Determining the use of Interdomain Shareable Pages using Kernel Introspection

Jacob Faber Kloster
Jesper Kristensen
Arne Mejlholm

{jk|cableman|mejlholm}@cs.aau.dk

at

DEPARTMENT OF COMPUTER SCIENCE,
AALBORG UNIVERSITY

JANUARY 2007

Department of Computer Science
Aalborg University

Title:
Determining the use of Interdomain Shareable Pages using Kernel Introspection

Semester:
Dat7
1. September 2006 -
16. January, 2007

Group:
Dat7103a, 2006-07

Members:
Jacob Faber Kloster
Jesper Kristensen
Arne Mejlholm

Supervisor:
Henrik Thostrup Jensen

Copies: 5

Report – pages: 52

Appendix – pages: 30

Total pages: 82

Synopsis:

This report investigates how interdomain shareable pages are used within the guest operating systems in virtualized systems. Specifically we use virtual machine introspection to bridge the semantic gap between the virtual machine monitor and the guest operating systems in regards to the use of shareable pages.

We find that the disk cache in the Linux kernel is generally responsible for most of the sharing. The files in the cache are, amongst others, standard C libraries and application binaries. Furthermore a significant amount of sharing is due to pages used by the kernel, e.g. its slab caches. These things leads us to believe that the more homogeneity in the virtualized system setup, the more redundancy in memory and thus the more memory can be reclaimed.

The implementation used to carry out the analysis is a driver for the Xen modified Linux kernel running on the Xen VMM patched with CBPS and Potemkin.

Keywords: VM, VMM, hypervisor, Xen, Potemkin, virtualization, paravirtualization, memory sharing, content-based page sharing, reverse mapping, Linux kernel, introspection, semantical gap, page cache, mapped pages, anonymous pages.

Preface

This report documents our work about the examination of what redundant pages in virtualized systems are used for. We present an approach of how to retrieve enough information to determine this, build an implementation to do it, and use this to evaluate a number of workloads. The report has been developed as part of the DAT7 semester at the Department of Computer Science at Aalborg University.

The reader is assumed to have some knowledge about operating system principles, virtualization in general and our previous work[20, 21].

Jacob Faber Kloster

Jesper Kristensen

Arne Mejlholm

Contents

Chapter 1

Introduction

The report starts with an introduction to virtualization for the readers convenience and to introduce the terminology needed to motivate and state the goal of the project. In order to motivate the project we then present some basic experiments on memory sharing in virtualized systems. Having presented these, we then state the goal of the project and argue that this is worthwhile.

1.1 Introduction to Virtualization

Virtualization[16, 29, 30] has gained popularity in the recent years, but it is by no means a new concept. It dates back to the 1960ies, where it was used to provide concurrency. The recent popularity is amongst other factors due to the fact that modern computers are often underutilized and sufficiently powerful to run several workloads.

The benefits of virtualization are plentiful. Examples include server consolidation to save power and hardware costs, sandboxing of OSes to achieve fault containment, rapid deployment of servers, providing a better platform for OS development and many more.

A virtualized system is based upon two central concepts: A number of *Virtual Machines* (VMs) and a *Virtual Machine Monitor* (VMM) running on a physical machine. A VM is a software abstraction of the physically available resources, i.e. the physical machine is partitioned into a number of abstractions. Each of these abstractions are able to execute an OS and a number of applications. The architecture of a typical virtualized system is illustrated in Figure 1.1 on the following page. To ensure that the VMs are only accessing the hardware resources that have been allocated for them, the VMM serves as a mediator between VMs and the hardware. The VMM must be able to preserve isolation between the VMs in order to ensure the safety of the whole system. Therefore it is crucial that the VMM is kept simple in terms of complexity in order to make it probable that it is correct.

The primary cost of virtualization is reduced performance, but there is also an increased memory usage for the VMM and VM metadata. These costs are however minor taking the plentiful computing resources of todays hardware into consideration.

