Karlsruher Institut für Technologie

Generalizing Memory Deduplication for Native Applications, Sandboxes and Virtual Machines

Diplomarbeit
von

cand. inform. Philipp Kern

an der Fakultät für Informatik
Institut für Betriebs- und Dialogsysteme
Lehrstuhl Systemarchitektur

Erstgutachter:	Prof. Dr. Frank Bellosa
Zweitgutachter:	Prof. Dr. Hartmut Prautzsch
Betreuender Mitarbeiter:	Dipl.-Inform. Konrad Miller

Bearbeitungszeit: 1. November 2012 – 30. April 2013

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ort, Datum Unterschrift

Deutsche Zusammenfassung

Die vorliegende Arbeit beschäftigt sich mit Speicherdeduplikation von Speicherseiten unter Linux laufender Programme. Bei der Deuplikation von Speicher wird versucht, möglichst viele Speicherseiten mit gleichem Inhalt zu finden und zusammenzulegen, sodass dieselbe Seite von mehreren Programmen gleichzeitig genutzt werden kann und insgesamt weniger Speicher für die Anwendungen benötigt wird.

Im Arbeitsspeicher vorhandene Redundanzen können auf aktuellen Plattformen durch zwei Ansätze erkannt und ausgenutzt werden: durch inhaltsbewusste Deduplikation und die periodische Überprüfung von Speicherinhalten auf Redundanzen. Die inhaltsbewusste Deduplikation nutzt Eigenschaften der Daten, z.B. deren Quelle, um gleiche Speicherseiten möglichst ohne aufwendige Analysen zu identifizieren. Die periodische Überprüfung liest die Speicherinhalte in regelmäßigen Abständen aus und sucht mit Hilfe von Hashwerten nach Duplikaten.

Den Hauptteil dieser Arbeit bildet die Offline-Analyse von vier Lastszenarien. Diese beeinhalten Desktopumgebungen mit in diesem Umfeld üblichen Programmen (Benutzeroberfläche, Webbrowser und Office-Programme) und ein Serverszenario mit einem dedizierten Server eines Computerspiels. Die Szenarien wurden insbesondere in Hinblick auf Redundanzen in Mehrbenutzerkonfigurationen untersucht, wie sie in Cloud-Umgebungen üblich sind. Die Offline-Analyse basiert auf Momentaufnahmen des Speicherinhalts, welcher mittels Schnittstellen ausgelesen und archiviert wird. Diese Speicherauszüge werden dann ausgewertet, um Redundanzen gemäß unterschiedlicher Vorgehensweisen zu finden.

Die Evaluation der Lastszenarien zeigt, dass Redundanzen im Speicherinhalt der untersuchten Anwendungen ausschließlich zwischen Dateien bzw. innerhalb von dynamisch allokiertem Speicher auftreten, nicht aber zwischen diesen beiden Domänen. Es bietet sich daher an, diese Speicherbereiche getrennt voneinander zu behandeln, um eine effizientere Deduplikation zu erreichen. Insbesondere wäre auch die Deduplikation von Dateien vorteilhaft, da diese bisher in Linux noch nicht implementiert ist.

Systemprimitive wie `mmap` sollten erweitert werden, um die Wiederverwendbarkeit von Speicherkacheln basierend auf deren Inhalt zu ermöglichen. Unterschiedliche Dateien mit gleichem Inhalt könnten so im Speicher durch die gleichen Kacheln dargestellt werden.

Linux unterscheidet dateibasierten und anonymen Speicher. Die Analyse hat gezeigt, dass nicht nur die bereits implementierte Deduplikation von anonymen Speicher verfügbar sein sollte, sondern auch diejenige von dateibasiertem Speicher.

Contents

Deutsche Zusammenfassung **v**

1 Motivation **3**

2 Fundamentals **7**
2.1 Origin of Virtual Memory 7
2.2 Hardware Support for Paging 8
2.3 Virtual Memory Representation in Linux 11
2.4 Sharing Memory Mappings 14

3 Related Work **17**
3.1 Content-Aware Memory Deduplication 17
3.2 Periodical Memory-Scan Deduplication 19

4 Implementation **23**
4.1 Tracing Memory in a Full-System Simulator 23
4.2 Implementing Named Merging on Linux 24
4.3 Dumping Process Memory Contents on Linux 25
4.4 Analyzing Offline Dumps 27

5 Evaluation **31**
5.1 Scenarios 31
5.2 Results 34

6 Discussion **39**
6.1 Sharing Opportunities in Applications 39
6.2 Previous Evaluations 42
6.3 Evaluation Results 44
6.4 Possible Improvements of Memory Deduplication 45
6.5 Summary 48

7 Conclusion **49**

Acronyms **53**

Bibliography **54**

1 Motivation

Computers ship with more memory every year. Whereas early applications tried to use memory efficiently, common applications nowadays embrace the availability of vast amounts of memory instead and use larger quantities of it. Many companies try to reduce and optimize the time spent on programming instead of optimizing the resulting application for speed or efficient use of resources. This keeps development cost down, as Random Access Memory (RAM) is considered a cheap resource. However, what works on a small scale on one desktop computer might not work in tomorrow's world of cloud computing.

The advent of cloud computing caused more applications to run on central servers again, not unlike the mainframes of the past. The central cloud server infrastructure provides companies with virtual servers for their services. Services that can be used by small devices like tablets to operate applications such as voice recognition or searching the internet that need several times more computing power than the device has available. Most companies let their employees use computers to boost their productivity. By providing them with devices that are less powerful whereas the applications are hosted in a central efficient data center, energy and overall costs can be reduced.

To increase this efficiency even more, the hosting of such central services is outsourced to cloud data centers, where multiple companies can run their servers on top of an abstracted infrastructure. To make efficient use of resources a single server machine can serve dozens of different clients. This requires multi-tenancy support, the most popular one being virtualization, but terminal servers are another option to consider. One bottleneck of supporting even more clients per server is the amount of memory that can be provided per physical host.

This is aggravated by the fact that virtual machines and applications often contain redundancies in their memory content. For instance, multiple instances of similar or identical operating systems contain portions of machine code that are identical. The same application being run multiple times by different clients might require the same data in memory to operate on. This redundancy that can be found and coalesced, resulting in less memory required overall. This can free up memory to allow an increased density of applications on a single host or enable programs to run more quickly by keeping more of their data in memory.

All this is possible because memory is organized in fixed-sized tiles (so-called *page frames*) that are not limited to single applications. This makes it easy to deduplicate identical tiles in memory to a single copy. An intuitive way of deduplication is looking at each tile, calculating a hash over the content, and comparing it to a list of already calculated hashes. If a match is found, memory can be freed up by merging the corresponding pages. It is also possible to infer knowledge about redundancies from certain observable actions. If a single disk block is read multiple times by different entities, it is likely that it will be saved as duplicates in different memory locations. Common operating systems avoid this, but virtualization reintroduced this problem as the host does not know how the guest manages its file cache.

Current systems include methods to primarily deduplicate memory used by virtual machines. Other scenarios, such as native applications or sandboxes, require a more generalized view on memory deduplication. Deduplication enables us to need less memory for applications. However, spending too many resources on this task, for example processor cycles, might void the benefits.

One benefit is to serve more clients on one host, for instance by fitting an additional virtual machine on it. Another benefit is being able to keep more disk content in memory, which can speed up applications. Disk I/O is costly, even more so in virtualized environments, where content of many hosts is stored on busy storage systems.

Objectives

The objective of this thesis is to gain insight on memory sharing opportunities in native applications and sandboxes on Linux. Previous works have focussed only on deduplication of memory content in Virtual Machines (VMs). They did not look at the applications within the VMs as to where redundancy is coming from. By eliminating the opaque layer of virtualization and looking directly at native applications on Linux we can determine if memory deduplication is beneficial even outside of the context of virtualization.

The scenarios we used to find redundancies in memory content include single-user and multi-user desktop environments, as well as a game server. From this we deduce in this thesis how memory deduplication techniques should be adjusted to cope with native applications, on which the Operating System (OS) has more information than it has about the memory layout of a guest OS within a VM.

Structure

This thesis is structured as follows:

Chapter 2 gives the reader background information on the creation of virtual memory and how it is implemented in today's most common processor architecture (x86-64). Furthermore, it outlines how the Linux kernel manages physical and virtual memory and how its file caching operates.

Chapter 3 focusses on related work in the field of memory deduplication. It presents two different techniques on how to find duplicates in memory, either by using knowledge about the data or by periodically polling the data for changes.

Chapter 4 describes three approaches that have been carried out in the course of this thesis to gather insight on sharing opportunities: full-system simulation, adjustments to a Linux implementation of memory deduplication and application snapshots. It describes how sharing information can be deduced from such application snapshots.

Chapter 5 introduces the different scenarios in which we applied memory snapshots and the results on memory content sharing within these snapshots.

Chapter 6 discusses the sharing opportunities within native applications, previous results on memory redundancies in VMs, our own evaluation results and presents possible improvements that could help with identifying memory redundancies in applications.

Chapter 7 concludes this and provides a summary of our findings and an outlook on future work.

2 Fundamentals

This chapter introduces the fundamental concepts used in this thesis: an introduction to the concept of virtual memory and how it is implemented. Section 2.1 explains how and why virtual memory was created. The implementation of hardware-assisted paging on the x86-64 architecture is described in Section 2.2. Section 2.3 describes the data structures used by the OS to keep track of virtual memory layouts in tasks. Finally Section 2.4 explains the Unix mechanisms `mmap` and `fork` and the concept of Copy-on-Write (COW) memory management.

2.1 Origin of Virtual Memory

Virtual memory was introduced in the 1960s as a means to ease programming. The first computers started off with one contiguous chunk of physical memory in which a single thread of execution operated. In addition to this, secondary memory in the form of magnetic drums held data that did not fit into memory. When writing a program that exceeded physical memory, the programmer needed to explicitly manage the fetches and stores into secondary memory. Virtual memory eliminates this need by providing the programmer with a large chunk of contiguous memory. This memory is called virtual memory as it does not exist as-is in physical memory. Instead, the hardware is now responsible for providing this virtual view on physical memory to the application. Content that cannot currently be stored in memory or is not needed for a prolonged amount of time is paged in and out by the hardware as needed.

This was based on the observation that most programs have a "working set" of data that varies over time as data is being processed [Denn96]. The parts of the data that are not currently needed by the program can be moved to some kind of stable storage system (disks or tapes back then) to be retrieved at a later point as needed.

Conveniently, the introduction of virtual memory also solved the memory protection problem. Two processes are commonly being run in separate address spaces, which isolates their data from malicious reads and writes and inadvertent modifications of data in memory. Only one address space is active at a time and a process cannot access memory content beyond all the mappings defined by the address space. Protection flags like write or read protect were included when virtual memory was integrated into the hardware, providing a trusted base for an OS to rely upon.

In general, virtual memory serves as an abstraction of the available physical memory that allows objects to be placed at any place within the available address space, relieves the programmer from concrete memory configurations, provides protection, and makes use of the memory hierarchy transparent. [Denn96]

On the successors of the PC that we commonly use (based on the x86-64 Central Processing Unit (CPU) architecture), the Operating System (OS) keeps track of all memory segments an application requested. It then fills CPU data structures with memory mappings, so that memory access can be resolved without further interaction with the OS. Both sides are discussed in the next sections en detail.

2.2 Hardware Support for Paging

Current processor architectures provide virtual memory mechanisms to distribute the available physical memory between multiple distinct processes, providing isolation and flexible memory layouts to applications. The mechanisms are implemented in hardware in order to reduce performance penalties introduced by the abstraction. Direct access to memory needs to be converted to the hardware view on every load and store operation, hence software interaction would be very costly. Physical memory is partitioned into page frames of fixed size to avoid external fragmentation. This means that fixed boundaries are used, regardless of the size of the data. This avoids memory areas that cannot be filled because the allocations would be too large. The page frames can be allocated freely by the OS kernel to processes and for its own use.

The mapping mechanism is provided by a Memory Management Unit (MMU) that can be imagined as sitting between the processor and memory. One early instance of this unit was provided by the TENEX Paged Time Sharing System for the PDP-10 (the "BBN Pager") [BBMT72]. By implementing this functionality in hardware it is also possible to track referenced pages to get hints which pages can be paged out because they are no longer part of the working set. The TENEX system also implemented page sharing: pages that can be mapped (i.e., that can be accessed) by multiple processes. The hardware of the PDP-10 already allowed trapping write accesses to protected memory which can be used to implement Copy-on-Write (COW). A trap interrupts the currently running program and gives the OS the opportunity to implement custom behavior before returning control to the userspace program. Whenever a write access to a shared page is made, the OS can allocate a fresh page frame, copy the content, and provide the resulting copy to the process to write to. Other processes will not see the changes, still in the non-write case memory is saved.

Two different types of addresses are used within the hardware: physical and virtual addresses. Physical addresses are used to address memory cells in Dynamic Random-Access Memory (DRAM) and might be required by devices for Direct Memory Access (DMA) operations. Userspace programs commonly use virtual addresses instead, which are mapped by the hardware to their corresponding physical addresses. This is accomplished by consulting paging structures in memory. One such structure exists for every protection domain, the so-called "address space", and only one is active on a core at a given point in time. Every userspace thread belongs to one address space. If only one thread exists, the combination between thread and address space is called a process. The OS retains the absolute control over the paging structures and userspace programs commonly cannot modify any page tables.

