or a regular, unlayered operating system depending on the context.

2.2. Virtual Memory Systems

Up to the early 1960's programs were loaded into and then run from physical memory directly. If a program was larger than the available main memory, the programmer partitioned his program manually into so called overlays or segments which were then swapped as a whole within the memory hierarchy [28]. Today, this kind of memory partitioning remains to be useful in very small embedded processors [53].

2.2.1. Indirect Addressing

When paging [31] and segmentation [75] based virtual memory systems made their way into computing systems in 1961, programmers were freed from the burden of manually partitioning their programs as this was now done transparently by the virtual memory system.

Virtual memory systems introduce an indirection layer for memory addresses. Programs can now use their own, contiguous address space by using indirect, virtual addresses to access memory from their private address space. Physical memory addresses are never used in programs directly. In consequence, the OS does not need to map entire virtual address spaces to physical memory. Memory can be loaded and mapped on demand, when it is accessed.

A co-processor, called memory management unit (MMU) translates virtual addresses into physical addresses at every load and store (Figure 2.1). The OS manages a translation database that stores how virtual addresses map to physical addresses. The MMU uses this database to autonomously translate addresses that have previously been mapped. If addresses that have not been mapped by the OS before are accessed, the OS is called through an exception to establish a valid mapping before restarting the access unless an error has occurred.

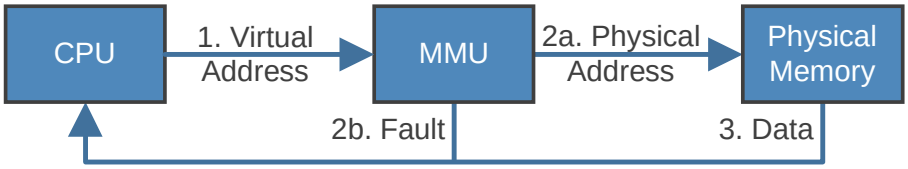

Figure 2.1.: Interplay of components when using indirect addressing. Processes load and store virtual addresses. The memory management unit translates those virtual addresses to physical addresses.

Virtual memory systems have the following key benefits over addressing physical memory directly [60]:

- **Protection:** Each process has its own address space. No process can observe or corrupt memory of another process.
- **Explicit sharing:** Virtual addresses make safe and explicit sharing of objects in physical memory possible by mapping different virtual addresses to the same location in physical memory.
- **Transparency:** Memory contents can be loaded into arbitrary locations in physical memory while being accessible at predefined locations within the process' virtual address spaces. Different processes can use the same addresses without interfering with each other.
- **Overcommitment:** The OS can allocate more virtual memory to processes than physically available. Overcommitted memory can be swapped from background storage transparently to the application.

When using virtual memory systems, there are always two viewpoints: the operating system's and the hardware's view. Those sides correspond to the policies and the semantic organization of the virtual address space (§2.2.2) and the mechanics of mapping virtual to physical addresses (§2.2.3) respectively.

2.2.2. Virtual Address Space

The operating system cares about the allocation of address space sections to programs, about their protection settings, and about where memory comes from and goes to. In short, the OS knows about the semantics of larger sections of memory.

When starting a program, the OS creates a new (virtual) address space for the resulting process. The process's binary file specifies the different address space sections the process requires and how these sections need to be initialized for the program to be executed properly.

Typically a UNIX/Linux process contains the following sections [59] (Figure 2.2):

- **Stack:** Local variables, function call parameters, return values, return addresses, memory allocated through `alloca`.
- **Heap:** Dynamically allocated memory for applications, that is memory allocated through `malloc`.
- **.bss: B**lock **S**tarted by **S**ymbol. Uninitialized static variables. Contains zeros when first accessed.
- **Data:** Initialized data from binary file, e.g., global variables.
- **Read-Only Data:** Initialized data from binary file that will never be written, e.g., constants such as strings.
- **Text:** Machine code to be executed by process.

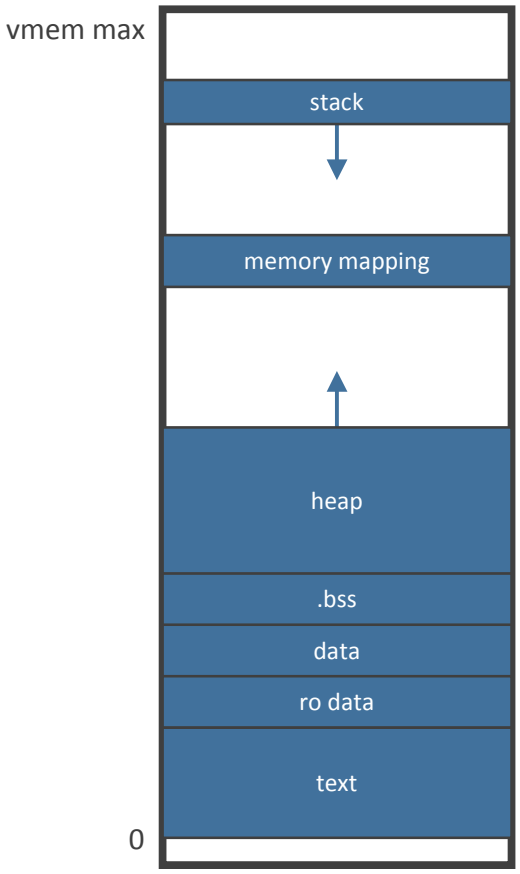

Figure 2.2.: A typical Linux address space. [59]

The OS allocates memory for those sections and stores their semantics, for example what files the memory contents come from or what protection bits they should have in memory, and the size of the section. The OS also sets up a translation table that the MMU can use to translate virtual to physical addresses.

Page Replacement Eventually, all page frames will be allocated to the kernel, the page cache, and processes. Then, unless processes voluntarily give back memory (`munmap`, `sbrk`) or exit, the OS needs to replace existing memory mappings when allocating new memory. To this end, the OS is responsible for the following operations:

- **Swapping out:** When a process allocates virtual memory, for example by requesting more heap memory (`sbrk`), or when a new process is started, the OS allocates physical memory to the process. As superfluous memory is generally used for caching, the OS first needs to free physical memory in order to reallocate those page frames. It can do this by flushing cached pages or by swapping not recently used pages to disk; the OS decides which pages to evict and how to properly evict them.
- **Swapping in:** When an address that is currently not in memory is accessed, for example due to memory over-commitment, the MMU signals this condition to the OS through a CPU exception (i.e., a segmentation fault or a page-fault). The OS can then suspend the process that executed the faulting instruction, bring in the wanted memory and restart the process right before the faulting memory load or store instruction.
- **Resolving illegal access:** If a process accesses an illegal address, which is an address that is not mapped into the process's address space at all, the OS receives a page-fault from the MMU due to the missing mapping. It can then notice from its recorded virtual memory sections that the accessed address is not mapped and forcibly quit the process.
- **Handling memory mapped files:** Only the OS knows how to use the file system. It is therefore the operating system's responsibility to bring in and flush out (dirty) memory mapped file pages. The same is true for memory mapped devices.

Working Set Every program has a memory sweet-spot: it requires the memory pages that it needs for the ongoing computation — the *working set* — to reside in main memory. Operating systems generally take the working set of applications into account when making page replacement decisions.

Denning defined a *working set model* that says that those pages that have been recently used are a good predictor for the pages that will be used in the near future [26]. This model is still in use today.

Thrashing If there is not enough memory available for keeping all active working sets in main memory, the system falls into a "condition of near-total performance collapse" [27]. This condition is commonly referred to as *thrashing*. Thrashing is caused by the steep performance drop in access latencies and throughput from main memory to background storage.

2.2.3. Physical Address Space: Paging

The hardware cares about how to translate a virtual address to a physical address. The MMU, today generally integrated into the CPU, can autonomously translate virtual addresses to physical addresses if the mapping is available in main memory. For everything else it signals the OS that then sets up the mappings for the MMU.

Segmentation and paging based virtual memory systems were published within the same year. Some memory management units (e.g., x86) support both techniques, segmentation and paging [21]. In x86, a logical (virtual) memory address is first translated to a linear address through segmentation and then further to a physical address through paging. However, today's modern operating systems predominantly use paging for virtual memory management. Linux [54] and L4Ka [23] for example use the *flat memory model*, meaning that they fix the kernel and user code and data segmentation registers to span the entire address space starting with zero.Those OSes then use paging for virtual memory management, only.

When using paging, the virtual address space of applications is divided into fixed size chunks called **pages**. Typical *page-sizes* are 4 KiB and 2 MiB, but the Intel architecture supports 1 GiB pages as well [21]. The Alpha 21064 for example gives the OS developer the possibility to use 8 KiB, 16 KiB, 32 KiB, and 64 KiB page sizes [45]. Pages are always a power-of-two bytes in size. Thus, larger pages are a multiple of the smaller page-sizes. Physical memory is divided into **page frames** of the smallest page-size and virtual pages are mapped to those **page frames** aligned to their page-size (Figure 2.3).

Each virtual address space that the operating system manages is associated with a **page table** that stores how pages currently *map* to page frames, that is where in physical memory the content of a page currently resides. The OS maintains this page table while the MMU consults the page table contents when translating addresses.

The customary *long mode* of the x86-64 processor architecture uses a 4-level page table (Figure 2.4) to translate virtual to physical addresses [21]. For every address space, it consists of a hierarchy of page tables, beginning with the *page map level 4* (PML4), which the MMU finds through an x86 CPU register (`%CR3`) dedicated to hold the PML4 of the currently active address space[1]. The PML4 points to multiple *page directory pointers tables* (PDPTs). Those, in turn, point to *page directories* (PDs), which finally point to *page table entries* (PTEs). At each level, the respective table can either point to a directory in the next hierarchy level, or to a PTE. PTEs in the first hierarchy level are called page directory pointers table entry (PDPTE), in the second hierarchy level they are called page directory entry (PDE).

[1] The OS sets the `%CR3` register to switch to another address space.

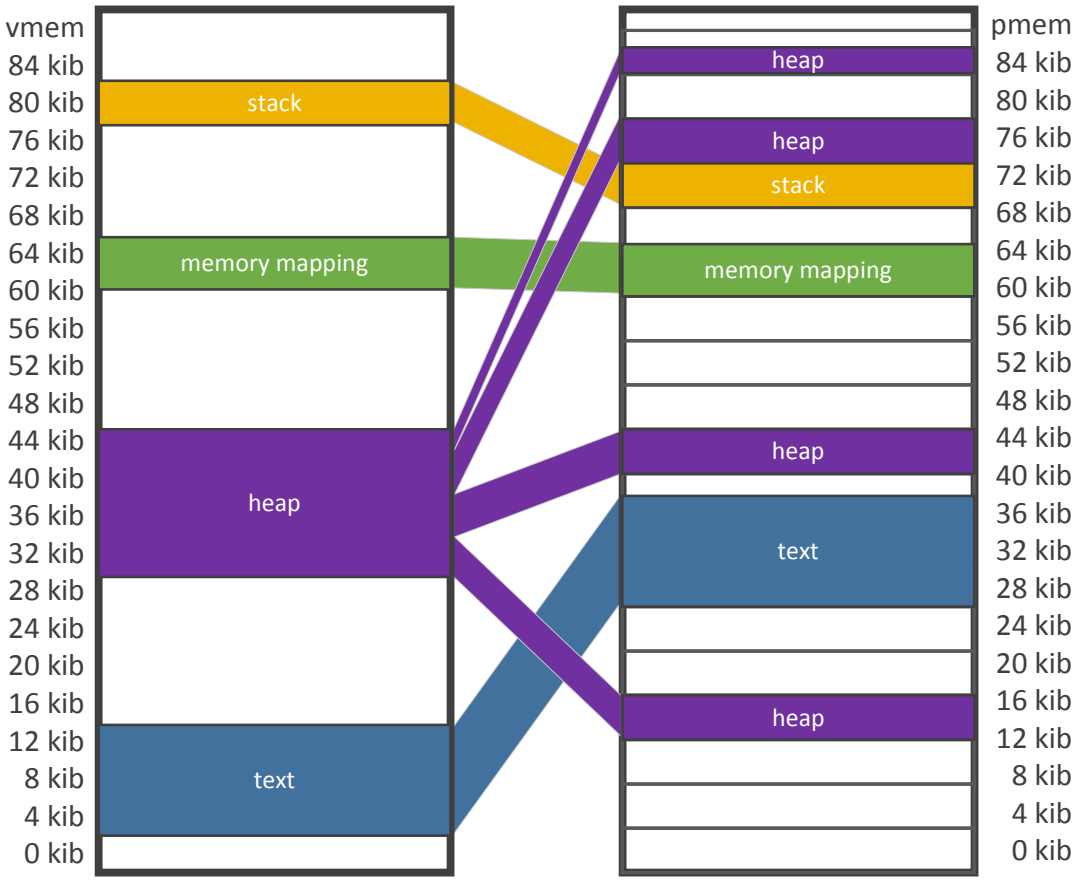

Figure 2.3.: Virtual memory areas are cut into virtual pages and then mapped to physical page frames individually using a paging MMU. Adjacent pages can but don't have to be mapped to adjacent page frames.

The hierarchy depth to a PTE determines the size of the referenced page. For example, a 4 KiB page uses the full depth of 4 levels, while 2 MiB pages are referenced by PDEs and 1 GiB pages are referenced by PDPTEs.

Each page table entry contains, among others, information about:

- **Present Bit:** Whether the page is currently available in memory or needs to be brought in by the OS, via a page-fault, before accessing it.
- **Page Frame Number:** If the page is present, at which physical address the page it is currently located.
- **Write Bit:** If the page may be written to. When a process writes to a page with a clear write bit, the MMU halts the operation and raises a page-fault.
- **Caching:** If this page should be cached at all and with which policy.
- **Accessed Bit:** Set if this page was touched since the bit was last cleared.
- **Dirty Bit:** Set if this page was modified since the bit was last cleared.

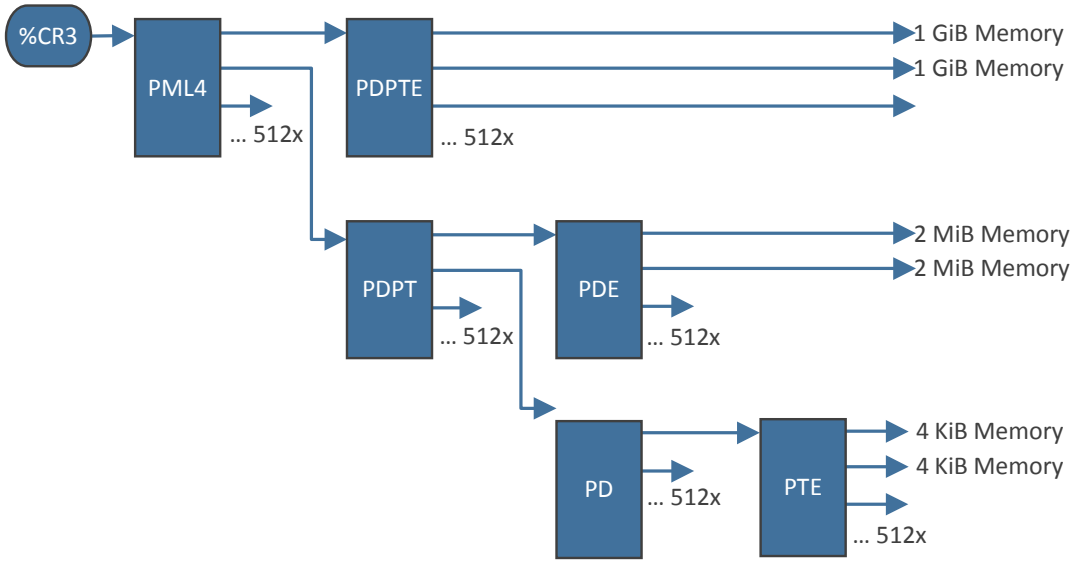

Figure 2.4.: Structure of Intel x86-64 page tables. [21]

The offset within the page is directly given in the virtual address and is just concatenated to the page frame number to form the final address. If the full page table hierarchy branch of a page table entry is currently mapped into memory, the MMU can translate addresses autonomously. If the target data page is also in memory, the access is done without OS invocation at all. In addition, the MMU caches recently used translations in the Translation Look-aside Buffer (TLB). However, all pages, including PDPT(E)s, PD(E)s, PTEs, and data pages can be swapped out to background storage. Only the PML4 is pinned to physical memory. When an address is resolved for a page whose page table entries or directories are not in memory, multiple page-faults can be cascaded to resolve the final address.

Details on the structure of x86-64 page tables were taken from the Intel Architecture Software Developer's Manual [21]. Details on sizes, contents and mechanics of page tables are out of scope of this document and can be found there.

2.2.4. Copy-on-write

The BNN pager in the TEXEX system [10] first complemented memory sharing with a copy-on-write (COW) mechanism. Using COW, memory pages are shared between processes without the involved processes seeing mutual modifications.

This is achieved, by marking the pages read-only in the page table (page-register in TEXEX). This way, the MMU suspends the current command and issues a page-fault on writes to those shared pages. The OS then creates a private copy of the to-be-written page frame, remaps the respective page to the new page frame and continues the write operation on the new, unshared memory.

The OS knows if the write operation to a read-only page is an error or a copy-on-write operation through its additional, orthogonal virtual memory translation database, so no additional hardware flag is needed to implement COW sharing.

2.2.5. Anonymous vs. Named Memory

Throughout this thesis anonymous pages are differentiated from named pages depending on their semantics.

Anonymous memory and consequently memory pages that make up this type of memory do not have a natural representation anywhere else in the storage hierarchy. Prime examples of such memory are the heap and stack segments of running processes.

Anonymous memory contents cannot be evicted from main memory without specifically writing them to a dedicated disk area on a background store (i.e., pagefile, swap partition, etc.).

Named memory pages in contrast do have a natural representation within the storage stack, generally in form of a file on disk. Prime examples of such memory are memory mapped files in general, and text segments of shared library and processes specifically. Note that files read through the VFS interface are generally buffered in a cache (i.e., page-cache, file-cache, etc.) in modern operating systems. This cache is also made up of named memory pages.

When named pages are being evicted from main memory, they can be just dropped when they still contain the unmodified contents of their source file. When, memory mapped file pages have been written, the modifications are written back to the original source file before evicting the page.

2.2.6. Paging Virtual Machines

Running virtual machines (VMs) adds another level of indirection to the virtual memory management stack. First, **guest-virtual** is translated to **guest-physical** memory, that is memory which is used like physical memory by the guest OS but in reality is virtual memory in the host (**host-virtual**). The guest-physical memory is then translated again into **host-physical** memory.

Hardware instructions to modify virtual memory state, for example setting up page tables and switching address spaces (setting the %CR3 register), are considered *sensitive instructions* that can only be executed by privileged software running in a special CPU mode. Generally, the OS kernel takes this role. When layering OSes, using virtualization, the question arises how virtual machines manage their

memory. The host cannot grant VMs full access to the hardware as it could not guarantee isolation between VMs and the host, then.

Today, the following three techniques commonly deal with the additional level of indirection:

- **Paravirtualization:** Paravirtualization was made popular by the Xen hypervisor [4]. Using paravirtualization, the guest OS is modified to use a hyper-call interface provided by the host. For example, when a new process is started within a VM, the VM registers the newly allocated memory with the hypervisor [4].

- **Shadow Page Tables:** This technique is used by the VMware virtual machine monitor. When using shadow page tables, the MMU traps to the host on changes to guest memory page tables, without the need for guests to cooperate. The host can then map memory appropriately on behalf of the guest, consulting a shadow page table which contains the physical location of guest pages [9].

- **Nested Page Tables:** Current hardware virtualization enabled CPUs are capable of *Second Level Address Translation* in hardware. This means, that their MMU can interpret page tables of nested guest OSes, effectively eliminating the need for separate shadow page tables. This kind of hardware also provides a more sophisticated TLB, which can handle address space switches with fewer TLB flushes due to an additional address space identifier. To this end, Intel ships CPUs with Extended Page Tables (EPTs) [21] while AMD implements this feature under the name Rapid Virtualization Indexing (RVI) [1]. Using hardware virtualization workloads can run up to 48% faster for real benchmarks, up to a factor of 6x faster in micro-benchmarks [9].

2.3. Traditional Page Sharing Approaches

Virtual memory systems and the copy-on-write technique, discussed in the previous section, lay the foundation stone for all transparent main memory sharing policies.

Traditional, non-virtualized operating systems share memory read-write between processes explicitly when programs request shared main memory (`shmem`) or implicitly when programs map the same file into main memory, for example using `mmap` with `MAP_SHARED`. Those operating systems can also share main memory transparently to the processes using copy-on-write semantics when cloning address spaces (§ 2.3.1), for example when `fork`ing, or when using files (§ 2.3.2).

2.3.1. Sharing Cloned Content

Memory already used to be short in early systems, but memory-shortness aggravated when time sharing systems were invented. Time sharing systems allow multiple users to launch several processes simultaneously. Virtualizing the CPU makes it possible to switch between those processes to make them appear to make progress concurrently.

Specifically in UNIX based operating systems, processes are traditionally not created and executed[2] independently. Instead, processes `fork` into two – the original parent process and a newly created child process which inherits almost all of its parent's state, such as address space, instruction pointer, open files, etc. This way all processes form a process tree with a single root – the `init` process – which is created at boot time.

The OS needs to perform the address space copy on behalf of the `fork`ing process for security reasons; modifying address spaces is a sensitive operation. In consequence, the OS knows of the copy operation and it's semantic. That makes `fork`ing a trivial target for memory sharing. Today, `fork` does not physically copy the entire address space. Instead, the *anonymous* parts of the parent address space, which are currently mapped in memory, are shared with the child process using copy-on-write. This way, unmodified memory pages remain shared throughout the entire life of the child. Memory pages that are written at run-time of either process are actually copied and thus not made visible to the other process.

2.3.2. Sharing the I/O Buffer Cache Among Applications

Data are made persistent for future use on background storage such as hard disk drives (HDDs) or solid state drives (SSDs). When programs are started, they are loaded from background store; settings and data are also loaded from there and are kept between runs of programs and even between reboots.

Long-Tail File Distribution Background storage itself is considered a system performance bottleneck. However, most systems only have few files in their active working set, and main memory is generally an order of magnitude faster than persistent storage devices, which make it a good case for caching. For example, a news web server accesses files that are visible on the front page at a very high frequency in contrast to older articles that are accessed infrequently. Such a long-tail distribution is actually common throughout the file system. Few files are accessed very frequently while many files are accessed infrequently [29]. This access distribution makes the use of file system caches reasonable as only few files need to be present for the cache to achieve a high hit-rate.

[2]The parent is responsible for collecting its children's termination state.

Unified File Caches A file system cache could be implemented in the user level on a per-application basis. A global OS cache which is used by all processes is a good design decision, however, for the following reasons: First, the same files are often used by multiple processes at the same time. An example would be the Microsoft IIS web server, which consists of multiple worker processes which all access the same data. Another example would be different independent processes that are linked against the same shared library, such as libc. Using a global cache the OS only needs to load the library from background store once. Second, files are often used again at a later time by another instance of the same program after exiting the previous instance. An example would be a cron job that runs frequently. Without a file system cache, the program would need to be loaded from background store at every run.

Consequently, modern operating systems use a portion of the main memory as a global cache for the background store. In Linux, for example, all unallocated memory makes up the page-cache which caches file data, file meta-data, etc. All processes then share this file cache, which is located in and managed by the OS.

The Virtual File System Programs access files through a virtual file system (VFS) interface. The VFS is an abstraction, which makes it possible to access every file in the same way, regardless of the underlying file system and storage media (Figure 2.5).

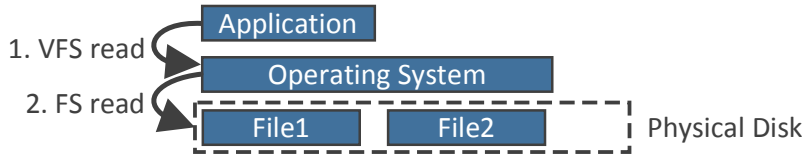

Figure 2.5.: The I/O-path of applications.

When a process reads files through the VFS, the `read` system call copies the requested data from the page-cache to the target buffer in the user's anonymous address space. If the requested file parts are accessed for the first time, they are first loaded into the cache.

Processes can also access page-cache memory directly, without copying it first. They do this by mapping the respective files into their address space (`mmap`). When multiple processes `mmap` the same file parts, they effectively share the buffer cache memory. Shared libraries are generally implemented to share their code section this way [30].

File-based Caching Sharing in file system caches is generally not based on the *contents* of the blocks that files are made up of. Instead, modern operating systems share the file system cache on a per-file basis. Files are identified by their device and file system internal file number. When using `mmap` for example, the <`inode`, `offset`> pair is used as the semantic origin.

Although that works very well when mapping the identical file, i.e., any one of the hard-links with the same `inode`, multiple times it fails to provide memory sharing between *copies* of the same file as all copies are represented by different `inodes` in the file system.

This is even true if a deduplicating file system is used: Although every block with a certain content ideally only exists once in the block-layer of such a file system, different files that contain the same blocks still have their own file control block (`inode`). When two files that are made up of the same data blocks are loaded into memory, these previously deduplicated file blocks are actually duplicated again in the file cache.

Content Caching Koller et al. have identified the duplication issue in file system caches and propose to replace traditional caches with a combination of two techniques: A *content based cache* stores each block content only once. Blocks originating from different files but with the same content are neither duplicated in nor are they retrieved from the background store again. The *dynamic replica retrieval* mechanism selects the replica to be fetched that leads to fewest retrieval overhead (e.g., disk arm movement). Additionally, Koller et al., propose to choose good candidates for selective disk deduplication through statistics generated from the dynamic replica retrieval. [48]

This system requires a strong linkage of the file system and disk cache as the file system needs to provide content hashes from metadata (i.e., without reading the data from disk). This makes a wide-spread deployment of such a system in the near future unlikely, as today's systems often orchestrate many different local and remote file systems in a single, global cache.

2.4. Deduplication for Virtual Machines

Although the concept of virtualizing resources, including virtualizing entire machines has already been invented in the 1960s, it has not had its breakthrough until the beginning of this millennium when infrastructure as a service (IaaS) and cloud computing became popular [36, 50]. In today's multi-core era we recognize the main memory size as a primary bottleneck when running multiple virtual machines (VMs, guests) on a physical machine [39].

Sharing Potential Multiple VMs running on a host often contain equal pages within and across the VMs as previous studies have shown (Table 2.1). In the best case the same OS, libraries or programs handle the same data. To increase the sharing potential, cloud providers can consolidate VMs with close workloads on the same physical machine [88].

Name	Source	Configuration	Equal pages
VMware ESX	[84]	10 VMs, SPEC95	65 %
Difference Engine	[39]	3 VMs, XP/Linux, RUBiS/LAMP	40 % – 85 %
Satori	[63]	2 VMs, Kernel-build/Apache	11 % – 66 %
Chang et al.	[14]	Hadoop, HOMP (MPI), LAMP	11 % – 86 %
Barker et al.	[5]	desktop/server snapshots	15 %

Table 2.1.: Memory sharing potential according to previous studies.

Memory Overcommitment for Virtual Machines Memory overcommitment is an important technique to increase the number of VMs on a physical host. Traditional sharing policies, however, cannot effectively share memory in virtualized environments.

Those policies are limited to scenarios in which sharing semantics are known a priori: Pages with the same origin are shared; the actual content of memory pages is not regarded for finding further sharing candidates. There is a semantic gap (§2.4.1) between the host and the VMs which makes those policies inapplicable for sharing memory without modification.

Address space as well as buffer cache sharing can, nevertheless, be extended in order to be useful in virtualized environments: Cloning of address spaces has been adjusted to make cloning and sharing entire VMs possible (§2.4.2). In addition, different propositions have been made in the past to make it possible to share file caches across VMs (§2.4.3). Moreover, new mechanisms and policies have been created to deal with the remaining shortcomings. In order to circumvent or mitigate the semantic gap, and make sharing of (a subset) of those targets possible across guests and the host, two different approaches have been taken in the past: Semantic agnostic inspection of memory page contents through memory scanners (§ 2.4.4) and the paravirtualization and introspection of guests through the instrumentation thereof (§ 2.4.5).

Memory Pressure vs. Memory Deduplication Memory deduplication increases the memory density across actively used memory in the system. Effectively, memory deduplication is a compression technique that pushes back the boundary at which the system starts thrashing.

In consequence, the justification of memory deduplication varies depending on the current system state. If the system is not under memory pressure, no memory deduplication may be needed at all. If the system is already thrashing, even putting many CPU cycles and much memory bandwidth into scanning can be justified if it brings the system out of the thrashing state.

2.4.1. The Semantic Gap

Virtualization hosts cannot easily share VM memory using traditional sharing policies due to a *semantic gap*, which is introduced through an additional level of memory indirection in the virtualization layer. Due to the semantic gap, semantic information about memory mappings is only available in the virtualized guest, but not in the host [15].

Semantic gaps can generally occur between any subsystems separated by an abstraction layer, whether it be a virtualization host and guest, or an OS and a running process.

Examples for information that is lost when traveling abstraction layers are:

- If two files are the same: The copy semantic is lost after the copy operation at the VFS layer.
- If a memory page is anonymous, named, or device memory: The mapping semantic is lost at the nested virtual memory translation layer.
- If two address space regions are two instances of the same program they likely contain similar data: The program semantic is lost when the guest OS is left as guest processes are not represented in the host.
- If a program or VM copies memory within its address space: The copy semantic is lost after the loop or the appropriate library call (i.e., `memcpy`).

2.4.2. Address Space Cloning in Virtualization

When `fork`ing, the parent process is used as the semantic origin for sharing memory and the entire address space is shared between the parent and child processes. The Android OS uses this property of `fork` to optimize its run-time environment [68]. Here, all processes are based on Java and run in Android's custom Java interpreter, the Dalvik virtual machine. In order to easily share the core libraries and the virtual machine itself (including some interpreter run-time state), all processes are forked off of a well-defined Dalvik VM base process, the Cygote process.

For VMs, this kind of sharing only makes sense after the guest OS is fully booted, as most of the memory contents change while booting. In consequence, mechanisms that work analogically to `fork`-sharing have been used to exploit sharing potential when *cloning* VMs in the past [50, 83]. Cloning VMs works very well for sharing initial memory images of virtual machines using COW semantics with a largely common state. Guests cannot share pages that will establish equal contents *after* cloning the VM using this technique, however. Using COW semantics when cloning to remote hosts can also be beneficial as it allows the cloned source VM to continue running while the cloning is in progress [50].

Virtual machine cloning has not caught on to a wide popularity as it requires spawned guests to be equal to be applicable. In contrast guests tend to be regarded as black boxes to the underlying host system.

2.4.3. VM I/O Path and Disk Cache Placement

Although file system cache placement is easy in a single OS (§2.3.2), it is hard to cleverly place this cache in the VM I/O path. Virtual machines do generally not share a file system with one another or their host. Instead, VMs implement their own file system on a virtual block device which is then mapped to a large file in the host called the **virtual disk image** (VDI). The VDI can contain an arbitrary file system determined by the guest OS. The host does not have to be able to interpret the guest file system and usually doesn't.

Figure 2.6 depicts the I/O path that applications running within a VM take when they access their background storage:

1. The application issues a file operation either through a VFS system call to the guest OS or by accessing a memory mapped file.
2. The guest file system translates the accessed file location to blocks on its (virtual) disk. The OS then programs the (virtual) disk's (virtual) DMA controller to access these blocks.
3. The host implements the virtual disk, generally mapping it 1:1 to a file – the VDI. The hypervisor consequently accesses the VDI file through a host VFS system call in order to satisfy the guests (v)DMA request.
4. The host's file system translates the VDI file location to blocks on the (physical) disk and accesses this disk through the (real) DMA controller.

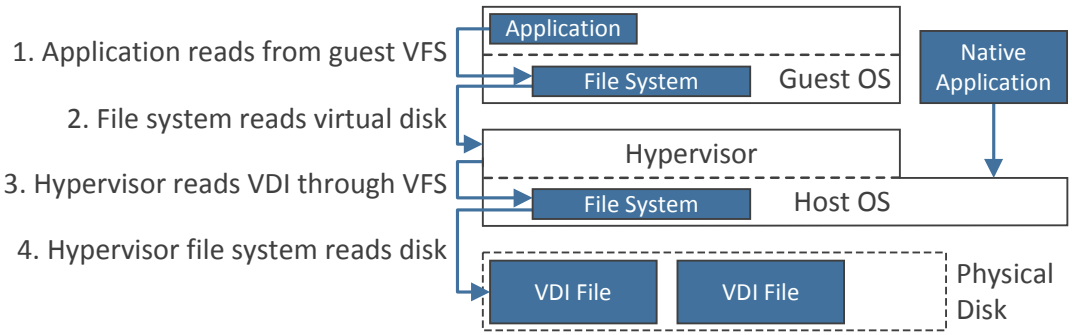

Figure 2.6.: I/O-path from an application through the guest and host to background storage when using virtualization.

Caching can take place in the host as well as in the guest, leading to four different I/O-cache configurations described by Zhang et al. [89] and by Balbir Singh [78]:

- **Host Cache: on | Guest Cache: on**

  This configuration is the default in KVM [22] and uses at least twice as much memory for caching as the other configurations due to the *double-caching problem* [77]. All data from background storage is cached twice: once in the host and once in the guest.

  Although this configuration has a high memory overhead, it also has the best caching characteristics. When a file is first accessed by a VM, there is still a chance that the file contents are already in the host's cache. Afterwards there is a great chance that it can directly be reused from the guest's cache without trapping to the host.

- **Host Cache: on | Guest Cache: off**

  This configuration is not common as it leads to high context switching overhead and poor performance isolation. For every background storage access, the VM must exit and trap to the hypervisor to satisfy the I/O request from the host's cache.

  In turn, this solution has a good chance for main memory sharing within the host cache if multiple VMs were booted from the same VDI file. Note, however, that no sharing would take place when booting from different VDI images even if equal blocks existed. Moreover, this configuration must be specifically supported by the guest, as its cache needs to be turned off.

- **Host Cache: off | Guest Cache: on**

  This configuration is common and is the default setting in VirtualBox [41] for example. Guests have the most accurate knowledge about the active working set of their applications and in consequence they know best what to cache.

  This configuration has no prospect of memory deduplication between VMs and in addition the VM cannot help the VM's I/O through prefetching and caching even if there is memory available.

- **Host Cache: off | Guest Cache: off**

  This configuration is generally not a good idea, because background storage I/O is often a bottleneck that can be significantly improved through caching.

2.4.4. Deduplication Through Inspection: Main Memory Scanners

Carl Waldspurger introduced several new memory management techniques for virtual machines to the VMware ESX server. The most relevant technique to the thesis at hand is *content-based page sharing* via main memory scanning [84]. Content-based page sharing takes a black-box approach to deduplication. The semantic of memory pages is not regarded in the deduplication process at all. Instead deduplication is done purely on the basis of the content of memory pages. Equal pages can be shared regardless of their semantics and histories. In consequence, the guest does not need to be modified in any way, which is one of the greatest advantages of memory deduplication scanners.

Memory scanners are a process, that wakes up periodically, chooses pages to scan, and adds the contents of those pages to an index before going back to sleep. When equal pages are found during the insertion, the affected pages are merged transparently using the regular COW mechanism.

Linux followed with the same general idea under the name KSM [2]. Lee et al. extended the scan process to host page cache memory [52]. Difference engine extended memory scanning to sub-page sharing and compression [39].

VMware ESX

In ESX, duplicate host-physical page frames are found using the *compare-by-hash* method, which associates every content with a hash value. Hash values can then be compared in proxy of the actual value, speeding the process up thanks to their much smaller size [40].

Hash values can, for example, easily be indexed through a hash-table; the method implemented in ESX (Figure 2.7). False positives, which occur when two different pages compute the same hash, are eliminated through a full comparison of the original data. When matching pages have been identified, these pages are merged using the regular copy-on-write mechanism. Merged pages remain in the hash-table in which they are marked as *stable* until they are written. If a page does not currently have a sharing partner in the system, the hash value is recorded in the hash-table without marking the respective page COW. Those values are treated differently than stable pages in the hash-table. On a future hash value match with such a page, it's hash value is first recomputed and updated before potentially going into the comparison phase described above. [84]

In ESX, pages are indexed at a fixed rate, in random order [84]. The original ESX paper hypothesizes that a better scan order heuristic could be beneficial, it does not propose a well suited policy, however.

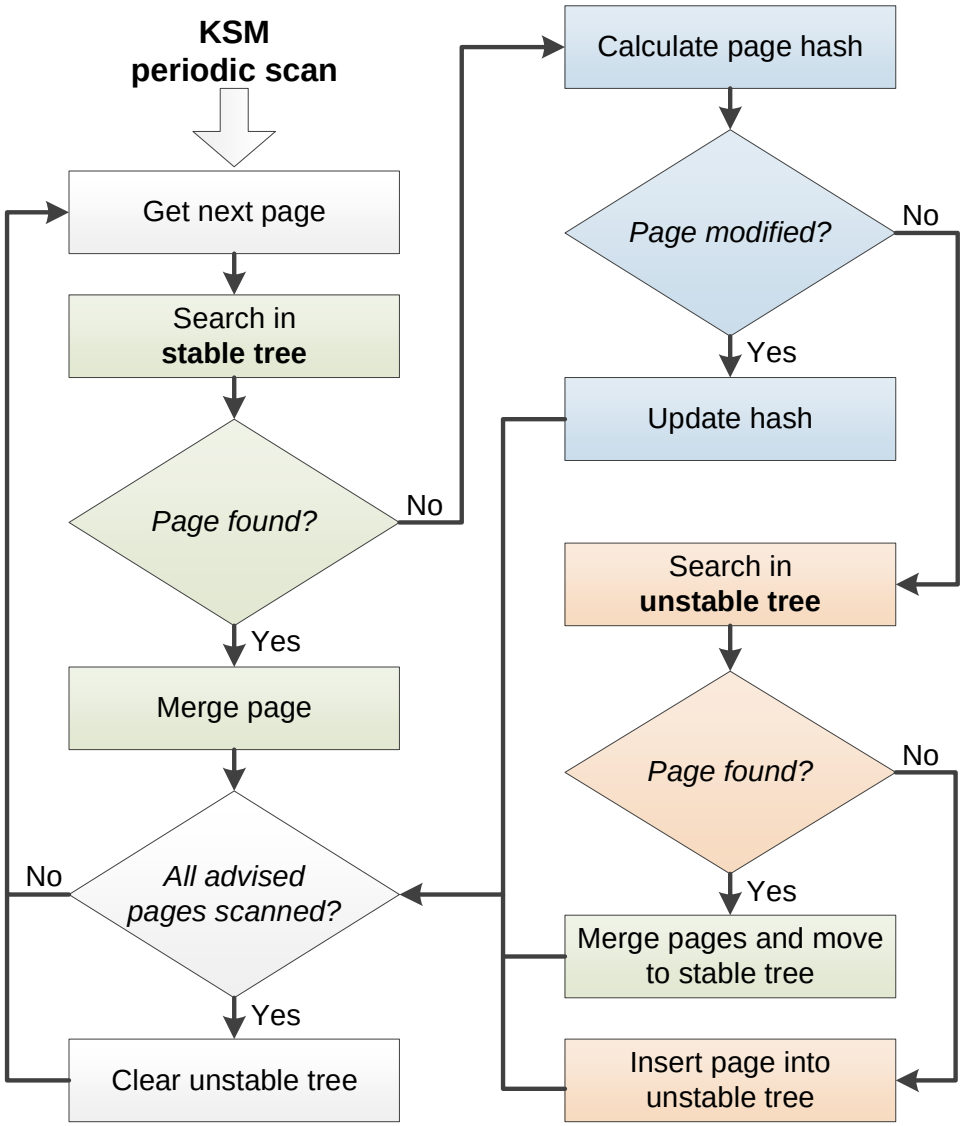

Figure 2.7.: High level overview of the ESX memory scanning process.

Linux KSM

Andrea Arcangeli et al. brought memory deduplication scanning into the Linux mainline kernel under the name KSM. The acronym KSM stands for either *Kernel Samepage Merging* or *Kernel Shared Memory*. [2]

In contrast to the method introduced by ESX, KSM does not scan the entire host-physical main memory when enabled. Instead, KSM operates only on anonymous host-virtual main memory regions that have been specifically advised to the OS to potentially contain many duplicate pages using the `madvise` system call. This is done from user space and popular virtual machine monitors `madvise` the entire VM memory space when bringing up VMs.