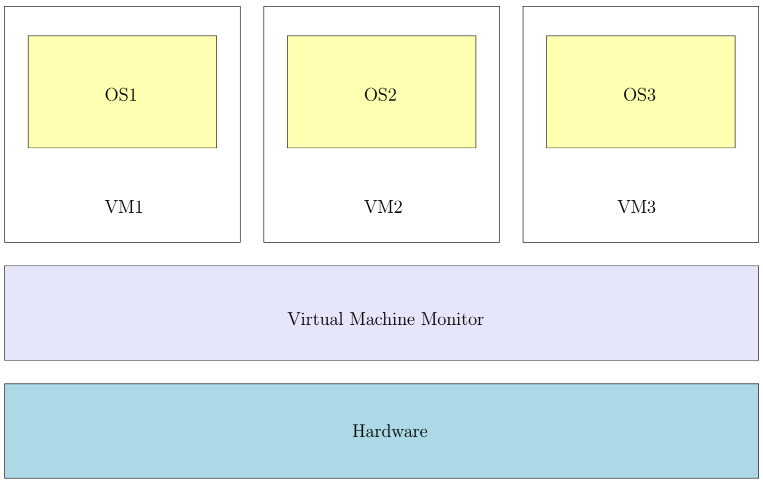


Figure 1.1: A traditional VMM architecture. The VMM arbitrates access to the hardware.

1.1.1 Approaches to Virtualization

The original approach to virtualization was to use *full virtualization*, which is to let the kernel run at a less privileged level than it is written for. Thus privileged instructions are not allowed to be executed and a fault is triggered. With the fault, also known as a *trap*, the VMM is reactivated and can handle the instruction, either by executing it at the sufficient level of privilege or by ignoring it. The definition of virtualization has three characteristics: 1) the environment must be essentially identical to the hardware, 2) there must only be a slight decrease in performance and 3) the VMM must control all physical resources. Thus virtualization differ from simulation or emulation, by a performance requirement, i.e. it must be ensured that as many processor instructions as possible are run natively on the processor [27, 18, 28, 2].

In order to use this approach it is necessary to have a processor that does not use *sensitive instructions*[27], i.e. instructions that fail silently when executed without sufficient privilege. Very few processors are able do this, including most processors of the 32 bit Intel Architecture (IA-32), so other approaches have been invented in order to remedy this. The current trend is to create processors that handle sensitive instructions and have virtualization assisting components, such as nested/extended page tables[3, 26]. Both AMD and Intel have launched different IA-32 compatible processors that support full virtualization, which are commonly referred to as Hardware VMs (HVM)[34]. Other approaches include *paravirtualization*[37], *paenevirtualization*[31], *previrtualization*[22] and *binary dynamic translation*[14]. Each of the approaches have their respective strengths and weaknesses, but all enable the concurrent execution of more OSes. A further description is however beyond the scope of this report.

1.1.2 Paravirtualization and Xen

Paravirtualization is an attempt to mitigate the cost of full virtualization by modifying the VM abstraction. Instead of being identical to the underlying architecture, the abstraction is only similar to the hardware. Sensitive instruction are avoided by modifying the kernel to make explicit calls that relinquish the control of the processor to the VMM, which can then emulate or execute the sensitive instructions. The drawbacks of paravirtualization are mainly the loss of compatibility for proprietary and legacy operating systems, as well as the cost of having to develop and maintain modified kernels, but performance that is superior to other approaches can be gained[4].

Xen is an open-source VMM that uses paravirtualization. When dealing with Xen it is worth noticing that the VMM is often addressed as the *hypervisor*. This term refers to the usual naming scheme used for the kernels privilege level, normally called *supervisor mode*. Because the kernel is running in this mode, a new term is needed for a VMM running in the most privileged mode, namely hypervisor mode. Features of Xen include live migration[9] and updates of page tables through either hypercalls or by the use of notoriously expensive shadow page tables. A patch to the Xen VMM was provided by the Potemkin[35] project, which introduced the possibility to clone VMs.

The VMM communicates with VMs using asynchronous *events* (for example to notify about network traffic) and the VMs use synchronous *hypercalls* to relinquish the processor and request modification of privileged state.