The remainder of this section will deal with paging on x86-64, the processor architecture used in the successors of the PC we use today.

Paging on x86-64

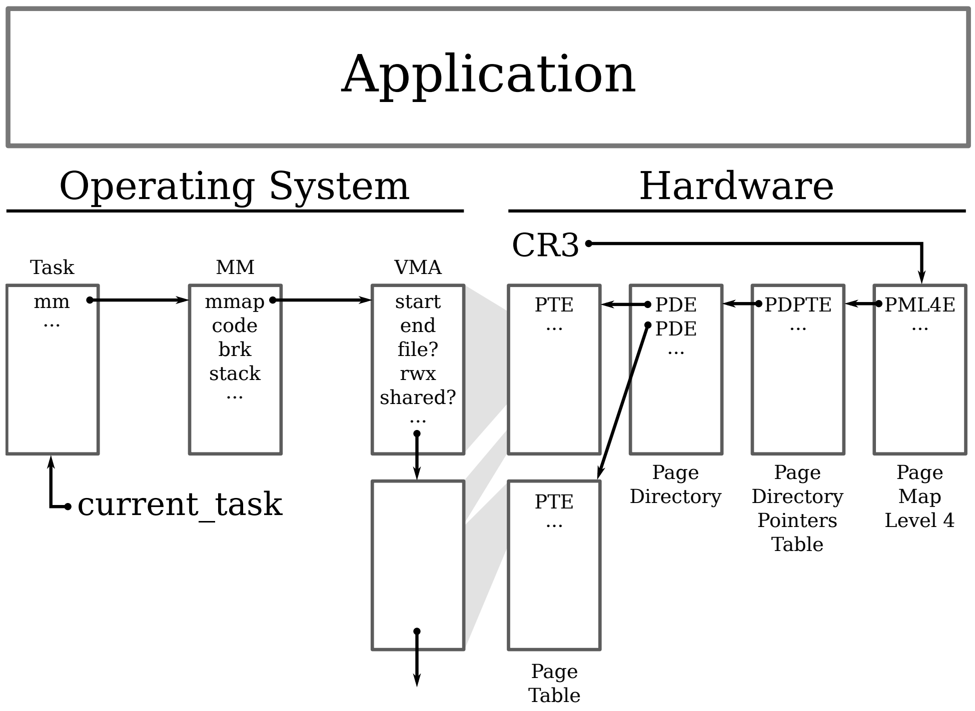


Figure 2.1: OS-level and hardware-level paging structures.

Paging on the x86-64 architecture (x86's "long mode") uses a four-level page table for the standard page size of 4 KiB, as depicted on the right hand side of Figure 2.1 on page 9. Every address space has a distinct "Page Map Level 4 (PML4)", to which a pointer is stored in the processor control register CR3. This register is updated on every address space switch. Traditionally this will also flush the Translation Lookaside Buffer (TLB) on the x86 architecture. Page tables can be shared between processes.

To avoid costly TLB flushes, the OS kernel will operate under the currently selected address space. Page tables provide special bits that restrict access to memory areas to machine code running with special supervisor ("Ring 0") privileges. Kernel memory can be mapped into the address space of every process without risks of malicious reads or writes. Furthermore, all available physical memory is mapped into a certain range of virtual addresses that is identical in every address space. These mappings can easily be shared by placing pointers to common paging structures in PML4.

These pointers point to the next level, as seen in Figure 2.1, to the Page Directory Pointers Table (PDPT). This page directory in turn contains pointers to Page Directories (PDs), which contain pointers to Page Tables (PTs). Their corresponding entries (Page Map Level 4 Entries (PML4Es), Page Directory Pointers Table Entries (PDPTEs), Page

Directory Entrys (PDEs), and Page Table Entries (PTEs)) are all one machine word (8 bytes) long and 512 of them fit into one page of memory. Every entry has a slightly different format depending on the level within the page table and whether the mapping terminates. This makes it possible to define pages larger than 4 KiB: termination at the PD level will create a page of 2 MiB, termination at the PDPT level will create a page of 1 GiB. Commonly the PTE is the final entry to resolve the virtual-to-physical mapping. The result of the resolving process is cached in the TLB to speed up further accesses to the same page. [Int13]

Page Tables in Virtualization

Full-system virtualization introduces another layer of indirection into the process of resolving memory locations. A complete second OS runs within a host OS, called the hypervisor. This means that not only need memory access to be translated from the application's virtual addresses to physical memory access, an additional mapping from a guest's physical memory to the host's physical memory is established. This is done to abstract the host's memory management from the guest and to look like real hardware to the guest OS. Such a guest is called a Virtual Machine (VM), as a complete machine with peripherals is emulated. The hypervisor sees each virtual CPU of each guest as a schedulable process and allocates it system resources similar to normal processes. The physical memory of the guest is hence userspace memory to the host (i.e., heap space), with a few extensions in hardware to make the translation be handled more quickly. Such memory can also be subjected to swapping if memory pressure arises. Different techniques were used to accomplish the additional mapping, to arrive at the Nested Page Tables we use today. They were as follows:

Paravirtualization Before the advent of hardware-assisted virtualization on x86 (Intel's VT-x and AMD-V), a hypervisor relied on paravirtualization, requiring changes to the OS of the guest. Virtual memory management was explicit by the use of hypercalls (calls by the guest OS to the hypervisor), with the hypervisor checking every page table modification for correctness with relation to isolation and taking care of appropriate TLB flushes if needed. The hardware page tables in use were the usual per-process page tables and hypercalls were needed on every process switch. [BDFH+03]

Shadow Page Tables Hardware extensions for Virtualization allow to trap the setting of CR3 in the guest. This means that address space switches can be detected without cooperation by the guest. However, the page tables provided by the guest still need to be checked for safety reasons. The guest must not be allowed to write or read outside of its confined boundaries. The hypervisor maintains an internal shadow copy of the guest's page table and provides this to the hardware. If a page fault occurs because a mapping is not yet present in the hardware-accessible shadow copy, the guest's page tables are inspected, verified and a mapping inserted into the shadow page tables. [DeBR02]

Nested Page Tables AMD's Rapid Virtualization Indexing (RVI) and Intel's Extended Page Tables (EPT) provide a more intelligent TLB, which is aware of Virtualization and the required additional mapping step. VM switches no longer need to flush the TLB as VMs and their mappings are now identified by a new identifier. Furthermore, the guest is able to set CR3 directly to its own set of page tables and the hardware extensions verify that the boundaries imposed by the hypervisor are obeyed. [Int13]

2.3 Virtual Memory Representation in Linux

An OS has a high degree of freedom in how it manages the available memory. This section contains the abstractions that Linux uses to describe its view of virtual memory.

VM Data Structures

Linux needs to keep track of processes, their address spaces and all individual mappings within these address spaces. The left hand side of Figure 2.1 on page 9 shows a graphical overview of the structures used.

Virtual Memory Area (VMA) A continuous mapped area in a process' address space is represented by a `vm_area_struct`. It contains a start and end address (both naturally virtual) for the mapping, page protection flags (read, write, exec, and shared) and a reference to a backing file if the area is not anonymous. Hints set using `madvise`(2) are kept in the VMA. Examples are mergeable (page can be merged using Kernel Samepage Merging (KSM), as described in Section 3.2.2), sequential read (pages read by the process are likely to be discarded sooner), or "don't fork" (do not inherit this area on `fork`). All VMAs belonging to the same address space are part of the same linked list and red-black tree, so that they can be retrieved. VMAs have a direct relation to `/proc/<pid>/maps`, which is described in more detail in Section 4.3.

Memory Management (MM) `mm_struct` describes the entirety of an address space. The structure is tied to a running task, it holds a reference to the binary that was loaded into the address space. Information from the Executable and Linkable Format (ELF) header of the binary are stored in this struct like the start and end address of the code and data segments. The kernel keeps track of the "program break", one way to allocate memory that uses `sbrk`(2) and `brk`(2), and the stack size. When a program is loaded the environment and command-line arguments are passed through defined memory locations, which are also tracked for later retrieval through the *proc* filesystem. Counters are included to export memory usage information to other processes in a quick fashion. The struct holds critical locks like the page table lock, which needs to be acquired for every modification of the process' page table.

Tasks For every task in the system a `task_struct` is kept to hold information relevant to the scheduler. Among this information is a pointer to the task's address space (`mm_struct`), which is relevant for context switches and to resolve page faults of the currently running process. Every CPU has a CPU-local variable called `current_task` that points to the `task_struct` of the currently running task.

Page Frames

The Linux kernel distinguishes between named and anonymous page frames in memory. For every page frame the kernel keeps a *struct page* that contains flags, for example from Least-Recently Used (LRU) tracking or if the page is not currently allocated. A *mapping* pointer signifies which type of frame it is, by pointing to either an inode (together with an offset) or to a list of related VMAs. If KSM is in use the page frame can also point to a merged page node in KSM's stable tree.

Anonymous page frames Anonymous page frames reference a list of related (anonymous) VMAs. The design of anonymous memory already includes copy-on-write semantics, so the relationship is one of one page frame to many applications, possibly even multiple locations within the same application. A VMA represents an area of virtual memory within a single process. For anonymous memory the relevant information are which virtual addresses are defined and page protection and flag bits (read, write, execute, and several more).

The Linux kernel already supports merging anonymous pages using KSM. This is implemented as a special case of anonymous page that coalesces multiple previously unrelated anonymous VMAs to point to a single page frame, which is read-only. If write accesses are allowed by the VMA a new page frame has to be allocated and the content copied. This is described in more detail in Section 3.2.2. [BoCe05]

Named page frames Named page frames reference exactly one inode in a file system. They are part of the page cache, an in-memory caching structure that holds the kernel's current view of all regular file I/O. Every file is represented by a radix tree of pages, which can be partial if a file has not yet been completely read from disk. Applications using *mmap* to map parts of files into their memory directly map the relevant part of the underlying page cache. Regular I/O using *read* and *write* (that is without the `O_DIRECT` flag on `open`(2) being set) modifies the page cache and is written back to disk after a configurable amount of time or if the user initiates it. I/O bypassing the page cache has to be done in a careful way as reading back from another file descriptor that does not have `O_DIRECT` set might yield stale data.

File Caching

The Linux kernel maintains a view on all mounted file systems in memory. With normal, non-clustered file systems the kernel's view is authoritative and it is assumed that no changes can be made on the underlaying disk unless they are committed by the kernel. This allows to provide a caching structure in memory into which content is read from disk and kept there as long as there is no memory pressure. If the cache content is modified, a periodic process will flush the changes to disk.

Such an in-memory caching structure is important when memory mappings are needed, memory areas that are directly accessible by the CPU and allow the direct access of file-backed data as memory. The *read* and *write* system calls could work with application-local buffers, but for mappings there should be a consistent view onto a file's content. This also allows the implementation shared memory through the use of a shared backing file. Modifications by writes into memory are tracked by the hardware and the page are marked as being dirty.

For every inode in memory a *struct address_space* is created that references the source device, contains the radix tree of all pages containing content of the file in memory, contains function pointers provided by the file system for various operations (e.g., read, write, and free) and allow to determine which processes have the specified file mapped.

The page frames are commonly populated by allocating a new one, submitting a DMA request to the disk to fetch the corresponding content, and by inserting the result into the radix tree. Page frames are first class objects that are passed to the file system layer directly. There is no additional abstraction layer that allows some kind of indirection. One such example would be the introduction of a Copy-on-Write (COW) logic when dealing with duplicate content in frames belonging to the page cache.

Linking and Virtual Memory

The execution of programs on a Linux system extensively involves virtual memory. When invoking an ELF binary on a Linux system, the Linux kernel will load its header into memory to determine the dynamic linker needed to resolve all library dependencies. It will then map all sections of the binary into memory that the header declares. The ELF interpreter will be loaded in the same address space and the control of the process is transfered to userspace, invoking the linker's entry point. The linker maps all needed libraries into the same address space and resolves all symbols in the application that depend on dynamic linking (relocations). The libraries are also ELF files with header containing information similar to binaries. Likewise libraries can depend on other libraries that need to be loaded. Only when the expected working environment within the address space is completely mapped and all relevant symbols resolved is the control transfered to the program itself.

Address Space Layout Randomization Address Space Layout Randomization (ASLR) is a technique that introduces randomness into the virtual memory layout of a process. This is done for security reasons to make target addresses for malicious jumps harder to guess. Exploits that try wrong jump addresses will cause their host program to be terminated and hence running the exploit's payload might be avoided.

For ASLR to be effective, it is required to compile applications as Position-Independent

Executable (PIE) and libraries as Position-Independent Code (PIC), which means that they can be loaded to any base address in virtual memory. This does increase the number of relocations needed because less jumps can be pre-determined and be made absolute. Instead additional tables (like Procedure Linkage Table (PLT) and Global Offset Table (GOT)) and indirect jumps are needed to resolve the code and data locations at runtime. Some of the runtime overhead can be eliminated by caching the symbol looking results and providing small trampolines at known locations.