The KSM scan process also differs from ESX in the policy that it uses to select the next candidate to be scanned. KSM scans advised virtual memory areas sequentially in a round robin fashion, as opposed to ESX, which randomly picks physical page frames to scan next.

KSM allocates a tree node data structure for every `madvised` page in the system. Those nodes contain, beside a pointer to their respective anonymous page, additional information such as the `jhash2` checksum that this page's content had at the last visit and a sequence number that is incremented at each visit.

The indexing data structure needed to be revised in Linux, because VMware had patented the use of compare-by-hash for memory deduplication [85]. Instead of using a hash-table, the nodes described above are linked into two red-black trees using their full page content as the key into the tree.

The first tree is named **unstable tree**, and records pages that have likely not changed since their last visit. The recorded checksum is used to check for this property. These pages are not protected from being written to, however, their content may thus still be unstable, hence the name. The second tree is named **stable tree**. In contrast to the unstable tree, it records pages that have already been merged and marked COW and in consequence, it contains pages that cannot be modified prior to a COW page-fault. Note, that KSM is not explicitly notified when such a page-fault occurs. The fault is handled by the page-fault handler alone. KSM can however notice that as page-fault has happened from its data structures and clean up the *stale* stable-tree entries to keep it consistent.

The KSM scan process is depicted in Figure 2.8. Every time a page is visited, KSM first checks the stable tree if a merge candidate is present, already. If this is the case, the new page is added to the sharing group and the scan moves on to the next virtual page. If there is currently no equal stable page, the `jhash2` checksum is calculated and the scanner checks if the hash value has changed since the last visit of that page. If it has changed, the hash value is updated and the scan continues with the next page. Otherwise, if the page's content has not changed since the last visit or if a hash collision has occurred, the page is looked up in the unstable tree. If a sharing buddy is found there, both pages are merged and inserted into the stable tree. Otherwise, the new page is added to the unstable tree.

When all advised pages have been scanned, the entire unstable tree is dropped and the process is repeated from the beginning, leaving only the stable tree and this round's checksums behind.

All information for the description of KSM have been taken from Arcangeli et al. [2] and from studying the Linux kernel source code 3.0 to 3.4, which can be obtained from `www.kernel.org`.

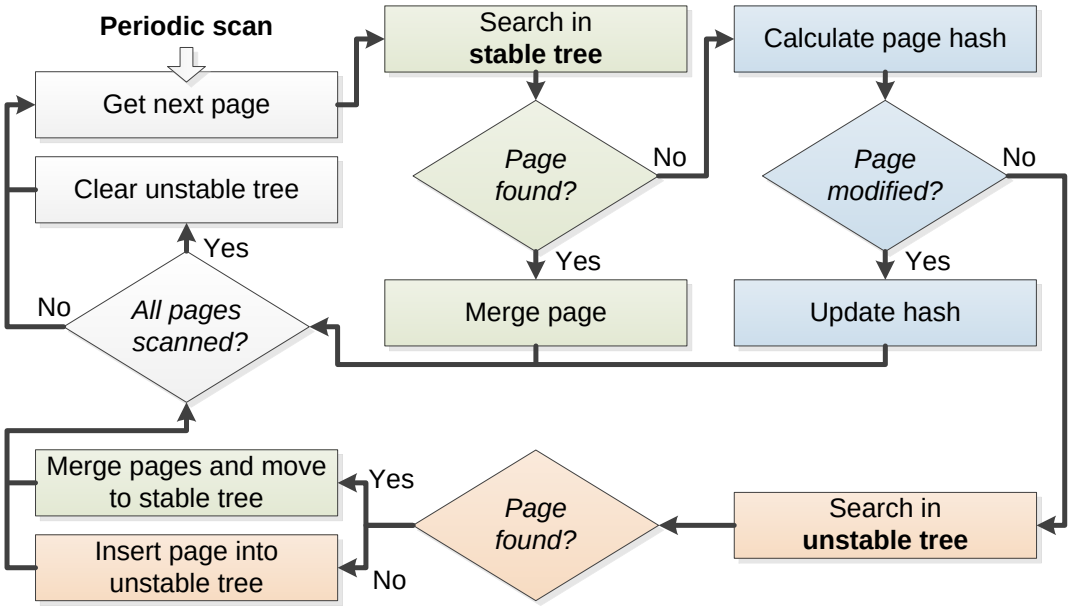

Figure 2.8.: High level overview of the KSM memory scanning process. [62]

Host Page Cache Deduplication

Lee et al. propose a twofold approach to deduplication. The unmodified Linux KSM memory scanner is used to deduplicate memory allocated by virtual machines. In addition, a second memory scanner is introduced that indexes the host's page-cache. The second scanner therefore merges files that are cached in the host, based on their content. [52]

Using both techniques simultaneously, Lee et al. can free more memory than KSM alone if VMs ship with their own VDI and do not use a static base image in conjunction with virtual copy-on-write disks. [52]

Difference Engine

Difference engine extends previous memory scanning approaches that were capable of deduplicating equal pages with sub-page sharing through memory patching that is capable of merging *similar* pages and through memory compression.

Difference engine, in addition, is also the first scanner that makes use of page table flags to efficiently classify pages based on their "freshness". Based on this metric for the expected page modification frequency, Difference engine chooses whether sub-level patching and compression are likely to cause high overheads, as those techniques must be reversed in a page-fault at each access. Note that this includes read accesses and not only writes as in other techniques. [39]

2.4.5. Deduplication through Paravirtualization and Semantic-Aware Inspection

Deduplication systems that fall into the category of paravirtualized- or semantic-aware inspection-based systems aim to explicitly track changes made in guests in order to observe the creation of duplicates. These systems find duplicate main memory pages either through intrusive instrumentation of guest operating systems (paravirtualization) or by inspecting the guest unintrusively.

All techniques introduced in this section require a strong interdependence between host and guest. Paravirtualization based techniques require the possibility to modify the guest OS in order to create an interface between the host and the guest which is used to communicate semantic information to surpass the semantic gap. Inspection based techniques, in turn, only work when the target is specifically known and supported by the host.

(Cellular) Disco

Disco is a virtual machine monitor (VMM) that can run multiple instances of the IRIX operating system on an SGI Origin 2000 system. The project's goal was to avoid extensive and complicated changes to commodity operating systems, such as IRIX at the time, to make them scale on many-core processor systems. Instead, servers were intended to scale by running multiple instances of available multi-processor (few cores) OSes on top of a virtualization layer. The processes are then parallelized on the application layer using the efficient communication mechanisms that the VMM provides. [12]

Bugnion et al. identified three challenges in their effort: Virtualization overhead such as an increasing number of TLB flushes, resource management issues such as lock holder preemption, and finally communication and sharing. All three challenges are attributed to the introduced semantic gap between the VMM and the workloads running within their guest OS. [12]

Addressing the semantic gap in the interplay between (virtual) disks and buffer caches, Disco introduces copy-on-write disks. Disco intercepts all DMA requests to those disks and transparently omits reading the same data multiple times through copy-on-write sharing previously read data in main memory. [12]

Cellular Disco extends the original Disco approach, which was viable for smaller scale servers, to be useful on large-scale shared-memory NUMA multi-processors. To this end, Cellular Disco adds hardware partitioning as well as resource management mechanisms and policies. Cellular Disco for example allows guest OSes to overcommit physical memory – it allocates more physical memory to virtual machines than available. [35]

Collaborative Memory Management (CMM)

Due to the semantic gap in virtualization, the usage semantic of guest pages is not known to the host. In particular, memory pages can be currently unused and marked as free in the guest but still carry their last content. Linux, for example, does not zero memory pages until it allocates them to processes, in contrast to Windows which has a thread dedicated to zeroing free pages.

Those free pages are consequently handled the same way as any other page in the system by the host regarding page allocation, page replacement, and swapping. Those pages moreover take up valuable space in physical memory that could otherwise be overcommitted to other guests.

Collaborative Memory Management (CMM) paravirtualizes the Linux guest to communicate memory usage semantics to the zSeries z/VM hypervisor [74].

Memory pages can be in one of the following states:

- **stable:** default
- **unused:** can be discarded
- **volatile:** guest can tolerate data-loss
- **potential volatile:** guest can tolerate data-loss if page is not dirty in PTE

The goal is for CMM to identify the working set of guests in order to trim the guest's working set size accordingly. Using CMM, unused pages can for example be discarded and reused by other guests instead of expensively and unnecessarily swapping them to disk. [74] The host itself can also put such memory to good use for example for caching.

Geiger

While the host can overcommit its physical memory to VMs, it needs to be careful that the active working set of currently running VMs needs to fit the total main memory size. Otherwise, the system will start thrashing, degrading the performance of all VMs. Thrashing is even harder to handle in virtual environments due to the additional layer of memory translation and the semantic gap between those translation layers.

Geiger unintrusively infers page usage information from guests in the Xen hypervisor [4]. The information collection is based on techniques that were previously used to optimize buffer cache placement [16, 87]. Applied to virtualization, these techniques can notice when pages are inserted into and evicted from the guest buffer cache, and also notice when the guest is swapping and replacing pages by tracing page-faults, page table updates, copy-, and disk operations [43].

With this semantic information, Geiger aims to make good resource decisions for VMs. An example application is working set estimation that can be used to prune VM memory. Another application can be to use the inferred information to improve second-level caching, for example by avoiding caching the same data twice (see §2.4.3).

Satori

Satori [63] is a Xen [4] based solution that uses paravirtualized, virtual, content addressable disks to detect sharing opportunities that originate from the guest file systems and are read into the guest's page cache.

Satori aims to prevent duplication from emerging instead of merging it after it has occurred. The mechanisms Satori uses to reach this goal are very similar to the mechanisms that have previously been used to implement memory deduplication in Disco. Both systems deviate in the location where the deduplication mechanisms are implemented. Satori implements a content addressable disk in the form of a paravirtualized virtual disk as opposed to Disco, which implements their content addressable disk in the host.

Satori performs very well in workloads that are not sensitive to I/O latency and jitter. It can for instance prevent 18%–50% of all duplicates from being created in a Linux kernel-build benchmark and share up to 94% of all duplicates in an apache web server scenario. The evaluation, however, shows high performance overheads of up to 34.8% in the Bonnie file system benchmark. [63]

Transcendent Memory

Transcendent Memory (tmem) aims to claim unutilized physical memory to make it available where it is needed. Using tmem, the virtualization host acts as a server that manages a pool of physical memory. Guest VMs are tmem clients that can claim memory from the memory pool through a well-defined interface similar to key-value stores (`get`, `put`). Using tmem turns classical static provisioning of main memory that can only be trimmed through techniques such as memory ballooning [84], into dynamic provisioning of main memory. [56, 57]

XHive

XHive is a cooperative caching scheme similar to Transcendent Memory. Instead of using a host page cache for VDIs, XHive implements a cooperative caching scheme between VMs. The host implements a content addressable key-value store that acts as a swapping device for the guests. [47]

XHive thus practically gives memory pages that are swapped out a second chance to remain in memory. This is implemented by moving swapped pages out of the guests' quotas to the host where they can be shared among multiple VMs.

Singleton

Singleton addresses the double-caching problem. Sharma et al. optimize caching in KVM based environments by eliminating duplicates from the host cache.

VM memory is deduplicated using the default KSM memory scanner [2]. At the same time, Singleton concurrently looks up host page cache pages in KSM's data structures that store the state of the ongoing VM memory deduplication. This way, singleton can identify if a host page has a high probability to already be contained in a guest's page cache and evict those pages freeing memory in the host. [77]

Small Is Big

In order to achieve a good compromise between VM switching overhead and cache deduplication Zhang et al. propose a combination of the two caching policies:

- Host Cache: on, Guest Cache: off
- Host Cache: off, Guest Cache: on

In their work, the guest is paravirtualized and decides based on the file semantic if it should be cached on the host or in the guest. Files that ship with a base OS image are cached in the host as these files exhibit the greatest chance for sharing in the cache. Private files that were created after the base image was first booted are then cached in the guest's cache to optimize the access latency of these files. On the one hand, if multiple VM instances are booted from the same VDI image, a great portion of the host file cache can be shared. On the other hand, all accesses to those files are then subject to extended access delays due to additional VM exits. [89]

2.5. Conclusion: Limitations of the State-of-Art

Virtual memory systems made transparent copy-on-write memory sharing possible in the early 60s. This technique has made its breakthrough when time sharing systems were built for the first time. Copy-on-write can be used to share anonymous address space regions between processes and to share the same buffer for files among different processes. Those traditional sharing policies cannot be used for

deduplicating VMs, as those mechanisms are purely based on sharing objects that come from the same source, known a-priori. Virtualization hosts do not have such knowledge, however, due to the semantic gap.

Paravirtualization-based approaches transport and leverage semantic information and thus exploit specifically considered sharing sources instantaneously at creation time. However, using this approach, the hypervisor needs to process all I/O and hooks right away. This can be a bottleneck for I/O-intensive workloads, and for applications that make frequent use of hooked calls. Moreover, paravirtualization directly implies, that the virtualization guest needs to be modified. This, however, is not always possible, for example if the source code is not available.

Memory deduplication scanners treat all memory equally regardless of their semantic and are thus able to exploit sharing opportunities from all sources without the need to modify the guest OS. However, scanners have their downside when it comes to deduplicate short-lived sharing opportunities quickly, as those systems can only index memory pages at a limited rate.

Chapter 3

Analysis of Main Memory Duplication and Sharing

This chapter motivates that memory deduplication methods can be improved by incorporating semantic knowledge into the deduplication process. Therefore we assess the opportunities for deduplication qualitatively and quantitatively.

Section 3.1 enumerates methods to measure and analyze memory duplication. The following Section 3.2 gives reasons for memory duplication. Then it quantifies the amount, sources, and characteristics of sharable memory regions obtained through various benchmarks and data acquisition methods. Section 3.3 concludes the chapter with a summary.

3.1. Measuring Main Memory Duplication

Main memory today[1] can be modified at peak frequencies of up to $6.3 \cdot 10^6 \frac{\text{pages}}{\text{second}}$, in single socket systems. When measuring memory characteristics in software it is unavoidable that the measuring process itself influences the target workload. In consequence, one needs to be very careful to use appropriate measurement techniques that keep the distortion within reasonable bounds.

In order to analyze the properties and development of sharing opportunities, we need possibility to inspect memory contents and to track their modification over time. The following sections discuss different techniques that can be used to trace main memory modifications which allows us to observe memory duplication to analyze its characteristics.

[1] November 2013, PC3-24000U DDR3 RAM

Uhlig and Mudge, consider different attributes in their review of trace-driven memory simulation techniques [82]. I redefine those attributes to fit memory duplication analysis and address those attributes in my following review of memory deduplication analysis methods:

- **Completeness:** Can all memory duplicates be observed?
- **Detail:** How much context, beyond the quantity, can be derived?
- **Distortion:** How does the analysis influence the measured workload?
- **Ease-of-use:** How hard to use is this method?
- **Slowdown:** How much longer does the workload run when analyzed?

3.1.1. VM Snapshots

Any privileged process can freely set-up virtual address spaces. This enables such processes to inspect memory contents of all other processes on a system. Operating System (OS) kernel modules for example can access memory pages of any process in the system; that is done in memory scanners to insert pages into the index. Hypervisors also use this privilege to create memory snapshots of active virtual machines (VMs) in order to suspend, migrate, and restart virtual machines.

Periodic main memory snapshots can be used to analyze main memory characteristics of processes such as virtual machines. This can be done by sampling and recording memory contents periodically. Afterwards, for example when the workload has finished, different properties can be investigated in the recorded data. One example is the average number of duplicates in each snapshot.

Completeness Following the Nyquist-Shannon sampling theorem, the sampling rate needs to be twice the *cutoff frequency* to see every memory modification. This means that two reads need to be performed to every memory page in the system for every write that can be done during that time. This is impossible, with random access memory (RAM) typical equal read and write speeds. Thus, VM snapshots can thus never detect all main memory modifications. In conclusion, VM snapshots are functionally incomplete for memory duplication analysis as a duplicate can be created and destroyed, before we can observe the duplicated state in a snapshot.

Slowdown There are two overheads that, taken together, make up the time needed to create a snapshot. First, a near-constant overhead in which the system scans main memory pages and detects which pages have been modified since the last snapshot. Second, the time needed to write the modified data to persistent storage, for example to disk.

Nikolai Baudis measured the overheads for creating incremental snapshots as a part of this Bachelor thesis [6] at our institute. His implementation records memory images of QEMU VMs [8], storing new memory pages on a local, high-performance key-value store. In a benchmark, incremental snapshots of a 2 GiB VM were taken in 2 second intervals. Between consecutive snapshots, the workload modified about 4000 pages. Those were written to a local Redis [73] server where a solid state device (SSD) was used as the background store. The snapshot downtime – the time between the beginning and end of a snapshot in which the VM is inactive – was around 400 ms. Parsing QEMU's bitmap that indicates dirty pages took 25–40% of each snapshot's time. Hashing the dirty pages which need to be stored took between 15% and 20% of this time and 40–60% of the time could be attributed to writing the modified memory pages to the background store.

Scanning for modifications on the one hand and storing those modified pages on the other hand limits the attainable snapshot resolution in practice. The lower bound for creating a snapshot is the (constant) time it takes to scan memory for modifications in a scenario where no pages have been modified since the last snapshot. The upper bound for creating a snapshot is the time is takes to scan memory when all pages have been modified and then persisting all (modified) pages.

Not all analyses require snapshots that encompass the full memory content. When quantifying the amount of memory duplication in a system, it is for example enough to record collision resilient hashes (e.g., cryptographic hashes such as SHA-1). Colliding hashes can then be used to count duplicates in the system. For our analysis we have implemented a kernel module that creates periodic snapshots using hashes [37]. We use this module to infer the sharing opportunities baseline from secondly snapshots when evaluating different memory deduplication strategies.

Distortion There is a trade-off between the resolution of this method, depending on the sampling rate on the one hand and its influence on the measured workload on the other hand. While creating a snapshot, the workload is typically not scheduled by the OS, to make those snapshots consistent. That results in a *downtime* in which the snapshot is taken while the workload cannot run. The length of this downtime determines the influence of this analysis method on the execution of the workload.

Specifying the downtime is not easy as it depends greatly on the amount of memory that the workload modifies, on the speed at which the workload modifies memory and on the sampling rate that the snapshots are taken in. When the snapshot frequency is high, the workload has less time to modify memory pages due to the shorter time interval between snapshots. Intuitively, one could think that the shortened snapshot time can even out the increased number of snapshot that have to be taken. Two things need to be taken into consideration in this process, however. We have seen, creating a snapshot can be subdivided into the time it takes to find the modified pages since the last snapshot and into the time it takes to persist those new pages. First, the time it takes to scan for modified pages is

constant; all pages in the system need to be checked for modification, whether this is done via the page table modified bit data structures which the VM provides or via hash-and-compare. Second, the total number of stored, modified pages is greater when creating snapshots at a higher frequency. Making more frequent snapshots increases the resolution of this method; more of the intermediate memory states are then recorded in addition. So, more frequent snapshots always lead to a longer total downtime.

Ease-of-use and Detail The development of memory contents can be easily tracked with snapshots, regardless of the workload itself. In the case of VMs this means that this technique can be used without knowing what guest OS is running. Some information can be directly extracted from each snapshot such as: the number of duplicates – pages that can be shared and then freed – and the size of sharing groups – then number of pages that make use of a sharing page. Other information require the correlation of different snapshots, such as the time that pages could remain shared. Extracting such information can easily be done.

Information that require semantic knowledge of pages are very hard to obtain. Gathering semantic information from a series of snapshots is hard, because the full chronological sequence of memory modifications is not available. In order to obtain semantic information from snapshots, reverse engineering techniques need to be applied to every snapshot. These techniques have to be tailored to each target workload, which requires time and internal knowledge of the analyzed system. For example, when analyzing a guest OS, the internal data structures that represent processes (PCBs) and address spaces need to be found. Those data structures reference other data structures with *virtual* addresses, as opposed to the physical memory addresses that are known in the memory dump. Virtual addresses need to be translated in the analysis using the page tables in the snapshot.

Jonas Julino has analyzed memory snapshots taken from the Android SDK simulator in his study thesis [44]. He has written a tool to find processes and correlate their address spaces through reverse engineering in single memory dumps. About 9.5% of all non-free and non-zero memory pages are duplicates in the used Android OS. Julino found the Dalvik VM to be the main source of duplication.

3.1.2. Page-faults

In paging-based memory management systems, virtual pages can be write-protected in their respective page table entry (PTE). When a write-protected page is written, the memory management unit (MMU) signals an exception which invokes the OS. The OS can then take appropriate measures to deal with the exception. Either, the page was intentionally mapped read-only and is not supposed to be written then the access has been correctly denied and the process is killed unless the process provides

an exception handler for such cases. Or, the page was mapped read-only for the OS to get a signal (the exception) on the next memory write.

A good example for the second scenario is the copy-on-write (COW) mechanism. The page frame is intended to be shared transparently, thus the write-protected page mapping is used to guard the page frames' content. When a sharing process writes his respective page that is mapping to this page frame, the MMU signals the OS, which then gets a chance to copy the page frame and map a private copy with read-write permissions to the writing process. The OS knows, that the write operation was in fact allowed despite the read-only mapping, through its internal virtual memory area records.

The same mechanism can also be used to analyze memory write bursts. To this end, the OS maps all pages that should be considered in the analysis read-only. When a page-fault is signaled, the COW mechanism is used to save the previous state of the page and transparently fork a new page for the writing process. The OS then sets a timer to take away write permissions on the copied page at a later time to repeat this process for the next write burst.

Completeness The copied page needs to be mapped read-write until the faulting process has written the page, but that there is no way to halt modification precisely after the first write operation completes. The OS has to monitor (poll) the PTE's modified flag, to learn about write operations. If it finds the flag set, it remaps the page back to read-only. This way, not all single write operations can be record. Write bursts at the granularity from the page-fault until the OS maps the page back read-only are recorded, instead.

Note that using this analysis method does not impede the original COW mechanism as the OS can infer the semantic context of the page-fault from its internal virtual memory area data structures.

Detail The amount of additional semantic context that can be derived from the analysis using page-faults depends on where the analysis takes place. The OS can easily do this analysis for native applications. It can moreover do it regardless of the position of the OS itself; the analyzing OS can run on bare metal as well as in a VM as a guest. When analyzing at this level, semantic information such as virtual address mappings are trivially known to the analysis by accessing the appropriate data structures in the page-fault handler.

Mapping pages read-only and using this method to analyze virtualization guests can also be done in the host by setting all guest pages to read-only in the extended page tables (EPTs). This modification is fully transparent to the nested page table entries used by the guest OS itself. If the analysis takes place on this level, the same semantic gap, as described for VM snapshots applies. Introspection methods are needed to infer the semantic context of written pages.

Distortion and Slowdown This analysis method influences the measured workload in three ways. All three distort the measurements by skewing with the timings in which instructions are executed and slowing down the progress of the workload.

First, the access latency is skewed. Usually, the read and write latency to RAM have equal access latencies. When using this method, the first write in a burst raises a page-fault which causes the OS to copy the associated page. The page-fault handling slows down the write latency by two orders of magnitude[2].

Second, the analysis distorts the write throughput due to the increased number of page-faults. As long as there is free memory available, the analysis can copy the faulting page and restart the faulting operation. Persisting or analyzing the page content can be deferred to a later time and in addition done by a different CPU core. After the respective data has been saved, the page can be added to the free pool again.

Even without persisting changes, the increased page-fault rate can slow down memory writes significantly. For example, on a laptop with an Intel i7-3520M CPU and 1333 MHz DDR3 RAM, writing a 1 GiB chunk of contiguous memory with `memset` is slower by a factor of 6x when every page experiences a COW page-fault on the first access compared to writing without page-faults.

The third kind of slowdown occurs, when the analyzing machine runs out of memory. This happens when the write-burst rate of the target is higher than the rate in which the copied pages can be analyzed and persisted. In such a case, the target workload must be halted until the memory pressure has lifted. If the whole content is important for the analysis, write-burst rate is limited by the write rate of the background store. For analyses with many similar memory regions, for example larger equally initialized regions, it may be beneficial to compress memory pages before persisting them. It may also be enough to record hash values of such pages.

Ease-of-use The implementation is not as easy as the previous method. When changing the memory subsystem of an OS, there are many pitfalls. Timing and locking bugs are hard to find at this level in addition. When this method is implemented correctly, however, the collected information is as complete as it would be using snapshots with a high sampling rate. The same algorithms for building semantic context for further analysis can also be used from the previous method.

[2]Measured 122.5x by touching 1M consecutive pages in a loop with and without COW breaks, subtracting the time the loop needs to run. Benchmark was run on an Intel i7-2600K with 1333 MHz DDR3 RAM running Linux 3.4.

3.1.3. Emulation

Programs that have been compiled for one computing system, can generally not run on another system unless the new system implements a compatible instruction set architecture (ISA). Moving from one computing system to another thus generally breaks the binary compatibility of programs.

Emulation, originally invented in the early 60s, eases the transition to succeeding computing systems by enabling old binary programs to run on a new system with incompatible ISA. Emulation achieves this in one of two ways [81]:

- **Simulation:** The machine code is interpreted at execution time, or
- **Binary Translation:** Old machine code is converted to the new ISA.

Completeness Emulation is done at instruction level, whether the instructions are interpreted or translated. The emulator keeps track of how instructions modify a (virtual) hardware model. In consequence, the emulator has access to all details that can be inferred from the executed instructions and the hardware states. This includes all memory accesses and the content of memory pages at all times.

Detail Emulation is commonly used to implement functional full system simulators. Such simulators can be used to thoroughly analyze entire computing systems for example to find programming errors or performance bottlenecks on a per-instruction detail-level.

In order to see additional, contextual information about the analyzed system's execution state, the simulator can make use of introspection techniques.

Binary translation can also be used to analyze applications in user space, where more contextual information is available for correlation. To this end, applications can be translated to collect data about their own execution. Well known examples are the dynamic recompilation based tools Valgrind [66] and Pin [55]. Another example is Electric Fence (eFence) [71], which can be statically added to a program in the linking stage. All three tools can be used to find memory leaks and memory usage errors (e.g., use after free).

Slowdown Full system *simulation* through interpretation is performed by simulators such as Bochs [51], which runs x86 binaries on different architectures.

Binary Translation can speed up the execution time by a factor of 30x compared to interpretation [8]. This technique is used by a variety of emulators such as QEMU [8] and full system simulators such as MARSSx86 [67].

The slowdown of emulators depends heavily on the additional bookkeeping and computation that is done at each interpretation step or in each translated *basic block* respectively:

- Full system simulators such as Simics can be many orders of magnitude slower than native execution when analyzing memory duplication properties. We have measured slowdowns of >7200x using Simics [69]. This results in a simulation time of 300 days for a workload of the length of a single hour.

- Quick emulators using binary translation and without instrumentation for measurements are still about 4-10x slower than native execution [8].

- We have found that instrumenting QEMU to trace memory accesses introduces another slowdown-factor of 4x over QEMU [38].

While working on analytical methods for memory accesses, we have realized that the largest speed-up potential for full system simulation lies in parallelizing the simulation of single cores. We have already published a paper [70] that outlines our ideas to realize this goal and quantifies the possible speed-up in a mathematical model. According to that model, near-native speeds are possible when simulating single cores on multi-core machines.

Distortion Emulation can be used to purely execute systems and it can also be used to analyze systems at different granularities. Instrumentation can range from the creation of a call-graph and functional analysis over the analysis of access patterns to caches and memory to microarchitectural analyses including precise timing predictions. The different granularities can be implemented using an appropriate combination of fast binary translation and slower, more detailed, interpretation.

Ease-of-use The ease-of-use varies greatly between the different, available frameworks. Tools such as Valgrind on the one hand, can easily and quickly be used to find pre-defined program errors. Full system simulators such as Simics on the other hand require deep knowledge of the analyzed systems and a lot of work to set-up a simulation with the required hooks and parameters. Contextual information that exceeds the hardware state and executed instructions needs to be gathered and correlated by hand, for example through introspection.

3.1.4. Trap and Emulate

Trap and Emulate is originally a technique that makes virtualization possible despite the lack of virtualization hardware support in the physical virtualization host [72].

CPUs only allow the execution of sensitive instructions, for example masking interrupts or changing address spaces, if the CPU is in a privileged mode (i.e., CPL0 on x86). For security reasons, only the host runs in this privileged mode and the virtualized hosts run with regular user privileges (i.e., CPL3 on x86). Guest OSes expect to run in privileged mode, however, as they are in charge of managing their (virtual) hardware. Calling a sensitive instruction without privileges, as a guest would perform it, traps into the host through a CPU exception similarly to a page-fault.

In case of a page-fault, the exception would be handled – the missing page mapped into memory – and then the original instruction would be restarted. When running sensitive instructions, the only possibility to successfully restart such a faulting operation would be to raise the privileges of the guest. As this is not intended, otherwise the isolation between guests and host could be compromised by the guest, the host needs to emulate the faulting instruction and advance the instruction pointer to return to the next instruction after the faulting one.

Analogous to page-faults, described in the previous section, Trap and Emulate can also be used for memory analysis. In contrast to the previous method, after trapping into the kernel due to a page-fault, the OS does not resolve the memory protection page-fault and restart the faulting operation. Instead, the host – which has a read-write mapping to the respective page frame – emulates the faulting operation and then increments the instruction pointer. This way, the guest skips the faulting instruction after emulation.

Completeness Trap and Emulate makes complete memory traces possible as the emulation renders remapping the accessed page unnecessary. This way, there is no time frame in which the page can be accessed unnoticed to the host.

Memory pages can also be mapped without access rights in the page table. Reading a page without access rights will then trap to the host which can emulate the read instruction. Thus, Trap and Emulate also works for tracing main memory read operations.

Relevance Trap and Emulate is inferior to emulation in combination with binary translation in any way. Using Trap and Emulate, the analysis is restricted to CPU-specific, fixed, predefined points in which analysis can take place. Binary translation delivers much higher performance. In addition it can modify the original binary code to trap into a more detailed emulation mode for critical parts of a program if necessary.

3.1.5. Custom Hardware

All previously introduced methods are techniques to analyze memory contents or access patterns in software. Specialized hardware can also record memory traces for analysis. Such hardware can be inserted along the bus between CPU and main memory in different places. All insertion places hide some details from the observer, however. When replacing the CPU, concurrent accesses to memory from DMA controllers and other CPUs are hidden. When sniffing the bus, cache accesses are missed.

The Hybrid Memory Trace Tool [3] replaces DDR3-RAM of a computing system and can record memory accesses with no slowdown. Context needs to be correlated similarly to the emulation based approaches. The benefit of such hardware is the increased recording speed compared to software methods.

3.1.6. Summary of Analytical Methods

When recording memory traces for memory (de-)duplication analysis several factors need to be taken into consideration. Different methods can be used to generate such traces; all of which have different characteristics (Table 3.1). All discussed methods need an introspection mechanism to correlate semantic with the trace.

Method	Complete	Distortion	Ease-of-use	Slowdown
VM Snapshots	no	based on frequency	easy	low–medium
Page-faults	no	high	harder	medium
Emulation	yes	depends	hard	high
Trap and Emulate	yes	high	harder	medium
Custom Hardware	no	low	hard	none

Table 3.1.: Overview and comparison of analytical methods.

We have mainly used *VM snapshots* and *Emulation* to evaluate our experiments. VM snapshots are an easy and quick way to analyze memory duplication quantities. Full system simulators can be used to gain more insight into the reasons behind duplication and the circumstances of their creation. Easy-to-use hardware was not available when we started this project.

3.2. The Anatomy of Memory Duplication

In order to instate a good deduplication policy, one first needs to understand the reasons for and properties of memory duplication. The following paragraphs provide an intuition where memory duplication comes from, in which quantities and distribution they are created, and how long they live.

First, I qualitatively describe why memory duplication exist and give reasons for highly varying duplication rates that can be measured in different workloads (§3.2.1). Second, I analyze spatial and quantitative characteristics such as the origin of sharing candidates and their cardinality (§3.2.2). The final paragraph describes temporal characteristics of duplicates: How long they remain sharable and the circumstances of their end (§3.2.3).

3.2.1. Reasons for Memory Duplication

Equal memory pages by pure chance are very improbable. A 4 KiB page can be in $2^{4096 \cdot 8}$ ($\approx 1.4 \cdot 10^{9864}$) different states, and even the birthday-paradox does not make content collisions likely for this many states when taking realistic memory sizes ($\ll 1$ TiB $\approx 1.1 \cdot 10^{12}$) into account. For a collision probability of 1% one would need to generate $1.6 \cdot 10^{4931}$ uniformly distributed random pages[3].

It is unlikely that we will ever reach memory sizes that make coincidental duplicates likely at uniform distribution for two reasons: First, we are unlikely to achieve a higher storage density than one bit per elementary particle[4]. Second, number of elementary particles in the observable universe is estimated to be 10^{97} [65].

Nevertheless, today's systems do contain duplicates. Their existence indicates that there are semantic correlations leading to duplicates. One goal of this thesis is to find policies, deduced from the creation semantics, to efficiently identify those duplicates.

Some sources for such semantics are:

Popular Pages and Initialization Memory page contents are not distributed uniformly across all possible states. Certain page values are recurring more often than others. Prime examples are pages filled with *patterns* such as 0's or 1's, but other initialization patterns are also likely[5]. Large arrays containing page-aligned objects (e.g., an OS's Process Control Block) may also be (pre-)initialized to recurring patterns.

[3] Calculated with the square approximation of the birthday problem.

[4] Examples of elementary particles are: quarks, electrons, photons, and neutrinos

[5] Some programmers initialize their data structures to values that the human eye can quickly identify in memory dumps, such as 0xDEADBEEF.

Direct Copies Within a program's address space, memory is often copied via `bcopy`, `memcpy`, and `memmove`. However, copying can also be implemented using arbitrary `mov` loops making the creation of copies hard to observe. An example for making a copy within a program would be the flyweight software design pattern [33].

Direct memory copies are, moreover, often made when transporting information between address spaces. A prime example is a program reading file contents. The OS first reads the requested file into a kernel-internal file cache and then merely copies the result into the target user address space. When files larger than the used page size are read and the target pages are aligned to that page size, this creates a duplicate page in memory.

Determinism Duplicates can also be the result of deterministic calculation results within programs. When two instances of this program are executed, this leads to duplicate memory contents between those processes.

An example would be an ELF-loader creating the same address space sections, initialized with the same data when `executing` a program. The *rodata* sections will for example likely contain the same constant strings in all processes of the same program. In contrast to the last paragraph, these memory areas are not copied in memory, but are autonomously created by each process.
Another example for this kind of duplication would be the creation of (the same) *lookup-tables* at the beginning of an algorithm.

A similar kind of duplication also occurs at the boundary between a hypervisor and a guest OS. Guest OSes ship with their own virtual disk image (VDI) containing all programs, shared libraries and data needed to boot the guest VM. When a hypervisor boots the *same VDI* multiple times, the same programs and libraries are loaded into the hosts' memory simultaneously.

This kind of duplication also occurs between programs that are *statically linked* against the same libraries [79], when using *zero-install*, and when running *appliances*. Starting any pair of those programs that were linked to the same libraries can lead to memory duplicates. An analogous problem exists when using *sandboxes*.

Different files with similar or equal content Files are loaded into the buffer cache based on their file identifier (e.g., inode), not based by their content.

Similar files are created when file contents evolve over time. Examples are shared libraries, databases and documents of different versions. Files with fully equal content can also exist in different parts of the file system. Examples are popular icons that are shipped with multiple software projects (e.g., freely available emoticons) or shared libraries that are bundled on distribution. It is for example common to bundle the QT-framework libraries with every QT-application that is distributed to the Windows OS.

3.2.2. Spatial and Quantitative Characteristics

This paragraph analyzes the spatial characteristics of memory duplication. That includes the quantity of duplication as well as their cardinality or *rank*, which is a metric for the distribution of shared and sharing pages. Spatial characteristics also include the classification into self-sharing and inter-domain sharing and an overview of the semantic origin of duplicate pages.

Duplication Quantity

The amount of sharable pages varies greatly between workloads. Previous studies have found redundancy on page granularity to make up 11% – 86% of the total amount of main memory allocated to VMs (Table 2.1). We have measured memory duplication in different workloads and scenarios as well. Two scenarios are introduced here. Duplication quantities of further scenarios can be found in the following chapters.

A **desktop workload**, in our case a PC with 2 GiB RAM with Linux[6] running LibreOffice[7] and Firefox[8] has a sharing potential of approximately 110 MiB, alone. The same OS running Gimp[9] and Eclipse[10] yields a sharing potential of approximately 93 MiB main memory (Figure 3.1).

Those measurements have been done using the Simics full system simulator. Note that the VMs did not use all of the available 2 GiB RAM when running the benchmark. When such a workload is run in a VM, the actual physical footprint is smaller than 2 GiB even without deduplication as only touched memory is allocated in the host. The actual amount of allocated memory varies greatly between workloads and also depends on the workload history. Disk I/O intensive workloads, that read (or write) many different files will use most of the available memory for disk caching. CPU intensive workloads on the other hand use hardly any memory at all.

A common application for VMs is to run multiple **batch-jobs** simultaneously. This way, the hardware utilization can be maximized while retaining isolation between jobs. On compile-servers for continuous integration [80], for example, the same code base might be compiled from different repository-copies (branches), in different VMs at the same time. Such scenarios yield a very high potential for memory deduplication on the one hand. Short run-times of single jobs make deduplication hard on the other hand.

[6] Ubuntu 11.10, 32 Bit, Kernel 3.3.2
[7] LibreOffice Writer 3.4.4 Build 402
[8] Firefox 12
[9] Gimp 2.6.11 displaying a 6.21 MiB picture
[10] Eclipse 3.7.0 Indigo

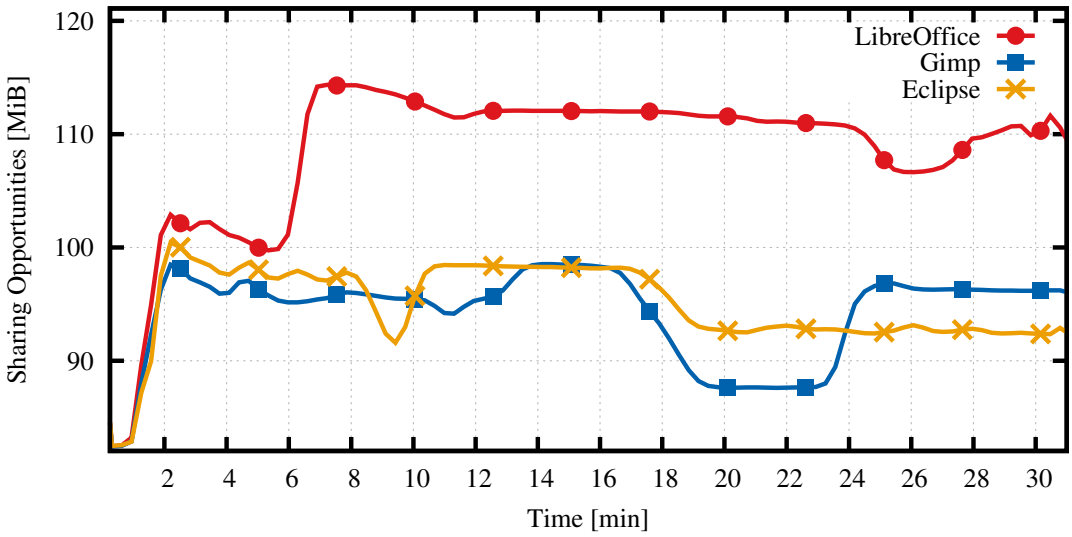

Figure 3.1.: Duplication in different Desktop environments. Each workload is running Ubuntu Linux as the OS. The three different workload were run independently of each other, they do *not* run simultaneously.