1.1.3 Removing Redundancy

What is of interest to us is that server consolidation introduces sources of redundancy. In particular we are interested in the phenomenon where there are two pages in memory that are identical. Usually the two pages belong to two distinct VMs, but a number of identical pages can usually also be found within a single OS. This redundancy is not uncommon in traditional server rooms, as many organizations prefer to apply the same OS and often run the same basic services. The more homogeneity between the services running in the different VMs, the greater the probability of redundancy. Sets of identical pages can be reduced to one shared copy using standard Copy-on-Write techniques, thus freeing a number of pages for the benefit of the whole virtualized system.

Several approaches has been introduced to do this. As explained before the Potemkin patch enables Xen to clone VMs. One of the goals of the Potemkin project was to ensure maximum scalability in terms of memory usage. Therefore they had to keep the memory footprint of a VM as small as possible. The solution was *delta virtualization*, based on the concept that a cloned VM only takes as much memory as is needed to represent the way it differs from the VM it was cloned from. Another promising project that addresses redundant memory contents is XenFS[38], which is an interdomain shared cache. Whenever a VM needs data from a block device, a request is issued to XenFS to bring that data into memory. If the data is already in memory, there is no need to read from disk, and a reference to the page, that is already in memory, is returned. Pages that are backed by block devices usually take up the majority of memory on many workloads, because all files read from disk are put into memory by the disk cache to maximize utilization of main memory. This is relatively inexpensive as the files often do not require synchronization with disk storage, before they are evicted in order to free up memory. So the XenFS approach should be able to reduce the total memory footprint of a virtualized system. This approach is also used in Disco[7]. VMware has introduced content-based page sharing[36], which compares pages in memory with each other to identify identical pages.

Previously we implemented a patch that enables Xen to do content-based page sharing. As this approach is the overall topic of this report, we shall explain this in more details.

Content-based page sharing essentially just compares every page in memory with all the others. As this obviously is inefficient, a hash value is produced from the contents of each page. This is then inserted into a hash table along with the machine address of the page. If at any time two hash values are identical and the machine addresses are different, there is a good probability that the pages are identical and thus are candidates for sharing. The candidate pages must still be compared bitwise, as there may have been an arbitrarily long period of time between the hashing of the two pages. Thus the pages may have changed since the pages were first hashed.

1.2 Motivation and Goal

As stated in the last section, we implemented content-based page sharing for Xen. To evaluate the implementation we carried out a number of experiments to investigate how much memory can be shared between VMs. We found that with identical OSes running in the VMs, most workloads have a minimal level of pages to share. We accredited this amount of shared pages to the Xen modified kernel running in each VM. If the VMs were running the same basic services, then the amount of shared pages would rise. So we assumed that the pages shared due to this were the pages used to contain the binaries of the services running. Finally some workloads exhibited a high level of sharing, e.g. when letting two VMs compile the same kernel at the same time. We speculated that this was mainly due to application binaries, i.e. compiler tools during kernel compilation. Furthermore we also suggested that some shared pages were due to the data kept in memory by the applications.

The Xen VMM generally is not aware of any data structures used by the modified VM kernels, except for a few things that are necessary to do paravirtualization, such as pages used for page tables. This lack of knowledge about VM runtime information is also referred to as a semantic gap[8, 19]. While the assumptions and suggestions above may very well be true, we could not tell for sure, as the VMM does not have sufficient access to the needed information. Determining what the pages were actually used for should prove rather interesting, so we can asses if we were correct in our assumptions. If, on the other hand, we were not correct, then determining what the pages were really used for should prove even more interesting. Furthermore to our knowledge no results about this have been presented to this field of research, so such an investigation should have some scientific research value.

In the next subsection we present results from new experiments. We present these for a number of purposes: 1) to demonstrate that sharing pages is possible, 2) to show the amount of pages that can be shared and 3) provide a basis for further experiments. We will return to these initial results later in the report as we investigate what the shared pages are used for.

1.2.1 Initial Experiments

The results presented here are new, but are conceptually identical to the results we presented in [20]. We have chosen to present these as they explain many of the observations from the thesis, but in a more concise manner.