The tables will most likely differ for each application because the load addresses will be picked at random on process creation, so only forking and preserving the process' memory image will retain the same tables. The application's address space will also span a larger area, which requires more page tables to be created in memory to cover these areas.

2.4 Sharing Memory Mappings

Virtual memory allows application to share memory by allowing multiple applications to access the same page frames. Sharing might both be read-only, with sharing being broken when an application tries to write, or completely read-write.

`mmap`(2) is a system call that provides applications with memory mappings within their address space. They can either be file-backed (named) or anonymous (by specifying no file, hence the name). Such mappings are created in the kernel's view on the process and reads and writes within these memory ranges are served on demand. This means that if a mapping is file-backed the data is read from disk when needed, not when the mapping is created, and the memory allocation for anonymous memory will commonly only happen on first access. The memory's protection bits (read, write, and exec) can be controlled when the mapping is requested. Furthermore, mappings can either be shared, which means that updates to a file will be visible to all processes mapping the same file, or private, which keeps modifications confined to the application applying them. [mma12]

Copy-on-Write (COW) is the technical term for the efficient sharing of significant amounts of data which is duplicated whenever a private copy is necessitated through a write operation. Such data can be stored on disk like base disk images of VMs with modifications for each VM being stored differentially in another file.

In the context of physical memory the base page frames will be marked read-only within the MMU's data structures (i.e., page tables) for all applications that share the frame. A write to the page frame by one of the sharing applications will cause a page fault and the kernel will copy (and hence duplicate) the content into a new page frame. The page table entry is then adjusted to point to the new page frame. This strategy also allows an efficient preallocation strategy, which is used, for instance, with a global page containing all zeroes. As this is a common pattern for allocation with programs written in C, one page frame can be reserved to contain just zeroes, which is then mapped into the requesting application's address space on memory allocations.

`fork`(2) is a Unix primitive that allows the creation of new processes based on the process calling it. The call will return twice, once in the parent and once in the child. On Linux, both processes will share memory in a COW fashion by default and inherit all open files. If a memory mapping is set to being shared, modifications to the memory will be visible in both processes. This works for both file-backed and anonymous mappings. [for12]

3 Related Work

This chapter presents two different deduplication approaches that have been proposed by the research community: content-aware deduplication that exploits knowledge about a content's origin to coalesce identical pages in Section 3.1 and periodic memory scanning in Section 3.2.

3.1 Content-Aware Memory Deduplication

Contemporary OSs employ several means of avoiding duplicate content in memory. For instance the content of files mapped into memory is shared among processes. Libraries are loaded into memory once and their binary code is available for other programs that link against them, as long as the linker resolves the dependency by using the same file on disk. This exploits knowledge about data on disk to deduplicate memory without scanning all or parts of the memory. Content-aware deduplication can extend this to other areas.

If content is fetched from a base system image that is shared by multiple VMs, a hypervisor can keep track of requested disk blocks and the target memory locations they were copied to. It can then refer further disk accesses from other VMs to the same locations on disk to the content that is already in memory. Disk content requested by a VM will be transfered to it by DMA, similar to what would happen on a real host talking to a disk. The hypervisor can mark the target area pre-emptively read-only and track if the content of the page frame changes. If not, it can simply be reused for another VM. [BuDR97]

On Unix-like OSs the `fork` operation will not copy every page of the forking process, but instead will share as much content as possible "copy-on-write". This means that child processes will reference pages of the parent until they or the parent write to that specific page in memory. It will then be copied and a new page frame be provided to the application. This is realized by marking the page read-only in the process' page table and resolving the resulting page fault on write appropriately.

Android – which is based on Linux – provides a central process called the Zygote which contains an instantiation of the Dalvik VM. Dalvik provides the runtime environment to Android apps by providing a bytecode interpreter and access to the libraries provided by Android. The Zygote `forks` a new process when a new app is started, sharing all Dalvik code and all initial heaps copy-on-write with its child environment. This reduces the memory footprint. [Brad08]

SnowFlock [LCWSP$^+$09] implemented this similarly for complete OSs running within VMs. They chose copy-on-access for the VMs based on a revision of a central base memory image that is guaranteed to be immutable in memory. Further changes to the base memory image will happen in a copy-on-write fashion.

For security reasons pages newly allocated to a process should be initialized with zeroes. This avoids leaking potentially sensitive content written by a previously run application to other processes [Wald02]. An OS can do this lazily, by providing one single central page filled completely with zeroes. This page can then be mapped read-only when an application is requesting memory that can be satisfied with a full and aligned page. Modifications to this page will be trapped (alike “copy-on-write”) and the page replaced with a separate page frame that really is initialized with zeroes. The advantage is that memory that is never written to but still allocated in advance does not need to be provisioned “for real” to the process [BoCe05].

3.1.1 Disk Content Sharing in Memory

Different approaches to disk content sharing in memory have been proposed. One is to be completely transparent. Examples of this are the interception of DMA and Multics’ approach of segments that will be retrieved on-access. Another approach is to require cooperation of multiple entities like it is the case with IBM’s Execute-in-Place filesystem or the Satori changes to the Xen hypervisor.

Intercepting DMA The Disco Virtual Machine Monitor (VMM) by Bugnion et al. [BuDR97] implements memory sharing through the interception of disk DMA requests. Disk blocks are read into memory once and kept in memory managed by the VMM. To satisfy the VM’s DMA request, Disco will then map the memory location read-only into the VM’s physical memory to the target location of the DMA request. In the VM, in this case running IRIX, the memory area will become part of the buffer cache. This effectively allows multiple VMs to share their buffer caches. Writes to the disk will be private to each VM and be logged by the VMM, providing a copy-on-write disk environment. Disks can optionally be made non-persistent, thus providing a snapshot-alike VM environment in which VMs can be restored to an original state with little disk space and memory overhead.

Multics Multics provides the programmer with direct access to all on-line information, by defining so-called segments that are transparently fetched from the I/O device containing them when the processor accesses them. Memory is then a transparent disk cache. All processes accessing the same data are using the same area in main memory. [BeCD72]

Execute-in-Place Filesystem IBM’s Execute-in-Place file system [IBM04] uses memory segments that are shared between multiple VMs to eliminate multiple copies of binaries in RAM. These segments are read-only and not backed by real disks. The key of this solution is that the CPU is able to address those shared segments directly including code execution without any need for previous load or fetch indirections. This eliminates the page cache copies which are needed when the backing store cannot be addressed directly. Pages of binaries found on this virtual disk are mapped from this segment directly into userspace programs. To address files a simple file system based on *ext2* is created in memory. Similarly it is possible on IBM mainframes to split the Linux kernel into a shareable (read-only) and a private (read-write) part. The read-only part can be shared between VMs running the same kernel version.

Satori Miłós et al. [MMHF09] propose in their paper additional "enlightenments" (paravirtualization) that can be added to the kernels of VMs to arrange sharing of identical content and to improve memory allocation to different VMs. One such modification is the introduction of a "repayment FIFO" that contains pages the guest is willing to give up in situations of memory pressure. The guest provides writeable private volatile pages that can be taken away without synchronous involvement of the guest. If the host would otherwise fail to allocate memory to a VM it can instead take a page out of such a FIFO instead of implementing paging to disk.

The proposed change for actual memory deduplication is the introduction of *sharing-aware block devices*. If multiple VMs run from the same disk image, only differences are stored per-VM. The storage layer is aware of these differential images and will map unchanged contents of the base image for all VMs to the same disk blocks. Such requests are tracked and the result be mapped read-only into the guest's memory. When another VM requests the same block, the hypervisor can then check if the content is still in memory and map the same page frame to the requesting VM, effectively saving a repeated read request going to the disks and saving the duplicate in the other VM's page cache. This approach does not involve memory scanning and does not incur a significant runtime overhead. Their sample implementation targets the Xen hypervisor.

3.2 Periodical Memory-Scan Deduplication

Another approach to find duplicate content in memory is periodically scanning it. The following sections describe two example implementations of such memory scanners: one in VMWare ESX and one in Linux.

3.2.1 Memory Scanning in VMWare ESX

Waldspurger [Wald02] describes how VMware ESX Server employs memory sharing between VMs to reduce overall memory pressure and provide higher levels of memory overcommitment on x86. It is assumed that modifications to the kernels within the VMs are not possible, for instance, in the case of Windows, the source code might not be available. Hardware virtualization extensions were not yet available in the CPUs of the time.

The general approach of Waldspurger is the following: Page contents are hashed with a 64-bit hash function and the resulting hash is used as a key for a hash table of page frames already marked as being read-only. Writes to these frames would cause any existing sharings to be broken, the content copied and a new page frame be given to the VM ("copy-on-write"). Hence their content is stable and not subject to changes. If a hash match is found within said hash table, a full content comparison is performed. If and only if both pages have the identical content, the redundant copy is reclaimed and the reference of the guest's physical memory adjusted to point to the shared copy. At the same time a reference counter on the shared page frame is increased.

To find sharings more quickly the page's hash will be opportunistically cached as a hint if no match was found in the hash table. If a later hash operation yields the same hash as a hint, ESX Server will establish if the hinted page changed in the meantime and merge both pages if this is not the case. Pages with identical hashes but diverging content will not be shared on the grounds that the hash collision probability is low.

The author proposes several strategies on how memory can be scanned: sequentially, randomly or by employing heuristics. Such heuristics can include pages marked read-only by the guest OS or pages from which code has been executed. Both could indicate binaries being present in those pages. VMware opted for a random scan strategy using a fixed rate of pages being scanned per time interval.

3.2.2 Kernel Samepage Merging (KSM)

Arcangeli et al. [ArEW09] implemented memory sharing in the Linux kernel. The module is called KSM and deals with the scanning for and merging of duplicate anonymous pages in memory. A kernel thread scans n pages of memory every m seconds for duplicate content.

Each page scan iteration involves checking the current page's content against the stable tree, a data structure containing all already merged pages. The corresponding page frames are referenced at least once and write-protected in memory. If a match is found, the reference to the page being scanned in the page table can be replaced with one to the merged copy and the page's frame can be reclaimed.

If no match is found, KSM checks if the page in question has not changed between two consecutive scanning rounds. If this is the case, it is compared against the unstable tree, containing all pages that have previously been scanned but not yet merged. If a match is found, a new copy of the page's content is created and both old copies are replaced with a reference to the new page frame. If no match is found, the page is inserted into the unstable tree.

KSM was designed with efficiency in mind: only pages that are likely sharing candidates should be scanned and lookups in the various tree structures should incur a reasonably low overhead. It relies on applications providing hints about which pages should be merged. These hints are implemented through the `madvise`(2) system call: applications set the `MADV_MERGEABLE` flag on such memory areas. This flag was introduced by KSM. Applications need to be manually adjusted to take advantage of this facility; one big user is *qemu* – originally a full-system emulator and now providing the hardware emulation for virtualization –, which brings memory deduplication to virtualization on Linux. *qemu* allocates the guest's memory on its heap and hints it, so that duplicate pages can be reclaimed. Arcangeli et al. cite scientific reconstruction jobs at CERN as a positive example: they were found to create many equal pages. These could be deduplicated by letting a wrapper around `malloc`(3) hint the resulting allocations. Furthermore, the authors explicitly states that it is possibile to merge kernel memory and page cache contents using one layer of virtualization, with KSM running on the host.