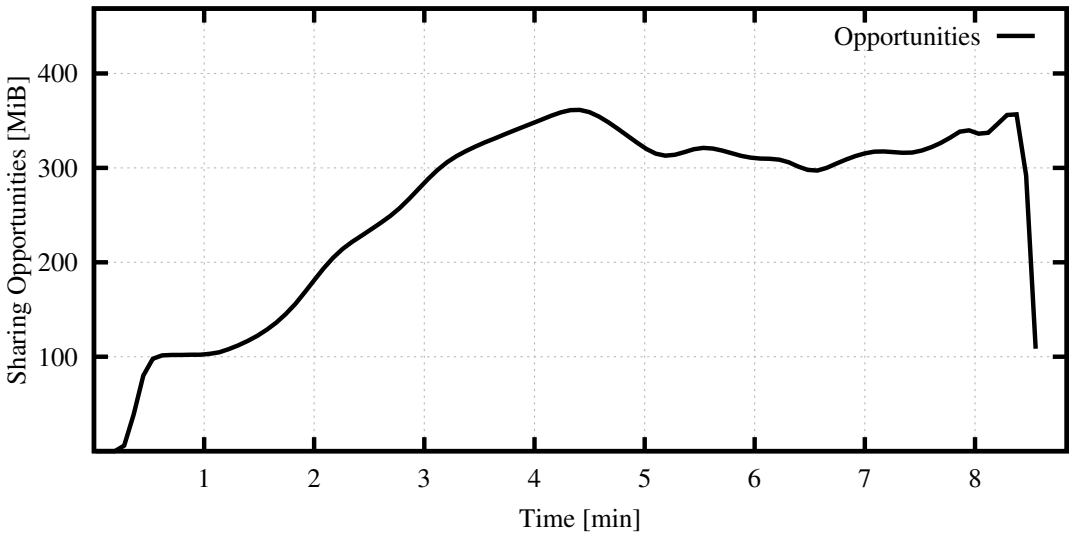

Figure 3.2.: Batch-job: Sharing potential of 2 VMs with 512 MiB RAM, each building the Linux kernel.

When running two Linux 3.0 kernel-builds (compiling the kernel) at the same time in VMs with 512 MiB each, more than a third of the total memory footprint for the VMs can potentially be saved using memory deduplication (Figure 3.2). We have gathered the number of sharing opportunities with secondly snapshots in this case. The run-time of the kernel-build is only 7:30 minutes without snapshots. We scale the time axis accordingly for the evaluation in Section 5.4.

Self-sharing vs. Inter-domain sharing

As we have seen, there are different circumstances in which main memory duplicates are created. When analyzing those sharing opportunities, we follow Barker et al.'s [5] notation of self-sharing and inter-VM sharing. As deduplication is not limited to virtual environments, we differentiate more broadly between *intra-domain sharing* or *self-sharing* on the one hand, and *inter-domain sharing* on the other hand:

- **Self-sharing:** Memory shared within a single semantic domain.
- **Inter-domain sharing:** Memory shared across different semantic domains.

This definition depends on the context that it is used in: When analyzing duplication in the address space of a process, sharing within a section (e.g., the heap) is considered self-sharing while sharing across sections (e.g., between heap and text segments) would be considered inter-domain sharing. When analyzing duplication in virtual machines, one would consider both of the above to be self-sharing if they occur within a single VM's memory partition. Equal memory pages that could be shared between VM instances would be considered inter-domain sharing in this case.

Figure 3.3 breaks down the total amount of sharing opportunities into self- and inter-domain sharing. The graph shows the distribution for three desktop workloads: LibreOffice, Gimp, and Eclipse each running in their own Ubuntu VM respectively. In this benchmark, inter-domain sharing dominates. This is not the case for all workloads, however. Barker et al. [5] have measured the contrary in their desktop workloads at the very high snapshot interval of 30 minutes per snapshot.

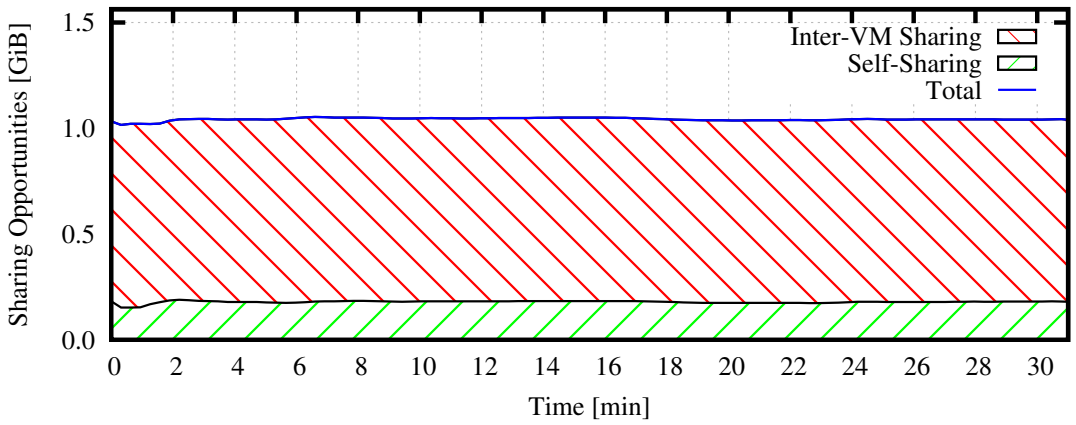

Figure 3.3.: Self-sharing vs. Intra-domain sharing: The previous three desktop workloads LibreOffice, Gimp, and Eclipse respectively run in three virtual machines with 2 GiB RAM. This scenario resembles a VM-based terminal server.

Sharing Rank

The difference between self-sharing and inter-domain sharing is directly reflected in the sharing rank. When merging equal memory pages, a single, shared COW page frame remains in the system to be referenced by all other pages.

The **rank** is the number of equal pages that can be merged [63]. For example, if two pages have the same content in the system, their sharing rank is two. One of those pages can be freed and **shared**, the other page remains in the system to be **sharing** (Figure 3.4). When more pages are equal, the sharing rank is higher than two. For a sharing rank of n, $n - 1$ pages can be shared, while one page remains, sharing the content using COW.

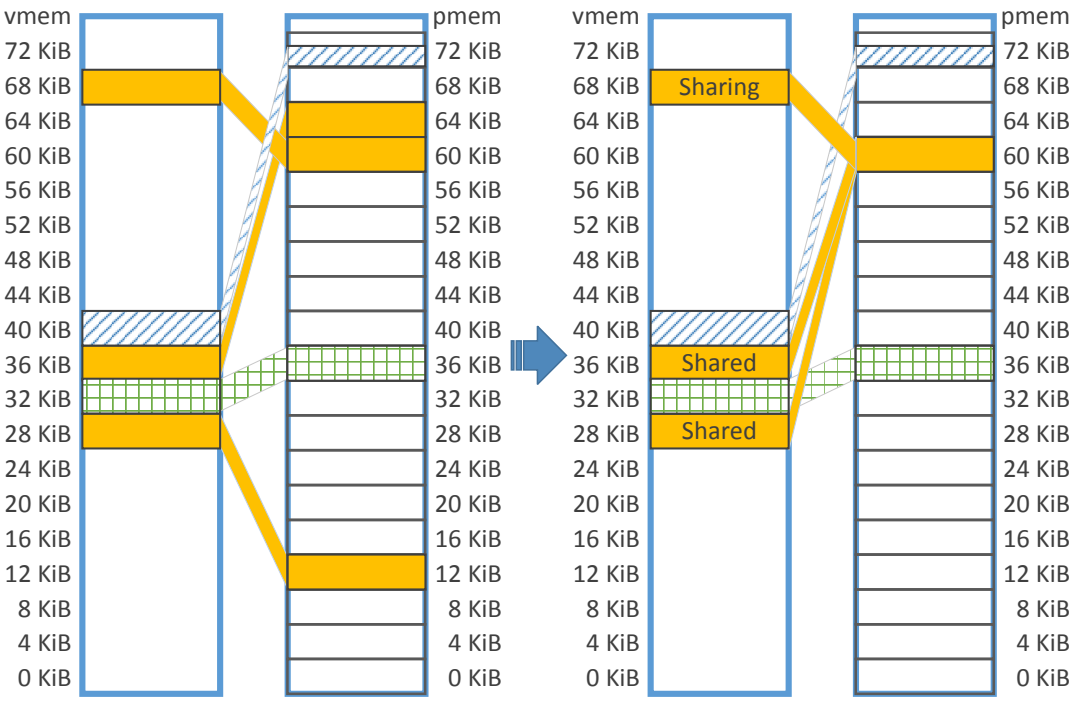

Figure 3.4.: Three equal pages are merged into a single sharing page and two shared pages. The two shared pages can be freed, the sharing page needs to remain allocated. The sharing rank is three.

A higher rank on the one hand means, that more memory can be freed and thus saved with the same total of equal pages. For example, if one page is sharing and 99 pages share this page (rank of 100), 99 pages can be freed. If 50 pages are sharing and another 50 pages share those pages (rank of two), only 50 pages can be freed. In both cases there is a total amount of 100 equal pages.

Self-sharing and inter-domain sharing scale differently with a growing number of sharing domains: The rank of (purely) self-sharing pages is hardly affected by the number of sharing domains. When starting another workload, additional sharing opportunities can generally only be found within the new domain. Inter-domain sharing in contrast generally scales proportionally to the number of sharing domains. Adding another (similar) domain, for example another Ubuntu Linux VM, increases the chance that another page is added to an existing inter-domain sharing group, increasing its rank. This is especially true, if a cloud provider consolidates VMs with similar memory footprints.

The sharing rank depends heavily on the workload and on the distribution of self-sharing and inter-domain sharing. Miłós et al. [63] have evaluated their approach using Linux kernel-builds, HTTPerf, and RUBiS. They have also measured the sharing rank in an aggregate of those benchmark scenarios. Almost 80% of the sharing opportunities had a rank of two. Rank three and four were assessed to make up 7% and 3% respectively. Another 6.5% of the sharing opportunities had a sharing rank of 14 or higher, while the remaining sharing ranks of 5 to 13 only contained between 0.5% and 1.3% of the sharing opportunities.

In our aforementioned desktop benchmarks, when running LibreOffice, Gimp, and Eclipse in three separate VMs simultaneously, we have measured the following sharing ranks, which are also summarized in Table 3.2: 15% had a rank of two, 83% had a rank of three. Ranks four to six made up only 0.1% in total, while 1.4% of the sharing opportunities had a rank of six or higher. Table 3.2 also breaks those results further down into inter-domain sharing and self-sharing. The inter-domain sharing rank clearly peaks at three due to the number of VMs that were run simultaneously in this benchmarks, which was also three. The self-sharing rank peaks between two and three pages and then at ranks of higher then six. Either pairs or triplets of pages are found within a sharing domain, or larger contiguous memory regions are initialized to a certain pattern.

Rank	Self-Sharing	Inter-domain Sharing	Total
2	55.2%	6.5%	15.4%
3	39.0%	93.5%	83.0%
4	0.2%	0.0%	0.1%
5	0.1%	0.0%	0.0%
6	0.1%	0.0%	0.0%
>6	5.4%	0.0%	1.4%

Table 3.2.: Sharing-rank distribution of 3 Ubuntu Linux VMs running LibreOffice, Eclipse, and Gimp respectively [69].

Semantic Origin of Sharing Candidates

When grouping sharing opportunities by their semantic origin, it is nontrivial how to attribute mergeable pages from different semantic origins. Two approaches are plausible: First, grouping can be done based on the number of pages that enter a sharing group. Second, grouping can be done based on the highest total number of cycles that pages of a certain origin remain in the sharing group.

Figure 3.5 illustrates this issue. If only the number of pages that enter the sharing group is counted, then pages that fluctuate between sharable and not sharable contents will distort which semantic origin this page frame will be attribute to. In the example, the heap pages that enter the sharing group five times makes the heap the prominent page type on that group. If the total share time is considered as the basis for a page type attribution of a page frame then the sharing group in the example is attributed to the page cache.

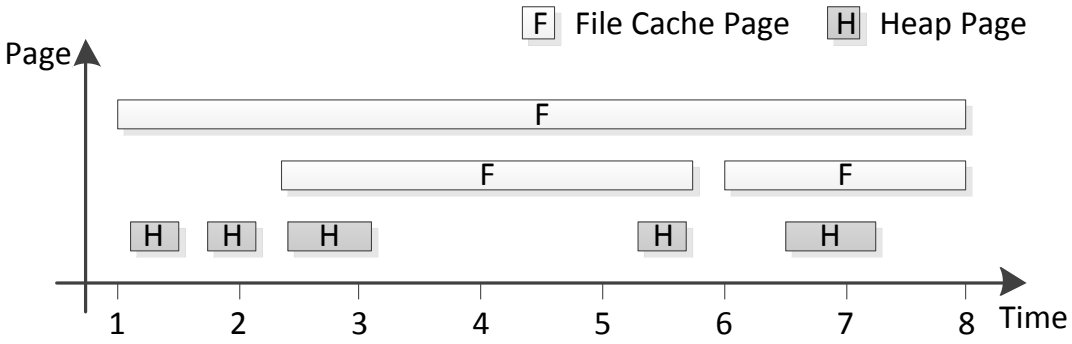

Figure 3.5.: Sharing groups can be attributed to the page type that enters the most into the group. It can alternatively be attributed to the page type that remains in the group the longest. [69]

We have decided that the second approach paints a better picture of the shared page usage. What counts after all is how much the sharing groups are used, not how many pages enter the group. Highly fluctuating pages would otherwise distort the distribution of sharing groups.

We have introspected Simics simulations to gather information about the semantic sources of the sharing opportunities [69]. Due to slowdowns of >7200x we were only able to analyze workloads with little computation and low memory pressure. The following results are aggregated numbers taken from introspecting the three previously presented desktop workloads, running Eclipse, LibreOffice, and Gimp atop of Ubuntu Linux.

Figure 3.6 visualizes our findings. Our experiments are limited to the terminal server and desktop domain only. The result, that file and heap memory make up the majority of the sharing opportunities, can also be deduced from measurements published by Miłós et al. [63]. Barker et al. [5] have found a majority of the sharing opportunities to come from heap memory (50%) followed closely by file-based memory (43%).

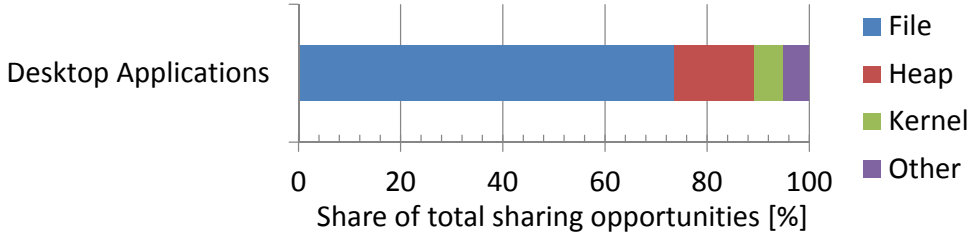

Figure 3.6.: Aggregated origins of sharing opportunities of 3 Ubuntu Linux VMs running LibreOffice, Eclipse, and Gimp respectively. [69]

3.2.3. Temporal Characteristics

This section describes temporal characteristics of duplicate memory pages. It highlights how long sharing opportunities live and when they are usually detected by a memory scanner. Moreover, it details on the time when sharing is broken.

Longevity

The time that pages remain equal is the **longevity** of sharing opportunities. Its distribution depends heavily on the workload and the semantic origin of the sharing opportunities that can be found.

Sharing opportunities that live for a long time, can easily be identified by brute force using memory scanners (see Section 2.4.4). Miłós et al. [63] have found that VMware's ESX server can find about half of the available sharing opportunities after 37 minutes using default settings. Even when using aggressive settings in the ESX memory scanner, Miłós et al. have measured that it takes up to 20 minutes to find half of the available memory duplicates in their benchmarks.

When running two Linux kernel-builds each one in a separate VM with 512 MiB RAM the sharing opportunity longevities that are illustrated in Figure 3.7 can be measured. When assuming a duplicate detection time of 30 seconds a large portion of the available sharing opportunities in this benchmark will not be detected by a regular memory scanner.

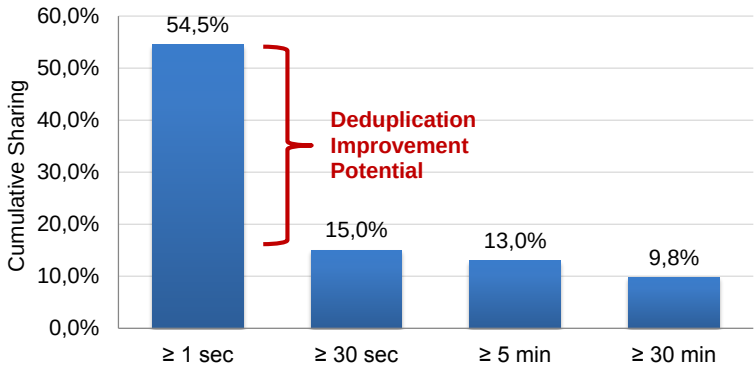

Figure 3.7.: Longevity of sharing opportunities when running two Linux kernel-builds in separate VMs.

Disappearance of Sharing Opportunities

Sharing is only broken when the content of merged pages is modified. Note that changing the status of a page, for example marking a page as free after a process ended, does not break sharing, yet. In Linux, pages that are freed are not zeroed until their next allocation. When no memory pressure is present, this prolongs the time that pages exiting the page cache can be shared. The consequence is, that memory pages can remain shared long after the respective pages have been freed.

Figure 3.8 illustrates the number of sharing opportunities that exist during the Bonnie++ file system benchmark and thereafter, when pages get freed by the OS. Bonnie++ deletes the temporary files it has previously created to measure the file system performance around 150 seconds after the beginning of the benchmark. The memory pages that have previously cached those file contents remain unchanged until they are allocated to another process.

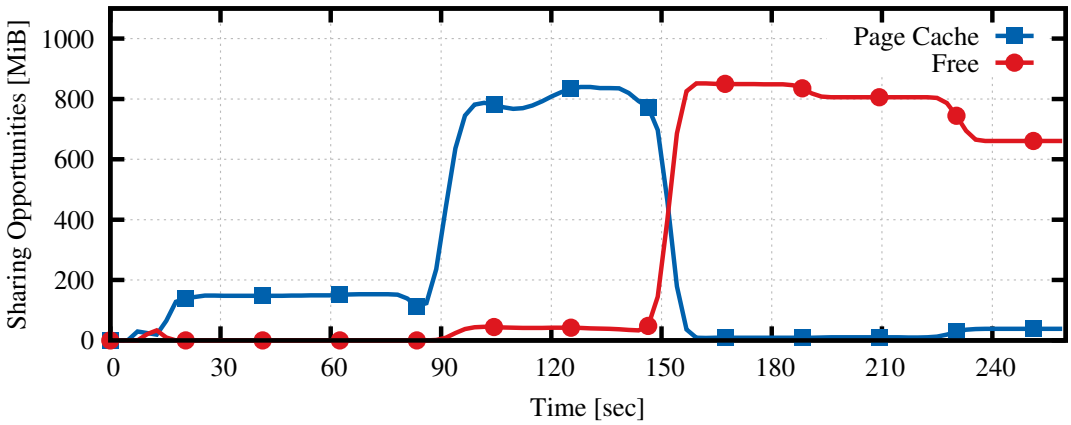

Figure 3.8.: Pages remain shared even after their memory state changes to *free*. In Linux, the content doesn't change from the last allocated content, as long as the state remains free.

3.3. Conclusion

We have introduced different techniques to gather information about memory duplication. In order to gain detailed insight into the semantics of sharing opportunities one always needs to go through the trouble to implement some form of introspection, however. We have used VM snapshots to gain insight into the duplication quantity of different workloads. Then we turned to emulation, by instrumenting the Simics full system simulator and QEMU with introspection methods, to get a deeper insight into the spatial and temporal characteristics of memory duplication.

Memory duplication has two main semantic sources: heaps and the page cache that is used to cache files in the OS. Many duplicates cannot be caught by today's memory scanners due to their short longevity. Those duplicates still live for minutes, so they could be shared with little performance impact, if they were identified more timely.

Chapter 4

Cross Layer Integration through Deduplication Hints

This chapter introduces the main memory deduplication scanner extension Cross Layer deduplication Hints (XLH), the main contribution of this thesis. XLH focuses the deduplication effort on memory parts with a higher prospect to yielding good sharing candidates. The idea is that hints, for what those areas are, come directly from the subsystems that can decide which pages are likely to contain duplicates. In the case of virtualized environments, hints can for example be generated by the host for recently modified memory areas that belong to the I/O-caches of virtual machine guests.

Our XLH prototype is based on Linux' KSM memory scanner. Consequently, Section 4.1 introduces the virtual memory management implementation in Linux before the following Section 4.2 reviews the KSM memory scanner in detail. Section 4.3 discusses different aspects that have been considered when designing XLH and Section 4.4 highlights some details of its implementation.

4.1. Linux Virtual Memory Implementation

In this section I introduce the key data structures and mechanisms of the Linux 3.4 virtual memory management (VMM) implementation. §4.1.1 describes how Linux handles basic internal memory allocation. §4.1.2 describes how the allocation of the address space layout of applications is represented. §4.1.3 describes the implementation of the Linux page cache, which is responsible for caching file system operations. Finally, §4.1.4 describes the mechanisms that use the previously described data structures.

This section does not intend to give a comprehensive introduction into Linux memory management. Instead, I focus on the parts needed to understand XLH, our approach described later in this chapter. The full Linux source-code is available to the interested reader at `www.kernel.org`.

4.1.1. Basic Linux Internal Memory Management

The Linux kernel represents every physical page frame of the system with a `page` data structure [54]. All `page` structures are stored in a large array, the `mem_map` (Figure 4.1).

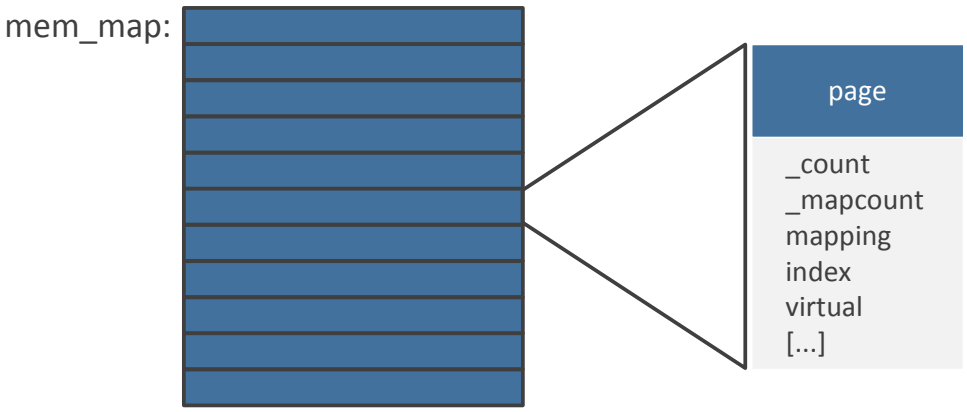

Figure 4.1.: The `page` structure representing a page frame.

The page structure stores the number of (virtual) pages referencing the page frame (`_count`). If `_count` is zero, the page frame is free. The `_mapcount` variable stores how many PTEs this page frame is referenced from[1].

Moreover, the page structure contains a `mapping` pointer. The `mapping` has a context-sensitive semantic. If the page frame holds file backed memory, `mapping` points to the data structure describing the file's mapping in the file cache (`address_space`) and `index` describes the offset needed to read and write the respective part of the file represented by this page frame. If, however, the page frame holds anonymous memory, `mapping` points to a linked list containing all memory areas (VMAs) that reference this frame (`anon_vma`).

The state (free/allocated) of physical pages are managed by a buddy allocator [58]. Often, smaller sized objects than pages are needed to store data structures. Only handing out memory on page granularity would lead to significant internal fragmentation. In consequence, Linux stacks a second memory allocator on top of the buddy allocator to dynamically allocate smaller portions of memory: the slab allocator [11].

[1] Recall, that PTEs can be shared.

4.1.2. Linux Address Spaces

The *address space* (AS) of each *task* (process or thread) is described in a *memory descriptor* (MM). This descriptor is organized as a collection of *memory areas* (VMAs), each representing a segment in the virtual address space of the task. Figure 4.2 visualizes the interplay between tasks, MMs and VMAs.

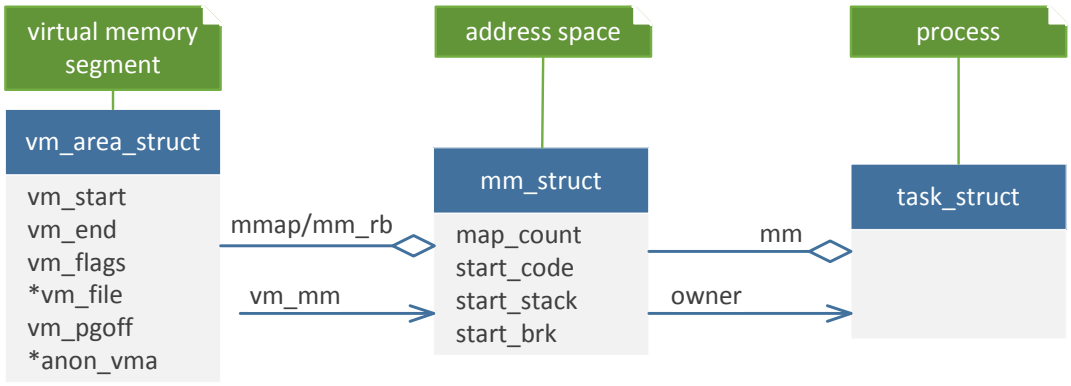

Figure 4.2.: Interplay of Linux virtual memory management data structures.

Virtual Memory Area (VMA) Every process's address space is made up of legal[2] and illegal address ranges. All legal address ranges are contiguous segments in the virtual address space and are described by a Virtual Memory Area (VMA) represented by a `vm_area_struct` structure.

All memory areas are assigned flags (`vm_flags`) by the OS. These flags describe how the memory shall be used. They also determine how the OS sets the respective PTE flags when mapped. Most important for this thesis is the `MAP_SHARED` flag, which indicates a memory area that is shared between tasks. The opposite of a shared mapping is the private mapping (`MAP_PRIVATE`). The read-only flag (`MAP_DENYWRITE`) is needed to implement copy-on-write.

VMAs can carry data of different origins, particularly the distinction between anonymous (`MAP_ANONYMOUS`) and named pages is done here (§2.2.5). Examples of anonymous VMAs are the stack segment which is created and modified at runtime, the data segment which is loaded from a binary but can later be written without modifying the binary, the heap segment, and shared pages mapped for example via `shmem`. Examples for named memory regions are the text segment which is loaded from background store containing program binary code and all pages from the page cache that are mapped into the address space via `mmap`. Not all variables in the VMA struct are always valid. For named segments, two member variables describe the corresponding file (`vm_file`) and offset (`vm_pgoff`). For anonymous files, `anon_vma` is set to signal, that this memory is not backed by a file.

[2] Those memory areas do not have to be mapped at all times. They can be invalid (i.e., currently not in physical memory) despite being legal (part of the virtual address space).

Memory Management (MM) The address space, which is the virtual memory layout of a process, is represented by a memory descriptor (`mm_struct`). Most importantly the memory descriptor contains a linked list of all contained VMAs. Dedicated pointers exist for the most important VMAs. Examples are pointers that point to the code (`start_code`), stack (`start_stack`), and the heap (`start_brk`) segments.

The list of VMAs is also referenced by a red-black tree (`mm_rb`) for easy lookup of addresses in addition to the list, which is well suited to efficiently traverse the VMAs in order. Note that these are merely two different indices for the same data nodes.

The VMAs that are included in the MM of a task can easily be inspected from user space via the `/proc` file system. The memory mappings of the own process can be listed through `/proc/self/maps` (Figure 4.3). Each line starts with the virtual address range of the VMA, followed by the access permissions in rwx (read, write execution) notation, if applicable the offset in the file, major and minor number of the device, the inode number and the file path and file name [18]. Not all mappings are based on files. Anonymous regions do not have device, inode or file path and name fields. More insight about the actual memory usage characteristics of each segment is stated in `/proc/self/smaps`. Figure 4.4 shows the output for the first segment of the previous example.

```
11:38:26 ~ cat /proc/self/maps
00400000-0040b000 r-xp 00000000 08:01 524484           /bin/cat
0060a000-0060b000 r--p 0000a000 08:01 524484           /bin/cat
0060b000-0060d000 rw-p 0000b000 08:01 524484           /bin/cat
00fd4000-00ff5000 rw-p 00000000 00:00 0                [heap]
32db200000-32db223000 r-xp 00000000 08:01 1179965      /lib/x86_64-linux-gnu/ld-2.17.so
32db422000-32db423000 r--p 00022000 08:01 1179965      /lib/x86_64-linux-gnu/ld-2.17.so
32db423000-32db425000 rw-p 00023000 08:01 1179965      /lib/x86_64-linux-gnu/ld-2.17.so
32db600000-32db7bd000 r-xp 00000000 08:01 1180132      /lib/x86_64-linux-gnu/libc-2.17.so
32db7bd000-32db9bd000 ---p 001bd000 08:01 1180132      /lib/x86_64-linux-gnu/libc-2.17.so
32db9bd000-32db9c1000 r--p 001bd000 08:01 1180132      /lib/x86_64-linux-gnu/libc-2.17.so
32db9c1000-32db9c3000 rw-p 001c1000 08:01 1180132      /lib/x86_64-linux-gnu/libc-2.17.so
32db9c3000-32db9c8000 rw-p 00000000 00:00 0
7f1d9361a000-7f1d938e3000 r--p 00000000 08:01 656788   /usr/lib/locale/locale-archive
7f1d938e3000-7f1d938e6000 rw-p 00000000 00:00 0
7f1d93905000-7f1d93907000 rw-p 00000000 00:00 0
7fff7bf12000-7fff7bf33000 rw-p 00000000 00:00 0        [stack]
7fff7bffe000-7fff7c000000 r-xp 00000000 00:00 0        [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

Figure 4.3.: The VMAs of a `cat` task.

