Karlsruher Institut für Technologie

Analyzing Shared Memory Opportunities in Different Workloads

Studienarbeit
von

cand. inform. Thorsten Gröninger

an der Fakultät für Informatik

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

Bearbeitungszeit: 8. August 2011 – 21. November 2011

Abstract

This thesis analyzes different workloads and their impact on memory sharing, sharing opportunities, and memory merging. To make the analysis possible, we have written a Linux kernel module to extract the needed information. It reduces memory dumps to 1.5% of the full memory dump size by hashing the page's content. These dumps show unused potential of page cache sharings between host and guest operating systems. Limitations of Kernel SamePage Merging as present in recent Linux kernels.

We analyzed parallel kernel builds in virtual machines with activated KSM and discovered that up to 60% of sharing opportunities remain unmerged, due to the slow scanning rate of KSM. In this scenario up to 80% of data is redundant and could be shared, because 54% of these sharing opportunities last longer than 20 seconds, even in VMs with high memory pressure. Most of this sharing potential originates from the page cache, which also contains redudant pages (up to 10%). A more sophisticated deduplicating file system and page merging approach could speed up the whole system.

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

Thorsten Gröninger
Karlsruhe, November 21st 2011

Deutsche Zusammenfassung

Diese Studienarbeit beschäftigt sich mit Methoden zum Einsparen von Arbeitsspeicher, der Analyse ungenutzten Einsparungspotentials. Ferner gibt sie einen Überblick über bereits existierende Deduplikationslösungen und zeigt deren Beschränkungen auf.

Arbeitsspeicher ist heute meist keine knappe Ressource mehr und wird daher oft verschwendet, allerdings gibt es gute Gründe Speicher zu sparen. Eingebettete Systeme, Smartphones und auch Cloudsysteme können es sich nicht leisten die selben Daten mehrfach im Speicher vorzuhalten. Deshalb gibt es verschiedene Ansätze, die den Speicherverbrauch reduzieren sollen. Kernel SamePage Merging (KSM) ist einer dieser Ansätze und ist bereits in aktuellen Linuxkernen enthalten. KSM untersucht den Speicher einer Anwendung (z.B. einer virtuellen Maschine) in regelmäßigen Abständen und legt gleiche Speicherseiten zusammen. Diese Arbeit beleuchtet KSM und seine Beschränkungen und zeigt ungenutztes Sparpotential, z.B. beim Bauen des Linuxkerns in verschiedenen Virtuellen Maschinen oder bei rechenintensiven Benchmarks auf.
Da der Linuxkern nur über unzureichende Schnittstellen für Redundanzanalysen verfügt und ein komplettes Arbeitsspeicherabbild zu aufwendig ist, wurde hierfür ein Kernmodul für Linux entwickelt, das eine Analyse des virtuellen, sowie des physischen Speichers ermöglicht. Es ist in der Lage einen Fingerabdruck (hash) jeder Speicherseite im System, Anwendung oder des Seitencaches, sowie die damit verbundenen Informationen (PFN, Page Table Flags, etc.) zu liefern. Ein solches Abbild umfasst nur noch 1.5% des zu analysierenden Speichers, lässt aber dennoch eine vollständige Redundanzanalyse zu. Diese Analyse wird vollständig im Userland durchgeführt, zeigt bereits geteilte Seiten (shared pages), potentielle Kandidaten (sogenannte „sharing opportunities") und deren zeitlicher Verlauf auf.
Die Analyse virtueller Maschinen ergab, dass es Einsparungspotenzial von bis zu 80% gibt, wenn die ausgeführten Programme eine sehr hohe I/O Last aufweisen, wie z.B. beim Bauen des Linuxkerns in verschiedenen virtuellen Maschinen.
Des Weiteren hat sich gezeigt, dass die momentane Implementierung von KSM nicht in der Lage ist das volle Potenzial schnell und vollständig auszuschöpfen.

Zum Einen kann KSM nur anonyme Seiten zusammenlegen und versagt bei Dateidaten im Seitencache (page cache) vollständig, zum Anderen ist KSM zu langsam, um alle teilbaren Seiten zu erfassen oder es verbraucht dabei zu viele Ressourcen. Die Analysen verdeutlichen, dass es noch genügend unausgeschöpftes Einsparungspotential und einige Geschwindigkeitsverbesserungen geben kann.

Contents

Chapter 1

Introduction

Memory saving seems old-fashioned, and a problem of the past, as memory prices are low and even falling.

This was true, but now, virtualization has become popular, even in small computer systems. "Cloud computing" is the buzzword of our time and in this context, it is more important than ever to save memory and deduplicate redundant information. Why should a cloud environment waste its memory for the same system image over and over again, or for the same data, loaded from different sources? Furthermore, embedded platforms, or even smartphones can profit from memory saving, power consumption of memory is not negligible. There are many different ways to reduce the amount of consumed main memory, e.g, by deduplication redundant data. Is there still unused potential, not yet exploited by any proposed solution and is the remaining redundancy worth searching for new techniques?

A common way for analyzing memory deduplication is dumping a the complete memory of a task or the whole system RAM to disk or network. To analyze existing sharings and sharing opportunities, it is sufficient to get a hash of each page's content and its corresponding physical address. Faster changing pages are hardly traceable with full memory dumps, because of there overhead. An existing memory deduplication, can get this information from its internal data structures, and need not to rely on memory dumps.

The default interfaces of Linux and Unix do not provide all required data to analyze sharing opportunities in userspace. This lack of information requires a new interface that can collect data within the kernel.

1.1 Basic ideas & Approach

We have come to terms, to collect PFNs (page frame numbers), as unique identifiers of each page, page and page table flags, to get the current page state, and the page's content, to detect redundant data. These information can then be used to get a deeper understanding of shared pages, deduplication mechanisms, and unexploited sharing opportunities. The retrieval process should be done with little impact on the analyzed system and without altering its state.

Our basic idea is to analyze a large amount of pages (associated with tasks, file cache, system RAM), without fetching their whole content from physical memory and dumping it to disk.

Our analyzing approach reduces the impact a snapshot has on the scanned system/application, reduces the size of memory dumps (approximately 100 times smaller), and allows an analysis of page information. Furthermore, compare-by-hash is the only practical way to compare memory content in realistic time and reduce its complexity from $O\left(n^2\right)$ to amortized $O\left(n\right)$. A question arising from previous works, why should a whole memory dump be transfered from kernel to user land or via networks, and hashing the data afterwards, when the hash can be already computed in kernel mode, and so reduce the amount of storage space and bandwidth needed. The analysis can then be done afterwards offline. Hashing swapped out pages should be avoided, to minimize the impact on the running system.

We were looking for workloads, which primarily use I/O operations and process large amount of data (mainly form disk) and in contrast workloads, which favor the CPU. We assume that this two workload classes will show all effects of memory sharing, merging, and unexploited sharing potentials.

With all the required information available and different workloads, there are some interesting questions:

- What kind of sharing is not exploited yet?
- How long does potential sharing last?
- Which kind of tasks can profit from host to guest sharing?
- How many pages are already shared between tasks?
- Why can existing "solutions" not exploit all sharing opportunities?

1.2 Contribution

The target audience ranges from every interested student, who wants a closer look at the memory layout of processes, the whole system, virtual machine content, page table content, or find files inside an address space. To researchers that try to evaluate their new solution, create a new deduplication technique, or just want to analyze new potentials before programing a concrete solution related to memory saving, sharing mechanisms. This thesis contains a analysis of unexploited potential that we discovered during testing and evaluation.

Our tools are not limited to one platform and the kernel module is already capable of ARM architectures and compatible with at least some Android kernels.

Chapter 2

Background & Related Work

Memory content has various sources, it can be unrelated to files in background storage (anonymous pages), it can map the content of a file directly into memory (named pages) or it can hold copied (unchanged) file content (also anonymous pages). Anonymous pages containing file content can not be treated like named pages by common sharing techniques in operating systems, since their origin is lost. It is the same problem most VMs suffer from, the source of the content is not clear, which is very important, if memory should be saved or shared with other tasks. Approaches have to overcome this semantic gap.

In general, there are four different kind of page sharing, created with a-priori knowledge or with posteriori knowledge after page scans.

- a-priori

  Anonymous shared memory sharing due to fork or shared memory regions

  Named shared memory sharing due to fork or mapping the same file

- a-posteriori

  Merged anonymous memory sharing due to memory sharing methods, such as KSM

  Merged named memory does not exist yet (for a system wide usage)

Many different proposals have been made to reduce the memory consumption. They fit into two different classes:

Instant Memory Sharing modify the host and guest system, and install sharing when it occur, or utilize regular operating system mechanisms, such as `mmap`. This is only possible with a-priori knowledge, due to this knowledge gap, memory scanning must be involved.

Memory Scanning scan the (unmodified) guests periodically and merge pages with the same content, e.g., KSM.

The instant memory sharing approaches are completely based on Xen or other cooperating operating systems (such as DISCO), and make extensive use of their internal memory management structures to do their memory analysis, e.g. Satoris' Page enlightenment [17].

Memory scanners on the other hand, introduce new data structures that can be used for memory analysis, e.g., KSM maintains two trees and reverse mappings, which otherwise would not exist in the Linux kernel [1].

Often it is not completely clear, how the other approaches acquire their analyzed data, in the most common cases, they use their internal data structures, their shadow page tables or just complete memory dumps, but there is no general fast and independent mechanism to get the actual sharing information. This makes analyzing sharing opportunities more easy, if you have developed a solution and can easily access these structures, otherwise exploring unexploited sharing potentials is difficult, this is were our tools kick in.

2.1 Identifying Memory Duplicates in Virtual Machines

The greatest benefit of virtual machines, is also their greatest weakness, they separate and virtualize hardware to allow multiple guest to run on the same hardware. This provides isolation, but leads to many challenges. Without VM, it is possible to share pages, because every task and the OS was aware of the source and of its content (file mappings), to exploit shared anonymous memory mappings, a application has to be modified. Memory sharing in VM is difficult, as there is not enough knowledge about the content, a *semantic gap*, e.g., a VM does often not know the source of a page, was it file backed or not. This problem exists also outside VMs, a regular `memcpy` also breaks the linkage between a file and page. This drawback was tackled. IBM introduced VMs in the early 1970s on their mainframe and made special hardware adoptions (instruction set and page table flags) that help the host to get extended knowledge of the guests activity, e.g.,

	Anonymous Memory	Named Memory
a-priori (shared)	fork, shared memory regions	fork, same mapped file
a-posteriori (merged)	memory sharing, KSM	does not exists yet

Table 2.1: Summery of page sharings

special page table flags for memory reclaim. VMware's solutions have special drivers for the guest OS to provide more information (e.g. unused guest pages) or even to compress the memory content inside the VM [8, 21]. Solutions running on paravirtualized system like Xen, have both external knowledge and internal knowledge about the guest OS activity and its memory content.