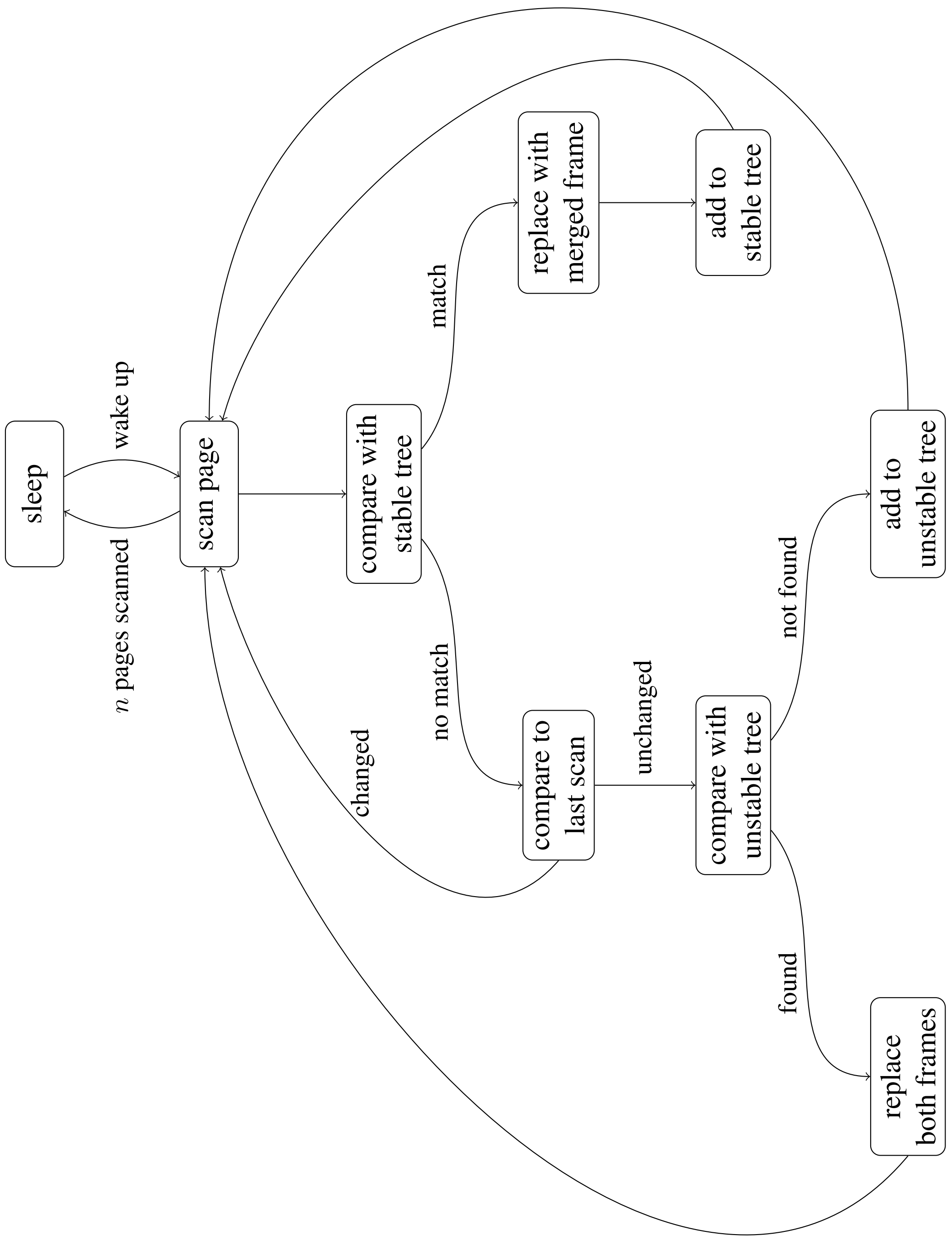


Figure 3.1: Flowchart of KSM's main thread.

Figure 3.1 on page 3.1 shows how the KSM thread operates. It sleeps a predefined amount of time (*sleep_millisecs*) and then wakes up to scan a maximum amount of pages (*pages_to_scan*) within the hinted areas of memory. The stable tree contains already merged page frames, their content is stable and will not be changed. Modifications will allocate new page frames. Scanned pages will be checked against this tree first and be replaced with the merged copy if they are identical. If the page in question does not match with any in the stable tree, KSM will do a comparision against the checksum results of the last complete run. If a match is found the matching page needs to be rechecked, as it might have changed in the meantime. Is this not the case, both references will be replaced with a copy of the data placed in a newly allocated page frame. If no match is found, the page will be added to the unstable tree. Finally, when the defined amount of pages was scanned or if there is nothing more to scan, the thread will go back to sleep.

ksmtuned There is little research as to how the KSM parameters should be set. One tool, *ksmtuned*, has been developed, which turns off KSM as long as some amount of memory is still free and turns it on otherwise. The scanning frequency (the variable "pages_to_scan") can be increased if more memory needs to be reclaimed. Still, *ksmtuned* does not have or provide any more information about reclaimable memory itself, it just tries to reduce memory usage at the expense of more CPU cycles spent scanning.

Exploiting Disk I/O for Scan Priorities XLH is an extension to KSM that introduces a priority queue of page frames to scan, based on them being the target of DMA disk I/O requests. These hints are stored in a bounded buffer and are considered as merging candidates first, with non-hinted page frames only being considered when the list of hints is exhausted. This approach can be implemented completely within the host without assistance of the guest or the storage system. [MFGR+13]

Difference Engine on Xen

Savage et al. [GLVS+10] add page sharing through memory scanning similar the approaches of ESX server as presented in Section 3.2.1 and patch-based memory deduplication to the Xen hypervisor. Patching allows to retain only one copy of the base content plus a small patch in memory if it is similar to an existing other page in memory. This is only beneficial if the patched copy is not currently part of the working set, as a memory access to this page will require the hypervisor to create a copy and patch it to restore the original content of the deduplicated page. Furthermore, Difference Engine adds page compression of currently unused pages to Xen, which allows to save some memory without paging it away to disk where it will be much slower to retrieve than an uncompress operation in main memory.

4 Implementation

This chapter presents the different approaches we took to determine memory deduplication opportunities of programs running on Linux. It documents in Section 4.1 an approach that was initially taken to measure memory deduplication opportunities over time, which did not succeed mainly due to the lack of trace files. Section 4.2 describes our efforts to implement the merging of file-backed pages in Linux and how they failed. Section 4.3 describes the interfaces Linux provides to access another application's memory image, which our memory dumping tool uses. Section 4.4 presents our approach on measuring the deduplication potential based on the data obtained by the memory dumping process.

4.1 Tracing Memory in a Full-System Simulator

We originally intended to use the memory access analysis framework developed by Marc Rittinghaus [Ritt12] as the foundation for the work. It extends the full-system simulator Simics by Wind River Systems, Inc. [Wind] with the possibility to collect and store any memory write operation during the course of the simulation. Simics is a powerful simulator that can emulate a wide range of CPUs and is commonly used as a virtual platform for embedded development and debugging.

The extensions by Rittinghaus store all reads and writes executed by the virtual CPU. Furthermore, the OS is instrumented to pass task information such as the currently running process, task creation and destruction, and memory map information to the simulator via the use of a special CPU instruction. This allows the inspection of the memory content at any given time, annotated with precise information which processes have mapped which page frames. For named page frames their source filenames are as well. A Graphical User Interface (GUI) tool is provided by Rittinghaus which parses the information up until a specific time code and provides the user with a visualization of this data.

Storing all reads and writes allows to see any sharing opportunity, even if it persists only for a small amount of time. Furthermore, information from the future could potentially be used. This would allow to implement a perfect oracle offline algorithm for memory deduplication, yielding the minimum amount of memory in use for every point in time. In addition one could distill prediction functions from the data that can be used by a memory merging algorithm to check if a page is likely to be shared in the future. This makes this approach a valuable data source, as it offers capabilities that go beyond running experiments on current hardware. It is not possible to track every write access on today's CPUs; memory snapshots have to be taken after predefined timeslices instead.

The framework also includes the possibility for users to provide scripts written in C# that analyze the data. The scripts receive every event sequentially in the order in which they have been recorded. The script is responsible for tracking all the state it needs for conducting the analysis, based on the information it receives. Random access within the data streams is not possible. Due to the vast amount of events and the time spent processing each of them, this analysis process is lengthy and very memory-intensive. Rittinghaus states that "for a single simulated minute of a single-core 20 MHz processor a billion trace entries (10 bytes each) are generated" [Ritt12]. Even though they are stored compressed, they still need to be decoded and fed into the analysis system. The scripts can only keep a fraction of this data in memory.

The main drawback with the extension of Simics itself was that the instrumented execution of any application within the OS on a simulated processor is very slow. A Linux kernel build, a common workload to be analyzed as, for instance, by Miller et al. [MFGR$^+$13], had to be terminated after a few weeks, as it progressed too slowly. The slowdowns encountered were more than $5000\times$, it would have taken ten months for the build to compile in this setup [RMHB13]. Traces that were created only lasted a single digit count of minutes of simulation (i.e., in real time as seen by the emulated 20 MHz CPU). These already took hours to create. These early experiments suggested that it would not be possible to generate meaningful traces in a reasonable time frame. Hence the amount of available memory traces, created with this tracing framework, was severely limited.

Although the Simics-based solution provides a lot of the capabilities we needed, it seemed impossible to generate enough different traces to draw meaningful conclusions from the gathered data. This was the main reason why we did not pursue this approach further. Instead we tried to improve Linux's own memory merging capabilities.

4.2 Implementing Named Merging on Linux

After it became clear that using the extensions to Simics was not a viable solution, we turned to merging file-backed pages and its integration into Kernel Samepage Merging (KSM) on Linux. This would allow workloads that have an inherent duplication of files on disk (e.g., zero-install or container environments) to benefit from memory deduplication. KSM only handles the merging of anonymous pages at this time. Furthermore, benchmarking would be based on an improved real-world algorithm. This would be a direct benefit for workloads on Linux, as their memory use would be reduced. However it was clear that it would provide less insight into what would be shared and why. Some debugging code could help to at least list the source files or processes that participate in a given page frame sharing.

A preliminary patch existed to implement the merging of file-backed pages, also called named merging, developed at our chair for earlier research on this topic. However, it proved to be too unstable for our purposes. It constantly crashed the kernel (through kernel panics) or caused exceptional, unexpected conditions (i.e., a kernel oops). Page frames were merged onto each other partially without write protections, that is, without Copy-on-Write (COW), which caused page modifications to leak into other processes. A deeper analysis revealed the following problems:

The sharing of named page frames in Linux is difficult due to missing abstractions. There is currently no abstraction in the kernel that could deal with one page frame containing the content of multiple inodes and hence different files. Instead, file system code gets pointers to a `page` struct, Linux' page frame descriptor, passed into its functions. The file system then needs to deduce the corresponding inode (through `page->mapping`, which points directly to it) and offset from it.

Whereas KSM was able to work with just modifications to the memory management layer, the implementation of named merging would need various patches across the kernel's source tree to adjust for the changed semantics. To complicate matters further, `page->mapping` is directly accessed by some drivers. This situation is exacerbated by kernel code implicitly assuming that these page frames are directly writeable by file system code, to the degree that DMA requests are posted with these page frames as their target. This makes intercepting changes and implementing COW on page frames belong to the page cache difficult, especially in the absence of an I/O MMU that could be programmed to trap write accesses.

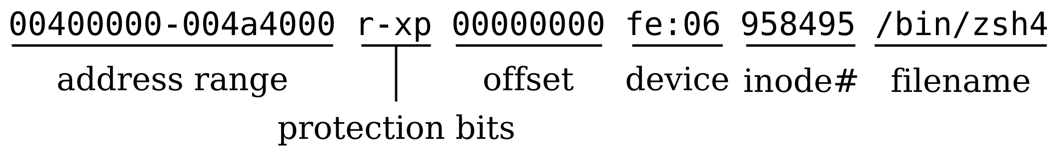
Modifying KSM to implement the merging of file-backed memory would have allowed KSM merge the pages in various scenarios and to retrieve the resulting memory deduplication counts as a result – even split by type, that is, how many are file-backed, anonymous, or mixed. The difficulties mentioned above let us refrain from the approach of implementing file-backed merging into KSM. In principle the merging can already be accomplished by running the workload in a VM, but the insight about the page type is then lost due to a semantic gap. The hypervisor does not know how memory is used within the VM. We then turned to an approach that dumps application memory content from within Linux.

4.3 Dumping Process Memory Contents on Linux

Instead of using either a full-system simulator or observing the runtime behavior of KSM we settled on producing memory dumps of scenarios we defined as representative of some of today's workloads.

This section explains the workings of specific Linux system calls and interfaces to present how memory content can be obtained. Further information about them can be found in the corresponding manual pages or kernel documentation listed at the end of each paragraph.

The Linux `/proc` file system, as documented in `proc`(5), provides data about the memory usage of every process in the system. The file `/proc/<pid>/maps` (with `<pid>` replaced in this and the following filenames with the process' numeric ID) lists all the "currently mapped memory regions and their access permissions" [pro12]. A sample line of the virtual file's content can be found in Figure 4.1, which shows a shell's binary being mapped into memory. The information found in this file includes the begin and end address of every mapping, the permissions (read, write, execute, shared, and private) and information about named mappings, if applicable (offset in the file, device, inode number, and filename). [pro12]



```
00400000-004a4000 r-xp 00000000 fe:06 958495 /bin/zsh4
```

Figure 4.1: An exemplary line showing the information found in `/proc/<pid>/maps`.