```
00400000-0040b000 r-xp 00000000 08:01 524484
Size:                 44 kB
Rss:                  28 kB
Pss:                  28 kB
Private_Clean:        28 kB
Private_Dirty:         0 kB
Referenced:           28 kB
Anonymous:             0 kB
AnonHugePages:         0 kB
Swap:                  0 kB
KernelPageSize:        4 kB
MMUPageSize:           4 kB
Locked:                0 kB
VmFlags: rd ex mr mw me dw
```

Figure 4.4.: Details about the memory consumption of the `cat` text segment. Shortened `/proc/<pid>/smaps` output.

4.1.3. Linux Page Cache

The Linux page cache speeds up accesses to block devices by sacrificing main memory for caching previously accessed contents. When reading from disk, for example, the `read` code path first checks if the desired data blocks are already available in the cache and only accesses the background store if they are not [54].

As previously noted, files can be directly mapped into a process address space as a new virtual memory area using `mmap`. This mapping can comprise any contiguous part of the file, including the entire file. Linux maps such pages directly to the buffer cache.

This way, direct, uninterrupted access is granted to applications if the file block has previously been loaded and is still in memory. Otherwise, the MMU issues a page-fault, as the requested page is invalid; the OS then loads the requested file-parts into the page cache and makes the new page available to the application by updating its page table.

Although 4 KiB disk blocks are becoming increasingly popular, the typical block size for HDDs is still 512 bytes. Blocks in a page of the page cache are ordered by the logical position they have in the file. This position does not necessarily correspond to the block's position on the background store, which is up to the file system.

Address Space Object Contrary to the name, the address space object does not describe the virtual address space of a process. Instead, the `address_space` object is the entity by which Linux caches pages. It abstracts the origin of the cached data by spanning an address space of offsets within a file and describing how these map to the containing backing device. This way, the page cache can be used to cache any data on page granularity instead of specializing on block devices [58]. The page cache can, for example, handle regular files, memory mapped files, block devices, and shared memory regions that are not backed by a regular file but by swap space.

Among other information, the `address_space` contains a pointer to the source device (`backing_dev_info`), inode (`host`), and offset (`writeback_index`). It moreover contains the number of writable references to this cache entry (`i_mmap_writable`), the number of read-only references (`i_mmap`), and the number of pages in this entry (`nrpages`). In addition, the `address_space` contains pointers to methods that are called on various events that happen, when handling cached data (`a_ops`). One example is a cache miss handler that the OS can call when it wants to write back a cached page.

For any page cache, it is crucial that the cached pages are found quickly when they are accessed. In Linux, a radix-tree is used as an index to hold all allocated address space objects. Additionally, the address space object can be accessed from the VMA, via the `vm_file` pointer. Figure 4.5 depicts the relationship between the data structures that hold the information needed to implement the Linux page cache.

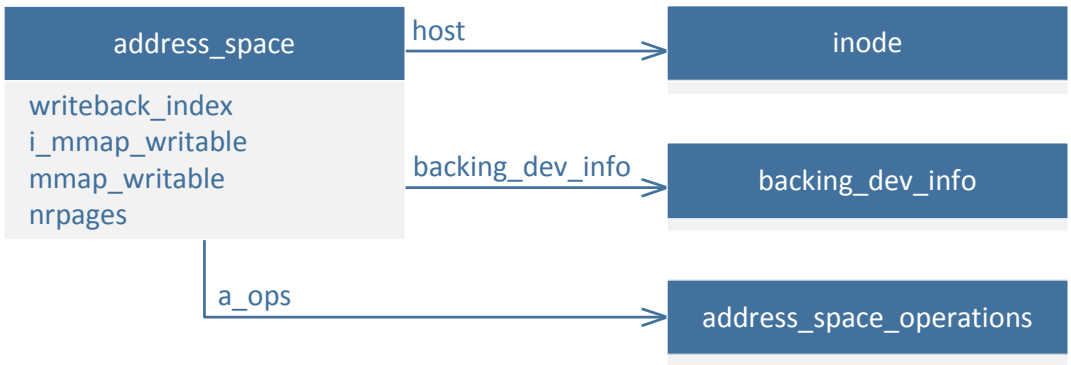

Figure 4.5.: The data structures used to implement the Linux page cache.

4.1.4. Page-Faults

The allocation and initialization of page tables is separated from the population of pages with data in modern operating systems. In Linux, virtual pages are not mapped to physical memory page frames until they are first used (demand paging).

The MMU triggers a page-fault if it cannot resolve a memory access by itself due to lack of information, or if the memory access is illegal according to the current mapping flags. An example for the former would be the access to a page that has previously not been referenced yet. Another example is the access to a swapped, and hence invalid page that has to be brought into memory before accessing it. An example for the latter case is a write operation to a page whose PTE protection bits are currently set to read-only. The page-fault invokes the OS to resolve such issues by taking the appropriate actions.

On legal accesses, page-faults associate pages with a page frame, potentially after evicting another page [58]. Before marking the page valid in the page table and returning to the faulting process, the OS initializes the new page frame with the data that is associated with the respective memory region.

Figure 4.6 is loosely based on Figure 4.17 in Mauerer's Book "Professional Linux Kernel Architecture" [58] and depicts reasons for page-faults and how page-faults can be resolved. A segmentation fault is either handled by the OS by calling the appropriate signal handler to let the application resolve the problem itself, if available. If no such signal handler is available, the OS forces the faulting process to quit.

4.2. Implementation of Kernel Samepage Merging

The design of Kernel Samepage Merging (KSM) has first been presented by Arcangeli et al. at the Linux Symposium 2009 [2]. I have given a high-level description of KSM in § 2.4.4.

KSM was first added into mainline Linux kernel 2.6.32 in December 2009 [42]. Since then, the code has undergone several revisions. A brief description of the key data structures, mechanisms and policies that implement KSM in Linux 3.4 follows. This is the version that we have extended to build our XLH prototype described throughout the rest of this chapter. Figure 4.7 gives an overview of the interacting subsystems.

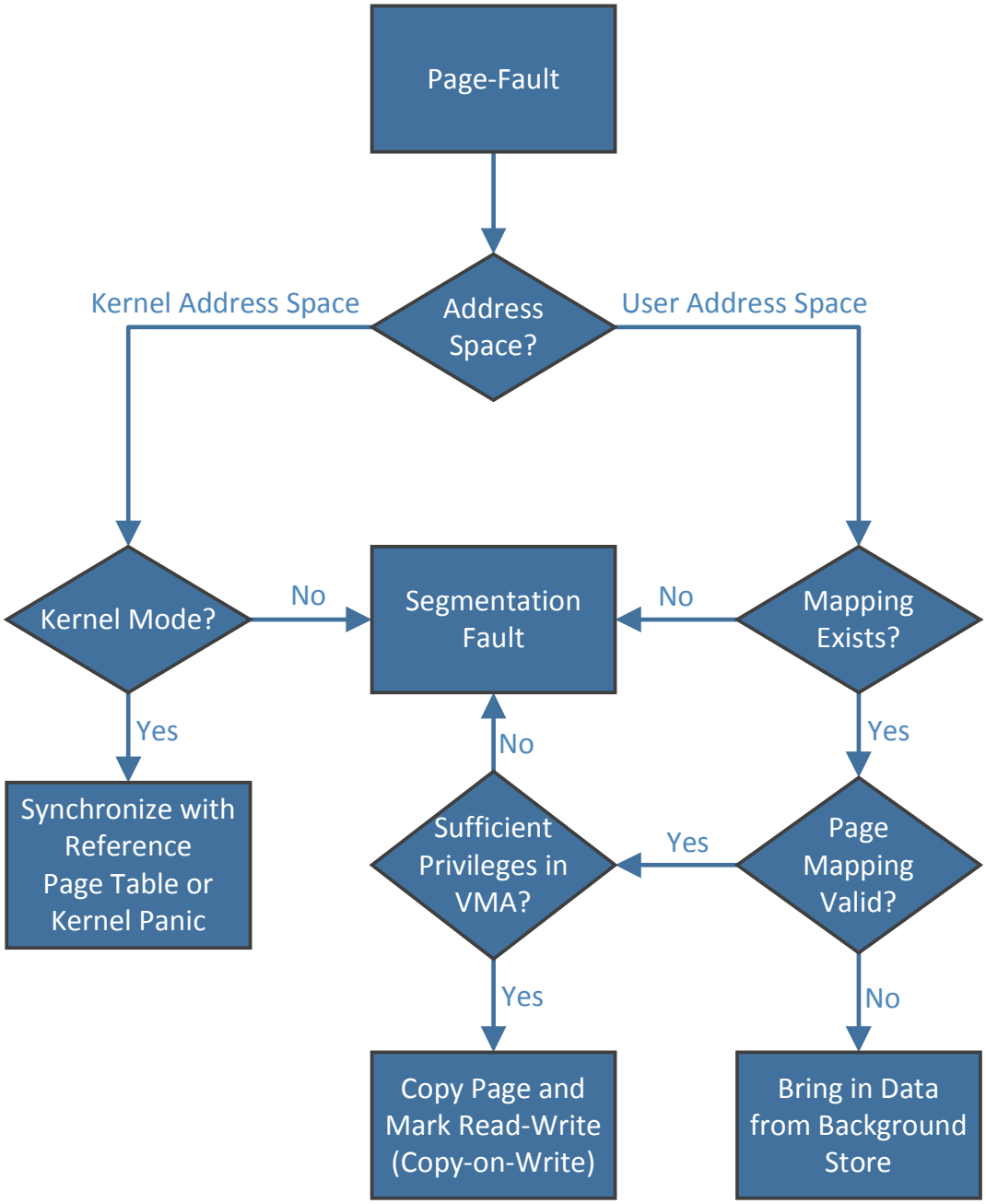

Figure 4.6.: Flowchart of page-fault reasons and how to resolve them [58].

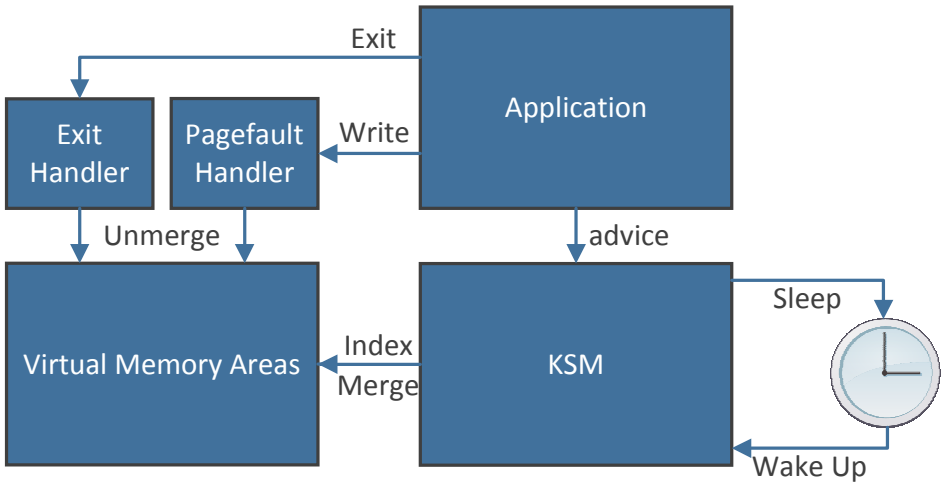

Figure 4.7.: The interplay of the application, KSM, the page-fault-, and exit-handler.

4.2.1. KSM Data Structures

The data structures storing information about the KSM scan process and index are described in the following. If no other reference is given, the provided information is derived directly from the Linux 3.4 source code.

Reverse Mapping Item KSM keeps a reverse mapping item (`rmap_item`) for every virtual page that can be deduplicated. It has its name due to its function to translate physical page frames (`kpfn`) for merged pages to the virtual `addresses` that use the page frame in a copy-on-write fashion. The `rmap_item` is also the common data node storing the state of each page in the deduplication scan.

The `rmap_item` comprises information such as:

- A sequence number (low bits of `address`) that indicates how often this page has been visited in the past.
- A `jhash2` checksum (`oldchecksum`) of the content that the page had when the scanner last visited that page.
- The state (also encoded in the low order bits of `address`) and consequently position of the item in the following data structures (`mm_slot`, unstable tree, stable tree).
- A pointer to the next item of its address space (`mm`).

All major data structures in KSM such as the `mm_slot`, the unstable tree, and the stable tree, which are described in the next paragraphs, use `rmap_items` intrusively. This means, that they merely hold pointers to the same nodes, they do not allocate or delete `rmap_items`[3].

mm_slot The `mm_slot` contains a sorted list of all address regions that are subject to deduplication, grouped by their containing `mm`. Applications can add new regions from their own address space to `mm_slot` using the `madvise` system call with the `MADV_MERGEABLE` flag. If there is already a mergeable region in the same MM, the new memory area is added to that `mm`'s `rmap_list` by creating a new `rmap_item` for every page in the area and inserting it to the list.

KSM Cursor A single instance of the KSM cursor (`ksm_cursor`) structure exists per system. It stores information about the current progress of the linear scan through all `rmap_items` in all `mm_slots`. To this end, it contains pointers to the `mm_slot` and `rmap_item` that currently processes and the next address to be scanned (`address`).

Unstable Tree The *unstable red-black tree* keeps `rmap_items` of pages that have neither recently been modified nor been merged yet (Figure 4.8).

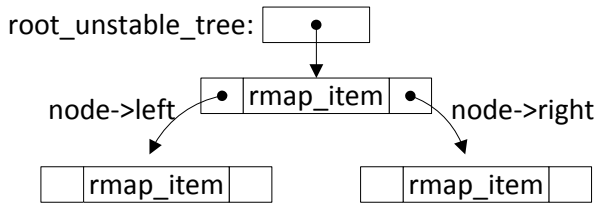

Figure 4.8.: `rmap_items` contain node pointers to form the unstable tree. If a certain `rmap_item` is part of the unstable tree is indicated by the `UNSTABLE_FLAG` in the low order bit of the `address` field.

The index into the tree is the full content of the inserted pages. To find a page with a certain content A, the first bytes of the root-node need to be compared to the first bytes of A until the bytes differ. If the different bytes in A are smaller than the bytes in the root node, KSM follows the tree to the left, if the bytes in B are greater, KSM follows to the right and repeats the process. If all bytes match KSM has found the `rmap_item` of another page with same content.

Pages whose references are linked into the unstable tree are not protected from being written by their respective processes. The content that was used as an index to insert the page can consequently change over time, distorting the tree. We have dubbed this property the "degeneration of the unstable tree" [2], which was adopted by following publications [76].

[3] Unintrusive containers copy data nodes on insert and free them after removal.

Entire tree branches can become stale due to write operations on intermediate nodes. For example, all branches to the left of the root node in Figure 4.9 become unreachable when the content is changed to something starting with the bytes “BA”. The entire index loses pages with a contents between pages starting with “BA” (the previous content) and pages starting with “EA”.

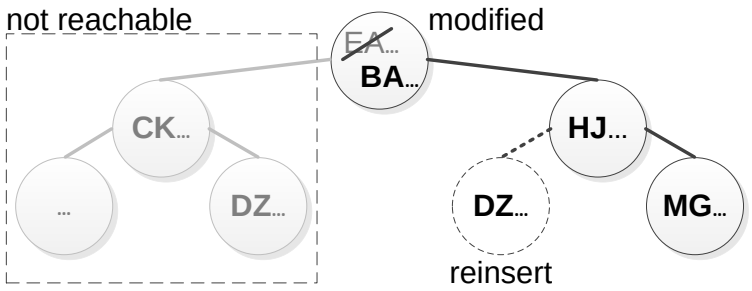

Figure 4.9.: Pages referenced by the unstable tree are not write-protected. The tree can thereby degenerate. [62]

The unstable tree is discarded after each scan round to clean up the degeneration. Dropping the unstable tree is just a matter of resetting the root node pointer (`root_unstable_tree`) pointer to `NULL` since no reverse map items are allocated or freed on tree operations. All `rmap_items` stay referenced through their respective `mm_slot` and can in consequence always be found by the scanner.

Stable Tree The *stable red-black-tree* references `rmap_items` that have previously been merged and are marked with the `STABLE_FLAG` in the `address` field.

Merged pages are set to copy-on-write and are consequently write-protected. Pages in the stable tree do either still have the content that they had when they were inserted into the tree or have been unmapped. Unmapping happens for example when copying the referenced page after a write. This state change can easily be detected at scan-time; **stale** items in the stable tree are removed when they are passed on lookup operations.

The garbage collection at lookup time is intentional. If the copy-on-write page-fault handler had to remove the written pages from the stable tree, concurrent access to the tree would need to be serialized. The single threaded scan process can omit locking.

Another difference between the stable- and the unstable trees is that in the unstable tree, it is an exception to find multiple items that point to the same page contents.

In the unstable tree this can only happen when:

- Page A is inserted into the tree
- Page B on the path to A is modified and blocks the path to A
- Page $\bar{A}$ is added with the same content as A
- Page B is restored, restoring the path to A

In the stable tree, however, the normal case is that multiple items share the same content; pages end up in the stable tree due to this fact in the first place. Stable nodes reference a single `rmap_item` when sharing with this item was broken, only.

Due to the different usage scenario two additional fields are used to reference the `rmap_items` while they are part of the stable tree. A linked list of `rmap_items` that reference the virtual pages that are currently merged to the same page frame is added (`hlist`) as well as the page frame number that they reference `kpfn` (Figure 4.10).

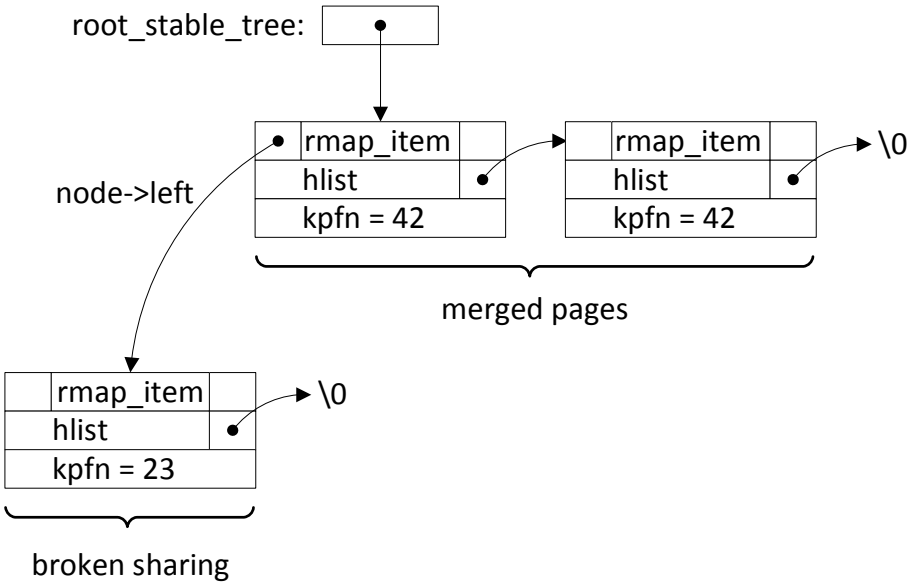

Figure 4.10.: Each `rmap_item` contains left and right pointers that make up the stable tree. In addition, each `rmap_item` contains a linked list holding references to the other items that reference the same merged page frame `kpfn`.

4.2.2. KSM Mechanisms and Policies

After describing the data structures that were added to the Linux virtual memory layer, the following paragraphs describe how these data structures are used to implement memory deduplication scanning summarized in Figure 4.11.

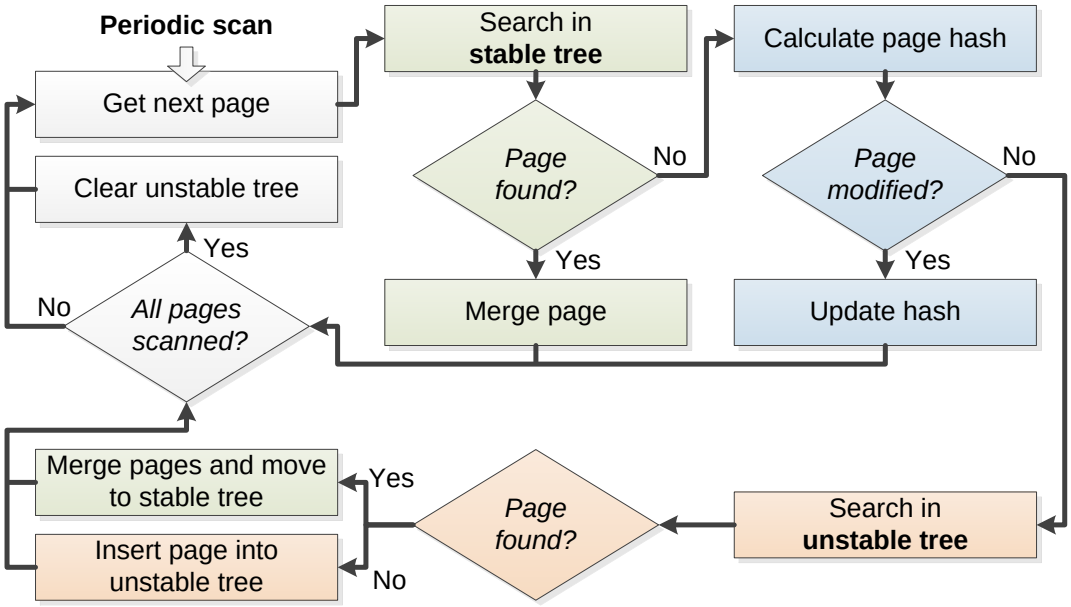

Figure 4.11.: High level overview of the KSM memory scanning process [62].

Selecting Subjects for Deduplication To omit wasting CPU cycles and memory bandwidth by scanning memory that is unlikely to contain duplicates, KSM only indexes pages that processes specifically select for being subject to deduplication.

Programs inform the kernel, that they want a certain address range to be deduplicated, using the `madvise` system call with the `MADV_MERGEABLE` flag. The OS forwards memory regions that are advised to be mergeable to KSM. The scanner then inserts the VMA and issued address range to the `mm_slot` data structure right before the current scan cursor. This way, the advised memory area has time to settle down before the scan thread visits it for the first time. In its current implementation, KSM only allows setting `MADV_MERGEABLE` on pages with 4 KiB page-size.

An address range can later be excluded from the deduplication scanner by issuing the `madvise` call with the `MADV_UNMERGEABLE` flag. Then, sharing for all pages in that address range is broken and the address range is removed from the `mm_slot`.

KSM does not increase the reference counter for pages in the `mm_slot`. This implies, that pages scanned by KSM can disappear at any time. On the plus side, KSM does not need to lock the respective pages when scanning, instead KSM uses a mechanism derived from `page_cache_get_speculative()` to access pages without referencing them.

Wake-ups and Pages to Scan The OS wakes up the KSM daemon periodically every `ksm_thread_sleep_millisecs` ms. The sleep time can be configured from user space via `/sys/kernel/mm/ksm/sleep_millisecs`.

On every wake-up, the scanner visits `ksm_thread_pages_to_scan` pages. The number of pages to scan at each wake-up can be configured from user space via `/sys/kernel/mm/ksm/pages_to_scan`.

Selecting the Next Page KSM scans virtual address spaces for duplicates, as opposed to VMware's ESX which scans physical page frames [84]. KSM selects the next page for scanning based on the order in the `mm_slot` data structure. The advised pages are scanned in increasing address order per `mm`. The `ksm_cursor` stores the current scan position between wake-ups.

This is the place where new `rmap_items` are first allocated if no such item exists for the virtual page that is supposed to be scanned next, yet. Otherwise, the previously allocated `rmap_item` is reused; it then already caries useful information from the last visit of the scanner.

When scanning, the scan thread skips invalid pages. That means, that it omits indexing pages that are currently not in memory, for example when those pages are currently swapped out to disk. Moreover, KSM does not visit pages that already reside in the stable tree, identified by the `FLAG_STABLE` in the `rmap_items`.

Merging with Stable Pages Every time KSM visits a page, it first searches the *stable tree* for a sharing buddy. If such an equal pages is found, the new page is remapped to the *sharing* page frame and freed. The scan continues with the next page.

Filtering Frequently Modified Pages The unstable tree is endangered of degenerating quickly if many frequently fluctuating pages are inserted (§5.4.1). KSM uses a heuristic to make the insertion of such less likely. Only pages whose content-based `jhash2` hash value has not changed since the last visit are inserted into the unstable tree. To this end, the `rmap_item` records the hash value that the page had at the last visit. If `oldchecksum` carries the same value as the current hash value, KSM tries to insert the page into the unstable tree.

Merging with Unstable Pages When inserting the page into the unstable tree, KSM may encounter another page with the same content to already reside in the unstable tree. In this case, the new page is marked COW for sharing. Then the equal page that was found in the unstable tree is remapped to reference the shared page before freeing the redundant, now unreferenced copy. Afterwards, KSM adds a new `stable_node` to the stable tree and appends the `rmap_items` that now share the same page frame number.

This sequence is not atomic. There can be a racing page-fault on another CPU which breaks the sharing before the `rmap_items` are recorded in the stable tree. KSM thus checks if the sharing is still valid when appending the respective `rmap_items` to the stable node and leave them unmapped.

Scan Rounds After scanning the last `memory area` of the last `mm_slot`, KSM is done with a full scan round. It then drops the *unstable tree* by setting the unstable root node pointer to NULL. Afterwards, the `rmap_items` are still reachable through their `mm_slot` and are not freed. The `oldchecksum` and the entire *stable tree* remain.

Breaking Merged Pages The scan process is fully single threaded. Locking is only required to protect the code from concurrent memory management operations of the Linux virtual memory subsystem. One of such operations is breaking COW pages in the page-fault handler.

The page-fault- and exit-handlers in Linux have not been modified when KSM was added. Shared pages that have been merged by KSM are broken using the same mechanism that is also used to break shared pages that have for example been created by using `fork`.

4.3. XLH Design

When extending a main memory scanner such as KSM, one needs to be very careful not to break explicit and implicit design decisions that were made. Today, many CPU cores often share the same physical memory and use a shared bus to access it. In memory management, concurrent operations on shared memory management data structures are serialized by spinlocks. KSM in particular, has been carefully designed to add as little locking as possible to the existing memory management code. Most of the design decisions in XLH were influenced by this circumstance.

The following paragraphs describe our XLH design: §4.3.1 describes the reasoning and goals behind XLH. §4.3.2 describes how XLH generates hints in the VFS layer of the host. §4.3.3 presents how XLH stores received hints until they are processed. §4.3.4 reports how XLH prioritizes processing hinted pages and how this prioritization fits into the design of KSM.

4.3.1. Design Goals

The foremost design goal of XLH is to detect equal memory pages more quickly than a brute-force memory scanner such as KSM [2] or ESX [84]. This shall be done by visiting memory pages with a high prospect of finding a sharing opportunity earlier. Pages with a high prospect to yield a sharing opportunity shall be communicated via a *hint* to the deduplication scanner.

Visiting and sharing duplicates earlier leads to more free memory available for other purposes such as caching or running additional VMs. Figure 4.12 illustrates the reasoning behind this claim.

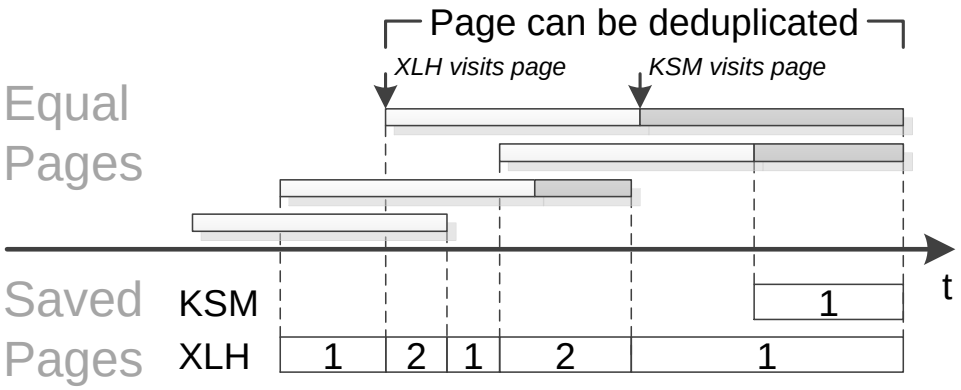

Figure 4.12.: Earlier sharing can cause more memory to be shared and memory to be shared for a longer time [62].

4.3.2. Hint Generation

We have extended `madvise` to pose a unified API to submit deduplication hints. To this end we have added the new flag `MADV_MERGEABLE_HINT` and made it possible to call `madvise` from within the kernel in addition to calling it from user space. An XLH-hint invocation looks like this:

```
madvise( hint_start, hint_length, MADV_MERGEABLE_HINT );
```

The parameters `hint_start` and `hint_length` are rounded to encompass only full virtual pages before passing the hint along and recording it.

I/O-Based Hints We have closed our analyses of the previous Chapter 3 with the discovery that memory duplicates primarily stem from process heaps and from background storage I/O. Our prototype currently supports generating hints in the virtual file system (VFS) layer with the incentive to find duplicates that stem from the background store more quickly than previously possible.

When using virtualization, the VM's file systems are implemented fully in the guests. The host solely exports a virtual block device to the guest. This is either done by passing a real device through to the host or by mapping the guest's block I/O directly onto a large file in the host. Generally, the latter is the case, as the former makes the flexible allocation and migration of VMs more complex.

Consider Figure 4.13. When an application accesses a file it can either go through the VFS using system calls such as `open`, `read`, and `write`, or it can map the file to memory using `mmap` (1). Then, accesses to the mapped memory regions are reflected in the file. For example, if an invalid page is read, the resulting page-fault will go through the file system and get the sought-after data from the background store (2). This behavior is the same, whether the application runs natively (i.e., without virtualization) or within a VM. In native environments, the access to the background store may be an actual DMA transaction to a local disk or go to a remote system (e.g., to NFS). In the virtualized case, the access will be handled by a virtual DMA controller and translated to another VFS call (3); this time in the host. The host then fetches the data from the guest's VDI file using the physical DMA controller (4).

When the DMA transaction finishes and the data returns to the second `read` VFS call (5), the host reports the target memory area of the I/O-operation as a hint to XLH using our `madvise` interface (6). Afterwards, the host returns the requested data to the guest (7), which passes the data on to the application (8).

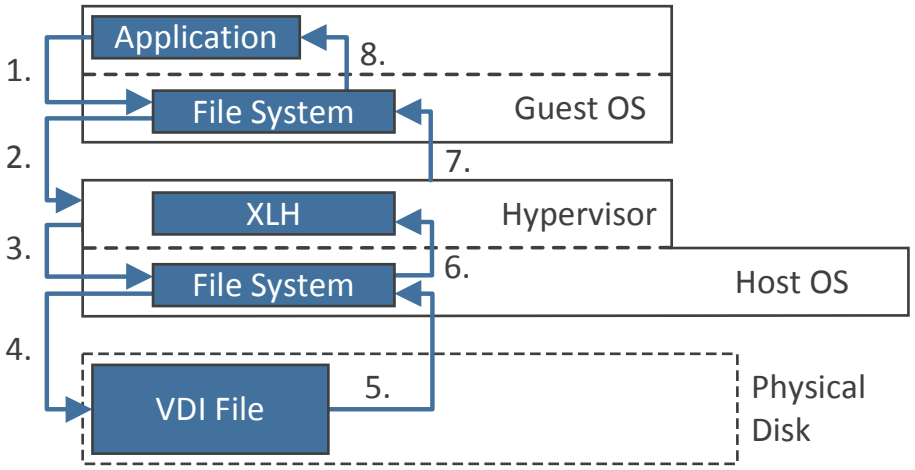

Figure 4.13.: Generating I/O-based memory deduplication hints.

In contrast to previous I/O-based approaches [12, 63], XLH does not *process* the hint before returning the result. Instead, it defers processing the hint to a later time and only *records* the target memory area. This policy minimizes the overhead for the I/O-transaction.

In addition, our system can be and has been implemented without modifying the guest OS, running within the VM, in any way. Using the VFS to emit hints even works for native applications running on the host. The only requirement is, that the target memory areas need to be `MADV_MERGEABLE`. This can enable XLH to efficiently deduplicate main memory for native server environments such as terminal servers, in the future.

Note that the VFS is not specific to Linux. All widespread operating systems use some form of a virtual file system layer to separate the file system implementation details from the application interface for file and directory access.

Other Hint Sources The hint generation and storage is fully decoupled from the memory scanner itself. Adding more hint sources is as easy as doing the appropriate `madvise` call from a different subsystem or even from a user level application. At this point, however, we haven't found another well suited source for hints.

4.3.3. Hint Storage

XLH needs to store emitted deduplication hints until they are processed by the scanner. There is a number of questions that play into the design of a good storage data structure. While evaluating how to design a well suited data structure for storing hints in XLH we have had the following insights:

Should all hints be recorded? This question boils down to the follow-up question: Can we filter bad hints and then only record good ones? Filtering hints is a *contradictio in adiecto*. We should never have to filter hints. If the hints are bad they shouldn't have been emitted in the first place!

Should all recorded hints be kept indefinitely? Hints are only useful if they are processed shortly after their submission. If pages are processed by the scanner before it processes the hint it will not only visit that page twice, but there is also no benefit of the hint whatsoever.

A hard upper bound for keeping hints ("aging") should hence be the time between page visits by a regular scanner with a given scan-rate and advised number of pages. XLH skips hints that were issued in the last scan round.

In our experiments with deduplicating virtual machine workloads, we have empirically found that a maximum age of around 15 seconds led to the most effective deduplication using XLH (§5.4.1). Processing older hints converges XLH to a regular KSM memory scanner. The main benefit XLH has is sharing hinted duplicate pages earlier than KSM. This is impeded by processing older hints.

Should there be an upper bound for the number of stored hints? The submission of hints is fully decoupled from their processing. The subsystem that submits a hint does neither know nor should it care whether the average rate of submitted hints is greater or smaller than the rate in which the memory scanner processes hints. In consequence, XLH needs a mechanism to discard hints ("pruning") when the number of incoming hints exceeds the number of processed

hints. There is no way to rank hints by their quality in advance to processing them. It is however better to discard old hints than new hints in order to both maximize the average time that merged pages remain shared and minimize the average merge latency.

Should the oldest (queue) or the newest (stack) hints be processed first? The same argument that was already used in the last paragraph applies here. New hints are more valuable to XLH than old hints. Therefore the hint storage data structure should return the hints to the scanner in ascending age.

Bounded Circular Stack Taking into consideration the previously laid out arguments, we have decided to implement the hint storage as a *bounded circular stack*, illustrated in Figure 4.14.

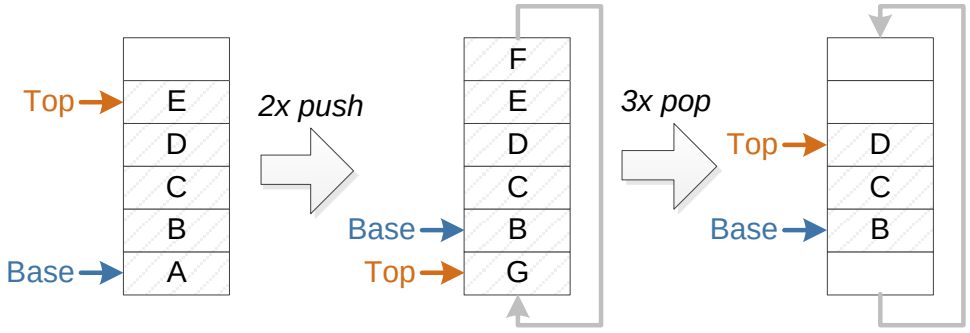

Figure 4.14.: The operation of XLH's bounded circular stack to store memory deduplication hints [62].

This data structure has the following properties:

- There is a hard upper bound for storing hints.
- No dynamic memory allocation is done when storing hints.
- The soft maximum age of hints can be adjusted by setting the stack size as a function of the scan-rate.
- The oldest, unprocessed hint is overwritten, when the stack is full and a subsequent hint is added.
- The newest hints are handed to the scanner first.

4.3.4. Hint Processing

We have answered the following questions before making the design decision to implement hint processing asynchronous to their submission and interleaved to the regular scan process.

Should hints be processed synchronously or deferred? Processing hints directly at the time of their submission (synchronously) can slow down the issuing task significantly. In the synchronous case, the task that issued the hint is paused until the hinted memory area is fully indexed.

Miłós et al. have found a slowdown of up to 35% when running the file system benchmark *Bonnie++* in their synchronous deduplication mechanism Satori [63]. Using synchronous deduplication of hints coming from frequent and latency sensitive tasks should in consequence be avoided.

We have decided to block the task that issued the hint as shortly as possible. Therefore, XLH only records the hint and then defers it's processing. This way, the processing is maximally decoupled from the submission of hints.

Should hints be processed in parallel, statically prioritized or interleaved? The design space has three trivial possibilities to process hinted pages:

First, hinted pages could be processed in parallel to the regular, linear scan process. This solution, however, would require serialization at all shared data structure accesses. In addition to complicating the process, this would also make it less scalable.

Second, hinted pages could be statically prioritized higher than the linear scan process. Then, however, the linear scan process would not make any progress if the number of incoming hints superseded the scan-rate. In our analysis (Chapter 3) we have noticed that often both, heap and file based pages provide deduplication candidates. Starving the linear scan process would cut the heap based deduplication opportunities off from the scanner.

We have decided on a third option. Interleaving the linear scan with processing hints (Figure 4.15). Upon wake-up, the scanner decides if it continues the linear scan or processes hints based on a defined interleaving ratio. If the hint buffer does not contain enough hints to satisfy the number of pages (`pages_to_scan`) that are supposed to be scanned in this wake-up, the remaining quota is filled with pages from the linear scan.

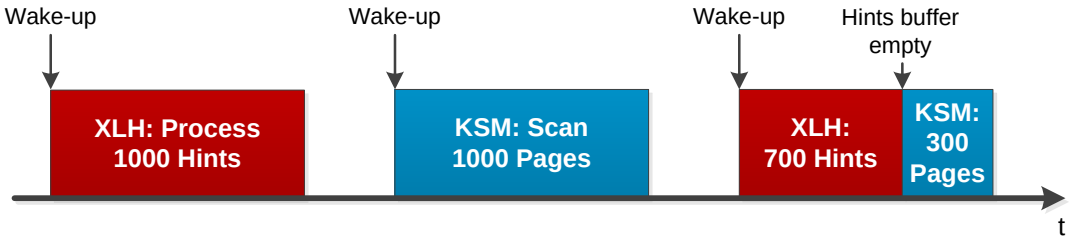

Figure 4.15.: Hints are processed interleaved to the regular scan process [62].

Should hinted pages be filtered for fluctuation first? We have decided, that by design, hints should only be issued on pages that are good sharing candidates.

This also implies, that their chance to be highly fluctuating should be low before they are submitted as a hint.

Pages from the page cache are inherently not highly fluctuating. In the contrary, as we have seen in the previous chapter, their content does often not even change right away after they are unmapped. In consequence, we do not filter hinted pages at all.

Resulting Design Following our previous discussion we have derived the XLH workflow depicted in Figure 4.16. A single scan thread is responsible for both, the linear scan and for following hints. At wake-up, it decides what kind of pages it indexes next. If the hint buffer is exhausted, it continues the linear scan, keeping the number of pages scanned per wake-up constant. This design has the additional advantage that the comparison between KSM and XLH can be done easily and fairly based on the same scan-rate setting.

When processing a page in the regular scan mode, the workflow resembles the one previously introduced in the description of KSM (§4.2.2). When processing a hint, however, the page is always looked-up in the unstable tree, regardless of its previously recorded hash value.

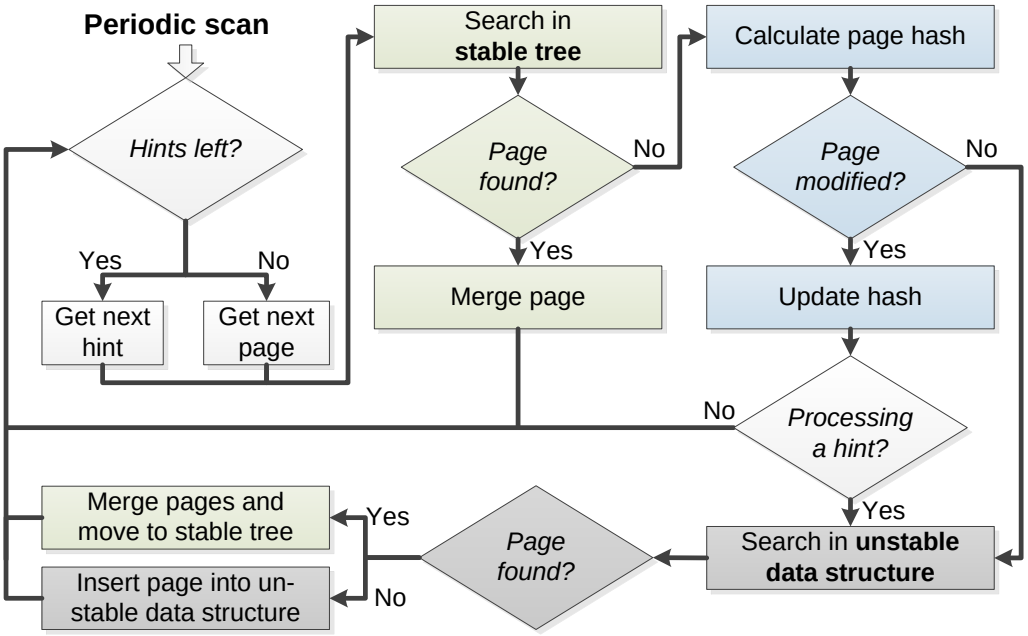

Figure 4.16.: High level workflow of the XLH scan process.

4.3.5. Mitigating the Unstable Tree Degeneration

As we have seen in §4.2.1, unstable tree nodes are not protected from modification. In consequence, the reachability of nodes in the unstable tree can be broken when pages that are indexed through the unstable tree are modified. We have analyzed the following three solutions that can stop the degeneration of the unstable tree.

Mapping Pages Referenced by the Unstable Tree Read-Only Instead of leaving pages that are indexed through the unstable tree writable, all pages that are inserted into the tree can be marked read-only analogous to the behavior of the stable tree.

This solution should not be blindly applied to all pages that are visited by the scanner as it can degrade the performance of scanned applications by an order of magnitude. Detailed measurements can be found in Chapter 6. Mapping the nodes that are referenced in the unstable tree read-only can however be beneficial if applied to a chosen subset. This approach is denoted as "KSM RO" in the evaluation.

Indexing Hashes instead of Page Contents Another possible solution to the degeneration problem would be to index hashes of the contents that the pages had at the time of insertion instead of indexing the volatile content itself. We have decided that if hashes were to be indexed then the appropriate data structure is a hash-table not a tree.

Replacing the Red-Black-Tree with a Hash-Table The main problem in the degeneration of the tree is not that nodes become stale themselves. The gravest problem is that modified nodes can make other nodes unreachable, too. Entire sub-trees (at most half of the tree) can become unreachable after the modification of a single referenced page (recall Figure 4.9).

The second option that we have implemented to stop the unstable tree from degenerating is to exchange the tree altogether with a hash-table. When using a hash-table, modifying a referenced page can at most break the reachability of that page from this hash. This is not a problem, as the referenced page's old hash value is not an indicator that the new page content is sought for merging. The modification of the page content can however not break the reachability of any other node in the unstable tree.

We have implemented this modification under the name "XLH HT" to compare it to the read-only unstable tree solution and the unmodified unstable tree.

4.4. XLH Implementation

We have first implemented XLH on the basis of KSM in Linux 3.0 and later ported it to Linux 3.4. Including all comments, debug code and statistics we have added 2350 and modified 250 lines of code. Almost 95% of the implementation falls into the main KSM implementation (`mm/ksm.c`). The remaining changes were done in the I/O subsystem (hinting) and in header files.

For our virtualized environment we have chosen QEMU [8] with KVM. A popular, open source, virtualization host that can be easily used to run batches of unattended benchmarks and is widely used in practice.

VFS Hinting We have directly inserted our hint submission mechanism into the VFS `read` and `write` functions. It is crucial, that the I/O operation is done first (`do_readv_writev`) before recording the hint. Otherwise, as hints are processed asynchronous and in parallel to the I/O itself, the scanner could access the hinted page before its content has been established.

Finding `rmap_items` in the `mm_slots` data structure KSM scans pages strictly linearly by VMS's. A linked list is an efficient data structure to implement finding the next page to scan in this scan order. To this end, KSM implements separate linked lists for each `mm` in the `mm_slots` data structure.

XLH needs to look-up `rmap_items` in random order, as hinted pages need to be visited out of the linear scan order. To efficiently support both, the linear scan and random access to the `rmap_items` we have modified the `mm_slots` to hold red-black trees of `rmap_items` instead of a linked list.

Insert operations only happen infrequently, when new pages are `madvised` to be deduplication candidates after booting a new VM and are thus not time critical. Linear traversal of a red-black tree can be implemented as efficiently as the traversal of a linked list (O(1) to get the next item). In addition, the red-black tree allows random look-ups in O(log n), asymptotically. In typical virtual environments, the tree depth is in the area of 18-22 (1 GiB-32 GiB).

If the look-up time ever becomes a bottleneck in the memory scanner, it is also possible to cache previously looked up rmap_item locations in the `address` field of the `page` data structure which is unused while the `rmap_item` is a part of the unstable tree. Then, the first look-up would need to be done in O(log n) while following look-ups could be performed in O(1).

Statistics When measuring the sharing time, we cannot use the time at which the respective `rmap_item` is removed from the stable tree as the end time. This is because (stale) `rmap_items` are removed from their `stable_nodes` when their invalidity is detected in a subsequent lookup operation. The item is not removed when the sharing is broken (recall §4.2.1).

We had to modify the page-fault handler (`do_wp_page`) to record the COW break time for memory that has been merged by the memory scanner; a special case that is not needed for regular operation without statistics.

Chapter 5

Deduplicating Virtualized Environments

Since the 2000s there has been a trend to move physical servers into virtual ones and consolidate those virtual servers onto fewer physical machines than previously possible. This strategy has many benefits such as a higher utilization of all subsystems in the remaining physical servers. Especially the CPU – which is otherwise typically idle in today's servers – benefits from consolidation. Moreover, this techniques yields the possibility to flexibly allocate and migrate server instances. This allows operators to increase the efficiency of data-centers by switching off unused physical servers, effectively cutting costs of data-centers.

As previously described in research papers (Chapter 2) and thoroughly analyzed in Chapter 3, such environments are a good target for memory deduplication for two reasons: First, the amount of duplicates can be large and further increased through smart migration [88]. Second, virtual environments benefit greatly from memory deduplication as their bottleneck is generally the main memory size [39].

This chapter analyzes the effect that XLH, in conjuncture with I/O-based hints, has on deduplication in virtual environments. We show that the deduplication effectiveness of XLH is superior to KSM at the same scan-rate settings. We also present where the increase in shared pages comes from. Moreover, we explore the impact that KSM's degenerating unstable tree has and analyze the effects that mitigating the unstable tree generation has on the memory scanning performance.

We first introduce our metrics to measure success in Section 5.1. Section 5.2 elaborates on the benchmark scenarios that we have used to evaluate our approach and Section 5.3 introduces common benchmark parameters. The following Section 5.4 shows the results of our benchmarks and discusses them before we conclude the chapter with a summary in Section 5.5.

5.1. Benchmark Metrics

The two most important metrics for any memory deduplication system are the amount of memory that can be freed using the technique and the resources that it needs to perform the deduplication. In the comparison between memory deduplication systems we say that one system is superior to another if it can save more memory than the other, while generating the same amount of overhead in a certain workload. This can also be expressed the other way round: Can one system deduplicate the same amount of memory while using less resources than the other?

We have set-up and run different benchmarks to compare the deduplication characteristics of KSM and XLH. In addition to the total run-time we have measured the following data when conducting our benchmarks.

Sharing Opportunities We have measured the amount of available sharing opportunities by periodically dumping the hash values of all allocated pages in the workload. The implementation details of the tool can be found in Thorsten Gröeninger's study thesis [37]. For the purpose of analyzing memory duplication, we output all content hashes every second.

Hashing all allocated pages and recording those hashes every second adds a runtime overhead in the area of 20%. This number can of course vary, depending on the workload, working set size, and hardware. In order to make the sharing opportunity quantities comparable to benchmarks without such slowdown, we have contracted the time axis to match the exact benchmark time.

Shared and Sharing Pages The number of shared[1] and sharing[2] pages can already be accessed from user space in the vanilla KSM implementation through `sysfs`. During the benchmark, we output such values every second.

Merge Latency Comparing the time between establishing a page content and merging this page is tricky without tracing the memory contents. We have relaxed this analysis to compare when KSM and XLH merge pages with the same content throughout the benchmark. Our approach is to filter out all shared pages that only occur in either one of the systems but not in both. For the remaining shared pages that are established in both systems we compare the times at which those pages are merged relative to the beginning of the benchmark. In consequence, we don't measure the actual, absolute merge latency but measure the latency difference between systems instead.

[1] `/sys/kernel/mm/ksm/pages_shared`
[2] `/sys/kernel/mm/ksm/pages_sharing`

5.2. Benchmark Scenarios

We have used the following benchmark scenarios to compare KSM and XLH regarding deduplication effectiveness.

Kernel-Build Measuring the sharing potential of parallel Linux kernel compiles has a long tradition in publications [39, 47, 61–63, 77] regarding memory deduplication. Compile jobs have the characteristic that they need CPU computation for parsing, lexing, and optimizing. They need file operations, as well. For example, when reading source files, writing intermediate object files, and when reading objects to link the final binary. Moreover, it is a common scenario to run multiple VMs compiling the same source code at the same time, for instance for continuous integration (CI). One representative for a cloud service supporting CI in such a way is TravisCI [80].

HTTPerf Benchmarking web-server performance, like the kernel compile benchmark, is also common in memory deduplication publications [39, 47, 61–63]. Serving static web-sites using the Apache web server [7] is an I/O-intensive benchmark which is sensitive to the access latency. HTTPerf [64] can measure this performance characteristic of web-servers and we can thus gain insight into the differences in deduplication quantity and into I/O overheads, at the same time.

Bonnie++ The file system benchmark Bonnie++ [17] is a local stress-test for the background storage I/O-layer. We chose Bonnie++ to check if the hinting mechanism is a bottleneck for I/O-based hints. Previous deduplication systems such as Satori have been prone to slowdowns of up to 35% in this benchmark [63].

The average disk throughput and access latency are direct result of the Bonnie++ benchmark. We have also compared the run-times – the time that benchmarks need from start to finish.

Mixed In this benchmark, we combine one VM from the kernel-build benchmark and another VM from the HTTPerf benchmark. We have modified the HTTPerf settings to match the kernel compilation time. This benchmarks compares KSM and XLH in a scenario that has only few sharing opportunities while many, mostly bad, hints are produced.

Micro-Benchmarks In the next Chapter 6 we analyze the performance characteristics using synthetic micro-benchmarks. Answers to questions about the performance of merging and copy-on-write breaks as well as on caching effects can be found there.

5.3. General Benchmark Set-Up

All benchmarks have been conducted on a PC with an Intel i7-2600 quad-core processor with 16 GiB DDR3 RAM. We have switched off the turbo boost feature in the BIOS and disabled dynamic frequency and voltage scaling, fixing the frequency at the maximum. We have set-up Ubuntu 13.04 as the host OS and replaced the original Linux kernel with our XLH extended version.

Ubuntu 12.04 LTS serves as the guest OS in the VMs, as this *Long Term Support* version is commonly already available in commercial cloud computing providers. The address space layout randomization (ASLR) feature decreases the amount of available sharing opportunities [5]. Nevertheless, we have left this feature at its default setting, which is: on. Each guest is assigned one virtual CPU (vCPU) and 512 MiB memory. One physical CPU (pCPU) is consequently used for the benchmark logic and for recording results, another pCPU is occupied by the memory scanner. The remaining pCPUs are used for running the virtual machines. This set-up is in line with the current developments in computer hardware where the number of cores raises faster than they are utilized. Consequently, we assume that KSM has a pCPU for itself.

Unless otherwise noted, every following graph shows the mean of six runs of a respective configuration. Table 5.1 summarizes the settings that we have used in the three configurations used throughout this chapter (XLH RO, KSM, KSM RO). The table also shows the XLH HT configuration benchmarked in Chapter 6.

Setting \ Name	XLH	XLH HT	KSM	KSM RO
VFS Hinting	●	●	○	○
Unstable Structure	RB-Tree	Hash-Table	RB-Tree	RB-Tree
Skip Reset-Tree	●	●	○	●
RO Unstable Structure	●	○	○	●

Table 5.1.: Default settings of the four configurations used in the benchmarks.

Scan-Rate We vary the scan-rate by adjusting the sleep time while leaving the pages that are scanned at each wake-up constant at 100 pages. The bounded circular stack sizes that we have used for each sleep time and the resulting scan-rates of the respective settings are listed in Table 5.2.

Size of Bounded Circular Stack The size of the hint buffer, in our implementation the bounded circular stack, is a parameter that we need to set for XLH in the following benchmarks. We have run kernel-build benchmarks to find good values for the hint buffer and applied those settings to all benchmarks.

Variable \ Sleep time	20 ms	100 ms	200 ms
Stack Size [entries]	40.960	8.192	4.096
Effective Scan-Rate [pages/s]	5.300	1.100	530

Table 5.2.: The stack sizes we have used for the varying scanner sleep times. See §6.1.3 for a discussion about the effective scan-rate.

Figure 5.1 and Figure 5.2 show the kernel-build benchmark at sleep-times of 20 ms, 100 ms and 200 ms, when scanning 100 pages at each wake-up, with various sizes of the bounded circular stack used to store hints before they are processed. The VMs are booted and four full scan rounds are performed to share static sharing opportunities in the base image before the workload starts.

The bounded circular stack size has a great impact on the deduplication quantity when using XLH. Three different outcomes can be observed:

- If set **too small**, the deduplication quantity of XLH is low. This is because XLH degenerates into KSM when the hint buffer size moves towards zero. XLH with a zero hint buffer size is behaving exactly like KSM as no hints are processed at hint-processing wake-ups. XLH then shifts directly to the linear scan for the rest (the entire) of the hint-processing quantum.

- With larger hint buffer sizes the deduplication quantity in our benchmark steadily rises until it is saturated.

- If the hint buffer size is **too large**, the deduplication ratio drops again. This can be observed best in the benchmark with a 200 ms sleep-time (Figure 5.2). The deduplication quantity is much lower with stack sizes that can store 8192 and 16384 hints before overflowing than with a stack size that stores 4096 hint entries.

The reason for the drop in deduplication quantity for very large hint buffer sizes is, that large hint buffer sizes store hints for a longer period of time. Once hints become outdated, they are of no use to XLH anymore. They do, however, take up some of the scan quantum to process and thus decrease the efficiency of XLH.

Currently, we only use a very course grained aging mechanism to filter out such outdated hints. If a hint was issued in the last scan round,[3] XLH drops it after popping it from the top of the stack. Dropping hints does not count into the hint processing quota as XLH discards those hints before visiting the hinted pages themselves.

[3] Which we can recognize by a sequence number that we store with a hint. The sequence number can be stored in the low order bits of the address at no extra storage expense.

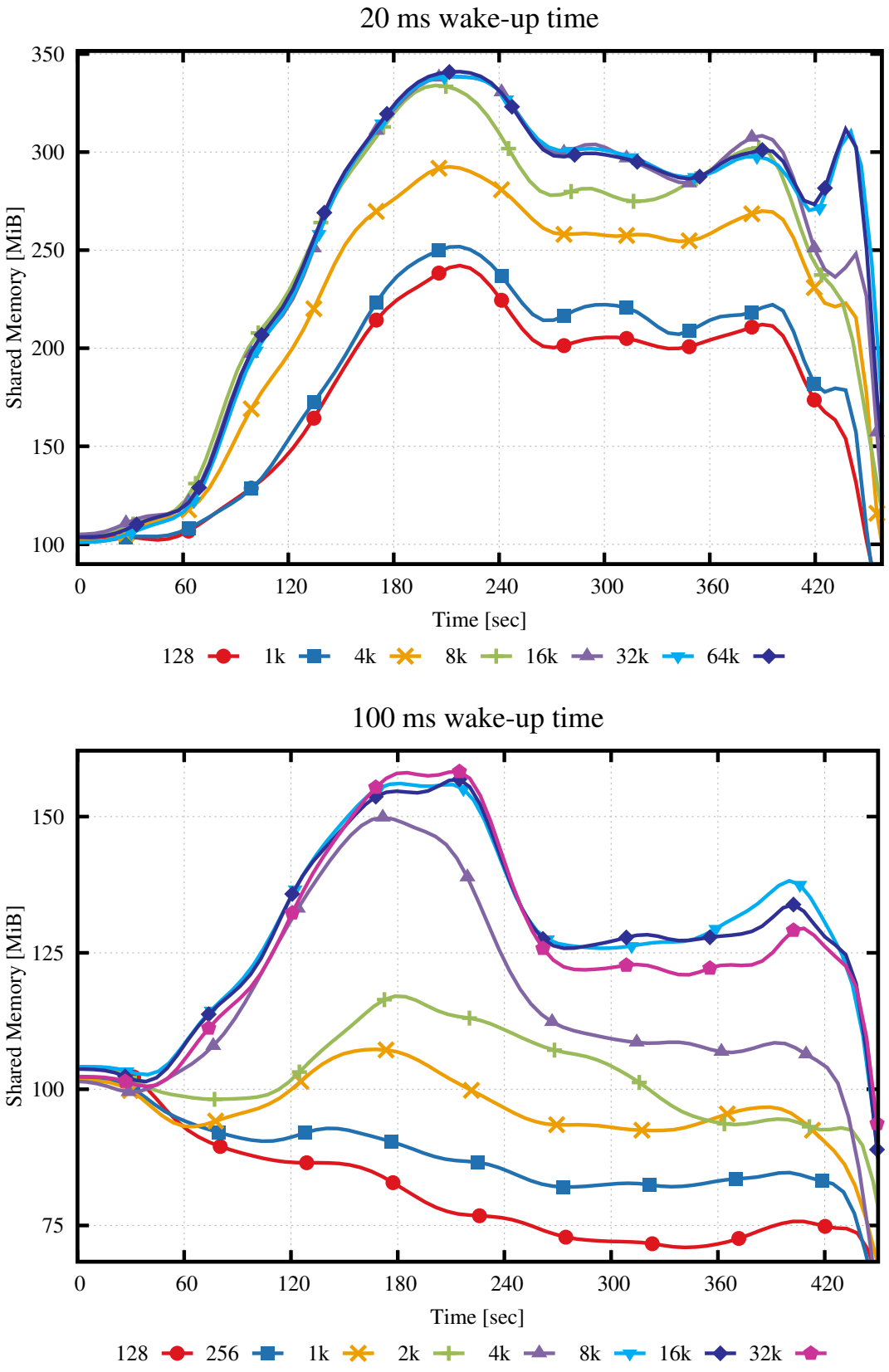

Figure 5.1.: Kernel-build deduplication effectiveness at fixed sleep-times of 20 ms and 100 ms, scanning 100 pages per wake-up with different stack sizes in the XLH RO configuration.

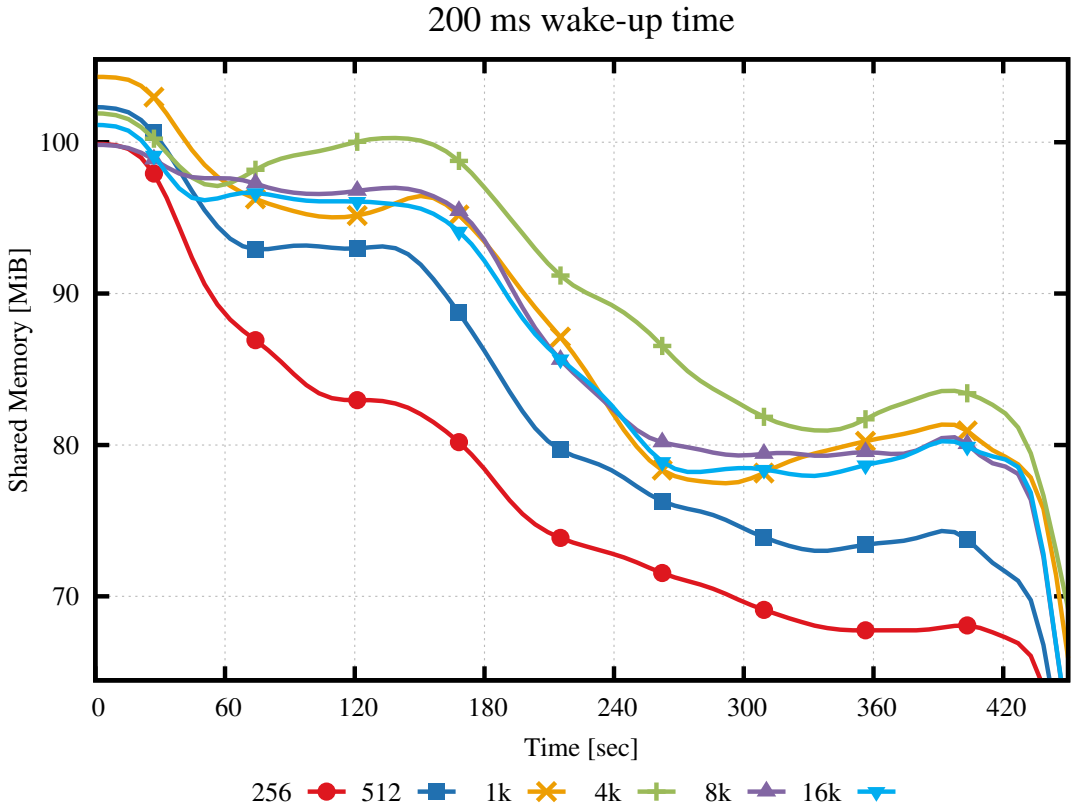

Figure 5.2.: XLH RO Kernel-build deduplication effectiveness at a fixed sleep-time of 200 ms, scanning 100 pages per wake-up with different stack sizes.

For the kernel-build benchmark, a simple rule of thumb applies for choosing a well suited hint buffer size. We have found empirically that the deduplication quantity is at its maximum, if a full hint buffer can be drained in about 15 seconds at the fixed scan-rate and pages-to-scan setting. Our default hint buffer sizes (Table 5.2) reflect this rule. Note that the rate at which hints are processed in a given benchmark is dependent on the scan-rate as well as on the interleaving ratio.

5.4. Evaluation Results and Interpretation

This section contains specifics about our benchmark set-ups, their outcome, and its discussion. The first two benchmarks, the *kernel-build* and *HTTPerf* show the deduplication performance when many duplicates are present. The following *Bonnie++* benchmark contains only few duplicates. The key focus in this benchmark was to stress-test the background storage system and in consequence test if XLH introduces a source for slowdowns with the hint generation and storage. The last section contains the *mixed* benchmark where we run one kernel-build VM and a VM running the apache web server simultaneously. In this benchmark we compare XLH and KSM in a scenario where only few deduplication candidates exist, yet many, mostly invalid hints are generated.

5.4.1. Kernel-Build

In the *kernel-build* benchmark, two VMs compile the Linux 2.6.27 kernel simultaneously. We have disabled the compilation of most kernel modules to reach a compile time of just above 7:30 minutes.

Benchmark Set-Up Figure 5.3 illustrates the set-up of the kernel-build benchmark. The benchmark logic and the memory scanner occupy one core each. In addition two VMs are run on the remaining two pCPUs, building a Linux kernel each.

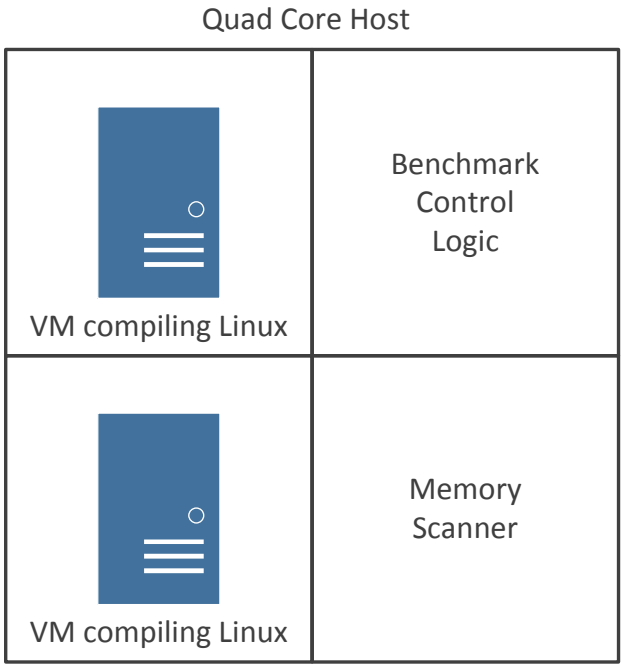

Figure 5.3.: Two VMs compile the Linux kernel, each one occupying a CPU core. The remaining two cores are used for the benchmark logic and the memory scanner, respectively.

Recall that the line illustrating the amount of possible sharing opportunities in Figure 5.4 and Figure 5.5 were measured in a separate benchmark without running a deduplication scanner. Instead, we took memory snapshots of the two VMs every second and later analyzed these for equal pages offline. Due to the high snapshot overhead, the run-time of the kernel-build increases. We have scaled the time axis to match the original time interval. Also, we have shifted the line to the left to match the points in time that the kernel-build starts in both benchmarks.

Deduplication Quantity In this benchmark, we assume that the VMs were specifically booted for running the benchmark workload. Hence, we start the memory scanner and the workloads at the same time. That way, in the beginning of the benchmark, in addition to the duplication caused by the workload, the scanner also has to merge static duplicates from the base system. This set-up is comparable to continuous integration services that boot a separate VM for each specific compilation and integration job.

The development of the deduplication quantities when running the benchmark with different scan-rates can be found in Figure 5.4. In our kernel-build benchmarks, XLH constantly outperforms KSM at the same scan-rate. At a sleep-time of 20 ms, XLH can share almost all available sharing opportunities. KSM in turn can almost constantly only find about two thirds of those sharing opportunities when running at this scan-rate. At the lower scan-rates KSM can share almost no sharing opportunities at all. XLH works much better at those low scan-rates. It is, however, also limited by the small rate at which pages are visited and merges much fewer sharing opportunities than available.

Deduplication Quantity After Warm-Up The base system already contains duplicate memory pages. In this benchmark, before starting the compilation workload, we have waited for four full scan rounds to complete before starting the kernel-build. This way all sharing opportunities from the base system are already shared when the workload begins.

Despite the deterministic nature of the experiment, the *initial sharing* quantity varies by around 7000 pages (~27 MiB) throughout the 72 considered benchmark runs. We have measured that between 20037 and 27340 pages (78–107 MiB) can be shared initially, after booting both VMs and before starting the kernel-build.

Figure 5.5 depicts the deduplication quantities after warming up the memory scanner with the VM base system. The graph with the fastest scan-rate almost resembles the previous one without warm-up. The other two benchmarks, however, paint a different picture. While KSM loses more than half of its previously merged sharing opportunities, XLH can at least keep a dynamic equilibrium between merges and COW-breaks.