The idea of "transparent sharing" was introduced in DISCO [6], an OS specially developed to reduce memory redundancies, it was further improved by Satori's page enlightenment. The sharing possibility and potentials of Xen were analyzed in [14] and showed a high sharing potential, in some workloads. Xen hooks, replaces the copy mechanism of the guest operating system, and creates the transparent sharing, by just mapping the page to both locations, without copying it, and mark it as a Copy-On-Write page. On the one hand, it seems clear that an instrumented guest OS can perform better in many situations [6, 14, 17], but on the other hand, there is often a need to run unmodified operating systems as well. This is were *memory scanning* kick in.

One way to get an insight into a VM's activity and memory usage, is to scan the VM periodically, e.g., the page tables of the VM. Many approaches scan memory from outside, so a guest OS can execute without modifications. This approach is taken by KSM [1] and Difference Engine [8]. Some inject special drivers, which gather this information and use it for memory reclaims or memory merging [21].

Memory scanning is done by KSM [1] as well as in VMWare ESX Server [21] and Difference Engine [8]. VMWare ESX Server tries to minimized the impact on the VMs, with low scan rates (completely scans every 20 min, only few pages at a time). Difference Engine goes beyond the other approaches and merges sub pages, it patches pages, after calculating the related Rabin fingerprints to find merging candidates.

2.2 Memory Merging

After having identified redundant page contents, it is possible to merge the pages with the same content. Not all pages with equal content are good candidates for merging, e.g., zero pages. Keep in mind, scanning, installing and breaking a sharing takes time and very short lived sharings are not worth the effort.

For a more detailed discussion of KSM see Section 2.5.5.

2.3 Memory Mapping

Files, can be mapped into address spaces [16]. This mapping can be shared among different tasks to preserve memory. The OS maps the pages into every address

Facility	Provided Information
`/dev/mem`	complete memory contents
`/proc/iomem`	physical memory layout
`exmap`	PFN, page flags, address space layout
`crash`	complete memory dump with data extraction helpers
`/proc/<pid>/vmmap`	address space layout, vm region flags

Table 2.2: Information facilities available for Linux and information they provide

space which requests a copy of the same file, and marks them as Copy-on-Write [18]. Shared libraries are mapped to memory in this way. Problems can arise, if two different versions of the same library are needed, they cannot be shared, even if large parts are identical [7]. This sharing approach is taken by Linux, Windows and many other modern operating systems and works very well with named mappings. In VMs however, the host does not know what content a virtual guest page holds. The guest file systems is (probably) unknown, to the VMM, which makes file mappings difficult or even impossible, so all types of memory are handled as anonymous memory. Their are some architectures that tackle this problem like the SystemZ series of IBM (for example, special page table flags) or some paravirtualization approaches like Xen, but for all this cases the virtual guest must be adapted to cope with this specialties of the hosting platform.

2.4 Memory Analysis

A common way to get information for an analysis is to dump the complete physical memory to disk or network. Linux provides some information through the Proc FS, and separate available tools (see Table 2.2). There is a special device (`/dev/mem`) for this purpose. Tools such as *crash* dump the memory and can extract meaningful information (e.g. page frame numbers, hashes per page). On the one hand, crash creates large dumps with all required information, which must be transfered (e.g. 24GB), but on the other hand it is primarily focused on dead systems [2] and can hardly be used for fast information acquisition. The exmap module [3] retrieves only a subset of memory informations, such as PFNs, but more than a regular `/proc/<pid>/vmmap`. It only provides the address space layout, but not enough to analyze memory sharing opportunities, it is only sufficient for already shared pages.

We took the basic ideas of KSM and VMware ESX Server and scan the address space (e.g. of a VM), or physical RAM periodically, without interfering with the guest OS or the tasks. Since we are not interested in actual page merging, we can use the compare-by-hash methodology and get the sharing opportunities and all shared pages without a complete memory dump. The risk of false positives is low even for weak hash functions (see Chapter 5 and the general discussion of hashing functions).

2.5 Linux Memory Subsystem

The virtual address space of every task, is represented by `mm_struct` (Figure 2.1). It contains all necessary memory management information, like code start, list of virtual memory regions, amount of used pages and much more. This structure exists for every task in the system. VMs are no exception to this rule on Linux based systems using the kernel virtual machine monitor (KVM) [1] as they are regular processes executing along every other process in the system.

The physical memory is divided into pages that consist, of general structure information of every page's content, such as address space mappings, associated file and flags.

The following section gives a short overview, how Linux manages the user address space and the processors specific details.

2.5.1 Virtual Memory Areas

Each address space is divided into areas of usable, allocated memory. These structures are represented in Linux by the `vm_area_struct` (Figure 2.1), establishing the complete available user address space. Every page fault must traverse these structures to find the corresponding physical frame to the faulting virtual address. Therefore the virtual memory region is not accessed as a linked list, although we use this way for our snapshots, it is represented as a red black tree instead for faster access [5]. This structure contains flags that describe the usage of every address space region. For example, if it can be read, written, executed, contains I/O pages or maps files to memory. Figure 2.1 shows the typical layout of an address space and its structures.

The actual virtual to physical address translation is done via the page tables.

2.5.2 Page Tables, Page Struct and Page Access

The smallest memory management unit, a page, is described by a page struct (`page` (Figure 2.3)). This structure describes the page content, and its location. Each page can be associated with a file (for named mappings) or without any further information (anonymous mappings). This structure tracks the current page's

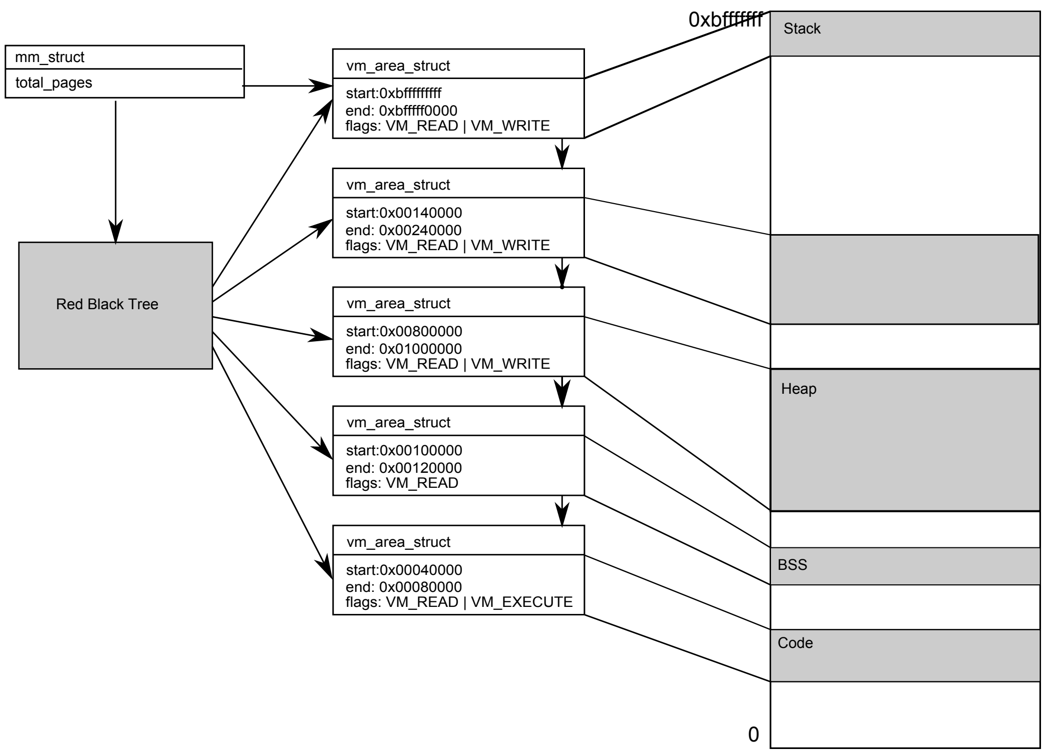

Figure 2.1: A simplified address space layout of a Linux task.

location, present in RAM or swapped out, and in how many address spaces it is mapped. Every page has an associated PFN (page frame number) that represents the physical address, the page is currently mapped to. This mapping is not fixed and changes over time.

Every task may accesses its pages via page tables [5], but in kernelspace it is also possible to get the associated page with its PFN.

The page table layout, is not specific to certain HW characteristics and thus can be applied to nearly every architecture with an MMU. Figure 2.2 show the page table layout of Linux version 2.6.11 and following. It consists of a global, upper, and middle directory. The upper directory can be fold for architectures with different needs, three example configurations of this scheme are shown in table 2.3. To collect all pages associated with an address space, the complete page table has to be traversed.

As the kernel also uses virtual addresses every page must be mapped to its address range, before it can be safely accessed, e.g., addresses above 3.2 GB, which are often configured as high memory by the kernel. The kernel cannot access high memory directly, for this purpose the kernel provides macros (`kmap_atomic` that map a user page to special slots - address windows - in its address range [5], so it

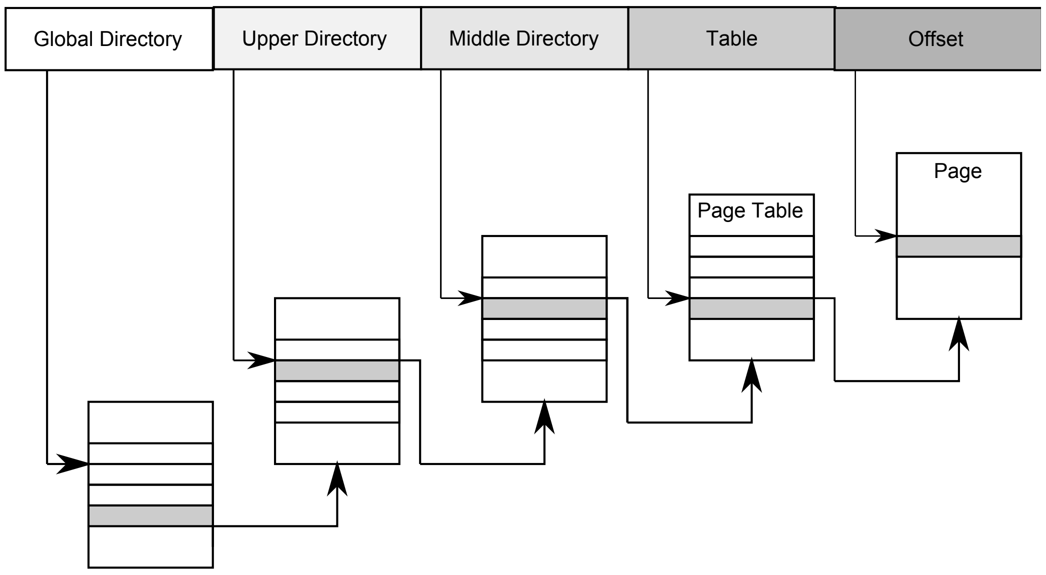

Figure 2.2: Page Table Lookup since Linux 2.6.11

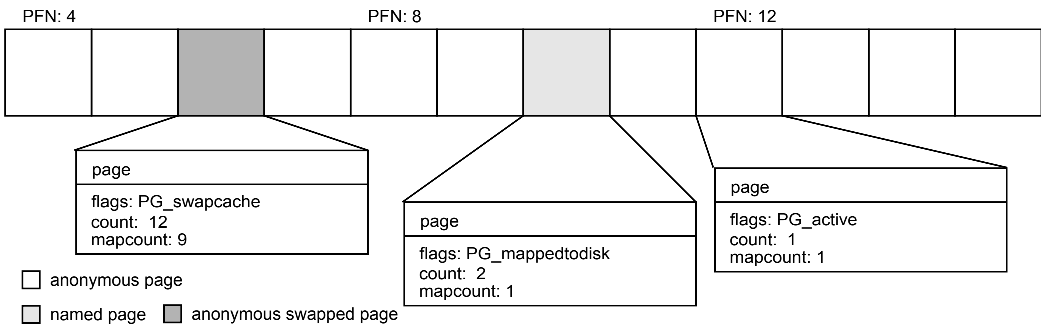

Figure 2.3: The page structure associated with every page in Linux, describing its current state and usage.

can access the stored page's content (in our case for hashing). Afterwards it has to been released with `kunmap_atomic`. If not released correctly, it might lead to corrupted kernel structures.