Reading a process' memory image It is possible to read and write a process' memory image by accessing `/proc/<pid>/mem`. To be allowed to use this facility from another process, it needs to become the `ptrace` parent of the process to be inspected. This is accomplished by calling the `ptrace`(2) syscall with `PTRACE_ATTACH` specifying the target process identifier. `ptrace` itself also allows *machine word*-based access using a peek/poke mechanism to the target process' memory, but this would require calling it many times, whereas `/proc/<pid>/mem` allows accessing the memory content by using `open`(2), `read`(2) and `lseek`(2). It is also possible to map memory regions of the target application into the inspecting application's own memory. The full memory image cannot be mapped at once because both processes are constrained by the maximum addressable virtual memory and one would require twice the available virtual memory for the inspecting process. [pro12] [ptr12]

Attaching to a process with `PTRACE_ATTACH` will stop the process for subsequent debugging. The kernel will send a `SIGSTOP` to the attached process and `waitpid`(2) can be used to wait until the process has actually stopped executing. This can be used to retrieve a complete memory snapshot of the process. [pro12] [ptr12] [wai10]

If a process uses multiple threads and thus represents a *thread group*, only one thread will receive the `SIGSTOP` when attaching to the process using `ptrace`. In this case all members of the same *thread group* can be retrieved from the directory `/proc/<pid>/task`. To prevent any thread from further modifying the process' memory image they all need to be stopped. This is accomplished by using the `tgkill`(2) system call. As this system call is not wrapped by glibc, it needs to be called directly for every thread ID in the specified thread group ID, which is the process ID of the parent process. [tki12]

Physical frame information Linux exports some information about a process' page tables through the `/proc` file system as well. `/proc/<pid>/pagemap` "contains one 64-bit value for each virtual page" [pag09]. This value contains the number of the physical page frame if mapped, a swap offset if swapped or indicates that no page frame has been allocated yet for the specified memory address. This information can be used to avoid allocating page frames for data that has not yet been needed by the inspected application. [pag09]

Globally, for the use of the system administrator *root*, two `procfs` files describe the use of the available physical memory: `/proc/kpagecount` contains one number per physical page frame, describing how often the corresponding frame is mapped into a process. `/proc/kpageflags` contains the in-kernel flags stored per page frame, which describe a page frame's content (e.g., if it's a named page, or anonymous memory, or a page shared by multiple processes through the use of KSM). [pag09]

Conclusion By using all these components together, we can attach to multiple processes at once and take a consistent memory snapshot of all of them, provided that swapping is disabled. Memory areas for which no page frame has been allocated yet can optionally be dumped, the default is not to force the kernel to allocate physical memory for these areas. Files can also be mapped with `MAP_PRIVATE` and changes to such memory areas are not visible to other processes or in the underlying files. To track such changes the original file content can also be included in the memory dump. [mma12]

4.4 Analyzing Offline Dumps

The data generated by the dumping process as described in the previous section is used by an offline analysis process to determine sharing opportunities within a set of user processes. To identify identical content it is enough to generate and retain hashes of the page content in memory when iterating over the files' content. For this work the MD5 hash algorithm [Rive92] was used, primarily for its size and speed. The analysis tool also allows to dump the page content for a given hash, in case interesting hashes appear in the analysis phase.

One interesting figure that can be deduced from offline memory dumps is how much physical memory was actually allocated to the set of inspected processes. The length of the set of unique page frame numbers that are found in the pagemaps across all dumped processes is the allocated memory footprint.

Linux already includes techniques like `fork`(2) and `mmap`(2) that allow less physical memory to be allocated (as described in Section 2.4). We can count how many processes have a given page frame mapped in their address space. The sum of these numbers indicates the amount of memory that would be needed to host the given applications without these deduplication measures in place.

Let H be the set of all different page content hashes and P be the set of all page frames. h is a single hash in H and p a single page frame found in P. p is assumed to be uniquely identified by its page frame number. The following functions are useful to describe properties of elements in P:

$$\mathrm{anonymous}(p) = \begin{cases} 1 & \text{if page } p \text{ is anonymous} \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

$$\mathrm{named}(p) = \begin{cases} 1 & \text{if page } p \text{ is named (file-backed)} \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

$$\mathrm{hash}(p) = \text{hash result of page } p \tag{4.3}$$

$$\text{anonymous\_exists(h)} = \begin{cases} 1 & \text{if } \exists\, p \in P : (\text{hash}(p) = h) \land \text{anonymous}(p) \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

$$\text{named\_exists(h)} = \begin{cases} 1 & \text{if } \exists\, p \in P : (\text{hash}(p) = h) \land \text{named}(p) \\ 0 & \text{otherwise} \end{cases} \tag{4.5}$$

4.4.1 Offline KSM

The following two sections present two different deduplication strategies. Shared pages are those remaining after applying such a strategy. Sharings denote the amount of pages that no longer need to be kept in memory. When multiplying the two values with the page size (in this work consistently 4 KiB), the memory amount of both values follows naturally.

KSM merges anonymous pages in memory by periodically scanning them for duplicate content. We are primarily interested in content that can potentially be merged, regardless if the content remains stable. Hence, assuming an ideal KSM scanner, we identify all identical anonymous content and count such page content just once. The remaining pages are added up as they have been deduplicated by normal Linux means, or not, which is the case for named pages.

The count of pages that remain post sharing (“shared”) and the deduplication potential (“sharings”) given a set of hashed pages are determined as follows:

$$\begin{aligned} \text{KSM shared} = \sum_{h \in H} \Big(& |\{p \in P : (\text{hash}(p) = h) \land \text{named}(p)\}| \\ & + \text{anonymous\_exists}(h)\Big) \end{aligned} \tag{4.6}$$

$$\begin{aligned} \text{KSM sharings} = \sum_{h \in H} \Big(& |\{p \in P : (\text{hash}(p) = h) \land \text{anonymous}(p)\}| \\ & - \text{anonymous\_exists}(h)\Big) \end{aligned} \tag{4.7}$$

4.4.2 Full Content-Based Page Sharing

KSM is currently not able to deduplicate all application memory in page-sized blocks, as it only considers anonymous memory. This is an engineering problem and hence we can assume that a perfect algorithm could find all redundancies within the application’s memory content. In this case, we can simply count how much different memory content can be found in the dumps. The sum of different content hashes will yield the maximum deduplication, if we do not consider sub-page sized blocks.

To determine the amount of unique page contents in memory we simply calculate the size of the set of different hashes. The deduplication potential consists of all the other pages that remain.

$$\text{CBPS shared} = |H| \tag{4.8}$$

$$\text{CBPS sharings} = |P| - |H| \tag{4.9}$$

5 Evaluation

This chapter presents the scenarios we used to determine memory deduplication potential in native applications. Desktop scenarios can easily be expanded to multi-user environments by invoking them multiple times on the same host machine. We also look at how a zero-install environment, that is, one that lets the user provide applications instead of them being centrally provisioned, impacts redundancy in memory. Section 5.1 describes single-user, multi-user and zero-install scenarios, as well as a game server scenario, which we used to represent realistic workloads. Section 5.2 presents the results we obtained by using the redundancy analysis process as explained in Chapter 4.

5.1 Scenarios

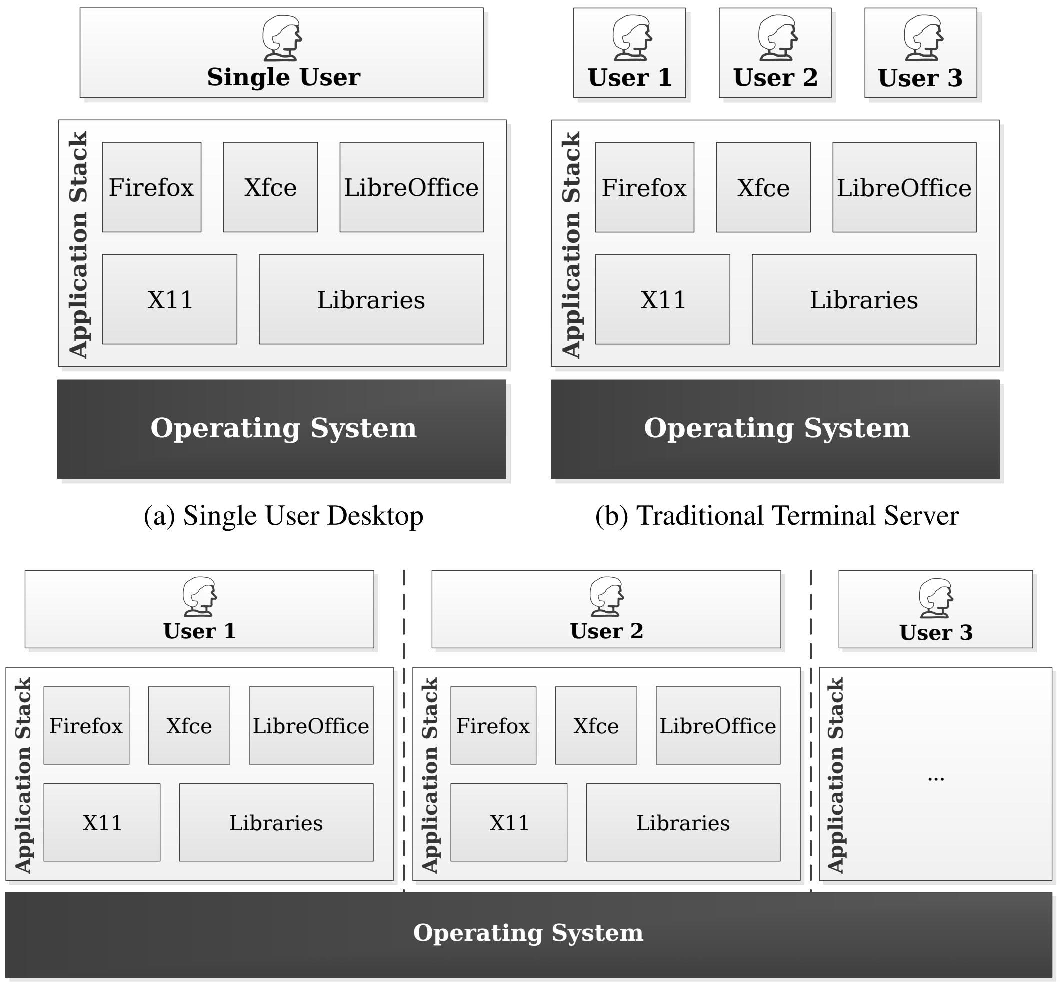
In order to determine the memory redundancy within native applications, we constructed four workloads. These consist of three desktop scenarios and one server scenario. All were run on a 64-bit installation of a pre-release of Debian 7.0 (Wheezy).

5.1.1 Desktop Scenarios

We picked three similar desktop scenarios, that differ on the number of instances on one host: single user desktop, traditional terminal server, and zero-install terminal server. Every scenario shares the same applications, in the following paragraphs we will describe them in detail.

Single User Desktop In our single user desktop scenario (visualized in Figure 5.1a on page 32), one user logs in and starts his desktop environment (Xfce 4.8), a web browser (Firefox 17) and a text processor (LibreOffice Writer 3.5). An instance of the X Window System is needed to display the windows on the screen and to process input. This scenario can serve as a baseline what memory could be saved on a traditional Unix development workstation using different deduplication techniques.

Traditional Terminal Server With traditional terminal servers, multiple users log into the same machine and have access to the same set of centrally installed software. Software will be shared in memory, but the desktop environments and the applications will still allocate data on the heap that might or might not be sharable. The concrete scenario mimics the Single User Desktop case by starting the same programs, but in three different X session. The binaries are provided by the system and are shared among the different users. This is depicted in Figure 5.1b on page 32.



(a) Single User Desktop

(b) Traditional Terminal Server

(c) Zero-Install Terminal Server

Figure 5.1: The desktop scenarios we used to evaluate redundancy in memory content of native applications. The scenarios differ in the number of users and which parts of the system are shared.

Zero-Install Terminal Server On zero-install terminal servers users can provide either just their own copies of all the applications or even core libraries they want to use. This increases the flexibility of users, as they no longer depend on administrators installing new applications or updating existing ones to specific revisions. This flexibility comes at the expense of increased disk and memory use if multiple users run the same applications. The OS kernel is still shared among the users. A visualization can be found in Figure 5.1c. This scenario could gain a higher importance with the emergence of Linux containers that provide full root access to users, while still being completely walled off from other users on the same system. Such lightweight VMs might allow users to install their favourite Linux distribution and hence duplicate the base OS and all applications on disk, causing duplication in the host's page cache as well.

Test Setup The desktop scenarios have all been scripted, to provide an automatic and repeatable way to gather memory content: a framebuffer-backed X server is started and various clients connect to it. The test framework waits for some time for the programs to settle and then starts taking snapshots. For the game server scenario three servers were started manually and snapshotted after they settled. Between all measurements the VM containing the tests was rebooted.

The framebuffer-backed X server has been provided by the `xpra` remote desktop tool which starts Xorg with the dummy video driver. This is alike to the Xvfb server which VNC uses, except that it uses an X server as released by the X.Org project instead of a fork.

5.1.2 Game Server

Game servers traditionally do not support multi-tenancy. Free modifications to the game tree are required, to incorporate new game content and functions. Often this is also used to modify a game's behavior, a process called "modding" (e.g., the introduction of new game modes). Not all configuration settings are exposed through proper configuration files or command-line switches, instead game files have to be modified directly. This implies that game content needs to be duplicated on disk for every customer. This is depicted in Figure 5.2. For licensing reasons, however, it was not possible to let different clients connect to the servers.

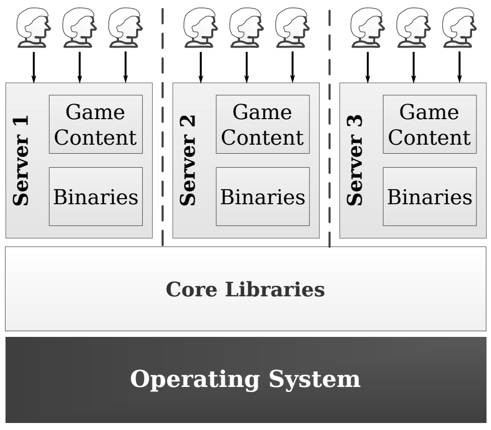


Figure 5.2: Approximation of the workload scenario "Game Server". Only few core libraries are shared between the different servers, all other content is duplicated.