In the benchmark with the most aggressive scan-rate, XLH can in total detect and merge 1.6 times more sharing opportunities with different content than KSM. While XLH merges 550411[4] different sharing opportunities, KSM can only find 342903[4] pages that can be shared.

[4]Median of 6 benchmark runs

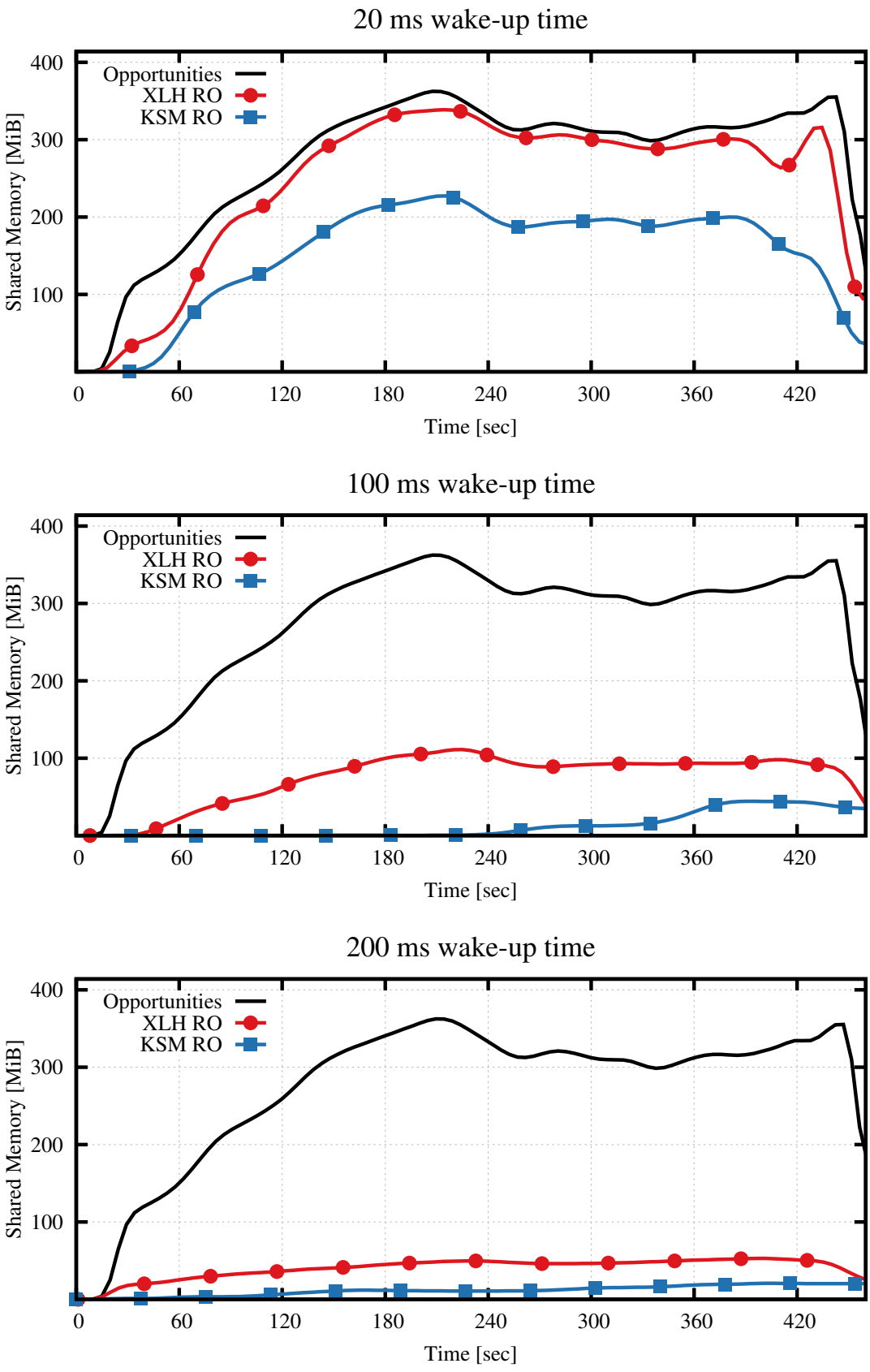

Figure 5.4.: Kernel-build merge performance with varying wake-up times. The memory scanner and kernel-build start at the same time.

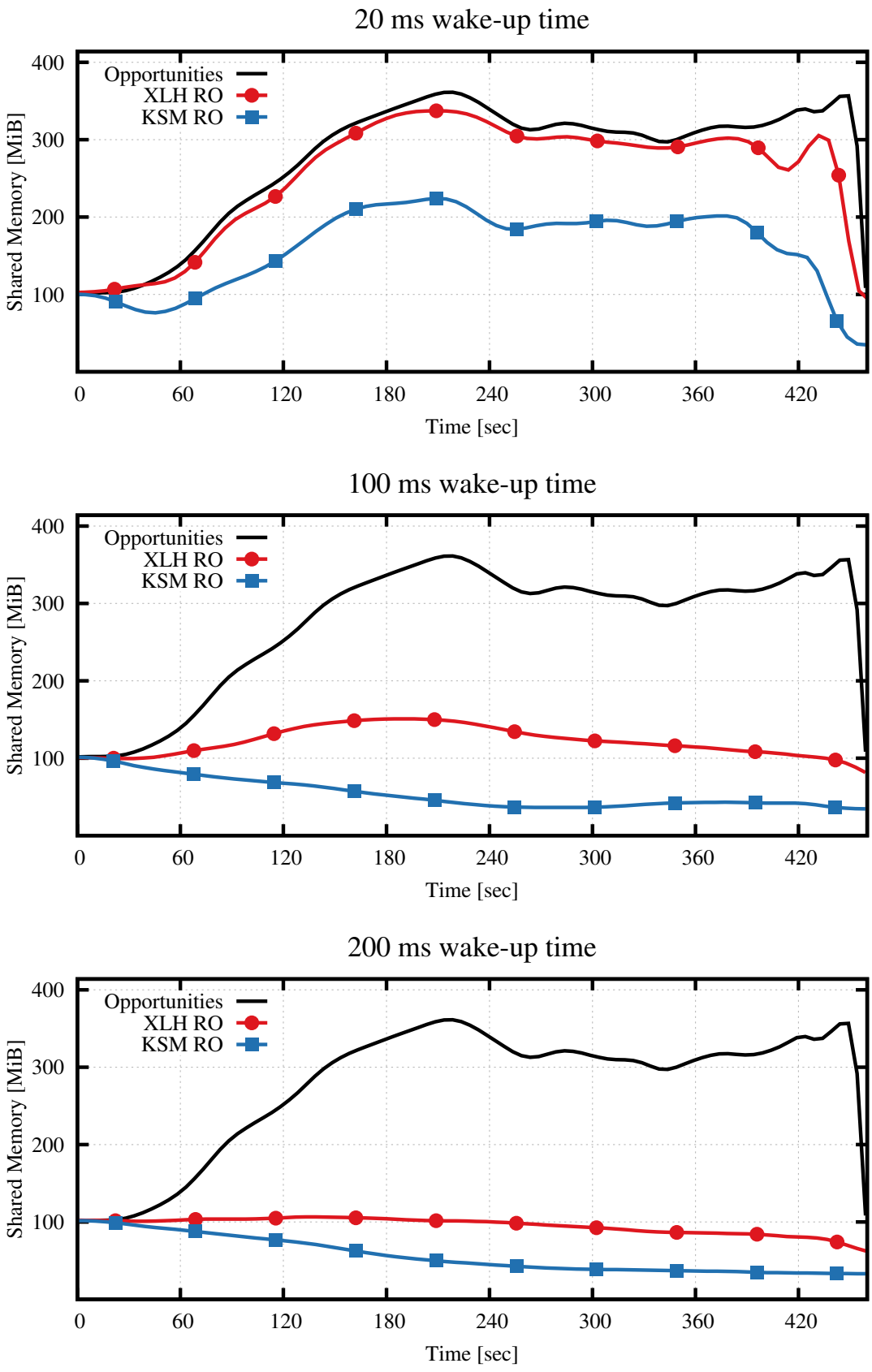

Figure 5.5.: Kernel-build merge performance with varying wake-up times. The scanner merges static sharing opportunities, then the workload starts.

Merge Latency There are two different possibilities for false positives when using the method previously described in §5.1. First, the matching algorithm records matching contents based on the hash value alone. Since we are using a weak hashing algorithm (jhash2), no such match needs to be in fact present in the original content in the case of colliding hashes. Second, it is possible, that one of the systems merges the first occurrence of a sharing opportunity while the other system merges a later sharing opportunity of the same content but does not find the first occurrence. Then, two semantically different sharing opportunities with the same content are compared. False positives will appear as outliers in the measured merge latencies.

From two benchmark runs, we were able to match 121813 sharing opportunities. To account for false positives we have filtered the top and bottom 1‰ (122 values) before visualizing data. Figure 5.6 depicts the resulting difference in merge latency times of KSM RO and XLH RO, sorted by time.

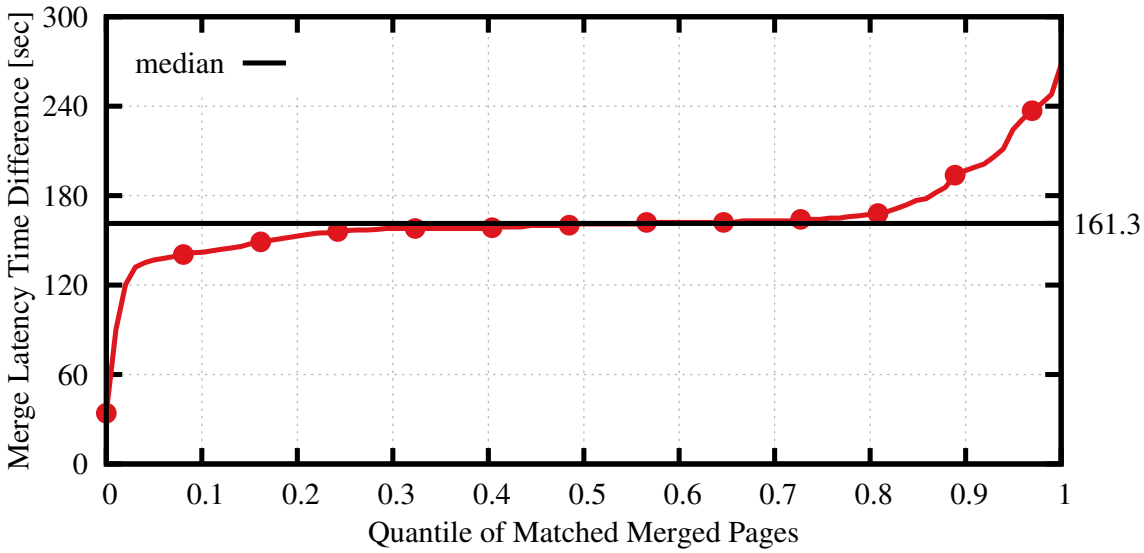

Figure 5.6.: Sorted merge latency difference of 121571 matched merged pages between KSM and XLH in the kernel-build benchmark at 20 ms wake-up time.

When comparing the time into the benchmark at which the respective scanners merge pages with equal content hashes, it becomes apparent that XLH detects those opportunities around 2:41 minutes before KSM at a 20 ms wake-up time. Negative values in this graph would mean that XLH merges pages after KSM. This only occurs in less than 0.14‰ (17 values) of the merges in our (unfiltered) sample, however.

Viewed from the opposite angle, XLH detects sharing opportunities more quickly than KSM in more than 99.9% of all merged pages that we have matched semantically. XLH thus achieves the superior deduplication quantity by checking and merging prospective sharing candidates earlier than KSM.

Merge Duration Finding sharing opportunities earlier has a twofold impact. First, XLH can find additional short lived sharing opportunities. Second, XLH shares sharing opportunities that can also be found by KSM for a longer period of time.

Both effects are confirmed by the histogram plotted in Figure 5.7, which shows the merge duration using the same data samples used in the previous paragraph. Here, the number of short lived sharing opportunities, up to 60 seconds, doubles when using XLH over KSM. Longer lived sharing opportunities that can be shared for over 150 seconds, also rise. The shift to longer sharing times can be seen especially well in the range between 360 seconds and 450 seconds.

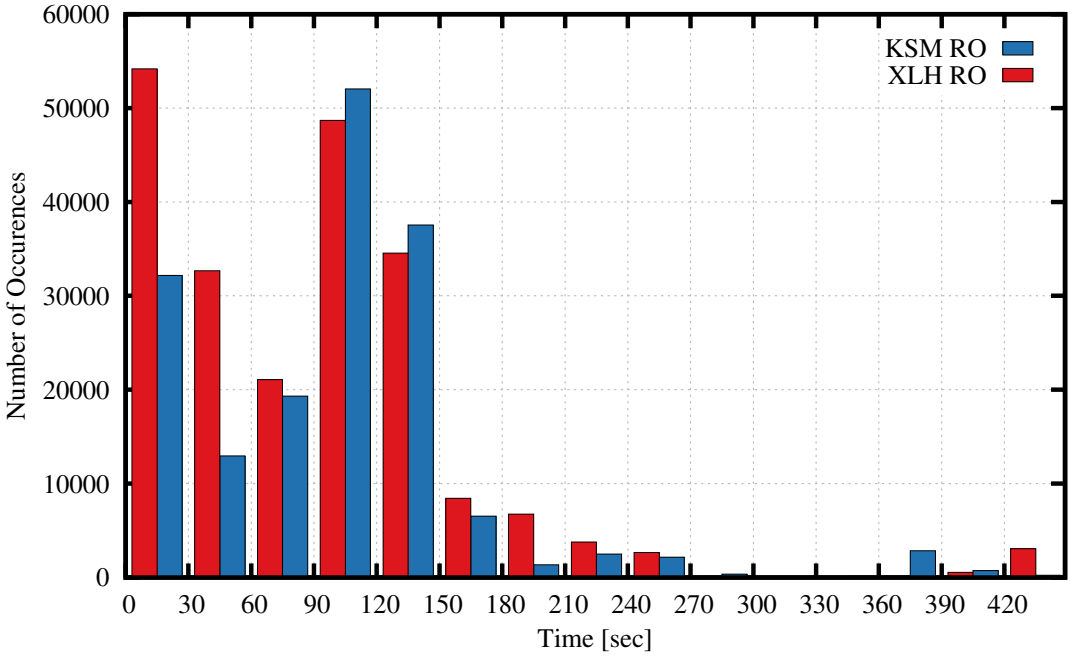

Figure 5.7.: Merge durations in the kernel-build benchmark at 20 ms sleep-time.

Kernel-Build Run-Time We have measured the total time that each VM needed to boot and compile its Linux kernel. Figure 5.8 depicts the resulting workload run-times averaged over 12 VMs respectively. These times come from the first kernel-build benchmark which was done without first merging static sharing opportunities from the base image.

XLH is always slightly slower than KSM on average. As we will explore in Chapter 6, this slowdown is due to the time it takes to merge and break the respective memory and is not attributed to finding the additional sharing attributes. If KSM and XLH were set to merge the same amount of sharing opportunities on average then both scanners would spend the same amount of time merging memory. XLH, however, would then spend less time finding the sharing opportunities and thus speed up the workload compared to KSM.

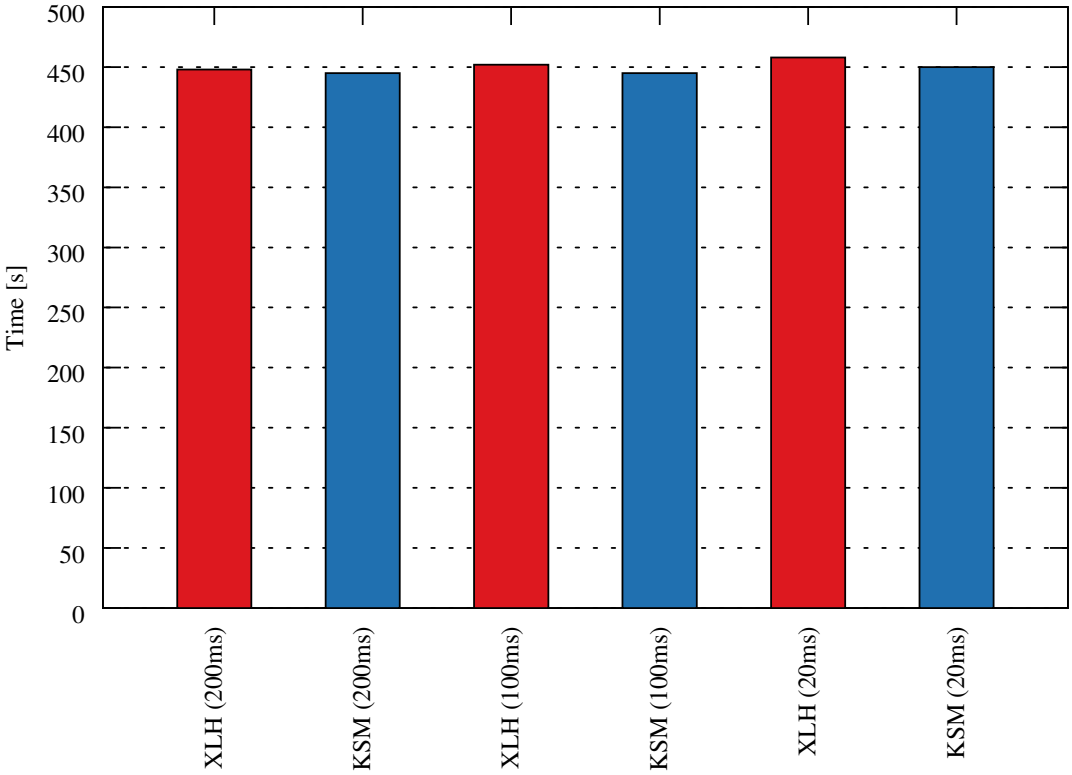

Figure 5.8.: Average time, the kernel-build takes to finish in the different configurations. Six runs were timed resulting in an average of 12 VM compile-times for each configuration.

Unstable Tree Degeneration The unstable tree data structure is meant to index memory pages that have not been modified between visits but that do not currently have a sharing partner. Just like the entire KSM memory scanner, the unstable tree works only well if the goal is to share long-lived sharing opportunities. If pages are written between scan rounds, the subtrees underneath those pages may get lost (Figure 4.9). This degenerating unstable tree problem aggravates the closer the modified pages are to the root of the tree as the loss of subtrees will then affect more pages.

It is easy to measure the number of affected pages in the unstable tree. All tree-nodes can still be reached for example using a breath-first search (BFS). To check which and how many nodes are reachable in the unstable tree, we have implemented a modified BFS: For every node N discovered through the BFS, the algorithm checks if the content of N can also be found when searching for it in the index.

When running this unstable tree checker at the end of each scan round, we get the node reachability summarized in Table 5.3. The numbers for each scan-rate were

Property \ Sleep-time	20 ms	100 ms	200 ms
Upper 95% confidence:	82%	76%	85%
Arithmetic mean:	62%	53%	75%
Lower 95% confidence:	41%	31%	65%

Table 5.3.: Two VMs running the kernel-build benchmark with the vanilla KSM. Percentage of pages that are reachable at the end of a full scan round.

generated from 12 kernel-build benchmark runs. Under the assumption that the reachability of unstable nodes follows a normal distribution, the 95% confidence interval boundaries indicate that 95% of the arithmetic means of all possible kernel-build benchmarks are within the given boundaries. It is thus very likely, that only between 30% and 85% percent of the nodes in the unstable tree are reachable at the end of each scan round.

The largest influence for the number of unreachable unstable tree pages is the number of total pages in the tree. If the number of memory pages that reside in the unstable tree rises, so does the probability for a modification of a page in the unstable tree. Every modified unstable tree page can make up to half of the tree unreachable if the root-node is modified.

The proportion of pages that is admitted into the unstable tree depends on the interplay of two factors: the memory modification rate and the scan-rate. If the modification rate is very high compared to the time that a scan round takes to finish, then only few memory pages will be inserted into the unstable tree, because the hash heuristic prevents the insertion of all modified pages.

Despite causing a higher possibility for breaking the tree, it is important to admit many pages into the unstable tree. This is because a larger number of indexed pages raises the chance to find a sharing opportunity. In conclusion it is important to address the degenerating tree problem.

Mitigation of the Unstable Tree Degeneration We have implemented two approaches to mitigate the degeneration of the unstable tree. First, mapping the memory pages that are indexed through the unstable tree read-only and using the page-fault as a signal to mark tree-nodes as stale[5]. Second, we have implemented a hash-table to replace the unstable tree.

[5]Stale nodes can then be cleaned up on the next KSM run without additional locking.

Figure 5.9 compares the deduplication ratios of different unstable tree configurations. Regarding the deduplication ratio, both approaches – the hash-table and the read-only unstable tree – perform exactly the same. For visual clarity, we have only printed the former line in the graph. *Reset* refers to dropping (resetting) the unstable tree at the end of a scan round. If the unstable tree does not degenerate at all, the memory scanner has no reason to reset the unstable tree at the end of a scan round, as resetting the tree is merely a clean-up mechanism for unreachable unstable tree nodes.

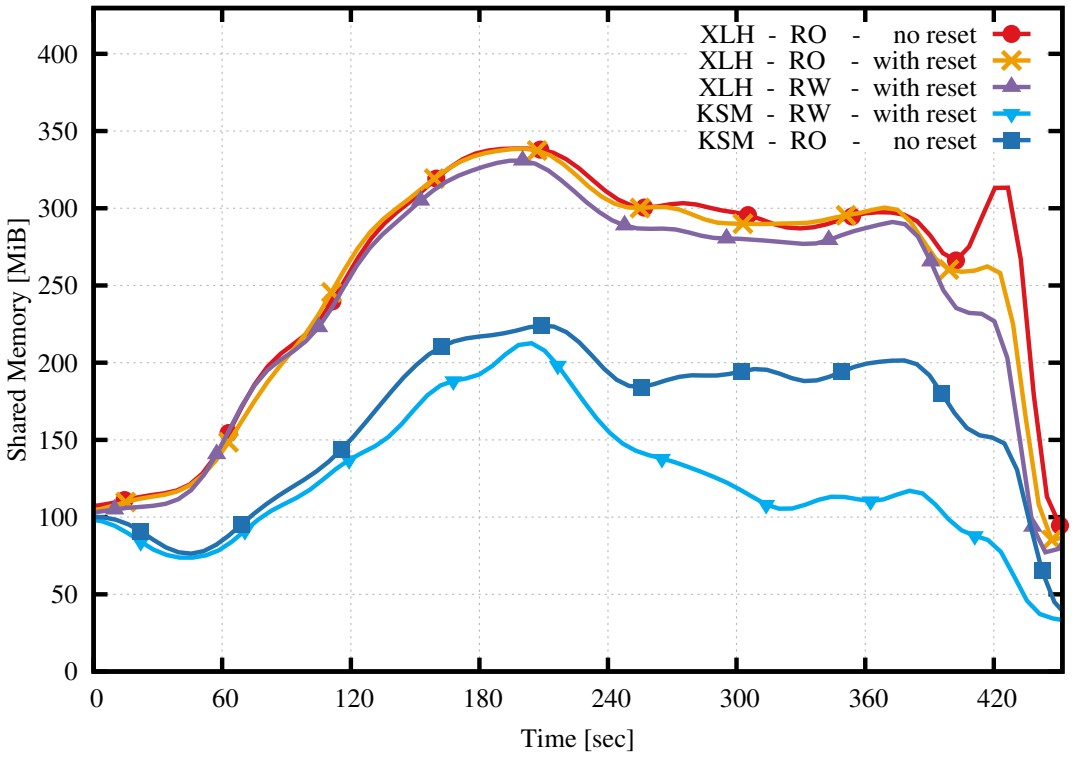

Figure 5.9.: Comparison between different *unstable data structure* configurations. Kernel-build merge performance with a 20 ms wake-up time after merging static sharing opportunities.

XLH shares almost the same amount of pages, regardless of the configuration. Only the linking phase, at the very end of the benchmark, shows a larger deviation between the three benchmark configurations. The main reason for this behavior is that I/O pages with the same content, that are later merged due to two hints, are inserted into the unstable tree in short succession. This decreases the likeliness for a concurrent unstable tree operation to degenerate this part of the tree in the meantime.

KSM's deduplication quantity, in turn, is affected severely by the degeneration of the unstable tree. To compare XLH and KSM more fairly we have used the "KSM RO" configuration in all our benchmarks instead of the vanilla KSM.

Conclusion of the Kernel-Build Benchmark First, we have further demonstrated that XLH can save much more memory than a linear memory scanner such as KSM can, at the same scan-rates. XLH especially excels in less aggressive scan-rate settings. We have determined that XLH achieves its superior deduplication ratio by exploiting additional short lived sharing as well as by prolonging the sharing time of sharing opportunities that can also be found by KSM by up to 4 minutes.

When evaluating the ratio of reachable nodes in the unstable tree, we have found a large proportion of the tree to be unreachable in all scan-rate settings. Our two solutions to the degenerating unstable tree problem differ only in the run-time performance which is evaluated in the next chapter. Here, we have seen, that it is mainly a problem for the vanilla KSM implementation. When keeping the unstable tree intact, the deduplication quantity stays almost the same for XLH.

5.4.2. Apache web-server and HTTPerf

HTTPerf [64] is a tool to measure the performance of web-servers such as the popular *Apache HTTP server* [7]. Serving static web-pages is latency sensitive. HTTPerf can evaluate this metric and enables us to compare XLH and KSM in this respect.

Benchmark Set-Up Figure 5.10 illustrates the set-up used for this benchmark. Apart from the benchmark logic and the memory scanner occupying one core each, the host runs two VMs; each executing the Apache HTTP server. An additional, physical computer runs two instances of the HTTPerf tool to measure the performance characteristics of both web servers. The virtualization host and the computer running HTTPerf are connected via gigabit Ethernet.

We have generated static files of 50 kB in size for the first web server and shuffled the order of those files for the second web server. This way, when the two HTTPerf instances access the files in the same order of file-names the contents are served in the shuffled order. We have used this approach to work around the lack of a feature in HTTPerf to access files in random order. The python code for the described generation of files is listed in Appendix B.

Deduplication Quantity The deduplication quantity is depicted in Figure 5.12. Alike the results in the kernel-build benchmark, XLH can constantly find and merge more sharing opportunities than KSM at all scan-rate settings. While KSM can only merge about 50 MiB memory at a 100 ms wake-up time, XLH can save almost 400 MiB memory; more than 8x as much. In reverse, XLH can save almost as much memory as KSM despite scanning 5x slower, as can be seen in Figure 5.11.

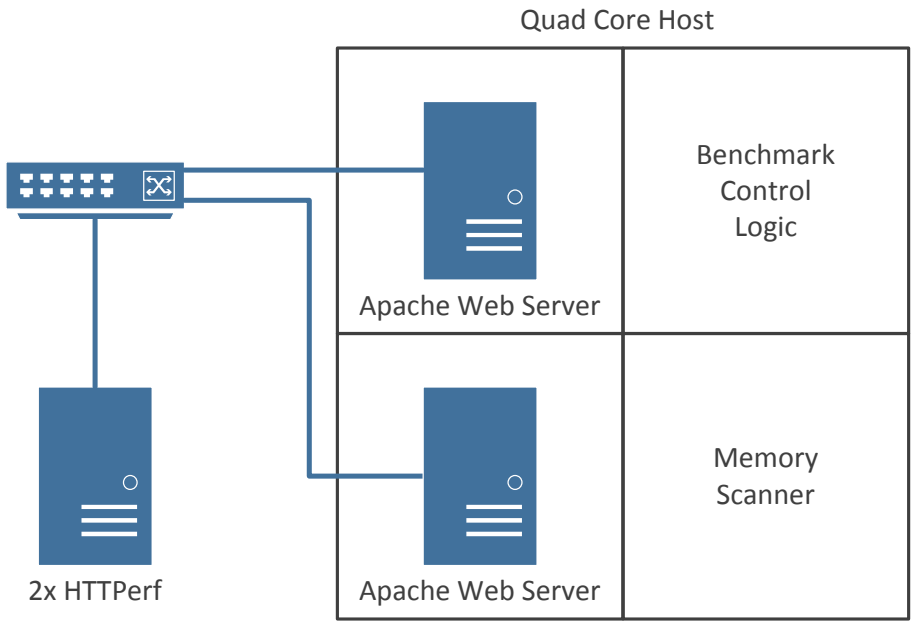

Figure 5.10.: Two physical computers are used for the HTTPerf benchmark. One running the two virtualized Apache web servers, the other running two instances of the HTTPerf benchmark.

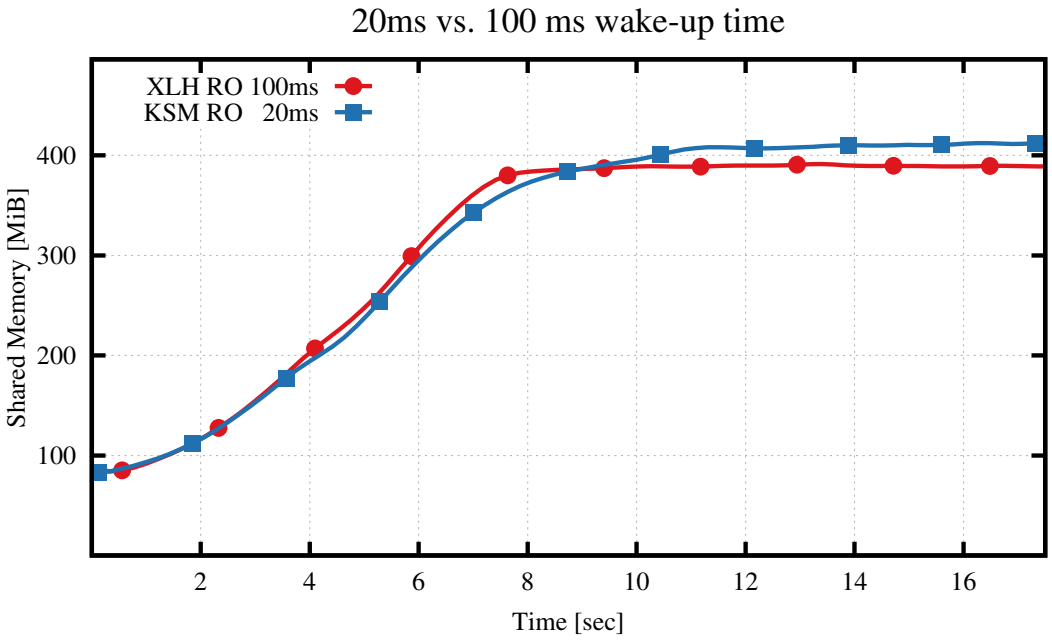

Figure 5.11.: Comparison when XLH scans 5x slower than KSM. KSM scans 5300 pages per second while XLH scans 1100 pages per second.

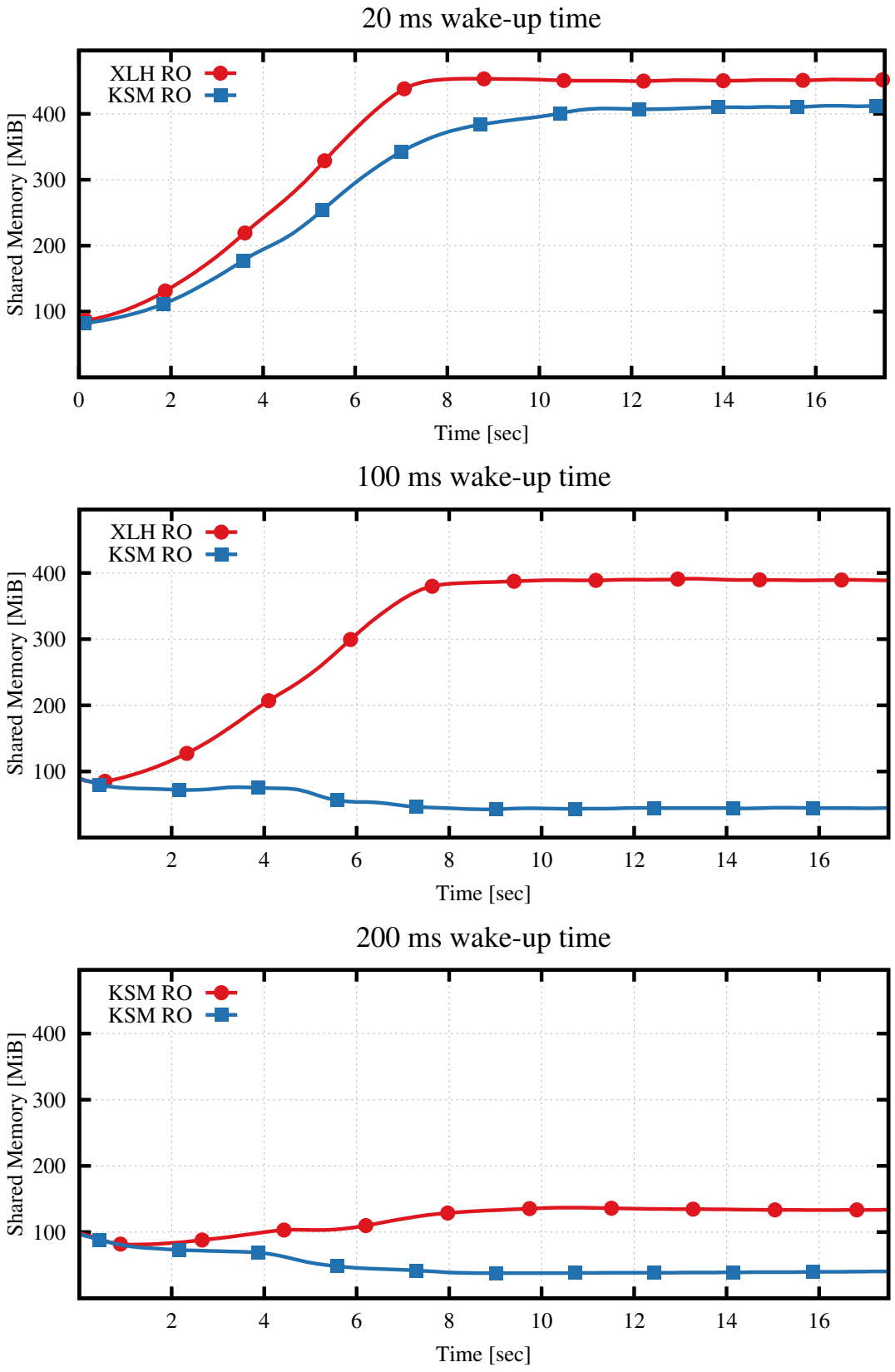

Figure 5.12.: Merge performance in two apache web servers serving static files to two HTTPerf instances requesting the same files in a different order. The scanner was warmed up, merging static sharing opportunities, before the workload started.

HTTPerf Latency HTTPerf directly outputs the average time it needs for connecting and transferring data from the tested web server. Figure 5.13 shows the 0.05 quantile, the median and the 0.95 quantile of the average transfer times of six HTTPerf runs in each configuration.

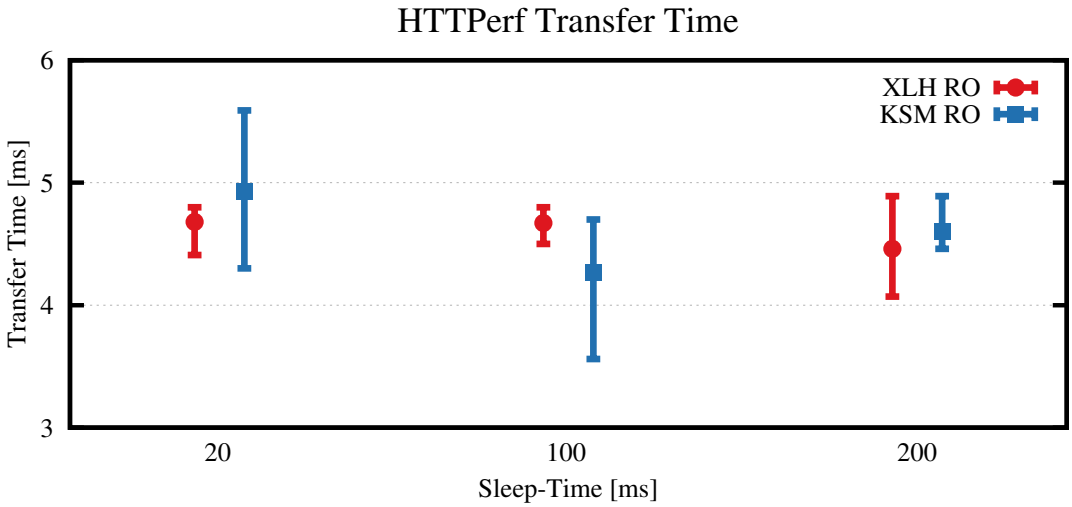

Figure 5.13.: Each bar represents the 0.05 quantile, median, and 0.95 quantile of the transfer time of 6 separate HTTPerf runs.

All benchmark runs resulted in transfer latencies between 3.5 ms and 6 ms. The variation is very low with a few outliers that are not fully filtered. The outliers are likely due to unrelated traffic in our internal network that was not exclusively used for the experiment. There does not seem to be a relation between the system used and the latency as XLH is sometimes at the faster end (20 ms, 200 ms sleep-times) and sometimes at the slower end (100 ms sleep-time). It also does not seem to be connected to the scan-rate in our settings; the 100 ms setting shows a slightly lower latency than the benchmarks in the other two sleep-times.

Conclusion of the HTTPerf Benchmark In the HTTPerf benchmark, XLH can deduplicate up to eight times as much memory as KSM at a sleep-time of 100 ms. XLH can deduplicate as much memory as KSM when KSM scans roughly 5 times faster than XLH (20 ms sleep-time). The transfer latency times do not show a clear trend which correlates the latency with our modifications or the scan-speed at the used settings.

5.4.3. Bonnie++

Bonnie++ [17] is a file system benchmark for UNIX. We use Bonnie++ to analyze the influence of our modifications of the I/O-Layer on the disk performance. We measure the total time that the Bonnie++ runs take. The throughput and access latencies are directly output in the benchmark results.

Benchmark Set-Up The benchmark set-up deviates from previous benchmarks in the respect, that Bonnie++ is only executed in a single VM to make the results more stable. If we had executed two Bonnie++ benchmarks at the same time, both instances would influence one another greatly as both try to saturate the background store.

The set-up is depicted in Figure 5.14. We have run 100 Bonnie++ benchmark runs and aggregated the results. A Hitachi `HDS72101` hard disk drive (HDD) posed as the benchmark target.

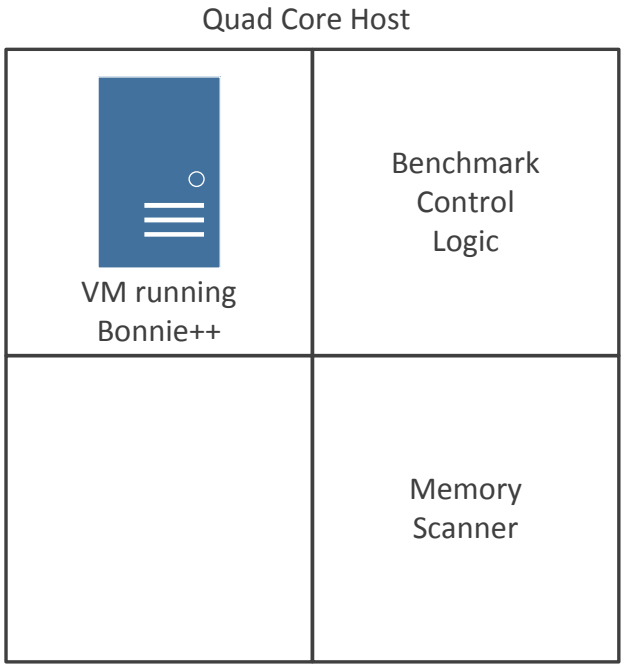

Figure 5.14.: A single VM runs the Bonnie++ benchmark.

Benchmark Run-Time The first graph in Figure 5.15 depicts the run-time distribution of 100 Bonnie++ runs. The marker in the middle represents the median, while the error bars show the 0.05 and 0.95 quantile. The dashed lines show the same quantiles for 100 Bonnie++ runs without deduplicating memory at all.

Figure 5.16 shows the sorted run-times of all 100 Bonnie++ benchmark runs that were used to generate the first graph in Figure 5.15. The run-times of about 60% of the benchmark runs are close together. Those runs take just below 12 seconds to run in total. A tail starts rising in the remaining part of the graph, maxing out at a run-time of about 17 seconds. There is no clear distinction between the minimum and mean run-times in the used configurations. Neither the scanner (XLH or KSM) nor the scan-rate seem to affect the result. The maximum run-time, however, is always slightly higher when using XLH over KSM in our experiments.

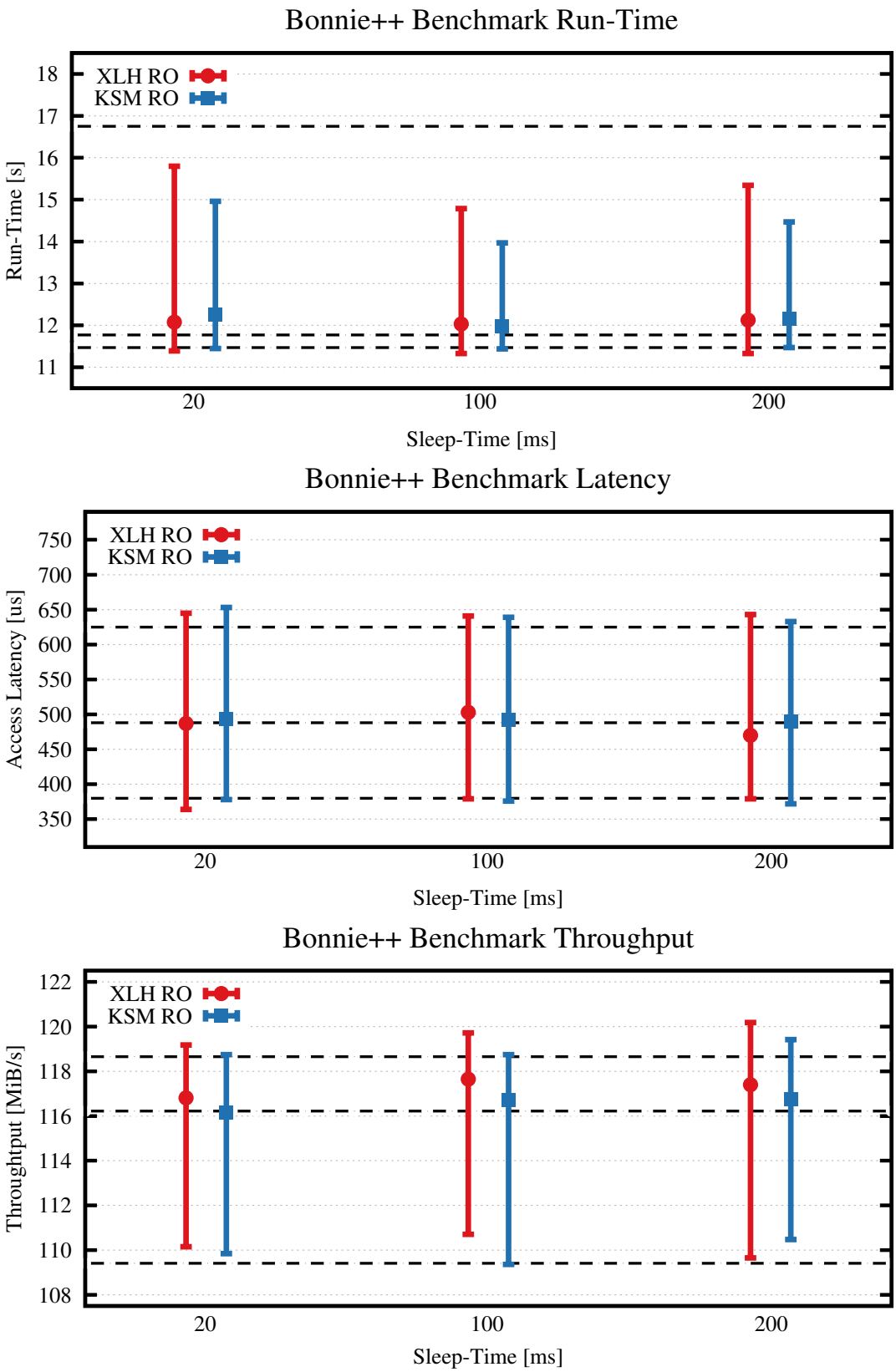

Figure 5.15.: Run-time, latency and throughput when reading block sequentially from HDD. 0.05 quantile, median, and 0.95 quantile of 100 Bonnie++ runs.

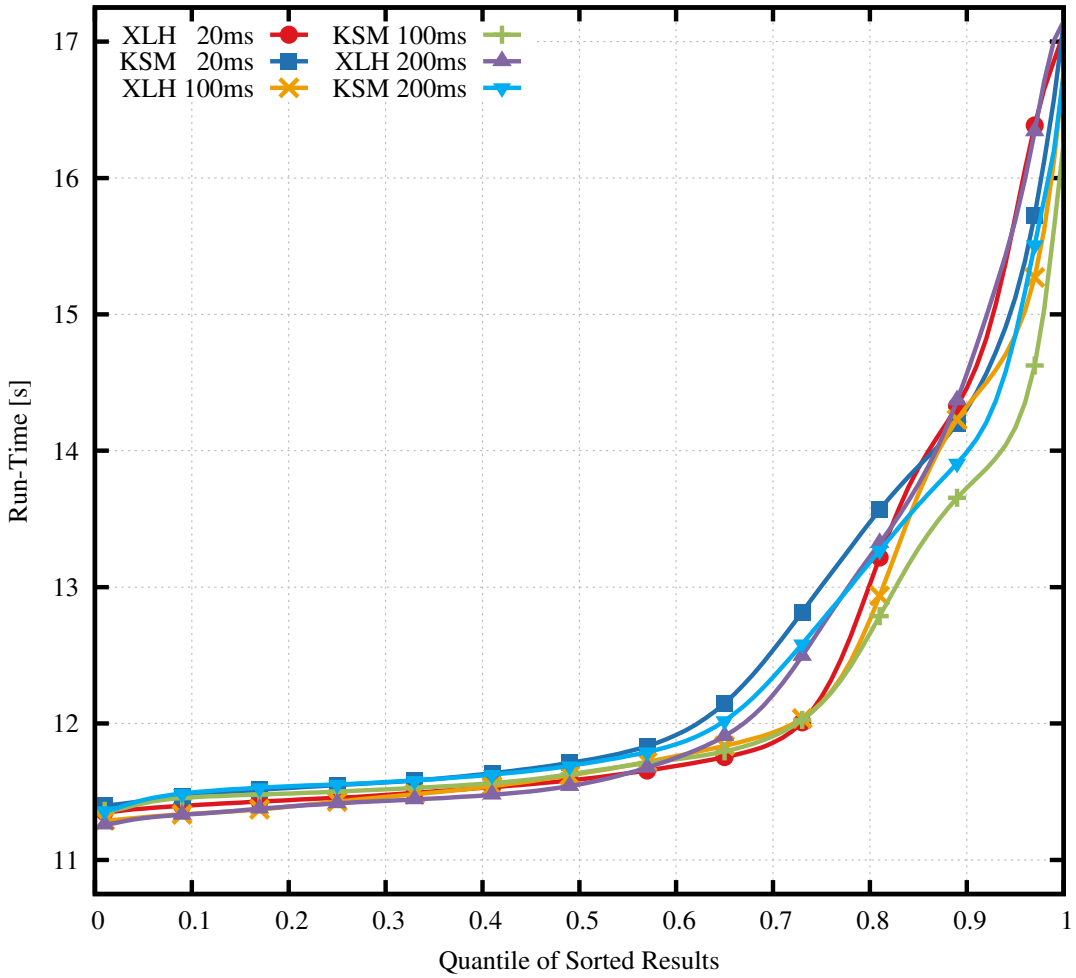

Figure 5.16.: Quantile plot of 100 Bonnie++ runs, sorted by time.

Latency The second graph in Figure 5.15 depicts the access latency when reading sequential blocks from the HDD. There is no clear trend that correlates the scan-rate with the mean access latency. All resulting graphs are very close together with little variation. The additional hint-storage in the I/O path is not noticeable in the disk access latency.

Throughput The third graph in Figure 5.15 shows the throughput that Bonnie++ measures when reading blocks sequentially. There seems to be a trend for XLH to get a slightly higher throughput than KSM. This is surprising, as the benchmark run-time and latency do not reflect this trend.

The benchmark runs are very short which limits the accuracy of the results. In addition, the difference is only between 3‰ and 8‰ of the total throughput. We neglect this result as the difference can as well be due to imprecision in the measurement [86].

Conclusion of the Bonnie++ Benchmark XLH adds code to the I/O layer that records every read and write operation. Using Bonnie++, however, we were not able to measure a clear performance difference, neither between KSM and XLH, nor between the two and the base system without memory deduplication.

5.4.4. Mixed

In this scenario we compare KSM and XLH in a set-up that exhibits only few sharing possibilities while many, mostly bogus hints are produced. We do this by mixing workloads. Then, apart from the same guest OS, code and data of the workloads are almost distinct.

Benchmark Set-Up In this benchmark we mix the set-ups of two previously presented workloads. As depicted in Figure 5.17 one VM runs a Linux kernel-build. Another VM runs the Apache web server that interacts with an HTTPerf instance running natively on a different physical host. The HTTPerf instance and the virtualized host are connected through a gigabit Ethernet connection. Compared to the first HTTPerf benchmark, we have modified the HTTPerf settings to shorten the benchmark run-time in order to match the kernel compilation time.

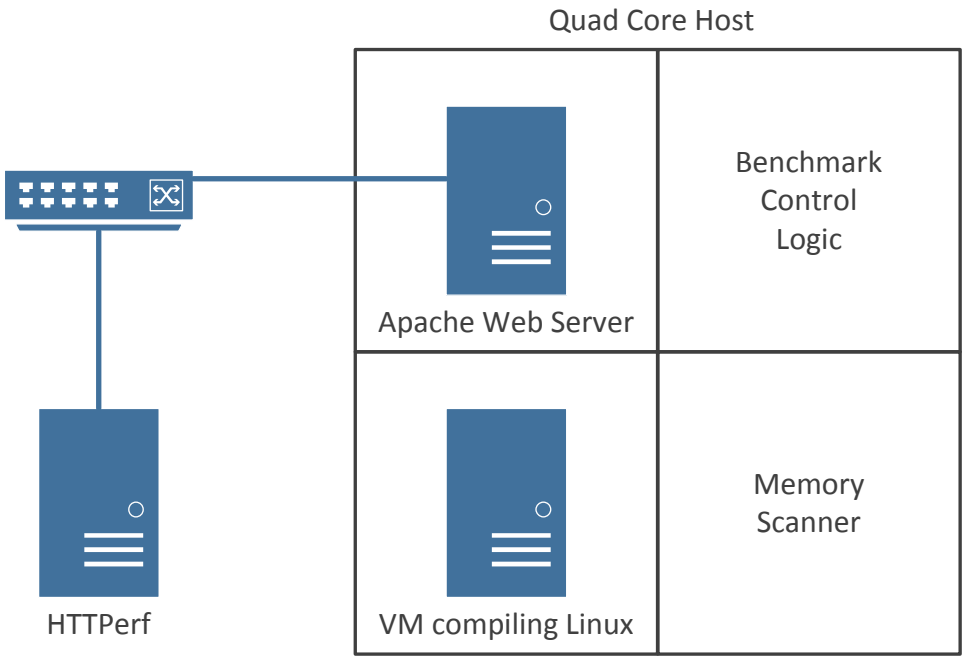

Figure 5.17.: Two physical computers are used for the mixed benchmark. One running a virtualized Apache web server and a kernel-build within another VM. The other computer is running an instance of the HTTPerf benchmark.

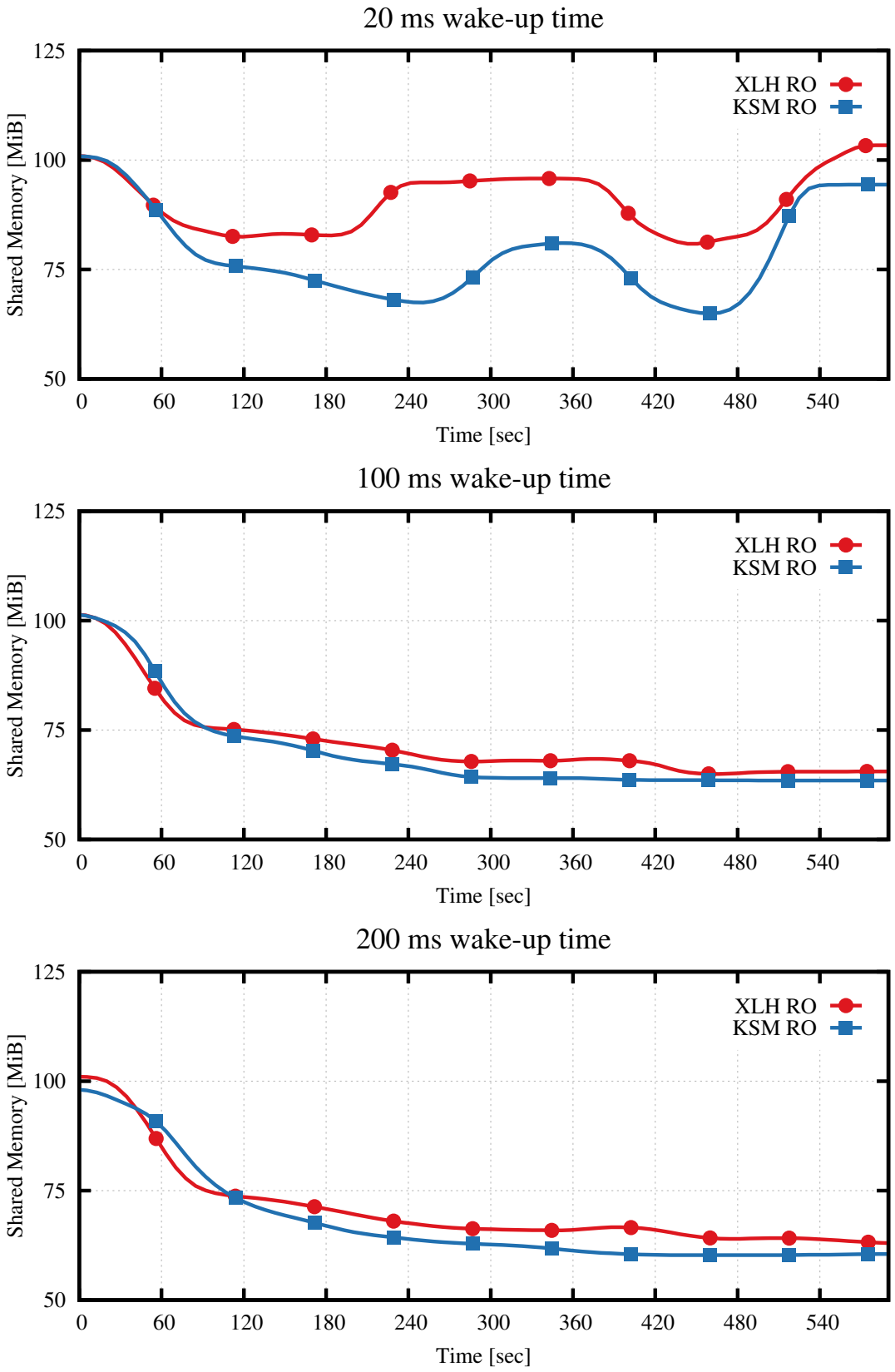

Figure 5.18.: Merge performance of simultaneous kernel-build and an HTTPerf workloads with varying wake-up times. The scanner merges static sharing opportunities, then the workload starts.

Results Figure 5.18 shows the deduplication quantity for three sleep-times. For sleep-times of 200 ms and 100 ms both, KSM and XLH, share practically the same amount of memory. Surprisingly, XLH can save up to 20% more memory than KSM in the benchmark with 20 ms.

The additional sharing comes from self-sharing in the kernel-build benchmark. When compilation results are flushed to disk, XLH can exploit the duplicates between anonymous and file memory before the anonymous memory region is reclaimed and reused by the OS as previously presented in §3.2.3.

Conclusion of the Mixed Benchmark Mixing workloads can make hints useless for inter-VM sharing. In our benchmarks, however, we did not see a decrease in sharing quantities when using XLH in this scenario as could be expected as XLH needs to follow many bogus hints. Instead, XLH can rise the deduplication quantity slightly by finding self-sharing opportunities more quickly.

5.5. Conclusion

In the past decade, there has been a trend to migrate servers into virtual machines. Although this strategy leads to a better resource utilization and flexibility, it introduces a bottleneck in the main memory size. Memory scanners can increase the memory density of virtual machines by finding duplicates in guest memory and sharing them in a copy-on-write fashion. Memory scanners such as VMware ESX and Linux' KSM identify duplicate memory pages by indexing all pages in linear or random order without regard to the memory semantics. XLH complements this technique by prioritizing pages of recent I/O activity.

In this chapter we have shown that XLH can share up to eight times more memory than previous scanners in typical virtualization scenarios at the same scan-rate. XLH achieves this increased sharing rate by finding sharing candidates between one and four minutes earlier than linear memory scanners and by finding additional sharing opportunities that could previously not be detected.

The increased sharing quantity comes at no measurable performance impact compared to KSM at the same scan-rate. Even in non-favorable scenarios, when most hints are bogus, XLH does not perform worse than KSM.

Chapter 6

Performance Considerations

Until now we have resolved that extending memory scanners with I/O-based hints can save more memory than pure brute-force, linear memory scanning at the same scan-rate settings. We have also shown that the performance in terms of benchmark run-time, I/O-latency, and I/O-throughput are barely affected by our extension.

This chapter analyzes the runtime effects of memory deduplication scanners more closely. We approach the problem from two different perspectives: From the scanner's point of view and from the application's point of view. Section 6.1 analyzes the work that duplication scanners put into the deduplication process. The following Section 6.2 elaborates on positive as well as negative effects that searching for and sharing equal pages has on workloads of different runtime characteristics. Section 6.3 concludes this chapter by summarizing the main results.

6.1. Scanning Overheads and Boundary

In this section we analyze the work that a memory scanner puts into the deduplication process directly. We begin by going through the KSM scan process and identifying how much time each part of that process takes to execute (§6.1.1). We also analyze the time that XLH spends on recording and retrieving hints in that paragraph. The next paragraph (§6.1.2) shows how often the code paths are executed in total and in relationship to each other. The section closes with an exploration of the scan-rate boundary, the maximum rate at which KSM can visit pages (§6.1.3).

6.1.1. Code Paths

We first measure how long each code path in KSM (vanilla Linux 3.4-rc3) takes. We have gathered this information by running a micro-benchmark in which we initialized the to-be-scanned memory in a way that forced the scanner to run solely along certain paths in the first scan rounds. While conducting the benchmark, we measured the time spent in the respective functions using *Ftrace*.

Ftrace *Ftrace* is Linux' internal function call tracer that provides information about function call frequencies, execution times, interrupts, preemption, and more. If Ftrace is enabled during kernel compilation, the compiler inserts the `mcount` instruction in front of every non-static function [49].

The space occupied by `mcount` is replaced by nop instructions when tracing is disabled. If, however, tracing is enabled, the `mcount` instruction acts as a trampoline to analysis code. The analysis code records the current time using the `Time Stamp Counter` (TSC) register and replaces the return address of the function with the `mreturn` trampoline that is used to record the elapsed time difference. Gathered statistics are output via the debug file system (debugfs).

Benchmark Set-Up The data region that is scanned in this benchmark is initialized as depicted in Figure 6.1. In the first scan round, the hash is calculated but the page is not indexed, yet. The data is initialized to contain distinct data in the first half of the scanned VMA. The second half contains an exact copy of the first half of the data. Thus scanning the first half of the data in the second scan round will result in adding all pages to the unstable tree, as all hashes are still valid. When scanning the second half of the data region, every page already has a sharing partner in the unstable tree and is thus merged.

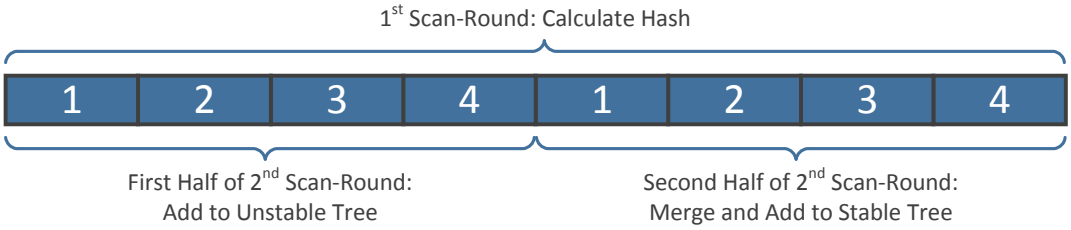

Figure 6.1.: The data-initialization that was used to force the scanner to run along known paths.

From the number of pages that we have indexed we know in which phase of the scan we are at every moment during the benchmark. This way we can differentiate between successful and unsuccessful tree lookups.

KSM Path Costs The resulting path-times can be found in Figure 6.2. For comparison: In our set-up it takes around 600 ns to read 4 KiB linearly from main memory. An LLC miss that fetches a cache-line from main memory takes around 50 ns [20].

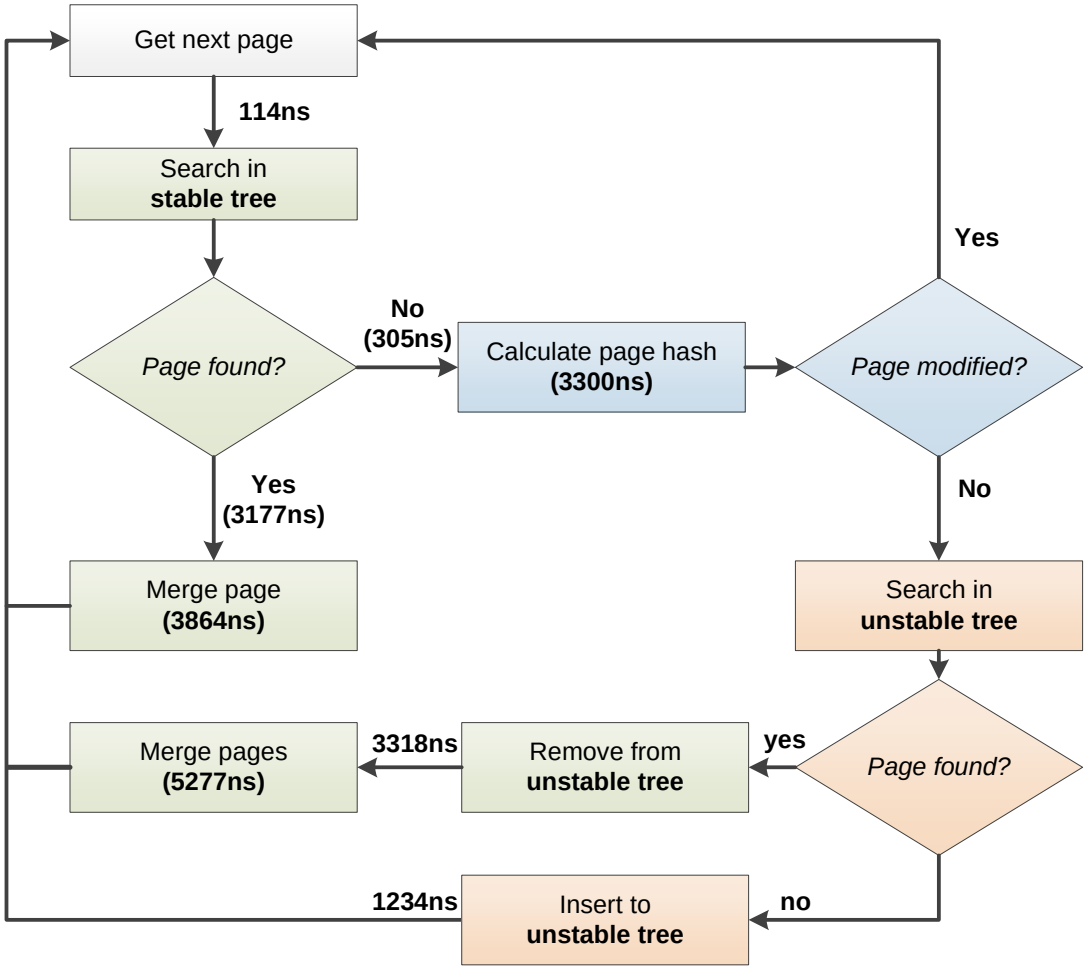

Figure 6.2.: The execution time of each sub-path of the scan process.

The times for stable and unstable tree look-ups differ greatly whether an equal page is already in the respective tree or not. If the page is not in the tree, the (lazy) comparison finishes much earlier in the look-up process. For this reason, we have annotated the outgoing edges for tree lookups instead of the nodes which we have annotated for the other operations. We do not annotate the pure look-up for the unstable tree, as the single look-up function already performs the next action – remove or insert – in compound. This avoids either locking the tree until the insertion takes place, or a second traversal of the tree.

Unstable Tree vs. Hash-Table Lookup Speed Our benchmarks have shown that there is a great difference between unsuccessfully looking up a page in the unstable tree (305 ns) and finding a page in the unstable tree (3177 ns).

This difference comes from the additional number of pages that need to be compared to the page that we are looking for, while traversing the tree, in case of a match. In addition, the whole page needs to be finally compared to make sure, that the pages are in fact equal. When the page is not in the unstable tree, the comparison can cease at the first differing byte in the leaf node.

When replacing the unstable tree with a hash-table, the look-up time is constant, as long as there is enough room in the table and no chaining is required. A well-tuned hash-table should therefore reduce the successful look-up time significantly, in the common case.

Figure 6.3 depicts the 0.05 quantile, the median and 0.95 quantile of the time required to look-up (and insert) a page in our hash-table implementation as well as the time to remove a page from the data structure. The numbers are generated using ftrace recording 56605 samples for the look-up and 62117 samples measuring the removal.

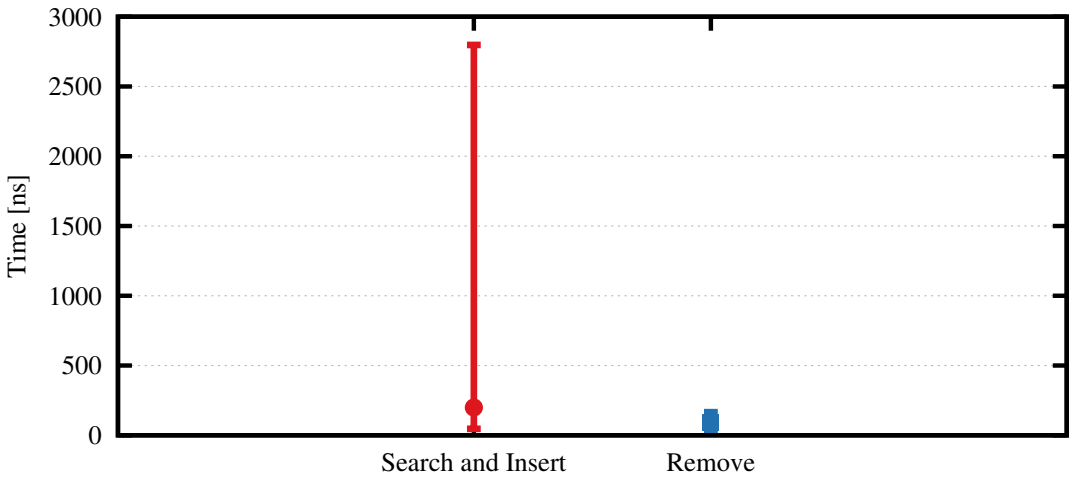

Figure 6.3.: Time to *search and insert* into the unstable hash-table and the time it takes to *remove* a page from the unstable hash-table, respectively. The bars show the 0.05 quantile, the median and the 0.95 quantile.

As expected, the common time to look-up and insert a page in the hash-table is shorter (199 ns) than when using the original unstable tree implementation (1234 ns). The difference is even larger for the removal from the unstable data structure. When a hash-table is used removal takes 91 ns in the median while we have measured 3318 ns for the unstable tree.

XLH's Bounded Circular Stack XLH extends the original KSM code by the bounded circular stack used for storing hints. Table 6.1 shows the execution times for adding a hint into and for removing a hint from the stack respectively. The additional push and pop operations are very fast, compared to the original KSM operations.

Property \ Operation	Push	Pop
Median	38 ns	111 ns

Table 6.1.: Execution times of bounded circular stack push and pop operations.

XLH's `mm_slot` modification The main difference between processing hints and following the linear scan is the choice which page to visit next. When processing hints, the next page to visit is taken from the bounded circular stack instead of following a pointer in the `rmap_item`. Then, a more costly red-black tree look-up is performed to find the `rmap_item` that was referenced by the hint within the `mm_slot` data structure.

Figure 6.4 depicts the time to extract the next `rmap_item` to process, when scanning and when following a hint. In our experiments it took a median time of 592 ns for the hint path and a median time of 114 ns for the scan path to return the next `rmap_item` to be processed. Compared to the time it takes to calculate the hash value of the visited page (3300 ns) or merging pages (3864 ns), getting the next page to visit does not seem to be a problem that is important to address at the moment.

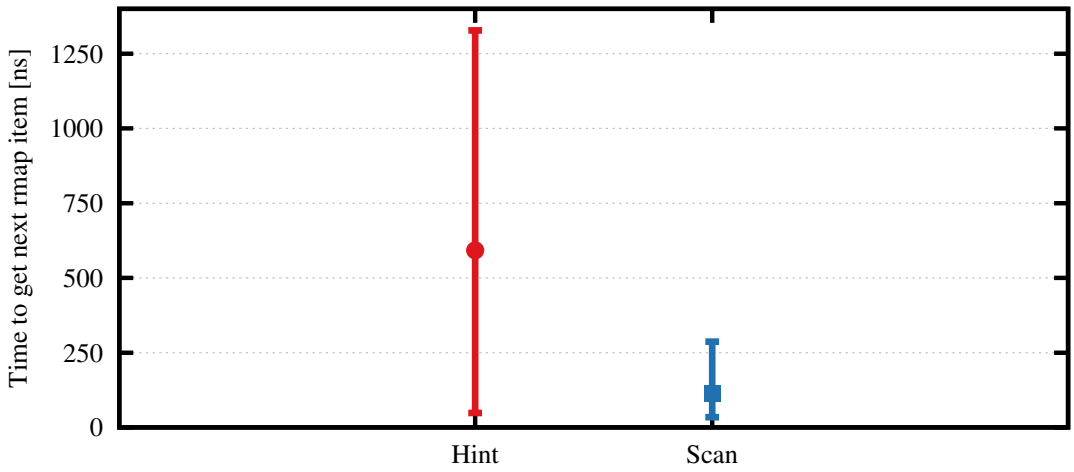

Figure 6.4.: Time to find the `rmap_item` for the currently processed hint or the time to find the next `rmap_item`, respectively. The bars show the 0.05 quantile, the median and the 0.95 quantile of 100 k samples.

We have run all of our benchmarks with this naïve implementation. It is, however, possible to extend the approach by caching the tree look-up in the `address` field of the `page` data structure. Then, every distinct `rmap_item` only needs to be looked up using the red-black tree once in O(log n) before it can be found in O(1) from the cache. This caching mechanism is prone to programming errors but can decrease the look-up performance to the same time that the scan path takes.

6.1.2. Code-Path Frequencies and Aggregated Cost

The absolute time it takes to execute each path in the KSM scan process and the time spent in the XLH extensions is meaningless without information about how often each code path is taken in a typical workload.

Figure 6.5, Figure 6.6, and Figure 6.7 show the relative distribution of the frequencies that each code-path is taken when running the kernel-build benchmark and scanning 100 pages per wake-up with sleep-times of 20 ms, 100 ms, and 200 ms respectively. There is a clear tendency that can be taken from the progression of the graphs: The more aggressive the scanner operates, the more likely it is that pages have the same content in successive visits. This is in line with intuition: if the modification frequency remains constant, the times at which pages still have the content from the previous visit rises if the frequency at which the pages are visited increases. It impacts the deduplication effectiveness significantly if many pages never make it into the unstable tree.

The *total number* how often each path has been taken in the entire kernel-build benchmark and the *aggregated path cost* (Equation 6.1) for all three sleep-times are given in Table 6.2.

$$\text{aggregated path cost} = \#\text{occurences} \cdot t_{\text{execution}} \tag{6.1}$$

Path \ Sleep-Time	20 ms	100 ms	200 ms
Merge to Stable Tree	20,465 (144 ms)	4,755 (33 ms)	1,190 (8 ms)
Differing Hashes	1,161,046 (4,186 ms)	447,126 (1,612 ms)	197,024 (710 ms)
Merge from Unstable Tree	47,931 (585 ms)	4,660 (57 ms)	782 (10 ms)
Add to Unstable Tree	1,442,546 (6,980 ms)	84,584 (409 ms)	8,090 (39 ms)
Total	2,671,988 (11,895 ms)	541,125 (2,111 ms)	207,086 (767 ms)

Table 6.2.: Number of times each path was taken and the resulting aggregated path costs in the kernel-build benchmark which runs around 7:30 minutes.

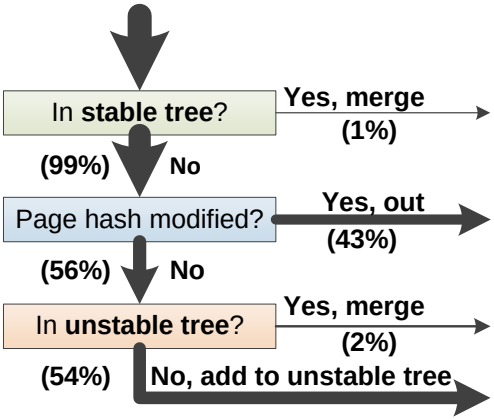

Figure 6.5.: Paths taken by visited pages through the KSM memory scanner when scanning 100 pages per wake-up and waking up every 20 ms.

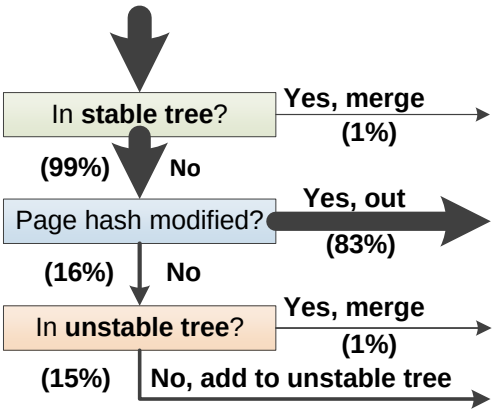

Figure 6.6.: Paths taken by visited pages through the KSM memory scanner when scanning 100 pages per wake-up and waking up every 100 ms.

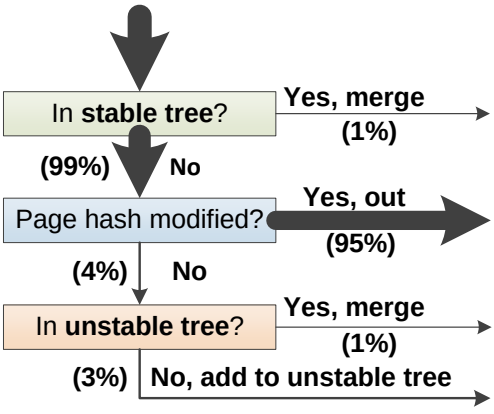

Figure 6.7.: Paths taken by visited pages through the KSM memory scanner when scanning 100 pages per wake-up and waking up every 200 ms.

The total time spent visiting pages that fluctuate frequently (differing hashes) and the time spend inserting pages into the unstable tree (add to unstable tree) dominate in all three scan-rate settings. This suggests, that it is a good idea to prioritize pages in the scanner that have a higher chance to lead to a merge.

Produced Hints The number of *produced* hints depends only on the I/O activity of the workload. Its rate is independent of the scan-rate, as the hinting mechanism is decoupled from the scan process. The accumulated number of produced hints, recorded from the beginning of the workload, after booting, is depicted in Figure 6.8.

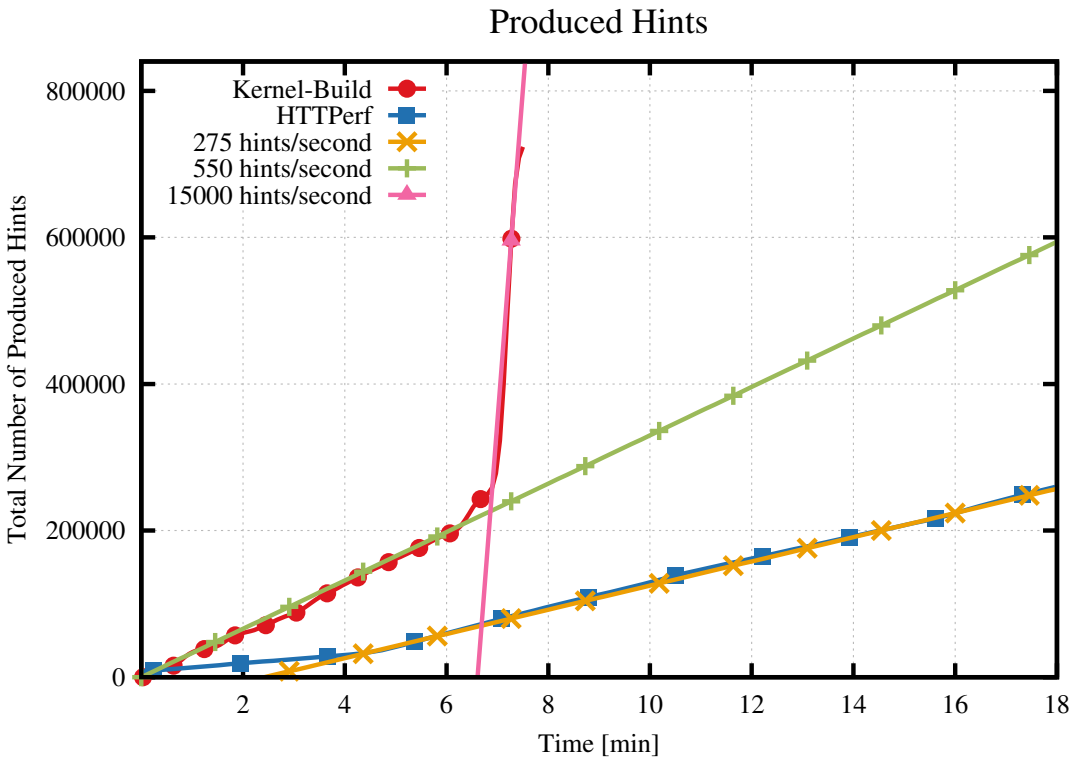