2.5.3 x86 vs. x64

The Linux kernel hides most of the architecture specific details behind a general interface, but some information is still important and special to the IA-32 and IA-32e platform. Page tables are hardware walked; their structure is fixed. This structure is called a page table entry (PTE) and has changed between the early IA-32 processors and the current 64-Bit platforms. A recent PTE is shown in Figure

Platform	Size	Global Dir	Upper Dir	Middle Dir	Page Table	Offset
x86	32	10	0	0	10	12
PAE	32	2	0	9	9	12
x64	64	9	9	9	9	12

Table 2.3: Address Bits and their usage; every address is divided in different parts associated with each page table level used for page address lookup [13].

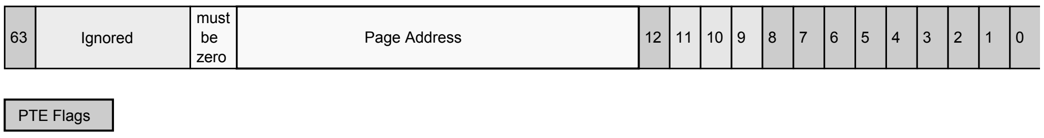

Figure 2.4: Page Table Entry on x64, table 2.4 describes the usage of the bits [13]

2.4. Each PTE consists of flags (see Table 2.4), describing the actual page usage and the associated physical address.

These flags are interesting for memory analysis, since they reflect the current state and usage of a page in an address space, e.g., it makes it possible to see if an application has written to a page or accessed it, after creation. In most VMMs, they are just set to some default values. Virtual machines often retain an internal state of each page (shadow page tables), so it makes perfect sense to use a default setting. Furthermore, page table entry flags are architecture specific and do not exist on ARM or MIPS [20].

0	Present Bit
1	Read/Write Bit - writable, if set
2	User/kernel - accessible in user space, if set
3	Page-level write-through
4	Page-level cache disabled
5	Accessed - page has been accessed, if set
6	Dirty - page has been written, if set
7	PAT
8	Global - global page, if set
63	No executions - page cannot be executed, if set

Table 2.4: Page Table bits and their meaning

```
00000000-0000ffff : reserved
00010000-0009fbff : System RAM
0009fc00-0009ffff : reserved
000a0000-000bffff : PCI Bus 0000:00
000c0000-000cffff : pnp 00:0c
000d0000-000dffff : PCI Bus 0000:00
000e0000-000fffff : reserved
00100000-bf77ffff : System RAM
    01000000-015e24ac : Kernel code
    015e24ad-01aca27f : Kernel data
    01bae000-01d03fff : Kernel bss
    2e000000-35ffffff : Crash kernel
bf780000-bf78dfff : ACPI Tables
bf78e000-bf7cffff : ACPI Non-volatile Storage
bf7d0000-bf7dffff : reserved
bf7e0000-bf7ebfff : RAM buffer
bf7ec000-bfffffff : reserved
c0000000-dfffffff : PCI Bus 0000:00
e0000000-efffffff : PCI MMCONFIG 0000 [bus 00-ff]
f0000000-fed8ffff : PCI Bus 0000:00
fee00000-fee00fff : Local APIC
ffc00000-ffffffff : reserved
100000000-63fffffff : System RAM
```

Table 2.5: I/O memory layout (of Ubuntu Server 11.04 on our test machine)

2.5.4 Physical Address Space

The physical address space is divided into accessible regions. Regions can be mapped to the system RAM, or to I/O devices. I/O mappings are common in many different architectures, such as x86, ARM and PowerPC.
Linux maintains special resource management structures to keep track of every region of system memory. A physical memory dump involves skipping unaccessible regions, this would otherwise cause a kernel panic, as a paging request cannot been fulfilled. Figure 2.5 shows a typical physical address space layout - as it is available through */proc/iomem*.

2.5.5 KSM

Kernel SamePage Merging is also known as Kernel Shared Memory. It finds anonymous pages with equal content through scanning. For performance reasons it does not scan the complete RAM [1], this reduces overhead greatly. The `madvise` system call has been extended to support a new flag (`MADV_MERGEABLE`) to advise virtual memory regions to KSM. Programs that are not aware of this new flag cannot profit from this new facility. Named mappings cannot profit from KSM as it is only implemented for anonymous memory. KSM made its first appearance in Linux kernel 2.6.32 [15]. It decides based on the algorithm, whether it should merge or leave the page untouched. KSM has two trees for this propose, a *stable tree*, containing all merged pages and an *unstable tree*, containing advised pages without any sharing candidates found yet. On a scan pass, KSM takes a page and searches the stable tree, if it finds the same content it merges the pages, the hashes are only used as an index in the tree, merging is done after a byte wise comparison. If it fails to find the page content in the stable tree, it takes the unstable tree into consideration, if a match exists the pages are merged and inserted into the stable tree, otherwise it is inserted into the unstable tree. Merged pages are set to read only and marked as Copy-on-Write pages [1].

QEmu calls `madvise` and advises the entire guest memory. KSM wakes up periodically scans the configurated amount of pages and sleeps afterwards for a configured period of time, and must be configured wisely. KSM exposes interfaces to dynamically change these settings (sleeptime between scans, amount of pages scanned per period) via the Sys FS. As we will show later, KSM is useful in some scenarios, but not fast enough in others. As the man page of `madvise` states, it is also dangerous to make excessive use of madvise, with `MADV_MERGEABLE` (or the kernel equivalent `VM_MERGEABLE`) flag. If someone advises pages, which are not good candidates, it decreases the effectiveness of KSM and can slow down the whole system [15]. On unicore systems it might be possible to stop execution completely, if the KSM configuration allows too many pages to be scanned.

2.6 Hashes

There are many ways to compare data. The most obvious way is to compare it byte by byte. In many applications it is infeasible to do such a comparison, due to bandwidth or storage space limitations.

Difference Engine and the VMWare ESX Server suggested to compare hashes instead of the entire contents, to shift spacial complexity to computational complexity.

A Hash function is a function that takes a variable length input and produces a fixed length output value – the *hash value* [10]. Hashing is widely used for file transfers (rsync, integrity check after download), to ensure integrity within a file format (PNG, PKZIP, ...), and to detect malicious tampering with cryptographic hashes.

Hashes vary in their collision resistance and size, which is why it is important to choose carefully.

We will give a short overview of hash functions and checksums, analysis of their applicability to analyze page contents.

Merging pages with the same hash, without checking the actual content might lead to data loss due to collisions [9, 10]. If two distinct data (pages) map to the same hash value, this is called a collision, and in our case this leads to false positives. Many checksums and hash functions are already part of the Linux kernel's Crypto API.

In the next sections we describe the candidates available in our kernel module.

2.6.1 Cryptographic Hash Functions

For a cryptographic hash function is by definition computationally infeasible to find a collision.

MD5 was for long years a very fast and reliable hashing function that was sufficient for all cryptographical uses, but nowadays it is broken [22] and not used for cryptographics anymore, but due to its speed and the difficulty to produces a reasonable collision, within minutes, it is still a often used function for integrity checks outside a secured environment.

The SHA-1 function is still in use for cryptographical applications, but was weakened years ago [10]. We will test all other hashes against SHA-1, since the probability to get false positives (hash collision) is very low (see Table 2.6).

A discussion on collision resistance and attacking cryptographic checksums can be found in [9, 10, 22]. SHA-1 hashes have 160 bits (2.2), which leads to a collision, if 2^{80} operations are executed, if not a more sophisticated approach is used. MD5 length is 128 bits (2.1), which leads to collisions within 2^{64} operations, see Table 2.6.

$$f : 2^{4096} \rightarrow 2^{16} \tag{2.1}$$

$$f : 2^{4096} \rightarrow 2^{20} \tag{2.2}$$

2.6.2 CRC32

CRC32 (Cyclic Redundancy Check) is not a hash in the cryptographical sense, it is a checksum, but is is often used for error checking in network protocols (e.g. IEEE 802.3), in popular file formats (PKZIP, PNG, MPEG-2), and produces a fast

Hash	Length	Collision Probability	Probability for false positive
SHA-1	160 Bit	$1 : 2^{80}$	$8.2718 \cdot 10^{-25}$
MD5	128 Bit	$1 : 2^{64}$	$5.4210 \cdot 10^{-20}$
CRC32	32 Bit	$1 : 2^{16}$	$1.5259 \cdot 10^{-5}$
SuperFastHash	32 Bit	$1 : 2^{16}$	$1.5259 \cdot 10^{-5}$

Table 2.6: Probability for false positives (in random data), it doesn't take the malicious usage or weakness of the underlaying algorithms into account.

way to check data integrity [4]. It has hardware support in some architectures, like the current Intel CPUs (`CRC32` instruction) [12]. Although it is not very collision resistant, it might still be useful in many cases, since it is fast compared to other hash functions [11]. It takes 4096 bytes and returns a 32 Bit checksum (2.3).

$$f : 2^{4096} \to 2^{4} \tag{2.3}$$

2.6.3 SuperFastHash

SuperFastHash is a very fast non-cryptographical hash function, which was suggested for usage in different papers [8] [21], as a hash function, to hash page content and us it as an index into a lookup tree (similar to a hash map) for page merging candidates. It is faster than a non-hardware-accelerated CRC32 checksum. Although it is just a checksum as CRC32 and has only 32 bits to represent the page's content, we guess it is a good candidate for comparison against other hash functions and checksums [11].

2.6.4 Hashes, Checksums and the Kernel

The Linux Kernel provides the complete set of cryptographical hash functions and checksums currently used, except for SuperFastHash, which must be include from a different source [11]. Every module may use the functions and might even profit from future improvements or hardware acceleration on different platforms. Unfortunately most of these functions do not make use of hardware accelerators (CPU instructions, or special Crypto Cards) yet [20]. For a speed comparison see Figure 5.1.

Chapter 3

Analysis and Design

Designing a new toolset is difficult, and you must be sure what problem you try to tackle. So we start with some definitions, give an overview of possible solutions and conclude, how and why we choose a particular approach.