This offers opportunities to deduplicate binaries, possibly content in RAM, depending on how the server is implemented (whether it is using `mmap` or not) and the heaps, which might be similar between different server instances. In this case we are measuring the memory consumption of Valve's dedicated server for Counter Strike: Global Offensive (CS:GO) on Linux. The reasons for picking this game are two fold: It is a recent game that enjoys some popularity at the time of this thesis and it is hence a real-world scenario to show if memory deduplication is worthwhile to apply in this setting.

Multiple servers are started on the same host, each with their own copy of the game's binaries and content, but with identical configuration options. The measurements include a few more processes like screen and bash, which only contribute a slight amount of memory to the total consumption figures. No clients are connected to the server, it is in hibernation.

5.2 Results

This section presents the results of our evaluations. We obtained them using the memory dump tool as described in Sections 4.3 and the following with the workloads outlined in the previous section.

The resulting tables list the following values. Memory consumption is measured relative to a base case that is noted in the caption of the table.

No sharing denotes the memory allocation if no pages were shared. This excludes sharing that happens naturally like `mmap` and `fork` and breaks up all coalesced mappings onto the same shared page frame.

`mmap/fork` stands for the actual memory allocation of the applications with swap deactivated. All deduplication that was already applied by the OS (zero page frame, shared memory, library sharing, etc.) is already accounted for. This means that page frames that are allocated once in memory are also just counted once.

KSM hint all denotes what would happen if all anonymous pages with identical page content were shared. All named pages are counted in the same way as `mmap/fork`. This is an oracle figure of the best case of all identical content being found and deduplicated.

Full Content-based Page Sharing (CBPS) points out the full page-based deduplication potential if all identical content could be shared, be it named or anonymous memory. This finds duplicate content within named pages as well, as **KSM hint all** focusses exclusively on anonymous pages.

Desktop Scenarios

The results of the analyzing process as described in Section 4.4 were normalized relative to the single user, `mmap/fork` scenario, which allocated 480 MiB of RAM. The input to the normalization was the median of 6 runs, to reduce the impact of possible outliers. However, the results were pretty stable. They can be found in Table 5.1. As can be seen, the

Figure 5.3 is a graphical visualization of this data for the one, two, and three user Terminal Server workloads. The same for the Zero-Install case, where the applications were not shared, can be found in Figure 5.4. It can be seen that the full CBPS sharings are more significant in the Zero-Install case, whereas KSM finds almost all sharings in the plain Terminal Server workload.

	1 user	2 users		3 users	
	single user	terminal	zero-install	terminal	zero-install
no sharing	1.10	2.20	2.20	3.32	3.33
`mmap/fork`	1.00	1.83	1.99	2.68	3.02
KSM hint all	0.48	0.64	0.83	0.81	1.20
full CBPS	0.45	0.60	0.61	0.77	0.78

Table 5.1: Relative memory consumption over the single user case that allocated 480 MiB of RAM.

Page Types Table 5.2 shows the distribution of shareable pages by type. It is interesting to note that there were no shareable pages across page types except for the special page containing all zeroes (the one mixed page that can be shared). All sharings remain within their own domain. This further discussed in Chapter 6.

	1 user	2 users	3 users
anonymous pages	29707	43520	60480
named pages	25490	30564	35675
mixed pages	1	1	1

Table 5.2: Shareable pages by type, taken from the results of the zero-install scenario.

Game Server Workload

Looking at the game server workload, we see in Table 5.3 that it exposes large sharing opportunities if multiple servers are started on the same host. There is, however, almost no self-sharing (i.e., sharing within a single server) to be found (3% within anonymous memory, comparing **full CBPS** with **`mmap/fork`** in the single server case).

The main data a game server works on is geometry information of a game environment (the "map"). This particular implementation of a game server loads map data into memory explictly instead of setting up memory mappings, likely because the map data is stored on disk in compressed form. Large heaps (anonymous memory) are allocated to hold the uncompressed map data. This makes hinting of all anonymous pages to be merged with KSM very useful, saving 110% of the single server case when running three servers on the same machine. Additionally a further 20% could be saved if the duplicated binaries and libraries were merged in memory.

Due to the game server install being duplicated across users, the only libraries `mmap` and `fork` are able to deduplicate are part of the system C library (glibc). Full CBPS can deduplicate the other game binaries and libraries found in other directories as well.

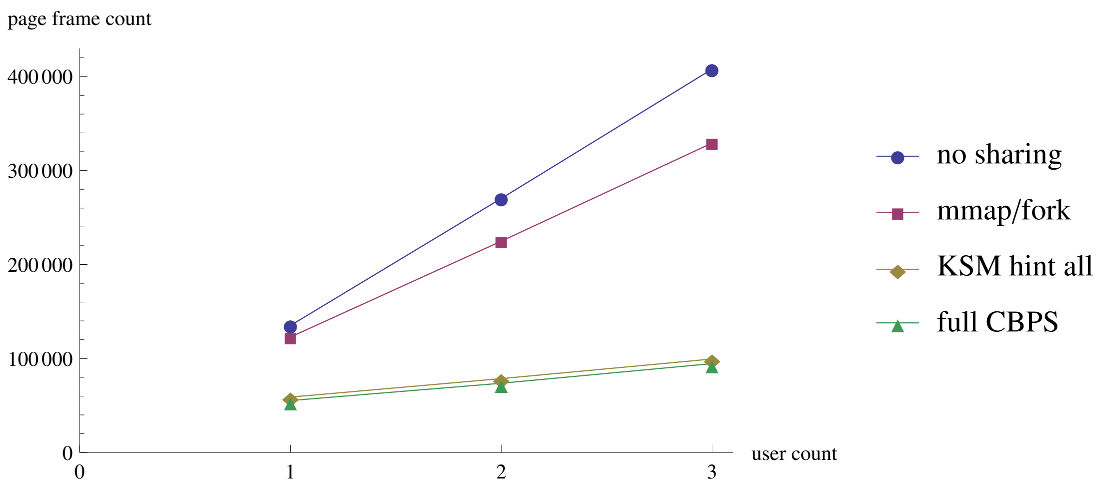


Figure 5.3: Memory consumption in the Terminal Server scenario in page frames. Full CBPS only brings marginal benefits over KSM. KSM is able to reduce memory use significantly.

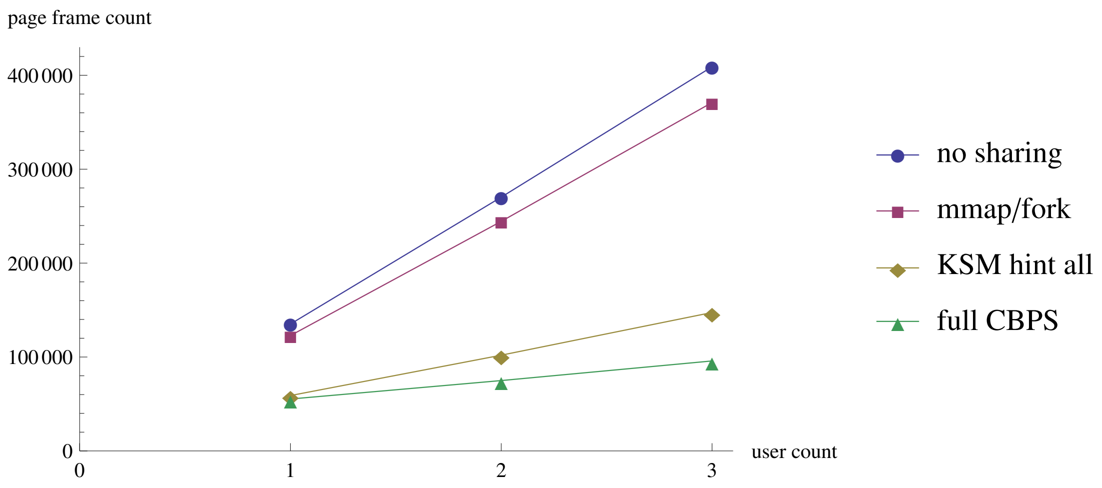


Figure 5.4: Memory consumption in the Zero-Install Terminal Server scenario in page frames. Full CBPS is able to reduce memory use to values similar to those being found in Figure 5.3. KSM identifies less redundancy and `mmap`/`fork` is less effective in this scenario.

	1 server	2 servers	3 servers
no sharing	1.00	2.00	3.00
`mmap`/`fork`	1.00	1.99	2.97
KSM hint all	0.97	1.44	1.90
full CBPS	0.97	1.34	1.70

Table 5.3: Memory consumption of CS:GO servers relative to the single CS:GO server case that allocated 147 MiB of RAM.

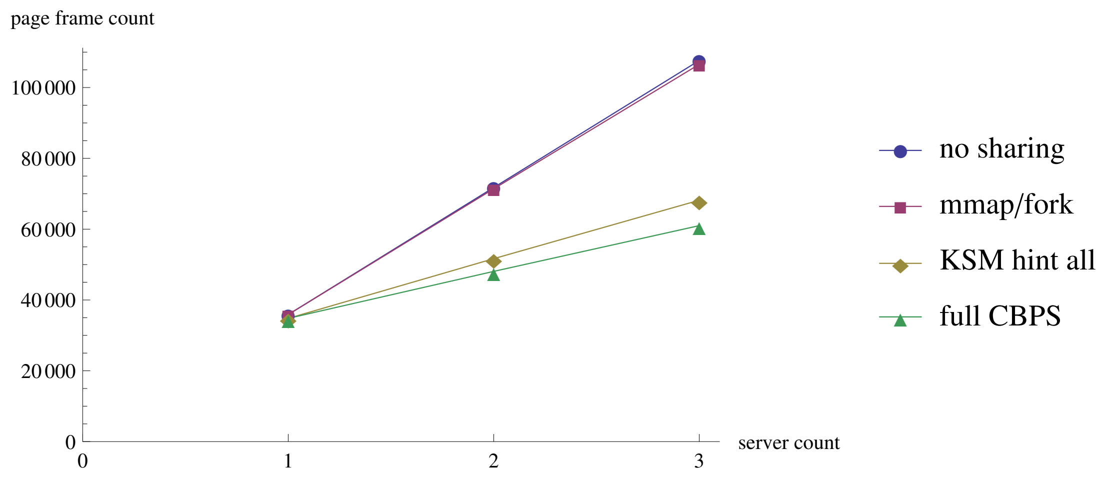


Figure 5.5: Memory consumption in the Game Server scenario in page frames. Almost no sharing is accomplished by `mmap`/`fork`. KSM is able to identify most redundancies and a few more are found by full CBPS.

6 Discussion

This chapter discusses sharing opportunities within native applications on Linux in Section 6.1. Section 6.2 presents previous studies on memory redundancies within VMs. Section 6.3 discusses our own evaluation results (as found in Chapter 5) and their implications. Section 6.4 gives a list of improvements which are desirable to benefit more from memory deduplication. Finally Section 6.5 provides a summary.

6.1 Sharing Opportunities in Applications

In the following we discuss common native applications, for which memory deduplication might be desireable.

6.1.1 Desktop Applications

Desktop applications differ from server applications in that they are usually long-running processes that load more libraries (graphical toolkits providing widgets, interface libraries to the display server, and the libraries that they need to fulfill their intended behavior). Server applications traditionally process many short-lived requests from various users, whereas interactive desktop applications can justify a longer startup time and larger memory allocations to provide immediate feedback to one single user once they are running.

Desktop applications Graphical applications typically rely on a GUI toolkit like GTK+ or Qt to provide a uniform appearance across multiple applications. A desktop environment usually mandates one single toolkit. It is responsible for providing widgets like buttons, menus, input fields and more, but also for window, color and buffer management. It also provides a main loop where all input events are processed.

Most widget toolkits support theming to provide a common look. These themes provide images, icons and color instructions that all influence how the application is rendered. These images and icons will be loaded in all applications anew. The GUI toolkits GTK+ and Qt rely on icon cache files that are mapped into all applications to avoid further file system lookups and to save memory. Large icon themes can take up to 150 MiB in size.

GUI toolkits also need a number of libraries to accomplish their task: interface libraries to the display server X11, rendering libraries for graphics and fonts, image and configuration parsers. They provide a rich development environment, which necessarily increases the memory footprint. The GTK+ 3 library on a typical Debian system links against 45 other libraries. Each of the libraries will be mapped in the process with at least one executable area, one guard mapping, one static data area and one writeable pre-initialized data area.

All except the latter can be shared with other processes that are using the library. The Evince PDF viewer as invoked on the PDF file of this thesis has 300 library mappings and 40 cache mappings on Debian Wheezy, whereas the application itself only has seven.

For the anonymous memory of a GUI application it is less clear where it is coming from. Libraries can allocate memory on behalf of the application or simply for their own use when they are initialized.

Multi-threaded applications Multi-threaded applications on Linux run multiple schedulable processes within one address space, each with their own Process ID (PID). The overhead in terms of memory consumed within the address space is quite small: each thread needs a chunk of heap for its stack. By default a process will receive 8 MiB of stack, but the corresponding memory area is *mapped* by the Linux kernel *on demand*. The full memory assignment is hence only visible in the process' mappings if it has been used by the process once. This is an exception to the usual handling of mappings and it is the only type of memory for which such a mapping happens. For threads, the Native POSIX Thread Library (NPTL) part of the C library requests the full 8 MiB on the heap from userspace, but the Linux kernel will not actually allocate this memory unless it is being used. Data on stacks is usually small and short-lived. It is therefore unlikely to find much sharing potential within them.