Figure 6.8.: Number of hints produced in the kernel-build and HTTPerf benchmarks.

Excluding the boot-process of both VMs, a full kernel-build benchmark produces around 750 k hints in the I/O subsystem, resulting in the same amount of bounded circular stack push operations. The kernel-build benchmark has two phases, compilation and linking. The compilation phase, which reads all source files and creates an object file for each module, produces around 550 hints per second. The linking phase reads all previously created object files and links it to the Linux kernel image. This phase requires less computation and has in consequence a higher I/O rate, producing around 15000 hints per second.

The I/O rate in the Apache web servers is dependent on the rate at which the HTTPerf tool requests files. In our benchmark XLH produces around 275 hints per second resulting in a total of around 250 k produced hints.

The total work that XLH puts into generating and pushing hints into the bounded circular stack is summarized in Table 6.3. In total, XLH needs less than 0.07‰ of the CPU time for inserting hints in the kernel-build benchmark and less than 0.01‰ of the CPU time in the HTTPerf benchmark. This overhead is negligible in the deduplication process.

	Produced Hints	**Time Spent Inserting Hints**	**Run-Time**
Kernel-Build	746,450	28 ms	≈ 7:30 min
HTTPerf	266,736	10 ms	≈ 18:00 min

Table 6.3.: Total amount of hints inserted into the bounded circular stack and total time spent pushing hints until the end of the benchmark run-time.

Consumed Hints The number of *consumed* hints and accordingly the number of pop operations, and `mm_slot` tree searches is highly dependent on the scan-rate and interleaving ratio of the scanner, and on the size of the stack. This is because old entries are overwritten when the stack is full and new hints are pushed. As long as hints are present, XLH withdraws them at a constant rate. The rates at which hints are consumed throughout the kernel-build and HTTPerf benchmarks in our default settings are depicted in Figure 6.9.

The total time that XLH spends inserting hints into the bounded circular stack and taking hints out of the stack is given in Table 6.4. The CPU overhead for consuming hints is below 0.04‰ in both benchmarks for a sleep-time of 20 ms.

Benchmark \ **Sleep-Time**	**20 ms**	**100 ms**	**200 ms**	**Run-Time**
Kernel-Build	163,454 (18 ms)	85,300 (9 ms)	55,017 (6 ms)	≈ 7:30 min
HTTPerf	173,361 (19 ms)	118,592 (13 ms)	78,039 (9 ms)	≈ 18:00 min

Table 6.4.: Total amount of hints extracted from the bounded circular stack and total time spent taking out hints until the end of the benchmark run-time.

For comparing the overhead that XLH has caused using the transition from an `mm_slot` list to an `mm_slot` red-black tree we use the *aggregated get next page time* (Equation 6.2).

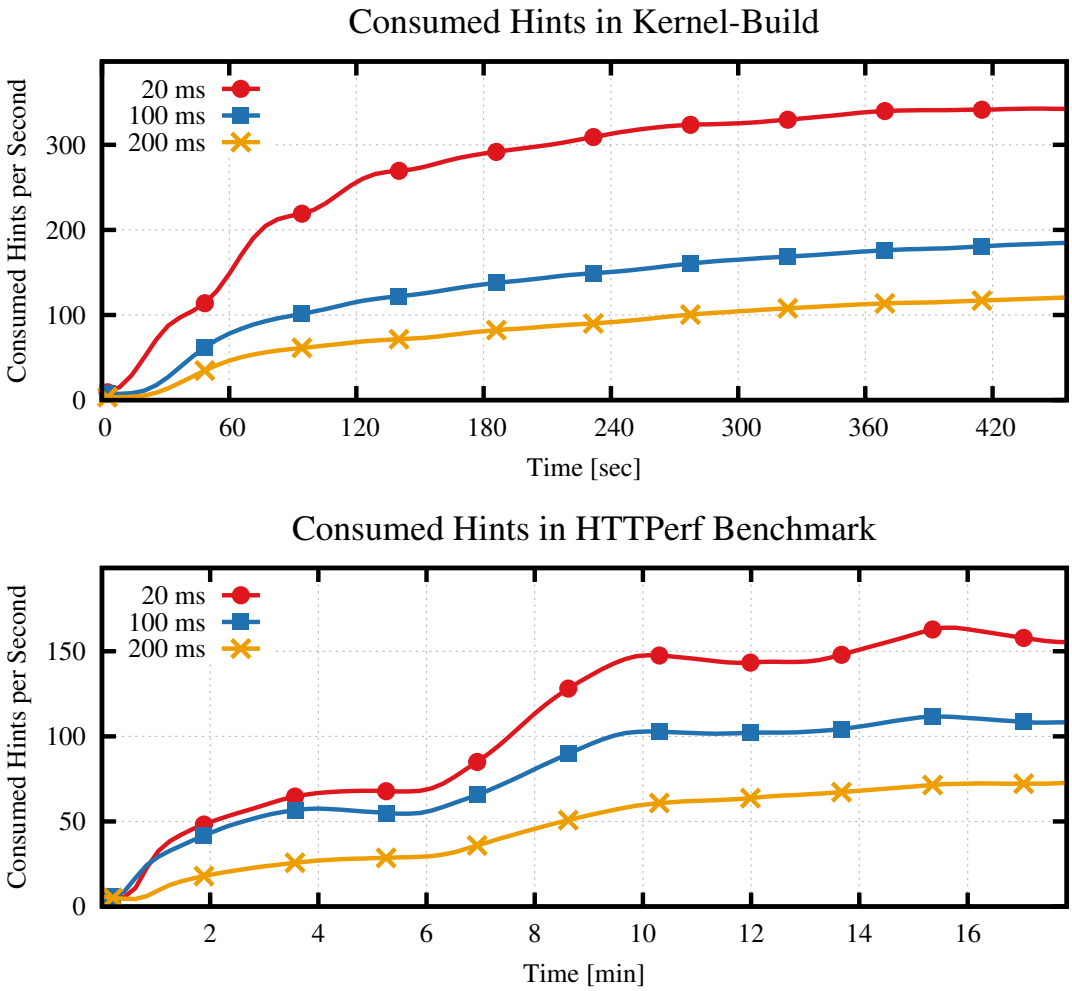

Figure 6.9.: Rate at which hints are consumed in the kernel-build and HTTPerf benchmarks.

$$\begin{aligned}\text{aggregated get next page time} = \#\text{scanned pages} \cdot t_{\text{linear page lookup}} + \\ \#\text{consumed hints} \cdot t_{\text{random page lookup}}\end{aligned} \tag{6.2}$$

Table 6.5 lists the aggregated get next page times for the kernel-build and HTTPerf benchmarks in different sleep-times. Each number of consumed hints and number of scanned pages that we have used for the calculation is the average of 6 benchmark runs. The times that linear and random page lookups take are the median numbers presented in Figure 6.4.

The absolute speed difference between linearly getting the next page to scan (114 ns) and getting a random page that belongs to a hint (592 ns) differs by a factor of 5.2x in the median. In a real workload, the average speed difference for getting the next page to visit is not as severe, between XLH and KSM, as the absolute numbers

indicated. XLH is only between 16% and 53% slower than KSM when getting the next page to process. This is because there is a mix of both operations in XLH and there are generally less hinted pages than scanned pages.

6.1.3. Scan-Rate Boundaries

There is a limit for the speed at which a system can scan memory pages for duplicates. To find this limit in the vanilla KSM memory scanner, we have run the following benchmark: First we have initialized 4 GiB of page-aligned memory with random memory contents and `madvised` this memory to be mergeable. Then we have run the memory scanner with varying settings for 240 seconds. In each 240 second interval we have measured how many pages were visited every second. We have then used the median of those 240 values as the number of pages that can be scanned per second in this setting.

The scan-rate is set through two variables (Figure 6.10). The *sleep-time* determines how long the memory scanner is inactive between scan spurts. The *pages-to-scan* setting determines how many pages are scanned at each wake-up.

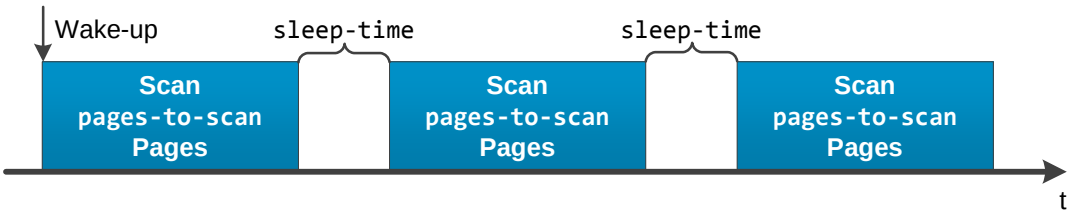

Figure 6.10.: Wake-ups do not occur strictly periodically. Instead the KSM daemon sleeps for a defined time between the *end* of the previous scan-spurt and the next wake-up. The time to scan *pages-to-scan* pages is variable.

Benchmark \ Sleep-Time	20 ms	100 ms	200 ms	Run-Time
XLH Kernel-Build	541 ms	200 ms	146 ms	≈ 7:30
KSM Kernel-Build	380 ms	134 ms	102 ms	≈ 18:00
XLH HTTPerf	926 ms	359 ms	220 ms	≈ 7:30
KSM HTTPerf	797 ms	235 ms	159 ms	≈ 18:00

Table 6.5.: Aggregated get next page time for XLH and KSM in the HTTPerf and Kernel-Build benchmarks.

We have first fixed the pages-to-scan setting at 100 pages and varied the sleep-time between 2 ms and 256 ms. The resulting numbers of pages that are scanned over the course of a second at those settings are depicted in Figure 6.11. The first 3 points, at 2 ms, 4 ms and 8 ms result in the same scan-speed. Around 10 ms is the shortest sleep-time that Linux can handle with the default slice-time length.

Then we have fixed the sleep-time at 10 ms and varied the pages-to-scan setting between 100 and 2600. The result is depicted in Figure 6.12. The number of pages that can be scanned in a second approaches around 385 MiB per second, asymptotically. In our benchmark set-up KSM cannot exceed this scan-rate.

Two things put these numbers into perspective. First, in a system with 32 GiB RAM a full scan round takes at least 1:25 minutes at the most aggressive possible scan-rate. With the two visits that are required to find a new sharing opportunity pair, new sharing opportunities are detected after an expected time of 2:13 minutes. Second, the most aggressive scan-rate has a large run-time effect on the scanned workload.

The effective scan-rate resulting from our default pages-to-scan and sleep-time settings can be obtained from Table 6.6.

Sleep-time [ms]:	20	100	200
Pages-to-scan:	100	100	100
Effective scan-rate [pages/s]:	5,300	1,100	530
Effective scan-rate [MiB/s]:	20.7	4.3	2.1

Table 6.6.: Effective scan-rate in our default benchmark settings.

6.2. Runtime Effects of Page Sharing

This section evaluates the overhead which applications incur when competing with a memory scanner for shared resources. First, we quantify the effect that rising scan-rates have on the run-time of concurrently running applications (§6.2.1). Second, we analyze the impact of additional page-faults and the copy-on-write overhead on the run-time of deduplicated applications (§6.2.2). Third, we evaluate the potential of main memory deduplication to increase the read speed and to lead to additional space in the CPU caches (§6.2.3).

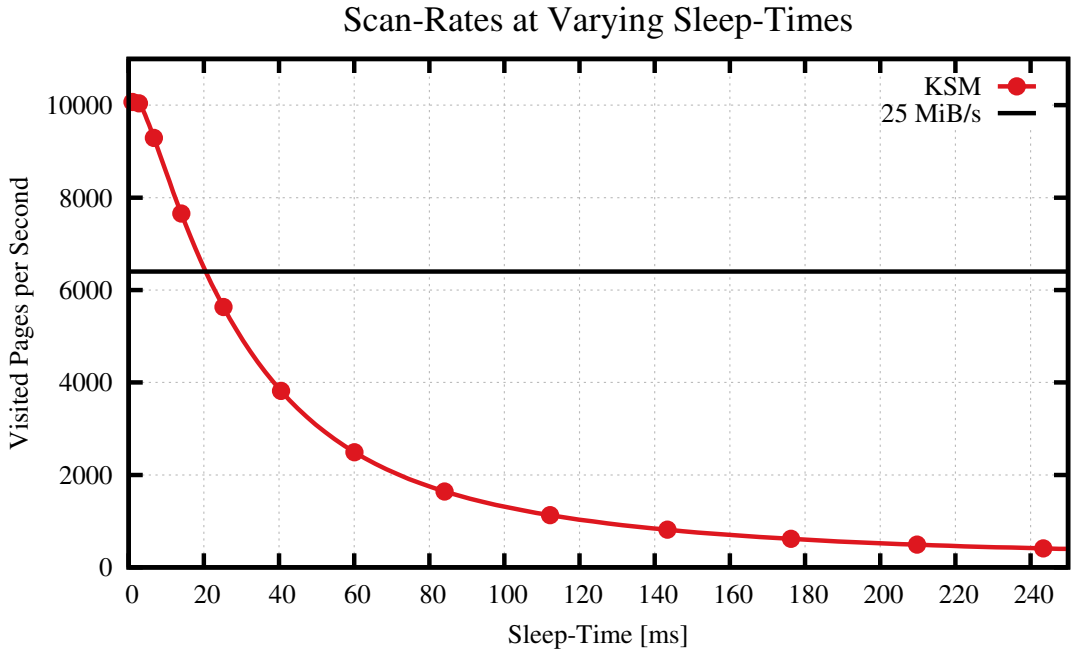

Figure 6.11.: Visited memory pages per second at varying sleep-times. The number of pages that are scanned at each wake-up is fixed at 100.

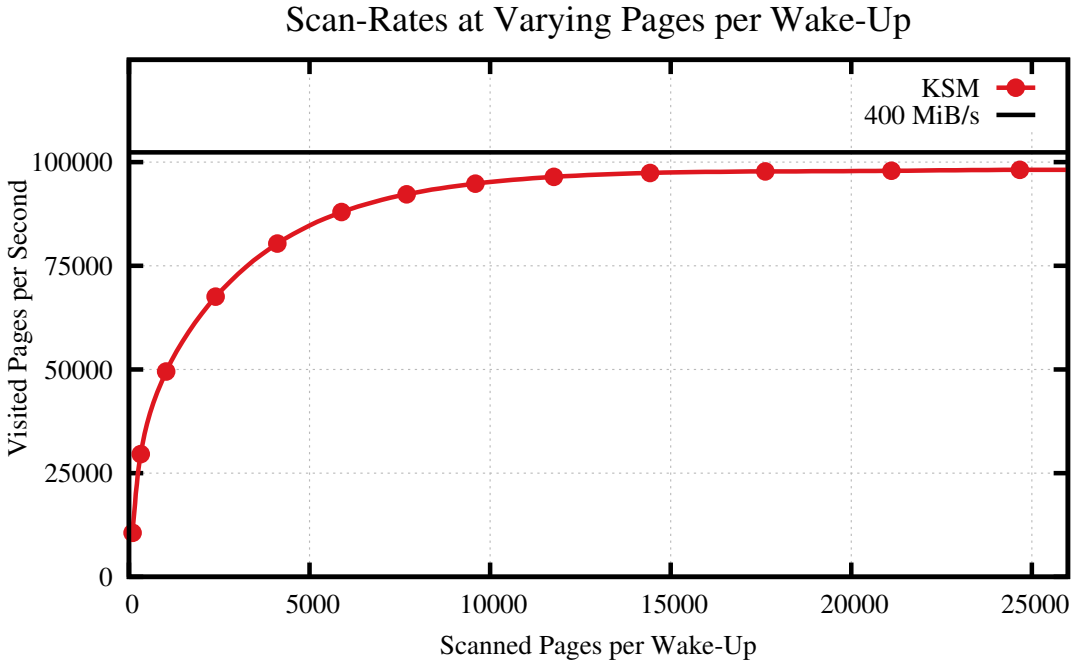

Figure 6.12.: Visited memory pages per second at varying pages-to-scan settings. The sleep-time is fixed at 10 ms.

CPU Overhead In accordance to the current trend that the number and speed of CPU cores grows faster than the size and speed of memory, we generally assume that the host has free CPU resources available. If the memory scanner runs on a shared CPU, it can be run at a low priority. This way, the scanner yields to a concurrently running CPU bound task. Linux, for example, runs the KSM daemon at a niceness value of 5[1].

6.2.1. Run-Time vs. Scan-Rate

KSM needs to scan at a higher scan-rate to find more sharing opportunities. To show that this causes slowdowns for the workload we have measured the run-time of the kernel-build benchmark when scanning, but *without actually merging memory*.

Figure 6.13 depicts the kernel-build run-times at a sleep-time of 20 ms at different pages per wake-up settings. The run-time clearly rises with higher scan-rates. Visiting 100 pages more per wake-up causes the kernel-build to run roughly 1 second ($\approx$ 2.5‰) longer in the median. This result is not directly comparable to the previous Figure 5.8, as the previous run-time was measured after warming up the scanner. Moreover, the previous benchmark also included the merge and copy-on-write overheads.

The only three ways in which the scanner influences the workload on a second core are memory bus contention, shared LLC cache thrashing and L1/L2 cache coherency protocol overheads.

6.2.2. Writing and Breaking Sharing

When shared pages are written, the page-fault handler creates a copy of the page frame and then remaps the written page to point to the new page frame in the involved page tables[2]. In consequence, the TLB entries and cache lines that point to the old page frame need to be invalidated.

Worst Case Write Performance Degradation To measure the worst case slowdown due to copy-on-write operations, we have set-up a micro benchmark that records the throughput when every write operation leads to a page-fault.

[1] On a scale from -20 to 19 where regular processes run at 0. Higher values mean a lower priority.

[2] Multiple address spaces can share the same page read-write. If the resulting page frame is transparently merged and later broken, all semantically related pages need to be remapped to point to the same copy.

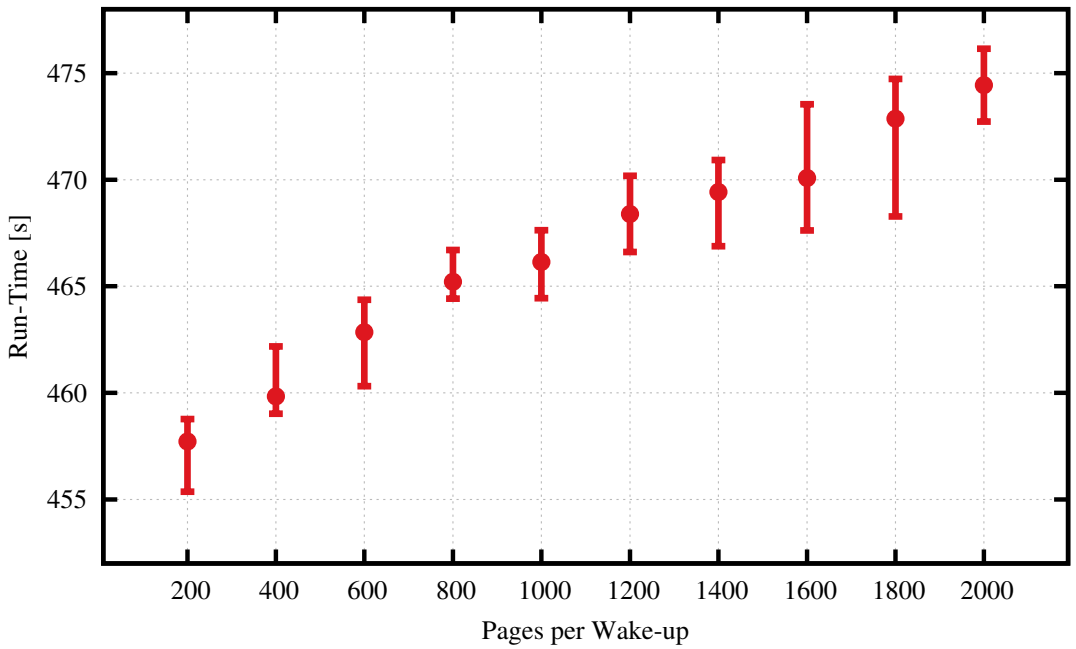

Figure 6.13.: Run-time of kernel-build while scanning, but not merging, VM memory with KSM (with writable unstable tree pages). KSM was configured with a wake-up time of 20 ms. The bars show the run-times of 3 kernel-build runs.

To this end, we have written a program that allocates 1 GiB contiguous memory. Then we have `fork`ed to create a second address space sharing the first in a copy-on-write fashion. Following this, the program executes a tight loop, writing 8 bytes (one 64 Bit integer) into each page. While writing, we sampled the progress and benchmark time 128 times.

Figure 6.14 depicts the throughput when writing in the set-up described above, compared to writing writable memory (without `fork`ing). The copy-on-write operations decrease the write speed by a factor of 140x in this (pathological) worst case scenario. When attributing the entire absolute overhead (1112 ms) to the time spent resolving copy-on-write page-faults, then a single copy-on-write operation takes 4242 ns on average.

Streaming Write Performance Degradation When streaming to COW memory, the relative performance impact is much smaller, as only one copy-on-write operation has to be done every 4096 bytes. We have repeated the micro benchmark described in the last paragraph. This time, however the tight loop writes the entire page (instead of a single integer) before continuing with the next page.

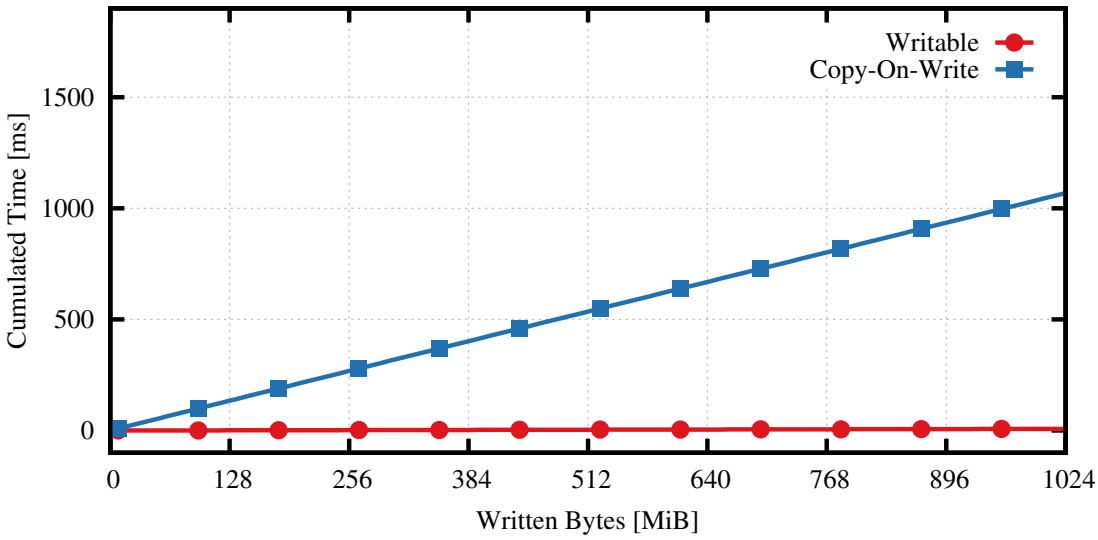

Figure 6.14.: Comparison between the time it takes to write single integers triggering a copy-on-write page-fault at each operation and without (hard) page faults.

Figure 6.15 depicts the resulting write performance of writing without (hard) page-faults and writing with one copy-on-write fault per page. The write speed without page-faults is faster by a factor of 3.6x. The absolute performance decrease (1108 ms) is very close to the last benchmark, however. On average, a single copy-on-write operation takes 4227 ns, in this benchmark.

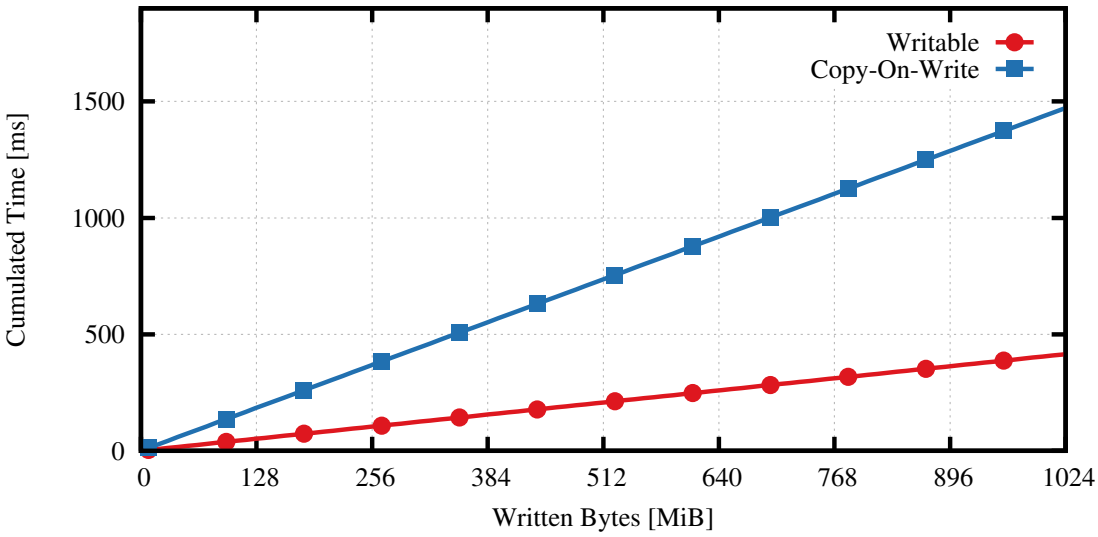

Figure 6.15.: Comparison between the time it takes to write entire pages without (hard) page-faults and with one copy-on-write page-fault at each page.

Copy-On-Write Operations We have recorded 332738 copy-on-write operations in the kernel-build benchmark at a sleep-time of 20 ms, scanning 100 pages per wake-up in the KSM RO configuration.

Figure 6.16 depicts the distribution of those operations across the benchmark time. In this benchmark, every write that leads to a broken sharing is recorded. On a shared page with sharing rank 3, all three pages will be broken separately and thus three COW breaks are recorded in the histogram. If a page is shared until the program terminates, the break is not recorded until it is reused (§3.2.3). Such copy-on-write operations to not appear in the graph.

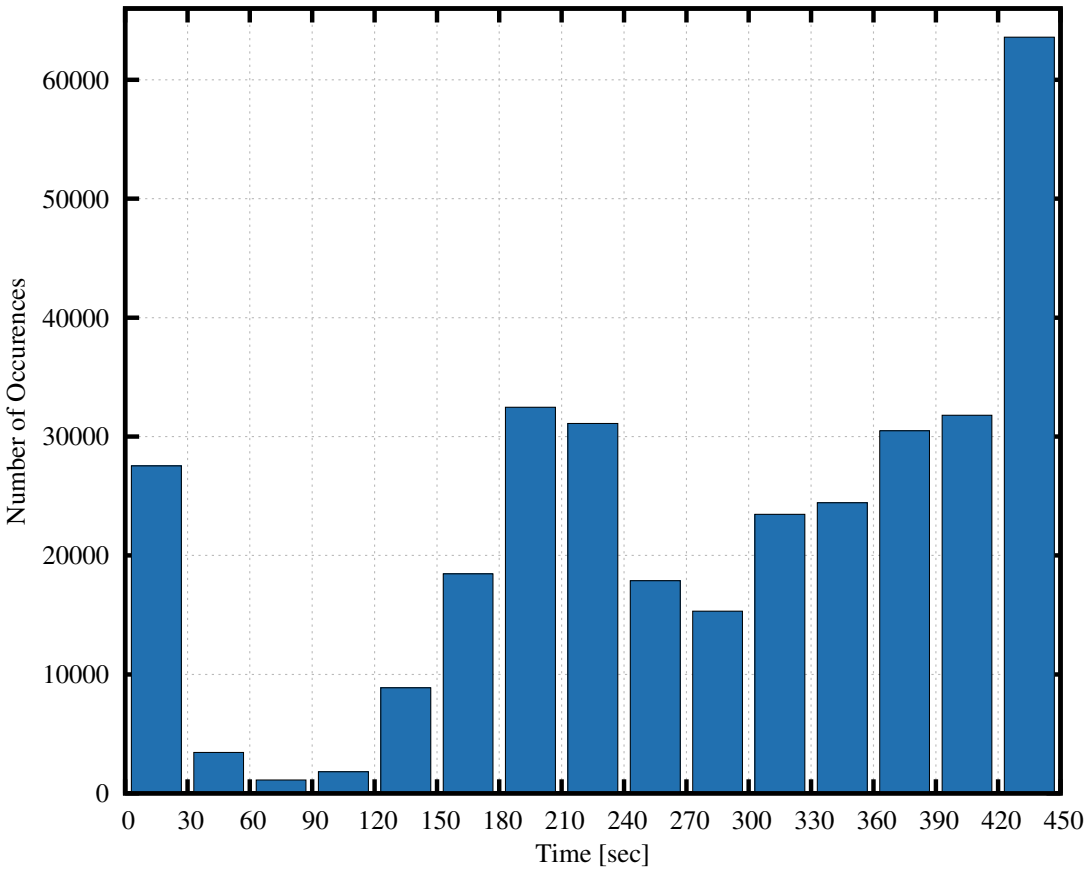

Figure 6.16.: Distribution of copy-on-write operations due to broken shared pages in the kernel-build benchmark. The benchmark was configured to run KSM RO with a sleep-time of 20 ms scanning 100 pages per wake-up.

Sharing is broken throughout the benchmark. Most shared pages are broken at the end of the benchmark, in the linking phase, however. If every break costs 4235ns (mean of streaming and single write overhead) then the total time the host spends breaking shared pages is 1.409 s. This equates to 3.13‰ of the benchmark time.

6.2.3. Reading and Caching

Although deduplication is often a source for overhead, memory deduplication also contains the opportunity to increase the memory access speed. Such speed-ups can result from reading shared memory, when a shared page is already present in a physically indexed cache-line. Another chance for a speed-up is that cache lines become free due to the deduplication and can be used for other data.

Reading Shared Memory Intel uses a physically indexed Last Level Cache (LLC). Thus, the memory read speed directly benefits from memory deduplication if a merged page has previously been accessed and is still cached.

We have conducted a micro benchmark on an Intel Core i7-920 CPU with 8MiB LLC and 24 GiB Kingston DDR3 memory (1333 MHz). In this micro benchmark we have allocated 1 GiB of memory and initialized it so that all pages can be merged to one. We have then measured the read performance from a fully deduplicated memory region and without deduplication.

Table 6.7 lists the median results of 128 runs of this benchmark. The upper bound to shared memory accesses is 19 GiB per second which is almost the speed of the last level cache; a speedup of 2x compared to unshared reads.

	Unshared Read	Shared Read	RAM	LLC
Throughput [GiB/s]	9.5	19.0	10.9	24.5

Table 6.7.: Median throughput measured in our micro benchmark. The throughput numbers for the Last Level Cache and for the RAM were reported by memtest86+ [24].

Increased Cache Capacity On physically indexed caches, merged physical memory page frames also lead to a decreased number of used cache lines, if both virtual pages are active. Caching generally operates on a smaller granularity than paging. On modern Intel processors, such as the Intel i7 and Xeon CPUs, cache lines on all three hierarchy levels (L1, L2 and L3) are 64 bytes long. This does not mean, that deduplication is performed on cache line granularity in the caches. Deduplication that has been done on page size is reflected in the cache, however.

Recall the desktop workload that we have described in §3.2.2. In that benchmark, we simulated and traced a PC with 2 GiB RAM running Gimp, LibreOffice, and Eclipse using the Simics full system simulator [69].

To learn about the effect that deduplication has on CPU caches, we have applied cache models of three different CPUs (Table 6.8) to a memory trace of the desktop workload. With those models, we have analyzed the amount of cache lines that could be freed if memory deduplication were used.

Name	Associativity	Size	Type
Intel Pentium 4 (Willamette)	8-way	256 KiB	L2 cache
Intel Core i7-2600K	16-way	8 MiB	L3 cache
Intel Xeon E5-2470	20-way	20 MiB	L3 cache

Table 6.8.: Caches that we have modeled and simulated.

Figure 6.17 shows the result of our simulation. With increasing cache sizes, memory deduplication increases the possibility to have a positive effect on the cache. Merged pages that are reflected in merged cache lines show the 2x increase in read throughput that was previously presented in the micro-benchmark. With larger CPU caches that we might see in the future, the positive effect of memory deduplication on the read performance will grow.

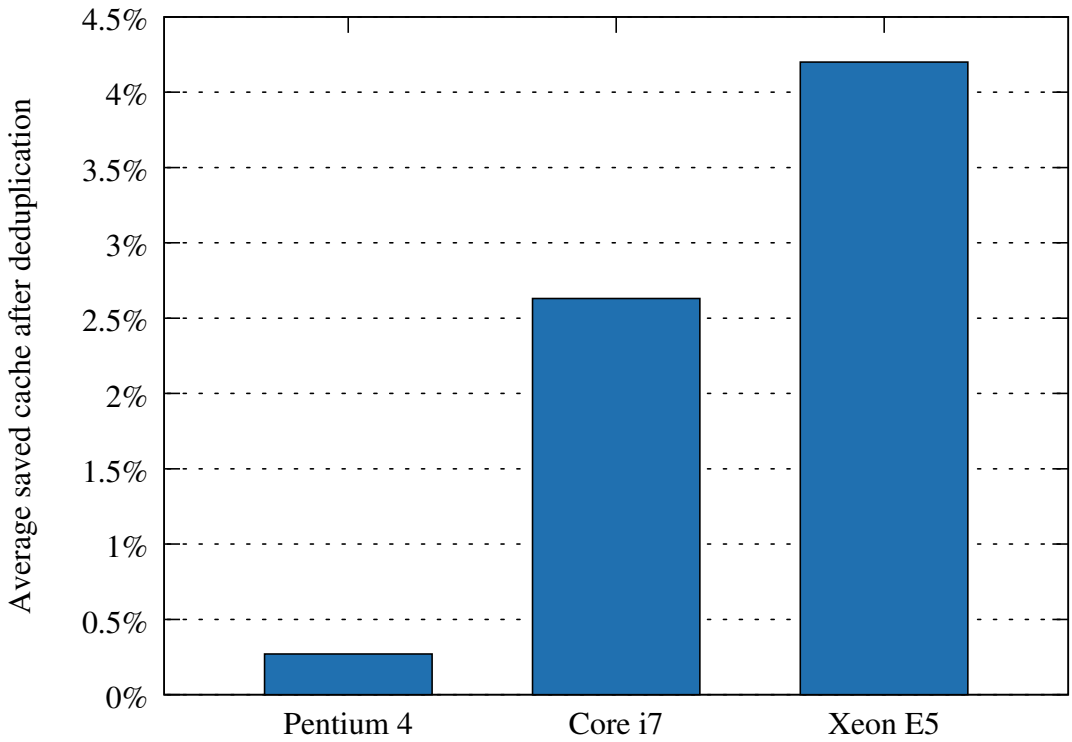

Figure 6.17.: Cache space that could effectively be saved with deduplication.

6.3. Conclusion

In this chapter we have first analyzed the work that memory scanners put into the scanning process. We have done this by measuring how much time each functional component in the KSM memory scanner takes to run in a micro benchmark. Then we have measured how often each functional component is used in the kernel-build benchmark to put the absolute overheads into perspective.

In KSM, unsuccessfully looking up pages in the unstable tree, calculating the hash value, and merging pages are the most costly operations. How often each operation is performed heavily depends on the scan-rate. The higher the scan-rate, the more visited pages are inserted into the unstable tree after calculating the hash value. On slower scan-rates, more visited pages are skipped after calculating the hash value due to the heuristic that keeps fluctuating pages from being inserted into the unstable tree. In absolute values, those two paths always dominate the run-time of the scanner. Merely which one of the two paths is causing more overhead depends on the scan-rate. We have also measured the time and number of times that XLH's modifications to KSM run with the result, that the overhead that XLH adds to KSM is negligible.

We have also analyzed the overhead that memory scanners impose on concurrently running application: First, we have established that scanning at a higher scan-rate does cause higher run-time overhead on real applications. Second, we have considered the write performance overhead and the read performance gain, caused by memory deduplication, individually. The copy-on-write overhead is constant per incident and takes around 4.2 µs per page-fault, regardless if the memory is written randomly or if the memory is streamed to. When reading, the maximum speed-up depends on the fastest physically indexed cache in the memory hierarchy. In our set-up, the speed-up that can be reached when reading is limited by the L3 CPU cache, which is roughly twice as fast as the RAM. The proportion of current caches that can be reused due to sharing is limited, however. We have measured 4.2% of cache space that can be reused in the kernel-build benchmark on a Xeon processor.

Chapter 7

Conclusion

Operating Systems can always put main memory to good use. Additional memory can speed up I/O through caching. Additional memory can moreover increase the degree of multi-programming. In virtualized environments this means that more virtual machines can be located on a physical host, reducing the number of servers to run a workload. The available memory capacity, however, is the primary bottleneck when it comes to consolidating virtual machines [39].

Virtual memory systems make it possible to share memory pages in a copy-on-write (COW) fashion. Previous work has shown that the memory footprint of virtual machines can be reduced significantly by merging equal pages using COW. The semantic gap between virtual machines and the host system makes it difficult to identify such pages, however. Until now, sharing opportunities have been identified either through brute force scanning, or using paravirtualization and introspection techniques.

A thorough analysis of sharing opportunities in virtual environments has given us the insight that I/O pages are a good candidate for memory sharing. Such pages are modified infrequently. They, however, often contain equal pages.

When using brute force main memory scanners, this opportunity for sharing is not exploited to its full potential. Furthermore, previous paravirtualization and introspection based methods that aim to prevent such memory duplicates from appearing at all have been prone to slowdowns of up to 35% in benchmarks that stress the I/O system [63].

We propose to introduce *Cross Layer Hints* (XLH) to memory scanners. XLH is a novel heuristic to focus memory scanning on areas of I/O activity. XLH hooks background storage I/O operations in the virtual file system of the host and stores the target I/O memory pages in a hint buffer. Those hints are later processed by the memory scanner, interleaved with the regular scan process.

We have implemented an XLH prototype, based on Linux' memory deduplication scanner KSM. Evaluating our prototype, we were able to show that compared to linear scanning at the same scan-rate, memory deduplication scanners deliver superior memory deduplication quantities when preferring recently modified I/O-pages. In the HTTPerf benchmark, for example, XLH was able to merge eight times more pages than KSM at the same scan-rate. XLH finds more sharing opportunities than KSM and detects them earlier by minutes. Thereby, XLH exploits sharing opportunities within and across virtual machines that were not detectable by linear scanners before and prolongs the sharing time of previously detected sharing opportunities.

Although the amount of saved memory rises, we have measured only minor increases in scanning overhead when moving from a purely linear memory scanner to XLH. When overcommitting memory, there are three predominant costs for the hypervisor: Finding duplicates, merging them, and finally breaking shared pages using copy-on-write. The identification overhead can be clearly improved using I/O-based hints. Merging pages can be costly, if the pages that are supposed to be merged are currently in use. Then, the merge operation has a negative performance impact on the application as previously cached pages and TLB entries need to be reloaded. Breaking shared pages using copy-on-write adds a fixed cost for each time a shared page is broken. Sharing pages that are likely to live for a longer period of time – such as pages in the page-cache of the guest OS – more quickly on the one hand, but merging other pages less aggressively on the other hand, can decrease the overhead of memory deduplication. XLH achieves this at medium scan-rates, such as our benchmark setting where we scan 100 pages per wake-up, with a sleep-time of 100 ms.

Reading memory can benefit greatly from deduplication. With growing cache sizes, we predict that we will notice this effect in real applications. Here, we have shown that the speed-up potential depends on the fastest physically indexed cache in the memory hierarchy. In our case the speed-up was limited by the L3 cache, which is twice as fast as main memory in our set-up.

The XLH design is robust enough to be of practical use in virtualized data centers. Interleaving hint processing with the scan process acts as a safeguard against starving the linear scan that can find non-I/O duplicates. The interleaving ratio and hint buffer sizes are tunable and can thus be adjusted to be applicable to many workloads.

We have left it open where hints can come from. I/O-based hints are just one well-suited example for such a hint source. In our design, hints can be issued concurrently from any number of sources by using the `madvise` system call. This way, applications can even advise the memory scanner to preferentially scan parts of their own address space.

7.1. Limitations and Future Work

Working on this project has not only answered many questions and brought insight into the area of memory deduplication. It has also raised questions and pinpointed challenges and thus directions for ongoing work.

NUMA Our solution currently solely targets single socket, multi-core systems with uniform memory access speeds. Non-uniform memory access (NUMA) systems distinguish between local and remote memory accesses that have different latency and throughput characteristics depending on their distance. As remote memory accesses are more expensive compared to local ones in those terms, in such systems, merging two pages from different nodes causes a higher overhead than locally. We have not regarded NUMA memory deduplication and the effect of our extensions on NUMA memory deduplication in this thesis. The analysis of memory deduplication on such systems will become interesting in the near future, as NUMA systems currently gain in market penetration.

Automatically Tuning Parameters at Runtime We have analyzed very static workloads in our evaluation. Beforehand, we have explored good parameters to maximize the duplication quantity running the kernel-build benchmark. We have then used those parameters in all other benchmarks.

Real workloads do generally not behave as uniformly. In consequence, the question is not only how to find good initial parameters. It is moreover a question of tuning the deduplication parameters at run-time.

Hardware Support for Efficient Deduplication We have found a heuristic to improve the overhead of identifying sharing opportunities. We have, however, not regarded the remaining sources of overhead in memory deduplication.

Three of the remaining overhead sources are purely mechanisms in nature and could thus be improved in hardware:

- **Checksums:** Main memory could calculate page frame checksums in hardware.
- **Merging:** Merging is prone to race conditions and must be very carefully implemented. After merging, the caches and TLB entries for the merged pages are flushed. The copy, page table, cache and TLB operations can potentially be performed atomically in hardware.
- **Copy-on-Write:** If the page table had a notion of “free page frame” the OS could reserve a small pool of copy-on-write targets. The MMU could then resolve copy-on-write faults without OS invocation.

Not all equal pages should be merged at all (e.g., in NUMA systems), and sometimes it can be beneficial to delay merging to wait and see if a memory region is highly fluctuating or not before going through the trouble of merging it. So, although mechanisms can be pushed down the systems stack, I would strongly suggest to leave policy decisions in the OS or even pushing policy decisions higher up the software stack.

Application to Native Applications KSM only supports deduplication of anonymous memory regions. This is sufficient to merge pages from virtual machines, because the host allocates all VM physical memory as anonymous memory in the host virtual address space.

If the memory deduplication mechanism was extended to support named memory, it would be possible to merge VM memory with the host page cache. Moreover, this would make it possible to deduplicate native applications. We have seen great potential for deduplication of native terminal and game servers in a preliminary quantitative analysis [46]. XLH could then be directly used in this scenario without modification.

Appendix A

Deutsche Zusammenfassung

Betriebssysteme können zusätzlichen Hauptspeicherplatz (RAM) nutzen um die Geschwindigkeit und Auslastung von Computersystemen zu erhöhen. Zusätzlicher Hauptspeicher kann beispielsweise genutzt werden um Hintergrundspeicherzugriffe zu puffern und somit zu beschleunigen.

Virtualisierung ermöglicht eine flexible Einplanung und Migration von Diensten sowie die Konsolidierung vieler virtueller Maschinen auf weniger physische Maschinen. Dabei bleibt eine starke Isolation zwischen den Diensten erhalten. Bei der Nutzung von Virtualisierung ist die verfügbare Hauptspeichermenge der hauptsächliche Flaschenhals für die Konsolidierung zusätzlicher virtueller Maschinen (VMs) auf einem physischen Server. Da die übrigen Ressourcen, wie beispielsweise die zentrale Recheneinheit (CPU), generell nicht vollständig ausgelastet sind, bedeutet eine höhere Integrationsdichte virtueller Maschinen hier eine effizientere Nutzung der verfügbaren Ressourcen und damit geringere Kosten für den Betrieb von Rechenzentren. Erhöht sich die Menge des aktiv genutzten Hauptspeichers, durch übermäßige Konsolidierung, über die physisch verfügbare Hauptspeichergröße hinaus, verringert sich die Rechenleistung aufgrund von Seitenflattern (thrashing) für alle Dienste auf diesem physischen Server.

Gemeinsam genutzter Hauptspeicher

Viele Hauptspeicherseiten (pages) enthalten im Allgemeinen, aber in besonderem Maße in virtualisierten Servern, den gleichen Inhalt. Diese Hauptspeicherseiten können mit dem Kopieren-beim-Schreiben (copy-on-write) Mechanismus auf einen einzigen Hauptspeicherseitenrahmen zusammengelegt und dort gemeinsam genutzt werden. Die Schwierigkeit hierbei liegt im Auffinden der Seiten mit identischem Inhalt.

In der Vergangenheit wurden zwei Verfahren entwickelt um gemeinsam nutzbare Seiten zu finden: Erstens solche, die auf Paravirtualisierung und Introspektion beruhen und zweitens Verfahren, welche auf dem Absuchen von Hauptspeicherseiten basieren (memory scanner). Verfahren die auf *Paravirtualisierung* beruhen, kommunizieren semantische Informationen zwischen virtualisiertem Gastbetriebssystem und dem physischen Wirtsystem. Mithilfe von veränderten virtuellen Hintergrundspeichergeräten in den Gastsystemen ist es Satori [63] beispielsweise möglich dateibasierten Hauptspeicher zwischen virtuellen Maschinen zusammenzulegen und somit einzusparen. Die Autoren geben für Satori allerdings indirekte Kosten von bis zu 35% gegenüber unveränderten Systemen an, wenn virtuelle Maschinen das Dateisystemleistungsfähigkeitsbewertungsprogramm Bonnie++ ausführen. *Hauptspeicherabsuchverfahren* erstellen kontinuierlich, mit einer vorher festgelegten Absuchrate, einen Index des untersuchten Speichers. Stellt der absuchende Prozess beim Einfügen in den Index fest, dass sich der einzufügende Inhalt bereits im Index befindet, legt der Absuchprozess beide Hauptspeicherseitenrahmen zusammen und gibt daraufhin den redundanten, nun nicht mehr referenzierten, Rahmen frei. Hauptspeicherabsuchverfahren sind gegenüber auf Paravirtualisierung basierenden Verfahren hinsichtlich der indirekten Kosten für ausgeführte Programme im Vorteil. Sowohl bei der Deduplikationseffizienz als auch bei dessen Effektivität hingegen sind Hauptspeicherabsuchverfahren im Nachteil.

Systemschichtübergreifende Hinweise

Um die Vorteile beider Verfahren zu vereinen, also um ohne die Gastbetriebssysteme anzupassen und um mit geringen indirekten Kosten effizient und effektiv Hauptspeicherinhalte gemeinsam nutzen zu können, schlagen wir vor systemschichtübergreifende Hinweise (cross layer hints, XLH) in Hauptspeicherabsuchverfahren zu integrieren. Mithilfe dieser Hinweise können Subsysteme, die über semantisches Wissen darüber verfügen, welche Hauptspeicherseiten gute Kandidaten für das Zusammenlegen sind, dieses Wissen an das Hauptspeicherabsuchverfahren übermitteln. Die Seiten, für die Hinweise vorliegen, werden daraufhin bevorzugt abgesucht.

Wir haben einen Linux-basierten Prototyp implementiert, in dem das virtuelle Dateisystem im Wirtssystem dem Hauptspeicherabsuchverfahren Hinweise darüber gibt, mit welchen Speicherseiten Hintergrundspeicherkommunikation stattfindet. In der vorliegenden Arbeit konnten wir zeigen, dass ein Absuchverfahren, welches diese Hinweise bevorzugt bearbeitet, bis zu achtmal mehr Hauptspeicher sparen kann als ein Absuchverfahren, das strikt linear im Gast-Physischen Addressraum nach Duplikaten sucht. Der zusätzlich gesparte Hauptspeicher kommt daher, dass Speicherseiten, die in beiden Absuchverfahren gefunden werden, in XLH um Minuten früher geteilt werden können. Weiterhin können durch die frühe Indizierung zusätzliche, kurzlebige Seiten gleichen Inhalts gefunden werden.

Die Auswertung der Kosten für Hauptspeicherabsuchverfahren bestätigte, dass eine erhöhte Absuchrate zu höheren indirekten Kosten für reale Arbeitslasten führt. Wir konnten außerdem zeigen, dass der Einsatz von XLH, bei gleichen Absuchraten, nur einen vernachlässigbaren Mehraufwand gegenüber einem linearen Absuchverfahren mit sich bringt.

Appendix B

Apache Static File Generation

The following python script generates a directory structure expected by HTTPerf and static files of 50 kB length with random content. The contents of the second directory are the same than the contents from the first directory but shuffled.