3.1 Sharing opportunities

Pages can be classified as shareable, shared and private. Shareable pages can be further subdivided in:

- Shareable Pages
  - Shared Pages (same PFN)
  - Sharing Opportunities (same content)

    Anonymous Opportunities: same content, different pages, not file-backed

    Named Opportunities: same content, different file, and therefore page

    Mixed Opportunities: same content, different page type)
- Private Pages (unique Pages)

Pages that have the same content are shareable, and if they are already shared, they have the same PFN, otherwise they are sharing opportunities. Pages which cannot be found in the own address space (internal sharing) or in other domains (caches, other tasks) are private or unsharable.

There is are third category, mixed sharing, anonymous pages containing the content of a named page, or vice verse are counted as anonymous sharing, since there is no way to exploit these sharings yet. We calculate the maximum of saved

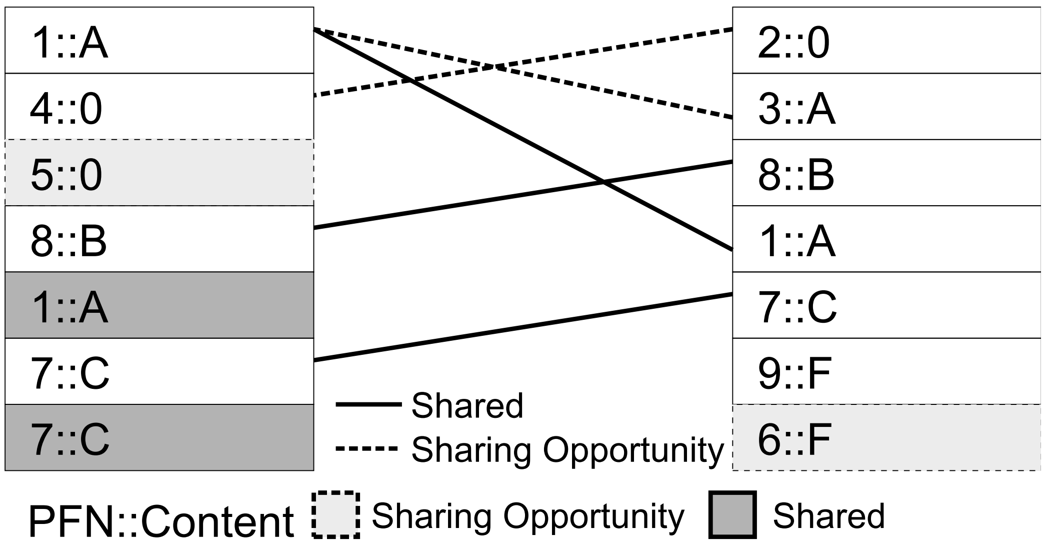

Figure 3.1: Simple sharing example demonstrating all common forms of sharing between two address spaces. It also illustrates the concept of internal sharing, which occur more often, if files are mapped into memory.

memory, by adding the amount of all internal and external sharing opportunities, to give an upper bound of sharing potential. Figure 3.1 shows a simplified sharing example between two address spaces.

In this particular case the page content is considered identical, if their hashes are identical, although [9] [10] state, their might the danger of losing data, on the one hand it seems very low, and on the other hand we will not make any merging decisions on these hashes. The likeliness of false positives is much lower as the flux of pages that change within scanning process, and furthermore we are not planning to use weak hashing functions.

3.2 Userspace Application vs. Kernel Patch vs. Kernel Module

The main question designing a framework is how to acquire the needed information? We had to ask ourselves, which of the following approaches to take. There are three different ways of acquiring information from the Linux kernel.

1. The *default interfaces* of Linux, provide only a small subset of the required data. Furthermore it would have been a puzzle game, with a lot of overhead.

2. The kernel can be modified to export information through the Proc File

System. If someone wants to use this new interface, the kernel would have to be patched and rebuild. It might be incompatible to kernel versions.

3. A third possibility is to write a kernel module. It can be loaded on demand, does not require a rebuild of the whole system, can be unloaded to avoid errors caused by it after analysis.

A kernel module is the most flexible solution to provide snapshots without hooking into a virtual machine monitor, kernel functions, or the guest OS itself. Only the hosting system needs some adaption, this reduces overhead and allows flexible analysis.

3.2.1 Communication with the userland

To keep the data acquisition fast, the kernel module is capable of raw output through the Proc FS, but to keep it compatible, it also provides data in human-readable form. The raw output must be explicitly triggered during snapshot creation.

The module collects every present page in the scanned task or system RAM, it leaves swapped out pages untouched, because this would cause the system to bring the page back into main memory or it would raise a page fault. In both cases this would harm performance and leaves the system in an altered state. For instance, an idling application, with 100 swapped out pages has more physical memory in use afterwards, an unwanted effect.
The general data acquisition is simple. First of all a PID must be specified, then the module locates the corresponding task structure and its associated memory descriptor. Then it starts walking down the virtual memory regions and their associated pages in the page table. If the page is present, the specified function hashes its content. When the snapshots is ready, it can be retrieved via the proc interface.

One special PID is reserved and never used by Linux and has no task associated. This PID (zero) can be used for special requests. If PID 0 is specified, the whole RAM or the file cache will be fetched, depending on the specified flags.

A full snapshot includes all information present in the page table, even absent pages, pages which have been allocated and not used yet. To reduce the data passed to userland further, these empty page table entries can be omitted. This can reduce the amount of transfered data in many cases (regular applications see Section 5).

Following, is a list of the most important information for every snapshot.

Snapshot Information

- Total Pages associated with Task
- Committed Pages and Swapped Pages
- Code Size and Position
- Heap Size and Position
- Stack Size and Position

Virtual Memory Area Information

These structures represent the data associated with every virtual memory regions.

- Begin and End Address
- Page Table Prototype Flags
- VMA Flags
- Associated File Mapping
- Associated File Name

Page/Page Table Entry Information

The structure describing every page in the scanned address space. Some of those fields exist only for task and will be set to zero, for full system memory scans.

- Page Flags
- Page Table Entry Flags
- Associated I-node, if any
- Hash of page content
- Reference Count of page
- Mapping Count of page

3.3 Userland Utilities

The userland utilities are based on the API and should assist the user to get the data from the kernel module as fast as possible and save them to disk or do some analysis on- or offline. They also capable of doing snapshots periodically, tracking frames, testing sharing opportunities and calculate the savable RAM. They do some preprocessing that reduces the amount of data that has to be saved to disk.

Chapter 4

Implementation

The following chapter gives a brief overview, what obstacles we had to master and what interesting aspects of the system are used for data acquisition.

The kernel module (vm_module) is the main component of this tool set. The Linux kernel does not export every function it uses internally to be used by modules. Most functions are only exported in a more general abstraction, e.g. `get_user_pages`, and the helper functions are unaccessible for modules. It was necessary to rebuild these functions or some of their functionality, to get around this limitation and acquire required data.

Since it uses infrequently changed kernel interfaces, it is compatible from Linux Kernel 2.6.11 to the latest release (3.1). And with small adaption working with ARM on a slightly modified Linux kernel (Android kernel), as the designated platform x64, x86 as well.

4.1 The scanning process

There are two different mechanisms. Acquiring data form regular tasks only, and acquiring data from the whole physical RAM.

The scan of *a task*, involves locking of the task itself and its associated page tables. If the semaphore of a task is not acquired, the task might exit during a scan. If the page table's spinlock is not held, the pages tables can be altered and the page table walk fails, the snapshot is in the best case incomplete and in the worst case the kernel module crashes, depending on the system configuration this leads to an unusable module or a panicked system. So the impact on the target application cannot be minimized completely. These locks imply that task are unable to modify page table content, allocate new memory or change any attributes of a page.

The physical RAM snapshot lacks semantic information. There are no page tables available, only the attributes stored in the page structure and the pages' contents. To determine which pages belong to the System RAM, the same structure is accessed by the Proc FS entry (*/proc/iomem*). Trying to access other regions, depending on the device, who is referred, causes the kernel to crash the module or the entire system.

4.1.1 User Page Access in Kernel

The kernel cannot access all pages directly, because it is working with virtual addresses. On 32 Bit system, it is thus limited to 4 GB of memory, but the system might be capable of more than 4 GB memory using PAE (Physical Address Extension). This is why the macros need to map the user addresses to special kernel address ranges, to ensure accessibility before hashing page's content. Special functions must be used for this mapping, because holding a lock or semaphore or even both, the regular mapping mechanisms `kmap, kunmap` cannot be used, these functions cause bug traces or even kernel panics on some systems. Parts of the crypto interface suffer from this problem on some kernels. They make use of a special structure that takes pages and hashes, encrypt, decrypted them and the kernel module cannot influence their mapping functions. They only possible way, would be to change the kernel for these platforms or waiting for an upgrade.

4.1.2 Hashes

Hash functions are part of the kernel's infrastructure. So the module will profit from any improvements done to this basic kernel functionality. For some architectures a new instruction set has been introduced and should therefore be accessible through those interfaces in the future.

The hash functions, implemented by the Linux kernel, have some overhead, because every time they are used, a string lookup is done and a structure is allocated and initialized for the hash function. All details are hidden behind that interface and cannot be used directly, as they are not exported. Our implementation uses the CRC32 macro of the kernel. The cryptographic hash functions are done via the crypto interface. Only the SuperFastHash requires external code, which we included and made it usable with the kernel.

The kernel module spends most of its time hashing page contents, so it is wise to consider, which hash function is suitable for our analysis.

The module provides support for different hashes (Table 4.1). The simplest way to get a fingerprint of data, is to take some bytes. This method is used for the so called "simple pattern".

```
for(i=0;i<12;i++)
    buffer[i] = page[1 << i];
```

It is very fast, only little data is touched and no calculation is done, but also it is very faulty and produces many false positives (collisions). More sophisticated functions are also supported. MD5 is commonly used for checksums. SHA-1 has no known collisions yet, and might therefore used as an reference. It is easy to add new hashes, in future releases of our module.

Method	Description	Resulting bytes
Simple Pattern	12 bytes + 4 bytes	16 bytes
SuperFastHash	`PAGE_SIZE` considered	4 bytes
CRC32	`PAGE_SIZE` considered	4 bytes
CRC32_EX	Page divided into 4 equal blocks	16 bytes
MD5	`PAGE_SIZE` considered	16 bytes
SHA-1	`PAGE_SIZE` considered	20 bytes

Table 4.1: Provided hash functions, sorted by expected speed (for speed measurements see Figure 5.2

We proposed CRC32_EX to make CRC32 more reliable, each page is divided into 4 equal blocks and each gets its own checksum, this provides speed like CRC32 and more collision resistance, furthermore it retains more information that can be used for subpage analysis.

4.1.3 Page Table Walk

The page table walk is the same on all IA-32 and ARM platforms, and is described in the function `get_user_pages` [20], which is used to get all user pages content. This function cannot be used for our approach. We need more information about every page and its hashed content. Also, this function swaps pages back in, which is a unwanted behavior as mentioned before.