Web browsers Web browsers are special among desktop environments, as more and more diverse workload is shifted to web applications. This includes traditional desktop applications like word processing and spreadsheets, but more intense calculations or access of remote desktops are also happening within browsers.

To realize complex applications, a JavaScript engine executes scripts with a Just-in-Time (JIT) compiler, which converts the platform-independent JavaScript source into machine-runnable code. Memory needs to be allocated for both the compilation result and for data used by the scripts, resulting in large heaps. Memory allocators in browsers are often custom-built for the application to manage memory more efficiently. Heaps of 1 GiB or more are common.

It is also typical for browsers to load plug-ins like Adobe's Shockwave Flash, which load other applications and render the result into a graphics area within the browser. These contain their own input, video, audio, and graphics processing and again an own JIT compiler. Depending on the browser this may happen in a separate process or within the same address space.

Contemporary browsers also try to reduce the attack surface by employing sandboxing. This involves creating a master process and forking a new rendering process for every new tab not belonging to the same web site. Memory allocated by the master process will be shared copy-on-write to its children. However, it still means that less memory can be shared within a single process than what would be possible if all rendering would be done in the same address space. For the communication between processes shared memory is used.

Looking at current means to share memory, web browsers would need to cache their compilation results across all sandboxes processes in files. This would allow the page cache to deduplicate them in memory automatically. Memory merging approaches would require that the results and, if we assume that multiple tabs with the same application are opened (e.g., web searches), the JavaScript heaps are page-aligned.

6.1.2 Multi-User Environments

Traditionally Unix systems were multi-user environments. Multiple users logged onto a single big system to execute their programs. Nowadays, when looking at workstations, the focus is on graphical applications rather than shell applications. For ease of maintenance these can be exported from a few powerful servers instead of being provided on each client workstation. On the shell side Virtual Private Servers (VPSs) are a cheap and popular way of getting one's own Linux server. The provider has an interest to support as many users as possible on one host.

Terminal Servers In the case of terminal servers multiple users login on the same machine. They run a graphical shell (i.e., a desktop environment), most likely a web browser and the applications they run to accomplish their tasks.

This scenario is often employed in thin-client environments, where few powerful servers serve dozens of low-spec clients. However, it can also ease administration, as applications are deployed and configured once on few hosts, instead of distributing them to many clients. In other cases it can be a key tool to reduce licensing costs.

Another related scenario is allowing multiple logins on a desktop machine. This can be realized through fast user switching, which allows only one user to use the machine at a given point in time, with the applications of other users being preserved in the background. Multiple users can even use the same machine at the same time through multiple graphics adapters (i.e., a "multi-head" setup).

Zero-Install Environments In zero-install environments users install their own binaries without any cooperation of the administrator. Two different degrees can be differentiated: the case where a desktop environment is provided centrally and the case where users maintain their own container with a full copy of the OS (except the kernel).

In these environments no file-based sharing is currently possible, due to Linux not being able to identify redundancy between named page frames. Binaries and libraries are hence duplicated in memory. Even caches that are intended to be globally shared will be container-local and hence user-local. This redundancy in memory content should be found and coalesced.

6.1.3 Virtualization

Virtualization is very important in cloud environments, to allow multiple users to use one hosts independently and isolated from eachother. The advent of virtualization on commodity servers also caused the introduction of hardware extensions for virtualization in consumer hardware. Two following two approaches are relatively novel, one using full virtualization and one trying to be more efficient by reusing the same OS kernel for multiple different user environments.

Virtualization Sandboxes Qubes OS [RuWo10] uses virtualization to provide isolation between different work environments (e.g., online banking, work, and private mails) on a single desktop. This can introduce redundancy not only on disk, which is solved by using a template VM that is shared in a COW fashion. For instance, a web browser is likely to be started in multiple isolated domains.

Container Environments Containers are a lightweight way of virtualization. A single host kernel provides virtual views on file systems, networking, processes, and permissions. This avoids the overhead of multiple kernels and allows more efficient use of the memory available in the system. Containers can be spawned for a single application or service to provide isolation and security. They can also contain full Linux distributions that will run their own init, the basic set of system services and the user's applications.

In the first case memory deduplication for file-backed pages is likely to happen in the host's page cache. In the second case, where binaries and libraries are duplicated on-disk, this is not easily possible. GUI-related OS-global caches like icon or font caches will be duplicated both in memory and on disk.

6.2 Previous Evaluations

Previous studies on memory deduplication potential focussed exclusively on Virtual Machines (VMs). They differentiate between intra-VM sharings, that is, sharings that can be found within the memory content of a single VM, and inter-VM sharings, which show redundancy of code and data between multiple VMs. Multiple systems were started on one host, heterogeneous workloads with different guest OSs and homogeneous workloads that basically executed the same OS and programs.

The VM's memory is one contiguous amount of heap memory in the hypervisor application. Periodical memory-scan deduplication on the host can share kernel and userspace memory of the OS within the VM. The benefits are then dependent on the guest OS in use.

Virtualizing Windows The Windows kernel is special in that it zeroes page frames for security and compliance reasons before returning them to the free memory pool, the "zero page list". Every page fault requiring a new page of memory will be served from this list. Windows will also zero out the entire memory on boot.

Memory deduplication is particularly effective with this strategy. This means that a hypervisor running Windows VMs which are not fully utilizing their memory allocation can re-purpose the spare memory in the host for other applications or VMs. However, this memory is only borrowed as the VM can, at any time, decide to request it by using it. If this strategy is employed to increase VM density, this might degrade performance a lot once memory pressure increases, as the VM cannot tell how much memory is actually available on the host.

Linux avoids this scrubbing in order to avoid increasing cache pressure: the zeroing process could push valuable content out of the cache and the zeroes are more likely to be read directly after a page fault is serviced, which would require that the caches refetch the zeroes from memory.

Implementation in VMware ESX When Waldspurger [Wald02] measured the effects of his implementation of memory deduplication in VMware server, he found significant inter-VM sharings. However, he also attributed most of the sharings related to Windows VMs to the aforementioned zeroing of free memory. This made it possible to reclaim 32.9% of the memory used by ten Windows NT guests (2 GiB in total) through the use of periodical memory-scan deduplication. Intra-VM sharings were mostly a sidenote, with 12.5% of RAM being reclaimed from a single Red Hat Linux VM, again 55% of it being sharing of the zero page.

Difference Engine Gupta et al. [GLVS$^+$10] found 65–75% memory savings in their benchmark consisting out of 1 to 6 VMs, each running Debian 3.1 and an e-commerce benchmark. They note that "the bulk of memory savings comes from page sharing" [GLVS$^+$10]. They also implement sub-page sharings through patches, which makes it hard to deduce CBPS sharings from their heterogeneous OS and applications scenario.

Empirical Study of Memory Sharing in VMs Barker et al. [BWSS12] conducted an empirical study on memory images of different virtual machines to determine the source of sharing opportunities. They observe that "absolute sharing levels (excluding zero pages) generally remain under 15%" [BWSS12], which indicates that synthetic benchmark situations might artificially inflate the sharing potential. The second important observation of Barker et al. is that "sharing within individual machines often accounts for nearly all ($> 90\%$) of the sharing potential within a set of machines" [BWSS12]. Past research focused on sharing as a means to support an increased count of virtual machines on a single host. The study indicates that sharing opportunities that can be exploited within a single non-virtualized operating system will alone contribute significant memory savings.

Assuming that self sharing would be possible for individual machines and that its share is that significant, it states that the potential for additional inter-VM sharing is low. At most one additional identical page per VM on the same host can be coalesced into a shared mapping.

The study relies on two data sources for their study: real-world memory traces by the Memory Buddies project [WTLSD$^+$09], which collected traces from desktop and server machines at the University of Massachusetts Amherst and own traces of VMs running a Linux desktop. Traces are taken by suspending the VM's execution and taking a snapshot of the guest's memory image. Each page-sized is then hashed. The original content of the page frame is not retained in the exported traces.

To get more insight into the sources of sharing opportunities, a Linux kernel module collects additional information from the inside of the VM. The module hashes the content of all page frames, records the "content type of the page" (e.g., heap, stack or a named page containing a library) and gathers a list of all processes mapping that specific page frame. This is used for a case study of Linux desktop applications.

This case study consists of a VM running Ubuntu as its base OS and Firefox, GNOME, OpenOffice and an X server as the user visible desktop environment. It finds that the largest single source of sharing consists out of heap pages ("50% of all sharing within the VM" [BWSS12]). Shareable library pages are "involved in 43% of all sharing" [BWSS12]. Stack pages are involved in less than 5% of all sharings. They observe that their memory traces containing GUI applications have a higher level of self-sharing, "also likely due to the tendency of GUI-related libraries to increase memory redundancy" [BWSS12].

As soon as the OS version did not match exactly, sharing opportunities significantly reduced in the base system. Common desktop applications were less affected by this version skew, however. In this work we did only evaluate identical applications.

The impact of Address Space Layout Randomization (ASLR) was explored by booting the same workload four times each with ASLR turned on and with randomization disabled. The study measures the negative influence of ASLR on both inter- and intra-VM sharing separately and found an overall 10% sharing reduction with the Linux workload. Self-sharing was reduced the most by about 15%, inter-VM sharing by about 8%.

The study also briefly covers variably-sized hashing. It suggests that sharing granularities beyond a single page do not yield significantly more sharing opportunities. The range of block sizes analyzed is from 0.4 to 2.4 times the system page size (4 KiB), in intervals of 0.1.

6.3 Evaluation Results

In the following we discuss the results obtained from the process described in Chapter 4. The raw numbers can be found in Chapter 5.

6.3.1 Desktop Applications

The desktop applications we looked at (Xfce, Firefox and LibreOffice) did expose a significant amount of redundancy in their heap memory areas In the single user case over half of the memory can be recovered (52%, comparing the "`mmap/fork`" and "KSM hint all" cases in Table 5.1 on page 35). The kernel does already take care of mapping fresh allocations onto the zero page (i.e., the page that only contains zeroes) to save memory. We find that most of this redundant content, 133 MiB, is found within the Xorg server as started by `xpra`. This is likely framebuffer content that needs to be allocated in advance to support large display sizes. This is consistent with the observation of Barker et al. that "the single most shared page, with 597 distinct copies (2.3 MB shared) was a heap page used by Xorg" [BWSS12]. It is followed by 1.7 MiB of sharing due to zero page deduplication.

Due to the fact that a massive amount of memory is deduplicated within the X server, additional merging across sessions only increases the sharing potential by about $7 - 9\%$. All this redundancy can be found and merged by KSM, as only anonymous memory is involved.

The zero-install scenario is one where the approach of full Content-based Page Sharing (CBPS) can shine. Being able to deduplicate the page cache allows the memory usage to drop to the same levels as the terminal server scenario (see Table 5.1 on page 35 for details). As the binaries are identical, this is not surprising. With three users deduplication with just KSM needs half of the allocated memory more than full CBPS.

6.3.2 Game Server

The case of the tested game server is particularly interesting from a deduplication point of view. The savings that can be achieved by activating KSM for processes that do not implement the hinting of their allocations, can be significant for some workloads.

In the three server scenario we see that it is able to merge more than the equivalent of a full server instance together (memory usage going down from $2.97\times$ to $1.90\times$ the baseline memory use of 147 MiB of RAM; Table 5.3 on page 37). Content-based page sharing that picks up further redundancy in named pages only finds 20% more sharings than KSM would. The numbers suggest that for every additional server 10% of the memory needed can be shared with the existing instances.

Here it would be beneficial for the application to wrap the calls to `malloc`(2) to hint all newly allocated memory areas. Alternatively it is possible to patch the Linux kernel to hint all anonymous memory areas within all processes on the same host.

This particular server setup does not benefit at all from `mmap` and `fork`. While it does use multiple threads, it does not use multiple processes. The sharings that can be found are exclusively system libraries like glibc. Commonly these will already be mapped by other applications, even though this was not the case in our setup as the server is shipped as a 32-bit binary which needs a 32-bit C library to run. All other binaries on the systems were 64-bit binaries.

6.4 Possible Improvements of Memory Deduplication

This section discusses possible improvements that would help to enable more efficient memory deduplication with native applications on Linux.

6.4.1 Improving Sharing of Anonymous Pages

Heap pages could be marked to be scanned when they are requested by applications. This would require no cooperation by the application, at the expense of scanning memory that could be highly volatile and hence was explicitly not hinted. This would require spare CPU cycles for the scanning process and cause more memory bandwidth to be used by scanning. On the other hand it could increase performance by freeing up memory for other purposes. In both the desktop and game server scenarios this would have helped reducing more usage significantly.

6.4.2 Multi-Tenancy as a Way to Avoid Duplication

Some programs avoid duplication by reducing the need of running multiple processes. Mail readers now support multiple mail servers in one program. Web browsers are able to fence off multiple windows against each other, not requiring the launch of multiple browsers – for instance to separate online banking or private browsing for gifts). Back in the old days connecting to multiple news servers required passing different configuration and state files on the command-line – which is the case, for instance, for `slrn`(1). Office programs launch a single big application, regardless if you request a spreadsheet application or a word processor. Documents are opened in existing windows.