```
import os, random
def createFiles( a, b ):
   r = file( '/dev/urandom', 'r' ).read(50000)
   print a, b
   file( a , 'w').write( r )
   file( b , 'w').write( r )

def generateDirs(l1, l2):
   os.mkdir( 'l1' )
   os.mkdir( 'l2' )
   for a in range(0, 10):
      a = str(a)
      os.mkdir( 'l1' + '/' + a )
      os.mkdir( 'l2' + '/' + a )
      for b in range(0, 10):
         b = str(b)
         os.mkdir( 'l1' + '/' + a + '/' + b )
         os.mkdir( 'l2' + '/' + a + '/' + b )
         for c in range(0, 10):
            c = str(c)
            os.mkdir( 'l1' + '/' + a + '/' + b + '/' + c )
            os.mkdir( 'l2' + '/' + a + '/' + b + '/' + c )
            for d in range(0, 10):
               d = str(d)
               f = a + '/' + b + '/' + c + '/' + d + '.html'
               l1.append( f )
               l2.append( f )
```

```
l1 = []
l2 = []
generateDirs( l1, l2 )
random.shuffle( l2 )

while len(l1) > 0:
   a = l1.pop()
   b = l2.pop()
   print str('l1/' + a), str('l2/' + b)
   createFiles( str('l1/' + a), str('l2/' + b) )
```

List of Tables

List of Figures

Bibliography

[1] Advanced Micro Devices, Inc. *AMD-V Nested Paging*, 1.0 edition, July 2008.

[2] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. In *The Linux Symposium*, OLS '09, Montreal, Canada, July 2009. Linux Symposium Inc.

[3] Yungang Bao, Mingyu Chen, Yuan Ruan, Li Liu, Jianping Fan, Qingbo Yuan, Bo Song, and Jianwei Xu. Hmtt: a platform independent full-system memory trace monitoring system. In *SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '08, pages 229–240, Annapolis, MD, USA, 2008. Association for Computing Machinery.

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *19th Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, Bolton Landing, NY, USA, 2003. Association for Computing Machinery.

[5] 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.

[6] Nikolai Baudis. Deduplicating virtual machine checkpoints for distributed system simulation, November 2013. Bachelor Thesis, System Architecture Group, KIT, Germany.

[7] Brian Behlendorf, Roy T. Fielding, Rob Hartill, David Robinson, Cliff Skolnick, Randy Terbush, Robert S. Thau, and Andrew Wilson. Apache http server. `https://httpd.apache.org`, 1995.

[8] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference - FREENIX Track*, USENIX ATC '05, Anaheim, CA, USA, 2005. USENIX Association.

[9] Nikhil Bhatia. Performance evaluation of intel ept hardware assist, March 2009.

[10] Daniel G. Bobrow, Jerry D. Burchfiel, Daniel L. Murphy, and Raymond S. Tomlinson. Tenex, a paged time sharing system for the pdp - 10. *Commun. ACM*, 15, March 1972.

[11] Jeff Bonwick. The slab allocator: An object-caching kernel. In *USENIX Summer 1994 Technical Conference*, BOS 1994, Boston, MA, USA, June 1994. USENIX Association.

[12] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *16th Symposium on Operating Systems Principles*, SOSP '97, pages 143–156, Saint-Malo, France, October 1997. Association for Computing Machinery.

[13] Chao-Rui Chang, Jan-Jan Wu, and Pangfeng Liu. An empirical study on memory sharing of virtual machines for server consolidation. In *9th International Symposium on Parallel and Distributed Processing with Applications*, ISPA '11, pages 244–249. Institute of Electrical and Electronics Engineers, May 2011.

[14] Chao-Rui Chang, Jan-Jan Wu, and Pangfeng Liu. An empirical study on memory sharing of virtual machines for server consolidation. In *9th International Symposium on Parallel and Distributed Processing with Applications*, ISPA '11, pages 244–249, Busa, Korea, May 2011. IEEE.

[15] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *Eighth Workshop on Hot Topics in Operating Systems*, HOTOS '01, pages 133–138, Elmau, Germany, May 2001. IEEE.

[16] Zhifeng Chen, Yuanyuan Zhou, and Kai Li. Eviction based cache placement for storage caches. In *USENIX Annual Technical Conference*, USENIX ATC '03, San Antonio, TX, USA, 2003. USENIX Association.

[17] Russell Coker. Bonnie++. `http://www.coker.com.au/bonnie++`, 1999.

[18] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly Media, Inc., 3rd edition, 2005.

[19] Intel Corporation. 2nd Gen Intel Core Processor Family Desktop Datasheet, Vol. 1. `http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/2nd-gen-core-desktop-vol-1-datasheet.pdf`, 2013.

[20] Intel Corporation, editor. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. July 2013.

[21] Intel Corporation, editor. *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 3. June 2013.

[22] International Business Machines Corporation. Kernel Virtual Machine (KVM) – Best practices for KVM. http://pic.dhe.ibm.com/infocenter/lnxinfo/v3r0m0/topic/liaat/liaatbestpractices_pdf.pdf, April 2012. accessed September 2013.

[23] Uwe Dannowski, Joshua LeVasseur, Espen Skoglund, Volkmar Uhlig, and Jan Stoess. L4 eXperimental Kernel Reference Manual, Version X.2. http://www.l4ka.org/l4ka/l4-x2-r7.pdf, October 2011.

[24] Samuel Demeulemeester. memtest86+. http://memtest.org.

[25] Peter J. Denning. Thrashing: its causes and prevention. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, AFIPS '68 (Fall, part I), pages 915–922, New York, NY, USA, 1968. Association for Computing Machinery.

[26] Peter J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[27] Peter J Denning. Before memory was virtual. *In the Beginning: Personal Recollections of Software Pioneers*, 1996.

[28] International Business Machines Corporation. Data Processing Division. *IBM OS Linkage Editor and Loader: (Program Numbers 360S-ED-510, 360S-ED-521 [and] 360S-LD-547).* IBM Systems reference library. 1972.

[29] Allen B. Downey. The structural cause of file size distributions. In *9th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOT '01, pages 361–370, Cincinnati, OH, USA, August 2001. IEEE Computer Society.

[30] Ulrich Drepper. How to write shared libraries. Technical report, Red Hat, Inc., 2010.

[31] John Fotheringham. Dynamic storage allocation in the atlas computer, including an automatic use of a backing store. *Commun. ACM*, 4(10):435–436, October 1961.

[32] Fabian Franz, Konrad Miller, and Frank Bellosa. Using i/o-based hints to make memory-deduplication scanners more efficient, July 2012. Diploma Thesis, System Architecture Group, KIT, Germany.

[33] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[34] Ilya Gavrichenkov. Samsung 840 Pro and Samsung 840 Solid State Drives Review. http://www.xbitlabs.com/articles/storage/display/samsung-840-pro_5.html, 2013. accessed July 2013.

[35] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. In *17th Symposium on Operating Systems Principles*, SOSP '99, pages 154–169, Charleston, South Carolina, USA, December 1999. Association for Computing Machinery.

[36] Charles David Graziano. A performance analysis of xen and kvm hypervisors for hosting the xen worlds project. Master's thesis, Iowa State University, 2011.

[37] Thorsten Gröninger, Konrad Miller, and Frank Bellosa. Analyzing shared memory opportunities in different workloads, November 2011. Study Thesis, System Architecture Group, KIT, Germany.

[38] Thorsten Gröninger, Marc Rittinghaus, Konrad Miller, and Frank Bellosa. On statistical properties of duplicate memory pages, October 2013. Diploma Thesis, System Architecture Group, KIT, Germany.

[39] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. Difference engine: harnessing memory redundancy in virtual machines. *Communications of the ACM*, 53(10):85–93, October 2010.

[40] Val Henson and Richard Henderson. Guidelines for using compare-by-hash. `http://valerieaurora.org/review/hash2.pdf`, 2005.

[41] Oracle Corporation innotek GmbH. Virtualbox. `https://www.virtualbox.org`, 2007.

[42] M. Tim Jones. Anatomy of Linux Kernel Shared Memory. `http://public.dhe.ibm.com/software/dw/linux/l-kernel-shared-memory/l-kernel-shared-memory-pdf.pdf`, April 2010. accessed September 2013.

[43] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Geiger: Monitoring the buffer cache in a virtual machine environment. In *12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 14–24, San Jose, CA, USA, October 2006. Association for Computing Machinery.

[44] Jonas Julino, Konrad Miller, and Frank Bellosa. Analysing page duplication on android, March 2012. Study Thesis, System Architecture Group, KIT, Germany.

[45] Randy Howard Katz. Lecture 19: Case study – virtual memory, alpha 21064 memory hierarchy and performance. `http://bnrg.cs.berkeley.edu/~randy/Courses/CS252.S96/Lecture19.pdf`, 1996.

[46] Philipp Kern, Konrad Miller, and Frank Bellosa. Generalizing memory deduplication for native applications, sandboxes and virtual machines. Master's thesis, April 2013. Diploma Thesis, System Architecture Group, KIT, Germany.

[47] Hwanju Kim, Heeseung Jo, and Joonwon Lee. XHive: Efficient cooperative caching for virtual machines. *Transactions on Computers*, 60(1):106–119, January 2011.

[48] Ricardo Koller and Raju Rangaswami. I/o deduplication: Utilizing content similarity to improve i/o performance. *Trans. Storage*, 6(3):13:1–13:26, September 2010.

[49] Marco Kroll, Konrad Miller, and Frank Bellosa. Performance analysis of memory deduplication, March 2014. Diploma Thesis, System Architecture Group, KIT, Germany.

[50] 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, 2009. Association for Computing Machinery.

[51] Kevin P. Lawton. Bochs: A portable pc emulator for unix/x. *Linux Journal*, 1996(29es), September 1996.

[52] Seho Lee, Inhyeok Kim, Dongwoo Lee, and Young Ik Eom. The page cache duplication mechanism in virtualized systems. *International Journal of Control and Automation*, 6(1):151–159, February 2013.

[53] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1999.

[54] Robert Lowe. *Linux Kernel Development - A thorough guide to the design and implementation of the Linux kernel*. Addison Wesley, 3rd edition, 2012.

[55] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, Chicago, IL, USA, 2005. Association for Computing Machinery.

[56] Dan Magenheimer, Chris Mason, Dave McCracken, and Kurt Hackel. Paravirtualized paging. In *First Workshop on I/O Virtualization*, WIOV '08, San Diego, CA, USA, December 2008. USENIX Association.

[57] Dan Magenheimer, Chris Mason, Dave McCracken, and Kurt Hackel. Transcendent memory and linux. In *The Linux Symposium*, OLS '09, Montreal, Canada, July 2009. Linux Symposium Inc.

[58] Wolfgang Mauerer. *Professional Linux Kernel Architecture*. Wiley Publishing, Inc., 2008.

[59] Konrad Miller. Betriebssysteme Übung 2 WS 2011/2012. based on Stanford class CS140, 2011, by David Mazieres. `http://www.scs.stanford.edu/11wi-cs140/`, 2011.

[60] Konrad Miller. Betriebssysteme Übung 4 WS 2011/2012. based on Stanford class CS140, 2011, by David Mazieres. `http://www.scs.stanford.edu/11wi-cs140/`, 2011.

[61] 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 *ASPLOS Workshop on Runtime Environments, Systems, Layering and Virtualized Environments*, RESoLVE '12, London, UK, March 2012.

[62] Konrad Miller, Fabian Franz, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. XLH: More effective memory deduplication scanners through cross-layer hints. In *USENIX Annual Technical Conference*, USENIX ATC '13, San Jose, CA, USA, 2013. USENIX Association.

[63] Grzegorz Miłós, Derek G. Murray, Steven Hand, and Michael A. Fetterman. Satori: Enlightened page sharing. In *USENIX Annual Technical Conference*, USENIX ATC '09. USENIX Association, 2009.

[64] David Mosberger and Tai Jin. Httperf - a tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review*, 26(3):31–37, 1998.

[65] Robert P. Munafo. Notable properties of specific numbers. `http://mrob.com/pub/math/numbers-19.html`, 2013. accessed November 2013.

[66] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavy-weight dynamic binary instrumentation. In *SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, San Diego, CA, USA, 2007. Association for Computing Machinery.

[67] Avadh Patel, Furat Afram, and Kanad Ghose. Marss-x86: A qemu-based micro-architectural and systems simulator for x86 multicore processors. In *1st International Qemu Users Forum*, pages 29–30, 2011.

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

[69] Marc Rittinghaus, Konrad Miller, and Frank Bellosa. Runtime benefits of memory deduplication, July 2012. Diploma Thesis, System Architecture Group, KIT, Germany.

[70] Marc Rittinghaus, Konrad Miller, Marius Hillenbrand, and Frank Bellosa. Simuboost: Scalable parallelization of functional system simulation. In *11th International Workshop on Dynamic Analysis*, WODA '03, Houston, Texas, March 2013.

[71] William K Robertson, Christopher Kruegel, Darren Mutz, and Fredrik Valeur. Run-time detection of heap-based overflows. In *17th Large Installation Systems Administration Conference*, volume 3 of *LISA '03*, pages 51–60, San Diego, CA, USA, October 2003. USENIX Association.

[72] John Scott Robin and Cynthia E. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *9th USENIX Security Symposium*, SSYM '00, Denver, CO, USA, August 2000. USENIX Association.

[73] Salvatore Sanfilippo and The Redis Community. Redis key-value store. `http://redis.io`, 2009.

[74] Martin Schwidefsky, Hubertus Franke, Ray Mansell, Raj Himanshu, Damian Osisek, and JongHyuk Choi. Collaborative Memory Management in Hosted Linux Environments. In *The Linux Symposium – Volume Two*, OLS '06, pages 313–328, Ottawa, Canada, July 2006. Linux Symposium Inc.

[75] Burroughs Corporation. Sales Technical Services. *The Descriptor: A Definition of the B5000 Information Processing System*. Bulletin (Burroughs Corporation. Sales Technical Services). Sales Technical Services, Equipment and Systems Marketing Division, Burroughs Corporation, 1961.

[76] Shengyang Sha, Li Jianxin, Nan Li, Wuyang Ju, Lei Cui, and Bo Li. SmartKSM: A vmm-based memory deduplication scanner for virtual machines. Poster presented at the 24th Symposium on Operating Systems Principles, April 2013.

[77] Prateek Sharma and Purushottam Kulkarni. Singleton: System-wide page deduplication in virtual environments. In *21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 15–26, Delft, Netherlands, June 2012. Association for Computing Machinery.

[78] Balbir Singh. Page/slab cache control in a virtualized environment. In *12th Annual Linux Symposium*, OLS '10, Ottawa, Canada, July 2010. Linux Symposium Inc.

[79] Kuniyasu Suzaki, Toshiki Yagi, Kengo Iijima, Nguyen Anh Quynh, Cyrille Artho, and Yoshihito Watanebe. Moving from logical sharing of guest os to physical sharing of deduplication on virtual machine. In *Proceedings of the 5th USENIX conference on Hot topics in security*, HotSec'10, pages 1–7, Berkeley, CA, USA, 2010. USENIX Association.

[80] TravisCI Community. TravisCI: continuous integration service. `https://travis-ci.org/`, 2012.

[81] S. G. Tucker. Emulation of large systems. *Communications of the ACM*, 8(12):753–761, December 1965.

[82] Richard A. Uhlig and Trevor N. Mudge. Trace-driven memory simulation: A survey. *ACM Computing Surveys (CSUR)*, 29(2):128–170, June 1997.

[83] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *20th Symposium on Operating Systems Principles*, SOSP '05, pages 148–162, Brighton, UK, October 2005. Association for Computing Machinery.

[84] Carl A. Waldspurger. Memory resource management in vmware esx server. In *5th Symposium on Operating System Design and Implementation*, OSDI '02, pages 181–194, Boston, MA, USA, December 2002. USENIX Association.

[85] Carl A. Waldspurger. Content-based, transparent sharing of memory units. U.S. Patent number: 6789156, September 2004.

[86] Vincent M Weaver and Sally A McKee. Can hardware performance counters be trusted? In *International Symposium on Workload Characterization*, IISWC '08. IEEE Computer Society, 2008.

[87] Theodore M. Wong and John Wilkes. My cache or yours? making storage more exclusive. In *USENIX Annual Technical Conference*, USENIX ATC '02, Monterey, CA, USA, 2002. USENIX Association.

[88] Timothy Wood, Gabriel Tarasuk-Levin, Prashant Shenoy, Peter Desnoyers, Emmanuel Cecchet, and Mark D. Corner. Memory buddies: Exploiting page sharing for smart colocation in virtualized data centers. In *International Conference on Virtual Execution Environments*, VEE '09, pages 31–40, Washington, DC, USA, March 2009. Association for Computing Machinery.

[89] Zhe Zhang, Han Chen, and Hui Lei. Small is big: Functionally partitioned file caching in virtualized environments. In *Fourth USENIX Workshop on Hot Topics in Cloud Computing*, Hotcloud '12, Boston, MA, USA, June 2012. USENIX.