Not all entries in the page table lead to pages that can be accessed directly, I/O pages have some restrictions and should not be touched. Large regions of a virtual address space are not mapped to physical addresses and have to be skipped.

4.1.4 Memory Consumption

Our kernel module and our tools need memory to store a snapshot till it is processed. The kernel allocates the maximum amount of memory to hold a complete snapshot (including the absent pages), the userland components need to allocate

	Real Memory Usage	Kernel Memory	User Memory	Peak
RAM	24 GB	402 MB	max. 402 MB	max. 804 MB
Process	1 GB	16 MB + VMRD	max 18 MB	max. 36 MB

Table 4.2: Memory Consumption during snapshots approximated with 64 bytes per Page. A regular dump would need the amount of the real used memory.

additional memory for the data transfer. The userland buffers fit exactly to the actual snapshot size, which in most cases is smaller than the kernel buffer.

The size of the page describing structure varies form platform to platform, so we assume a size of 64 byte per page for our calculations. On IA32 architectures the structure is 52 bytes (without SHA-1 support 48 bytes) and 68 bytes on x64 architectures (64 bytes with disabled SHA-1 support). Additional memory is needed for the virtual memory region descriptors (VMRD). Table 4.2 shows some common scenarios and their memory consumption in kernel, userland and during data transfer. To dump a complete system equipped with 24 GB of RAM (such as the machine in our test setup), you need 804 MB of free memory, during the kernel to userland transfer, which reduces the dump size to 3.4% of a full dump.

4.2 API & Tools

The API manages all tasks of communication with the kernel, memory management and conversions to human-readable strings or CSV format. We choose the CSV format, because it can be processed with common spreadsheet programs and has a simple structure.

The part of the API to communicate with the kernel module, is done in pure C90 using the system supplied functionality, where needed, a port to a different OS should be simple, but has not been done yet. The analysis programs are written in C++, since it needs some features that are only available in general implementations of C++ (since Technical Report 1), the `unordered set`, a hash map implementation, to find shared/shareable pages.These tools are command-line based and calculate the maximum of savable memory, they fetch, save and print snapshots, track the amount of changing pages. Some test tools to validate the kernel module exist. This following command takes 200 snapshots of the task 1 and task 42 every 1 second, calculates the maximum of savable memory and prints the CSV output to stdout.

```
./savedmemory -t=1 -n=200 -c 1:29 42:29
```

Chapter 5

Evaluation

To begin with, we had to find the best fitting hash function. Best fitting in this case means a tradeoff between collisions (false positives) and speed. After these initial testing, we evaluated the different workloads and their potential sharings.

5.1 Testing Environment & Procedures

The testing environment (Table 5.1) is completely based on Linux, the host OS. The guests may vary, but we stick to Linux. We decided to use Ubuntu, since it is available and widely used. The common configuration (Table 5.2), which is used for all benchmarks, with the default KSM configuration of Ubuntu Server 11.04, which scans 100 advised pages every 200 ms, if not noted otherwise.

There are many different benchmarks out there, some of them more computational intensive some more I/O bound. Although a kernel build is not a distinct benchmark, it is still a good I/O bound job and was suggested in [17]. Kernel building process and SPEC CPU2006 are our candidates for testing. It is possible to see all effects with these two benchmarks.

We executed the kernel builds in parallel in two VMs booting the same disk image, with the same amount of memory. The snapshot tool triggered a snapshot every 2 seconds, and merged the results in a hash map, to divided the shared pages, from the sharing opportunities and the unsharable onces. No disk access were invoked from the snapshot tools, they work completely in memory. Although the tools increase their memory usage over time, it had no impact on the target VM, since the system was well equipped with RAM and there were enough free pages, so Linux had not to reclaim, or swap pages.

To measure the exact usage of the host file cache, we rebooted the machine before each test and no other programs were running before and during the tests, to keep external impacts on the file cache as low as possible. For every kernel build,

CPU	Intel Core i7 980 @ 2.67GHz
RAM	24 GB DDR3
OS	Ubuntu 11.04 Server running Kernel 2.6.38-11-server

Table 5.1: Testing Machine Configuration

CPU	x86_64
OS	Ubuntu 11.04 Server running Kernel 2.6.38-8-server
KVM	enabled
RAM	configuration depends on experiment

Table 5.2: Virtual Machine QEMU Configuration

we restored the original image, so every build started from the same basis. The kernel build used the original Ubuntu build scripts, as available in the repositories of Ubuntu Server 11.04. All test run till they have finished or they started to build human interface drivers (approximately 45 min later in this configuration).

The same rules apply to the SPEC benchmarks, we aborted them after 45 minutes. We executed them with different memory configurations, but we became no new insights.

The kernel builds were also used to test the tools, the kernel module and acquire the data for analysis as well.
The first task is to determine the best fitting hash, according to its throughput and its collision resistance. The selected hash function will than be used for the concrete analysis.

The idea here is to take large address spaces resulting form a kernel build in a VM – and pausing their execution and take snapshots with different hashes. The SHA-1 is the reference hash function and every other hash has to compete with it. SHA-1 is too slow to be used for fast analysis.

5.2 Tool Set

The following section contains the tests of our tool set, their overhead and the performances graphs for the different supported hashes.

5.2.1 Linux Kernel Module

We developed test tools, with which we validate the collected information, and compare our results, with the internal counters of Linux, for example with `/proc/<pid>/statm`

	Task Memory Usage	Kernel Memory	User Memory	Peak
FileCache	11.7 GB	402 MB	185 MB	587 MB
RAM	13.1 GB	402 MB	207 MB	609 MB
bash	2.4 MB	48 KB	48 KB	96 KB
QEmu	13.7 MB	260 KB	260 KB	520 KB

Table 5.3: Memory Consumption during snapshots with 68 bytes per page on test machine. Some small real world scenarios - a bash and a QEmu with booted Linux kernel. A full memory dump would need 13.1 GB disk space.

and the amount of consumed memory by the file cache with `/proc/meminfo` or the amount of shared pages with the KSM counters. It passed these tests. The fluctuations were about 300 of 250 000 scanned pages, in the worst case it was around 0.5%. The calculated sharing opportunities and the actual shared pages were even better, the error was less than 100 pages in average, about 0.31% in our test scenarios. The slight time difference was the main cause for this differences, and it is rather complicated to eliminate this problem completely, but its estimated error is less than 1%.

Our snapshot mechanism produces overhead. It is mainly caused by the copy operation from kernel to user land – one improvement for future releases would be direct file system access, page donation from kernel to userland, or network transfer. Table 5.3 shows the memory consumption of our tools during snapshot acquisition.

This reduces the memory consumption in contrast to other solutions, which make a full memory dump, to approximately 1.5%. And also the temporal drift within a snapshot is lowered, due to faster acquisition process. The actual used User Memory depends on how many pages are populated. In regular processes (user applications common in Ubuntu), only 50% of the reserved memory is actually used. This is also true for most VM on startup, but once all pages were touched, there must be a reclaim mechanism to shrink the VM again, which is not done by default.

5.2.2 Hashes

After testing with different workloads, it became obvious that CRC32_EX should be sufficient, for all planned tests and fast enough (see Figure 5.2). In every tested scenario it performed as well as MD5 and SHA-1, but with much better performance. Its performance is nearly as good as the regular CRC32 and is only outperformed by SuperFastHash (Figure 5.1).

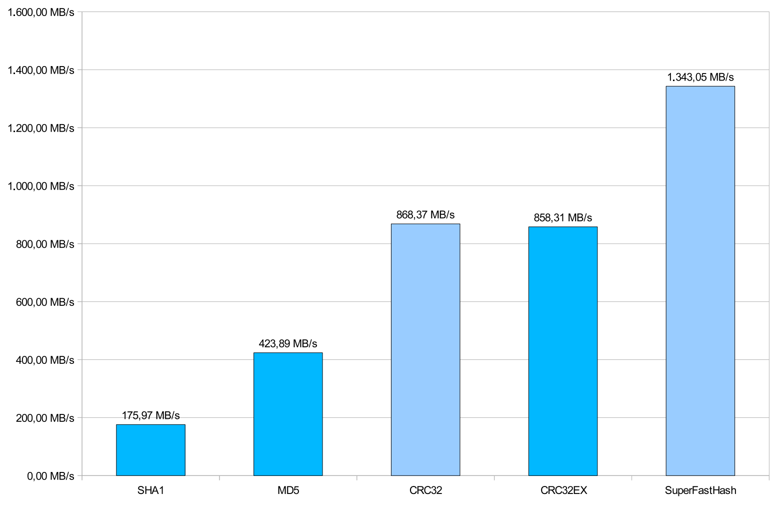

Figure 5.1: Comparison of the different hashing functions and their throughput. Measured with a VM building the Linux kernel and equipped with 1024 MB of memory.

Algorithm	False Positives
SHA-1	0
MD5	0
CRC32	7
CRC32_EX	0
SuperFastHash	7
Simple Pattern	9736

Table 5.4: Comparison of collisions. Here a 1 GB address space running a Linux kernel build. Reference is SHA-1. It clearly shows *simple patterns* are too faulty, but every other hash is a candidate for most testing scenarios.

As table 5.4 shows the amount of colliding pages is low. So all functions, except *Simple pattern*, were candidates for the following tests. There is of course a flux between every snapshot, which might be even higher than the false positives, this is an other reason to choose a robust candidate.

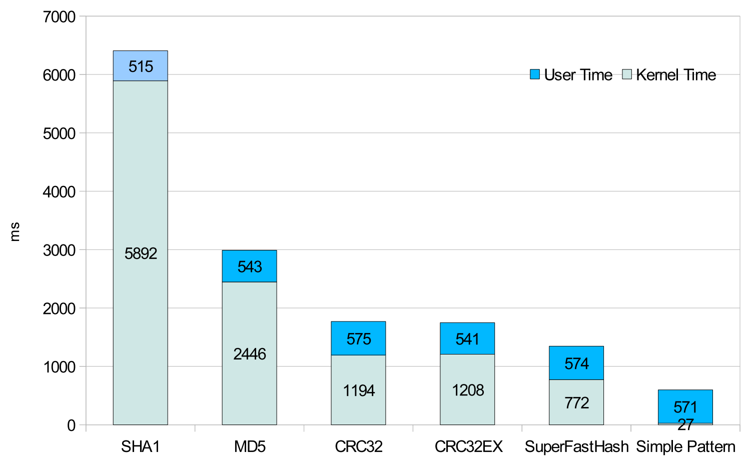

Figure 5.2: Time it takes to hash 1 GB of a frozen VM (QEmu), which executes a Linux kernel build. The test has been done 10x and the average processing time is displayed in this diagram.

5.3 Benchmarks

We executed the benchmarks on the testing machine as described in section 5.1. The regular scan period was 2 seconds, which leads to an effective resolution of 4 seconds, for large address spaces (> 1 GB), since this period is large enough, we were able to do an online analysis. The CPU has enough cores (4 cores with 8 threads) to run the VM on one and the analysis on a different core. An offline analysis would have on the one hand polluted the Page Cache, using cached I/O or on the other hand slowed the snapshots period down even further, using direct I/O.