Avoiding to invoke a process per file or task saves memory without elaborate deduplication mechanisms. Instead memory can be shared at least for one user. On servers this has been commonplace for web servers, which are able to serve the content of multiple users without the need of a dedicated web server per user.

The game server scenario would have benefited from this improvement, as one copy on disk and in memory would scale much better to multiple servers on one host than the current implementation of Counter Strike's dedicated server.

6.4.3 Per-User Scanner

As noted in Section 6.3.1, most of the anonymous sharing potential comes from single sessions, i.e. from processes of the same user. Currently the run time of KSM is not being accounted for and every user has to bear the costs. The fact that there is sharing locality within the processes of one user can make a user-local scanner feasible and useful. If interfaces were in place to allow special processes (e.g., ones that are in a debugging relationship) to modify mappings of anonymous pages across address space boundaries, such a scanner could periodically inspect the various address spaces of a user and coalesce them on account of the user in question. Currently it is only possible to establish shared memory through System V shared memory primitives [shm08] or file-backed memory. However, if this is extended, the user could spend some of his CPU time on merging, to consume less memory or run yet another application.

6.4.4 Improving Memory Sharing through Hardware Support

Some extensions of the hardware to better support memory sharing have already been proposed. It would be useful to incorporate such methods to reduce the load that periodic scanning induces.

HICAMP Cheriton et al. [CFSS$^+$12] propose a new hardware architecture called HICAMP that identifies memory cells by their content. This allows sharing memory at the line level instead of the coarse-grained page level. Memory is organized in segments, which are "variable-sized, logically contiguous regions of memory". They are defined by a Directed Acyclic Graph (DAG) of content identifiers.

The architecture inherently avoids all deduplication scanning, at the expense of treating modifications of such memory vastly different than common DRAM. Instead of modifying individual RAM lines, every line with unique content is immutable and a tree of references needs to be updated on every write access.

Hardware-based Page Checksums The memory architecture could offer additional memory lines containing a checksum of the page frame's content. This would speed up the scanning for duplicates, which would need to touch fewer bytes in memory.

To be efficient, the checksum used would need to be efficiently updateable on writes, so that it's not necessary to reread the whole page frame for the checksum generation.

6.4.5 Improving Sharing of Named Pages Using Storage Deduplication

Bugnion et al. [BuDR97] used Copy-on-Write (COW) disks to deduplicate storage blocks automatically. Nowadays, with content-addressable storage systems this could be extended to the sharing of complete files in memory. File systems would merely keep a hierarchical mapping of files in directories to content identifiers (e.g., checksums).

For the zero-install scenario this would make the identification of duplicate applications and libraries very easy. As long as the files map to the same identifier, they can be shared in a unified page cache.

The main problem with this approach is that filename identifiers need to be kept even though content of different files in memory is being reused by multiple processes. The OS needs to be aware which file was actually mapped by a process to instantiate the correct COW mapping.

6.4.6 Efficient Caching of Currently Unneeded Pages

Currently unneeded pages are commonly swapped out. Gupta et al. [GLVS$^+$10] propose that pages that are relatively similar could be stored as patches to existing pages if they are not currently part of the working set. Reconstruction of pages on access does require some processing overhead, as does the fingerprinting of all pages to identify similarities within pages. Furthermore, pages that were not recently used could be stored compressed in memory, instead of being swapped, to be uncompressed whenever they are needed again. Further evaluations should be carried out to identify the potential of patching and compression of memory found in native applications.

6.5 Summary

We see that the means to deduplicate memory could still be improved. Not being able to merge file-backed pages causes KSM to not find all redundancies in the native applications we started. The game server we used should be modified to support multi-tenancy to put less strain on the OS which then needs to deduplicate the memory content. What we have seen has been consistent with what a preliminary analysis by Barker et al. [BWSS12] has shown.

7 Conclusion

We have shown in this thesis that memory deduplication is very useful in the context of native applications and sandbox-like scenarios on Linux. The concept that was developed with virtual machines in mind should be extended to cover applications running on a single host as well.

We looked at a common desktop scenario that involved starting a desktop environment, a web browser and an office suite to determine sharing opportunities. This has been executed in parallel with varying degrees of duplication involved. We either reused the system libraries and applications when multiple users were logged into the system or we duplicated them all (except for the OS kernel) to simulate a zero-install scenario. Furthermore, we looked at a game server that has been instantiated multiple times to serve multiple users.

The evaluation led us to the following conclusions:

There are indeed redundancies in the scenarios we analyzed. Partly they are quite significant, even deduplication of just anonymous memory can yield savings of up to half the memory used without deduplication.

In the scenarios we analyzed there were very few sharing opportunities between anonymous and file-backed pages. Redundancy within anonymous memory was found to be mainly within similar programs, a user's session or a single VM, whereas file-backed pages were more likely to be shared with programs outside of a user's domain.

To improve memory deduplication across these domains, an improved `mmap` that uses content deduplication in the block layer would help to catch redundancies between multiple sessions in a zero-install scenario. For instance, simply basing the sharings on inodes is not sufficient to deduplicate redundancies arising from user-local caches, that are identical.

Deduplication of anonymous memory as implemented in KSM already uses hashes to avoid pairwise comparisons, which are in $\mathcal{O}(n^2)$. However, data structures used for this could be kept more local, to decrease lookup times and increase hit rates.

The implementation of named page sharing in Linux can be beneficial in single kernel scenarios that do not employ full system virtualization. Lightweight virtualization like containers commonly duplicate the base Linux distribution, of which parts can be merged in memory if the operating system's base version is identical.

A generic implementation of page frame sharing would be useful to achieve small memory footprints. This is currently possible by moving the workload into a VM and running KSM on the outside to coalesce identical pages. Introducing such a layer of indirection seems disproportionate if the only goal is memory deduplication.

Future Work

The benefits of memory deduplication are still dependent on how many CPU cycles can be spared for scanning. A future analysis should look at new ways to deduce memory characteristics from both *a priori* and *a posteriori* knowledge.

It is clear that a memory snapshot mechanism as presented in this thesis cannot predict the future. Pages would need to be tracked and inspected as to what causes yield a sharing opportunity.

A more efficient full-system emulator could give detailed information as to which writes caused a sharing to appear or to disappear and how long a sharing would have persisted if detected immediately.

It would be interesting to know why applications allocate pages that are for all means and purposes empty (zero pages) and why these are not mapped onto the zero page frame as provided by the operating system. Unfortunately, due to time constraints, we were not able to analyse this in detail.

List of Acronyms

ASLR Address Space Layout Randomization. 13, 44

CBPS Content-based Page Sharing. 34–37, 43, 45
COW Copy-on-Write. 7, 8, 13–15, 24, 25, 42, 47
CPU Central Processing Unit. 8, 10, 12, 13, 18, 19, 23, 24, 45, 46, 50

DAG Directed Acyclic Graph. 47
DMA Direct Memory Access. 8, 13, 17, 18, 22, 25
DRAM Dynamic Random-Access Memory. 8, 47

ELF Executable and Linkable Format. 11, 13
EPT Extended Page Tables. 11

GOT Global Offset Table. 14
GUI Graphical User Interface. 23, 39, 40, 42, 44

JIT Just-in-Time. 40

KSM Kernel Samepage Merging. 11, 12, 20–22, 24–26, 28, 34–37, 44–46, 48, 49

LRU Least-Recently Used. 12

MM Memory Management. 11
MMU Memory Management Unit. 8, 14, 25

NPTL Native POSIX Thread Library. 40

OS Operating System. 4, 7–11, 17, 18, 23, 24, 32, 34, 41–44, 47–49

PD Page Directory. 9, 10
PDE Page Directory Entry. 9
PDPT Page Directory Pointers Table. 9, 10
PDPTE Page Directory Pointers Table Entry. 9
PIC Position-Independent Code. 14
PID Process ID. 40
PIE Position-Independent Executable. 13
PLT Procedure Linkage Table. 14
PML4 Page Map Level 4. 9
PML4E Page Map Level 4 Entry. 9
PT Page Table. 9

PTE Page Table Entry. 10

RAM Random Access Memory. 3, 18, 33–35, 37, 43, 47
RVI Rapid Virtualization Indexing. 11

TLB Translation Lookaside Buffer. 9–11

VM Virtual Machine. 4, 5, 10, 11, 14, 17–19, 25, 32, 33, 39, 42–44, 49
VMA Virtual Memory Area. 11, 12
VMM Virtual Machine Monitor. 18
VPS Virtual Private Server. 41

Bibliography

[ArEW09] A. Arcangeli, I. Eidus and C. Wright. Increasing memory density by using KSM. In *Proceedings of the Linux Symposium*, 2009.

[BBMT72] D. G. Bobrow, J. D. Burchfiel, D. L. Murphy and R. S. Tomlinson. TENEX, a paged time sharing system for the PDP-10. *Communications of the ACM* 15(3), March 1972, pp. 135–143.

[BDFH+03] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt and A. Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP ’03, New York, NY, USA, 2003. ACM, pp. 164–177.

[BeCD72] A. Bensoussan, C. T. Clingen and R. C. Daley. The Multics virtual memory: concepts and design. *Communications of the ACM* 15(5), May 1972, pp. 308–318.

[BoCe05] D. Bovet and M. Cesati. *Understanding The Linux Kernel*. O’Reilly & Associates Inc. Third edition, November 2005.

[Brad08] P. Brady. Anatomy & Physiology of an Android. In *Google I/O Developer Conference*, 2008.

[BuDR97] E. Bugnion, S. Devine and M. Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, SOSP ’97, New York, NY, USA, 1997. ACM, pp. 143–156.

[BWSS12] S. Barker, T. Wood, P. Shenoy and R. Sitaraman. An empirical study of memory sharing in virtual machines. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC’12, Berkeley, CA, USA, 2012. USENIX Association, pp. 25–25.

[CFSS+12] D. Cheriton, A. Firoozshahian, A. Solomatnikov, J. P. Stevenson and O. Azizi. HICAMP: architectural support for efficient concurrency-safe shared structured data access. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, New York, NY, USA, 2012. ACM, pp. 287–300.

[DeBR02] S. W. Devine, E. Bugnion and M. Rosenblum. Virtualization system including a virtual machine monitor for a computer with a segmented architecture, May 2002. US Patent #6397242.

[Denn96] P. J. Denning. Before Memory Was Virtual. In *In the Beginning: Personal Recollections of Software Pioneers*, 1996.

[for12] fork(2) manual page. In *Linux Programmer's Manual*. October 2012.

[GLVS$^+$10] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker and A. Vahdat. Difference engine: harnessing memory redundancy in virtual machines. *Communications of the ACM* 53(10), October 2010, pp. 85–93.

[IBM04] IBM Corporation. *How to Improve the Performance of Linux on z/VM with Execute-In-Place Technology*, July 2004.

[Int13] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1*, January 2013.

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

[MFGR$^+$13] K. Miller, F. Franz, T. Groeninger, M. Rittinghaus, M. Hillenbrand and F. Bellosa. XLH: More effective memory deduplication scanners through cross-layer hints. In *Proceedings of the 2013 USENIX Annual Technical Conference*, USENIX ATC'13, Berkeley, CA, USA, 2013. USENIX Association.

[mma12] mmap(2) manual page. In *Linux Programmer's Manual*. April 2012.

[MMHF09] G. Miłós, D. G. Murray, S. Hand and M. A. Fetterman. Satori: Enlightened page sharing. In *Proceedings of the 2009 USENIX Annual technical conference*, USENIX'09, Berkeley, CA, USA, 2009. USENIX Association.

[pag09] pagemap, from the userspace perspective. In *Linux Documentation*. June 2009.

[pro12] proc(5) manual page. In *Linux Programmer's Manual*. October 2012.

[ptr12] ptrace(2) manual page. In *Linux Programmer's Manual*. October 2012.

[Ritt12] M. Rittinghaus. Runtime Benefits of Memory Deduplication, July 2012. Diploma Thesis.

[Rive92] R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321 (Informational), April 1992.

[RMHB13] M. Rittinghaus, K. Miller, M. Hillenbrand and F. Bellosa. SimuBoost: Scalable Parallelization of Functional System Simulation. In *Proceedings of the 11th International Workshop on Dynamic Analysis (WODA 2013)*, Houston, Texas, March 2013.

[RuWo10] J. Rutkowska and R. Wojtczuk. *Qubes OS Architecture*, January 2010. Version 0.3.

[shm08] shmop(2) manual page. In *Linux Programmer's Manual*. June 2008.

[tki12] tkill(2) manual page. In *Linux Programmer's Manual*. July 2012.

[wai10] wait(2) manual page. In *Linux Programmer's Manual*. September 2010.

[Wald02] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th symposium on Operating systems design and implementation*, OSDI '02, New York, NY, USA, 2002. ACM, pp. 181–194.

[Wind] Wind River Systems Inc. Simics. `http://www.windriver.com/products/simics/`.

[WTLSD$^+$09] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet and M. D. Corner. Memory buddies: exploiting page sharing for smart colocation in virtualized data centers. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, New York, NY, USA, 2009. ACM, pp. 31–40.