5.3.1 KSM Activities

KSM activities are slow and computationally intensive, even the corresponding man pages contains a warning: "It can consume a lot of processing power; use with care." [15]. Figure 5.3 shows the work of KSM over time, in three different KSM configurations, which increased "pages_to_scan" values (200, 500, 1000), the scan period ("sleep_millisecs") remains at 200 ms. The maximum savable memory, is the amount of memory that would be saved with all possible memory sharing and reclaiming techniques, a calculated value.

We used a balloon test, a simple program that can be inflated with different page contents, allocating 512 MB of RAM in two different VMs. Every negative peak indicates a change in page data content of one or both VMs.

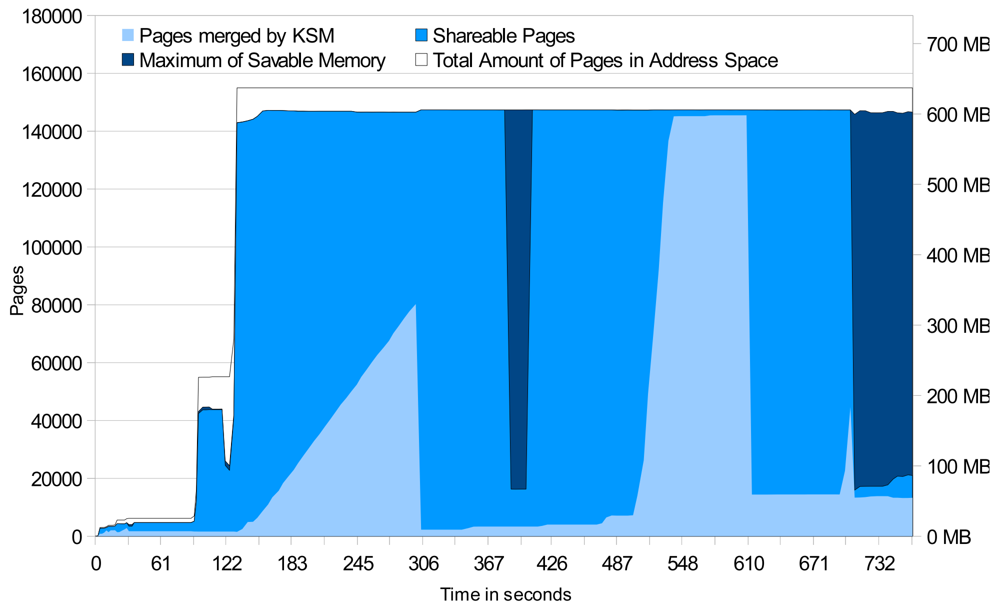

Figure 5.3: KSM activity during two VMs executing a balloon test, it clearly shows how lazy KSM scans and merges pages, in its default setting. Even with more aggressive scans (second half), it takes time and computational power to merge pages.

KSM needs more than one minute to merge 245 MB of data, in a increased setting (200 pages scanned every 200 ms), and the whole machine is just idling (no changes in the guests memory allocation). With much aggressive scans it is faster, but takes more CPU power. KSM is only a good solution if enough computational time is available. If two programs open the same file content from different files and map it into their address spaces, KSM is unable to merge any page (since these pages are named and not anonymous). The effect would be the same for user programs installed with 0install, or redundant disk data. Figure 5.4 illustrates this scenario. There is no way reducing the pages, although the files are mapped shared, they cannot be shared – and consume two times the size in system memory. Anonymous mappings cannot save the day, it just creates new overhead (2 pages in the file cache + 1 page mapped anonymously into two processes), if the file cache is not flushed afterwards.

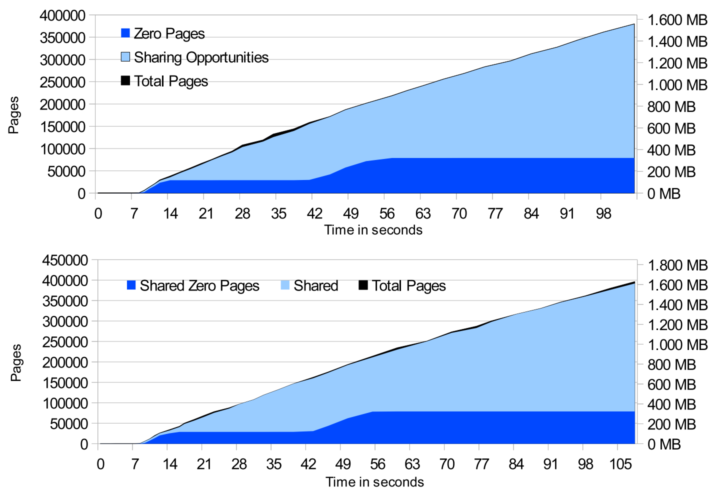

Figure 5.4: Two processes opening the same file and map it into their address spaces (second diagram). The residual set of each process is extreme low. If the same processes open files with the same content, but with different inodes, there is no default sharing, furthermore KSM cannot merge any of them.

5.3.2 Linux Kernel Builds

The kernel build process reads many source files, compiles them and writes object files, which are later linked into executable binaries. This causes a lot of I/O operations, which pollute the file caches, both of the VM guest and the host system.

We tested different memory configurations, Satori suggested 256 and 512 MB of RAM, this was a good starting point, but then we turned to even smaller 128 MB configuration and bigger 1024 MB or even 2048 MB. As figure 5.8 shows the fluctuations over time, it is clear that kernel build expects are working set with more available memory, 256 MB seems sufficient, but 512 MB is more common.

As you can see, the amount of memory for a VM is essential, less memory leads to less shareable pages and slows down the execution as shown in Figure 5.6. The more memory is available the more pages can be shared and the faster the kernel builds. This was also shown in [17].

If there is not enough memory for a kernel build working set (see Figure 5.12), most of the potential sharable pages are short lived, their big sharing potential is wasted (compare Figure 5.5 with Figure 5.6).

The testing VMs have been reduced in memory size to show the effects of low memory conditions. Now the sharing opportunities are changing quickly and KSM is not able to merge all shareable pages. QEmu has been configured to use

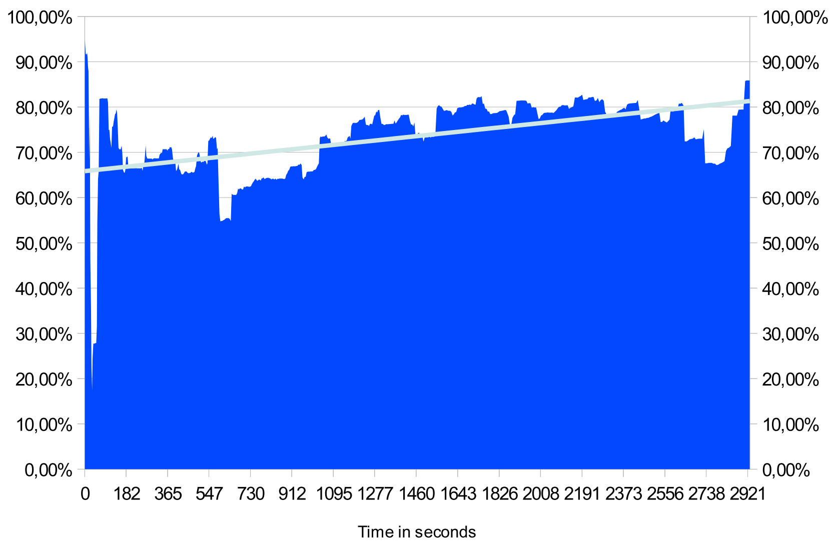

Figure 5.5: The potential sharing between two VMs building the Linux kernel with 1024 MB allocated memory per machine. This figures shows the maximum percentage of pages that can be shared between VMs at every moment in time and their internal redundancy. And it is increasing as processing continues. With a average saving potential of 75%.

128 MB of RAM for the guest OS, since our observations and [17] suggested that it needs at least 200 MB (256 MB) of memory to work fast and hold the minimum of the building process' working set. The contrast between Figure 5.7 and Figure 5.8 is enormous, the maximum amount of shared pages has dropped drastically - to an average of 25%. furthermore the theoretical memory savings are fluctuating even faster.

We wanted to know, where most of the sharing potential originate from, so we compared the file cache (of the host) and a VM building the Linux kernel. We discovered that most pages can be shared between the guest and the host file cache, the astonishing results (Figure 5.10) show the increasing amount of file cache pages of the host and the growing memory consumption of the guest in the VM. The host file cache keeps growing, whereas the VM is limited to 1 GB. The inability of KSM to merge pages with the file cache is shown in Figure 5.9.
There are sets of 3 pages with the same content in the testing system. Two of them can be merged because they are anonymously mapped into the VMs, but one page, the original in the file cache is still there, although this page could also

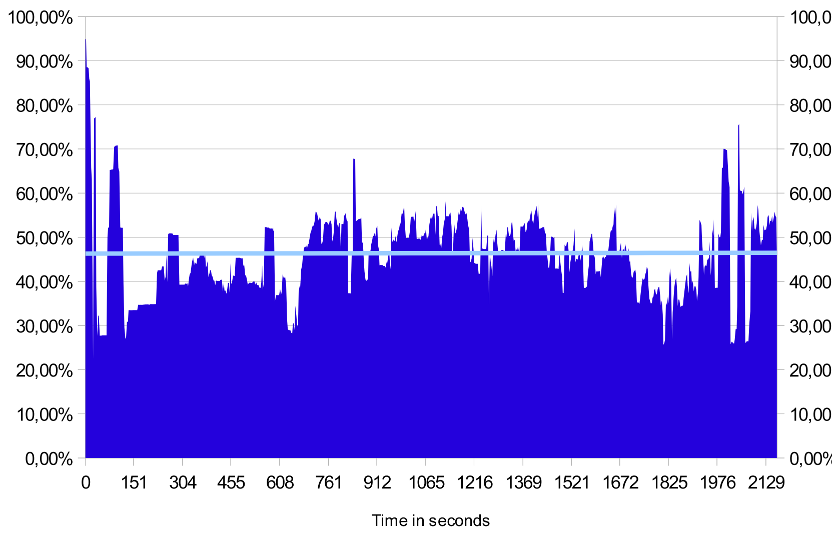

Figure 5.6: The potential sharings between to VMs building the Linux kernel with 128 MB. The sharings have more fluctuations compared to Figure 5.5. With only an average saving potential of 47%. This is a theoretical saving and it is hardly possible to exploit all opportunities, due to the fast changing page's content, in real systems.

be merged. This could be done without scanning, since every I/O operation must be processed by the VM monitor, or the underlying OS, it can instantly map the page into this address space and make it a Copy-on-Write page.

Another problem of KSM is illustrated in figure 5.13. When only one sharing candidate exists in the system, in this case one VM builds the kernel and the other just doing a balloon test, there is no merging candidate available for the current implementation of KSM.

The previous tests suggested that the amount of shareable data increases with the amount of memory allocated to each VM. Figure 5.11 shows the different VM configurations and their average sharing potential. This clearly shows the correlation between allocated memory and sharing potential. The private pages are growing linearly, whereas the shareable pages growing exponentially. The amount of private data is low, compared to the shareable pages. If a file cache sharing system exist, it would allow many different kernel builds to execute with 200 MB of private memory, and complete their task as if they had 1024 MB of RAM allocated to each.

As figure 5.12 shows, about 47% of these sharings last longer than 16 seconds,

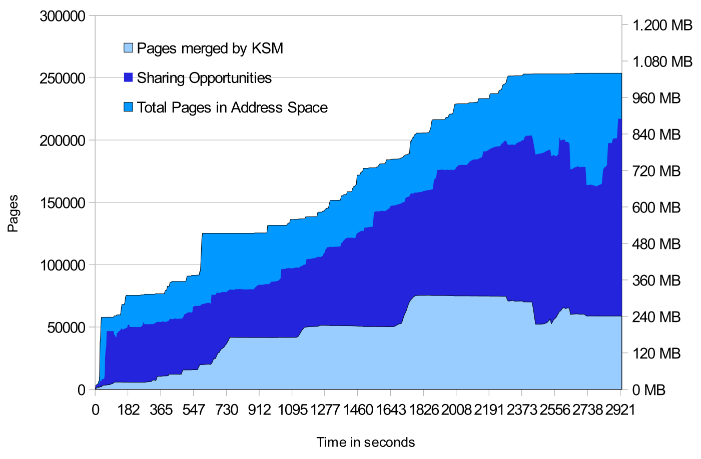

Figure 5.7: Two VMs executing a nearly synchrony Linux kernel build with 1024 MB of RAM. It clearly shows the growing of the VM during execution and their sharing potential between these VMs and the amount of memory that KSM merges during execution.

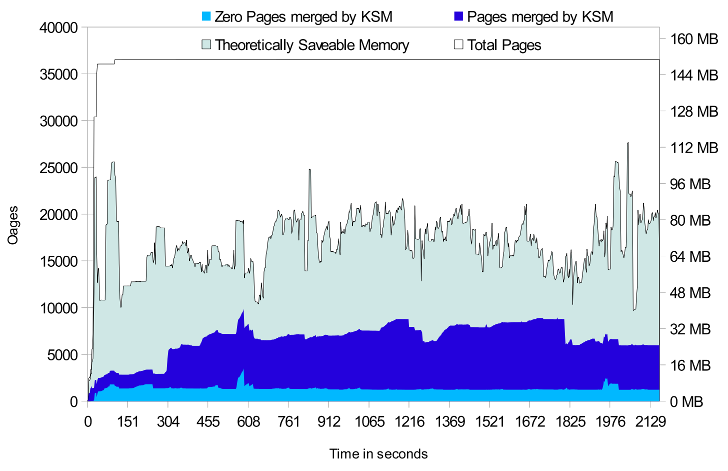

Figure 5.8: Two VMs building the Linux kernel with only 128MB of RAM allocated. The building process is slower and the page cache of the guest OS can hardly be utilized due to high memory pressure in the VM, it has to reclaim pages very fast.

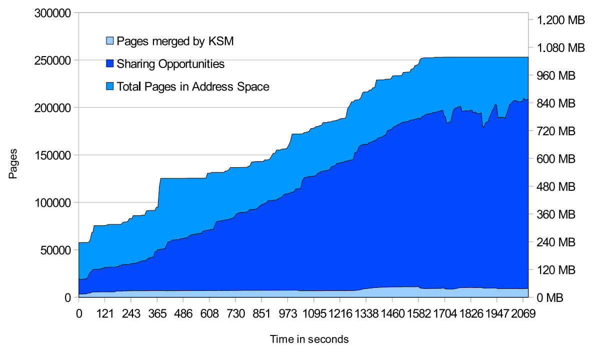

Figure 5.9: The sharing potential between a VM building the Linux kernel and the host file cache with 1024 MB. The freshly rebooted machine has a small file cache, which is growing over time, in the same way the amount of shareable pages grows. When we compare this figure with Figure 5.7 it become obvious that most sharings belong to file caches of the two VMs. In this scenarios KSM is unable to merge pages, since it cannot get hold of the named file cache pages.

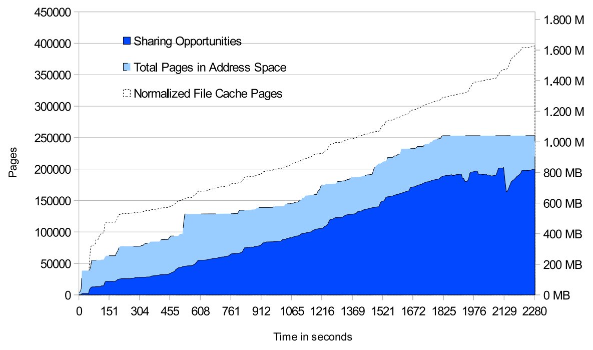

Figure 5.10: The sharing potential between a VM building the Linux kernel and the host file cache (VM equipped with 1024 MB). The rebooted machine has a small file cache, which is growing over time, in the same way the amount of shareable pages grows. When we compare this figure with Figure 5.7 it become obvious that most sharing belong to file caches of the two VMs. In this scenarios KSM is unable to merge pages, since it cannot get hold of the named file cache pages.

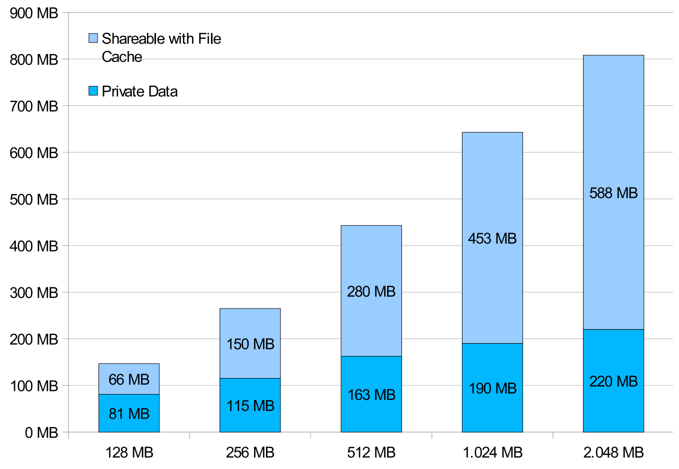

Figure 5.11: Sharing potential in different VM memory configurations. This figures shows the average amount of unsharable (private) data and the potentially sharable data. This clearly shows the correlation of memory associated with a VM and its saving potentials. The amount of private data increases linearly, whereas the sharing potentials are rising exponentially until they reach a certain level.

this is approximately the same amount that can be shared with the file cache. But for a complete run this reduces to 32% due to the enourmous memory usage, the file cache in the guest machine has to shrink every time, the building process requires more memory.

It is clearly possible to reduce the amount of shareable pages, when one VM executes a benchmark and they others just idling with large amount of memory used. These machines can surely be compressed by KSM or other means, but a machine executing, without a sharing partner. Figure 5.14 shows such a situation, at the beginning of this test, the kernel build has not been started yet, and the sharing potential with other VMs is high, since the executing the identical Linux kernel and balloons. Then both begin their work, one starts the kernel build, the other leaving their balloon untouched – just changing its content over time. The file cache is still growing but not used for the VMs, but would be a good candidate.

Even the file cache has redundancies and so unexploited sharing opportunities, that could be merged, so at least 10% (9.53% in this example) of memory can be saved. If a system detects this kind of named sharing, it can furthermore deduplicate these files back to physical storage, and save even disk space. For this porpuse it need not to scan its disks periodically, it just gets this information for free, if the file cache supports sharing and a self redundancy check.

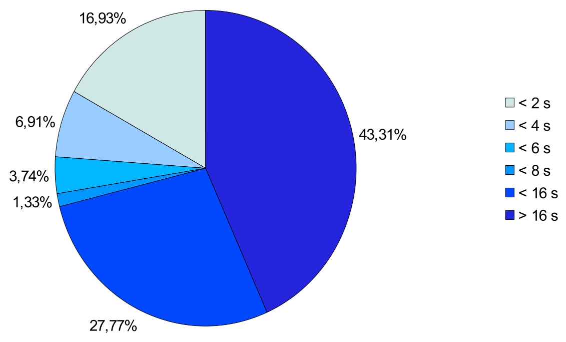

Figure 5.12: This 1 minute snapshot during kernel builds with 128 MB of RAM gives a impression how long sharings last in a high memory pressure environment. 47% last longer than 16 secondes, but only 32% last longer than 10 minutes. Whereas in a bigger VMs, more sharings, last longer, this was also discovered by the Satori paper [17].

5.3.3 SPEC Benchmark

SPEC (Standard Performance Evaluation Corporation) benchmarks such as bzip2 in CPU2006 are computationally intensive and work completely in memory [19].

To show the effects of KSM and the general sharing opportunities not touched by KSM. We executed 401.bzip2 in two VMs. The peaks in figure 5.15 illustrate its work and its progress. The operations are repeated, so the same peaks appear again and again. It is an interesting task and shows the limitations of a memory scanner, every time the work seems to be finished the shared pages are lost and have to be merged again. One third of all allocated pages are merged in average, for short periods of time, although more than 60% of all pages are shareable, excluding the zero pages (with zero pages up to 80%).

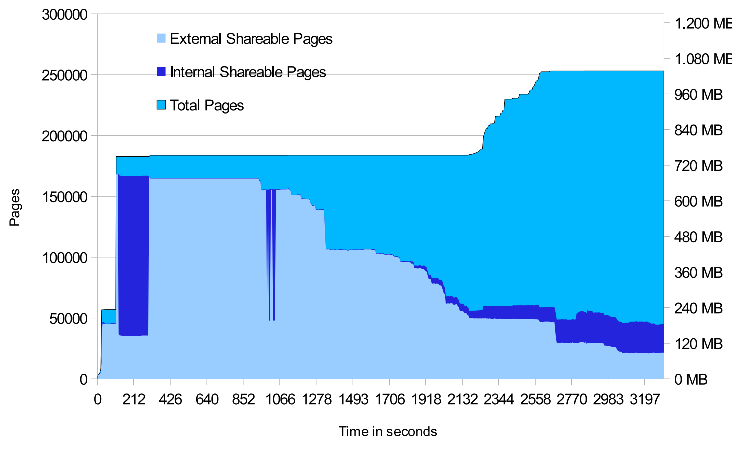

Figure 5.13: A kernel build versus VM with a balloon in it, both configured to use 1024 MB of RAM. KSM is able to merge the parts of the disk image that both machines have loaded into anonymous pages, but is unable to merge the pages in the file cache of the host with the guest.

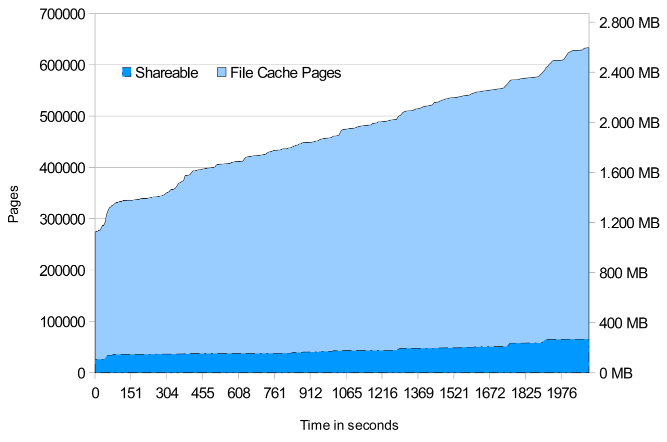

Figure 5.14: This figure shows the development of the file cache during a Linux kernel build and its redundancy. Approximately 9.53% of memory occupied by the file cache can be deduplicated.

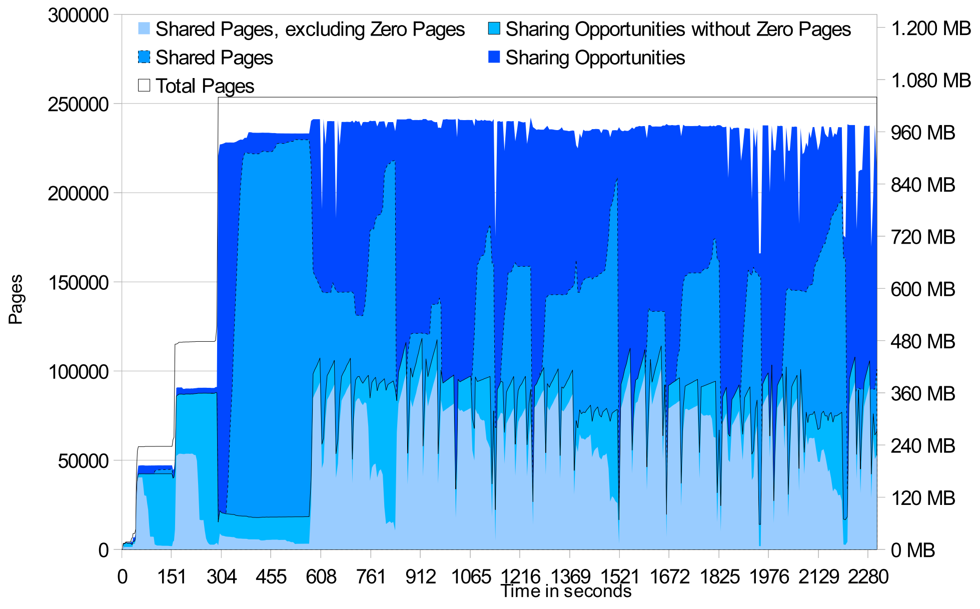

Figure 5.15: SPEC CPU2006 401.bzip2's execution in two VMs (equipped with 1024 MB of RAM each) over time, with KSM scanning 200 pages in 200 ms. Every time a sharing has been established it is destroyed within 15 seconds due to the changing compression workload of the benchmark. Although the theoretically sharing potential is high (60% of the pages could be shared, even without the large amount of zero pages), it can hardly be exploited by KSM. The sharing opportunities are non-zero pages, which have not been merged. KSM excludes zero pages, but they still slow down the scanning process.

Chapter 6

Conclusion

We discovered that impressive sharing potential exists, between VMs executing the same tasks or just using the same source data. The most sharing potential originates from VMs executing code with a lot of disk I/O, utilizing the file caches. This potential cannot be harvested by KSM, since it is unable to merge named pages. Furthermore it takes a lot of CPU power to compare and merge pages, if it is possible to get such sharing with the file cache of the host OS, they could be instantly in place and would only break on write operations. For example two VMs – maybe in an cloud environment, try to build the Linux Kernel in parallel. In most case there would be three times the same page, one copy for every VM and one page for the file cache. KSM would take its time to reduce this to two pages, still one for the file cache left. Some might argue that it would be possible to use direct I/O for the VMs, but there still would be two pages that KSM would have to merge and every VM must reread their data from disk or a special cache in the virtual machine monitor.

KSM does a good job, when only anonymous pages are involved and computational power is cheap (compared to memory) or the guest OS cannot be instrumented to give hints about the page's content.
A complete and general solution to every problem described earlier is hardly possible, but a combination of memory scanners for anonymous pages and a similar approach for named pages, might reduce the demand for physical memory drastically.

The CPU2006 benchmark suite can hardly profit form cached files or a similar mechanism. CPU2006 benchmark does not write or read from disks, except the basic parts of the OS and the actual executable code. And most of the sharings are short lived, there might be only a instrumented guest OS, an approach done by Satori [17] or by balloon drivers [21], which might be able to regain some memory from time to time. If all CPUs are in use for computations, it might be impossible to do a reasonable merging or reclaiming, without freezing the VM.

6.1 Tools

The tools performed as expected, we reduced the overhead of each memory dump to 1.5% and fetched all information required for sharing analysis. The speed is acceptable (see Figure 5.1), but further improvements are possible. When the Linux kernel is capable of accalerations for hashing, the kernel module will profit automatically.

These tools make it possible to scan, compare and explore new sharing potentials, on Linux based systems. It also runs on the slightly changed Android kernel on ARM. Future ports to other operating systems, such as Windows are easily possible.

6.2 Benchmarks

The benchmarks have shown the potential of new unexploited sharing opportunities and how computationally intensive benchmarks with nearly no disk access profit from Kernel SamePage Merging, if enough CPU power is left, for the KSM daemon. But on the other hand, it also shows the limitations of KSM. Sharings are often broken and the work of KSM was a waste of time, e.g. int the CPU2006 benchmark suite.

The benchmarks show the potential for named page sharing between host to guest and guest to guest. As anonymous pages are already merged by KSM, it would be great to have a similar mechanism for named pages or a mechanism instant installing sharing without scanning and without a instrumented guest OS. This all should be transparent for the guest, so even older, unmodified operating systems can profit, the necessity is shown by Waldspurger et al [8, 21].

All in all, it seems to be a good idea to combine these approaches, to save the maximum amount of memory with less overhead and small impacts on the guest. If you prefer a unmodified guest operating system, then a yet-to-develop named page merger and Kernel Shared Memory will save a great deal of memory. If a guest can still contain adapted drivers, it is best combined with a balloon driver, which can fast reclaim memory for other usages and might even be easily extended to control the file cache of the guest OS and so provide even better performance. For example, this specialized balloon driver can reclaim the guest's unused file cache pages immediately, if the host has to shrink its caches. So a VM would have 256 MB of private memory and the whole file cache of the host available.

6.3 Limitation & Future Work

Our kernel module is not able to take snapshots on different cores simultaneously and has not been tested on different architectures and different configurations, e.g., with huge pages enabled. Although some initial test on ARM and Android kernels were successful. Furthermore, it has only one interface to acquire the collected information (through the Proc FS), future versions might include a direct to disk interface, network interface (Android cellphone to host computer) or memory donation from kernel to userland, which reduces the overhead caused by transferring the snapshot's data to disk or user space memory. The amount of free memory required for snapshots can than be less than 2% of the installed memory.

Although we provide many different hashing methods, it might be still useful to test a specific hash before using it. The module can be easily adapted to new hashing functions and different architectures.
Furthermore, the userland tools are not capable of every possible comparison, but with the provided API it should be easy to create specialized tools very fast, with less knowledge of the underlying acquisition functionality.

It would be interesting, if someone exploits the potentials of file cache merging, provide a driver, which allows simple host file cache access, or special hints for KSM to work faster and more efficient.

Chapter 7

Glossary

Hash in this thesis, everything that reduces data to a fixed size hash value

File Cache also Page Cache, Buffer Cache, pages which contain data associated with files

Frame also physical page, machine page, it is the physical representation of the host pages

KSM Kernel SamePage Merging, also Kernel Shared Memory

KVM Kernel Virtual Machine, a hypervisor included in the Linux kernel

Page smallest administrative structure of the physically available RAM

System RAM physically installed memory, excluding memory allocated by devices

Task also Process, address space with at least one thread

x86 also IA-32 (Intel Architecture for 32 Bit)

x64 also amd64
the architecture introduced by AMD for 64-Bit extension to x86

Bibliography

[1] I. Eidus A. Arcangeli and C. Wright. Increasing memory density by using ksm. *Linux Symposium*, 2009.

[2] David Anderson. Red hat crash utility. `http://people.redhat.com/anderson/crash_whitepaper/`, 2011.08.05.

[3] John Berthels. exmap module. `http://www.berthels.co.uk/exmap/`, 2011.08.05.

[4] Glenn; et al. Boutell, Thomas; Randers-Pehrson. Png (portable network graphics) specification, version 1.2. `http://www.libpng.org/pub/png/spec/1.2/PNG-Structure.html`, 2011.08.20.

[5] Marco Bovet, Daniel Pierre ; Cesati. *Understanding the Linux kernel : [from I/0 ports to process management; covers version 2.6].* OReilly, Beijing, 3. ed. edition, 2006.

[6] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *SIGOPS Oper. Syst. Rev.*, 31:143–156, October 1997.

[7] Ulrich Drepper. How to write shared libraries. `http://www.akkadia.org/drepper/dsohowto.pdf`, December 2010.

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

[9] Val Henson. An analysis of compare-by-hash. *HotOS*, pages 13–18, 2003.

[10] Val Henson and Richard Henderson. Guidelines for using compare-by-hash. `http://infohost.nmt.edu/~val/review/hash2.pdf`, 2011.08.09.

[11] Paul Hsieh. Comparision of superfasthash. `http://www.azillionmonkeys.com/qed/hash.html`, 2011.08.09.

[12] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2: Instruction Set Reference, A-Z*, 2011.

[13] Intel Corporation. *Intel Architecture Software Developer's Manual Volume 3A: System Programming Guide*, 2011.

[14] Jacob Faber Kloster, Jesper Kristensen, and Arne Mejlholm. *On the Feasibility of Memory Sharing in Virtualized Systems: Content-Based Page Sharing in the Xen Virtual Machine Monitor.* PhD thesis, University of, 2007.

[15] Linux Kernel manpage. *Linux manpage madvise*, 2010-06-20.

[16] Linux Kernel manpage. *Linux manpage mmap*, 2010-06-20.

[17] Grzegorz Miłós, Derek G. Murray, Steven Hand, and Michael A. Fetterman. Satori: enlightened page sharing. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, pages 1–1, Berkeley, CA, USA, 2009. USENIX Association.

[18] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating system concepts.* Wiley, Hoboken, NJ, 7. ed. edition, 2005.

[19] SPEC.org. Spec cpu2006 benchmark description. `http://www.spec.org/cpu2006/Docs/401.bzip2.html`, 2011.09.07.

[20] Linus Torvards et al. Linux 3.0 kernel.org. `http://kernel.org`, 2011.08.12.

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

[22] Xiaoyun Wang and Hongbo Yu. How to break md5 and other hash functions. In Ronald Cramer, editor, *Advances in Cryptology EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 561–561. Springer Berlin / Heidelberg, 2005.