The experiments were carried out on a number of VMs running on our implementation and with page sharing enabled. The VMs ran Debian on a modified 2.6.16 Xen kernel and with a bare minimum of system services, such as cron. Each VM had a memory allocation of 128 MB and used shadow page tables. For a more detailed description of the actual machine setup and benchmark descriptions see Appendix A.

The first experiment, as pictured in Figure 1.2 on the next page, illustrates the implementations ability to share pages. We have chosen to let the VMs run kernel compiles on identical setups in order to ensure that we get a high page sharing percentage. The experiment represents a best case workload, as the VMs are running the same OS with the same kernel and the same workload. Once the kernel compile is done, the memory of the VMs should be in approximately the same state, thus ensuring redundancy. In the experiment we incrementally started one VM at a time, ran a kernel compile on the VM and left it idle afterward. After 30 minutes we measured the amount of shared pages in the whole system and then started another VM.

The metrics used in Figure 1.2 have previously proven to be the source of some confusion, so an elaboration is in order:

VMs Mem Total: is the amount of pages allocated for the VMs without any shared pages. This is also the upper limit to the number of pages that can be shared in the system.

Shared: is the amount of pages that are set up as shared pages at the moment.

Reclaimed: is the amount of pages that have been reclaimed from the VMs due to sharing.

Zero Pages: are the pages consisting of only zeros; often these have not been used by the guest OS.

Shared - Reclaimed: reflects the amount of pages used for shared copies.

Returning to the actual results presented in Figure 1.2, the first observation is that of all the shared pages, there is a very little amount of zero pages. As more VMs are started, the amount of shared and reclaimed pages increase almost linearly, indicating that the VMs are sharing approximately the same sets of pages. Generally 44% of all the memory allocated to the VMs can be shared. Having illustrated the implementations ability to share pages on an artificial workload, we then turn to a

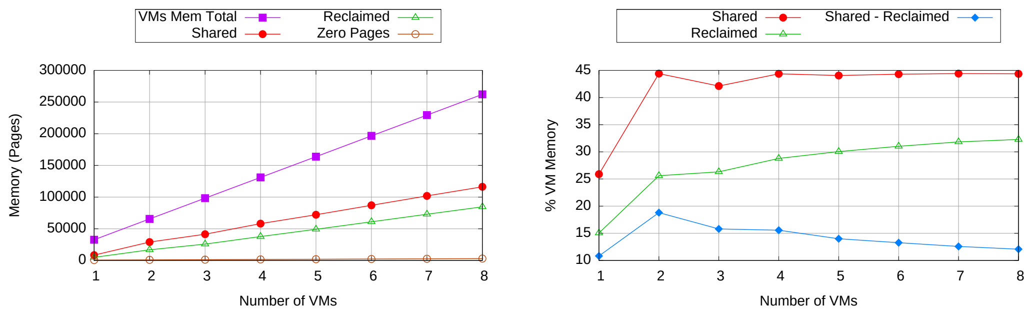


Figure 1.2: On the left the absolute graph and on the right the aggregate graphs for eight incrementally started VMs, each having performed one kernel compile. The share percentage is high, while there is a very low amount of zero pages.

set of less artificial workloads. These are however still synthetic workloads with VMs running different benchmarks, as we do not have access to real world workloads.

In the second experiment, we examined the level of page sharing, as it changes over time, on five different workloads. During each of the workloads, we start four VMs concurrently, let them run idle for a few minutes and then start the benchmarks. The five workloads are, as shown in Figure 1.3 on the facing page, Dbench, Kernel compile, OSDB, SETI and Mixed. In the first four workloads the four VMs are running the same benchmarks simultaneously, while the VMs are running one of the different benchmarks in the Mixed workload.

Except for the kernel compile, the benchmarks all run for approximately 80 minutes. As can be seen from the figure, the kernel compile stops after approximately 42 minutes. It should be noted that during the benchmarks, the VMs are subjected to very heavy loads. As the implementation only scans for eligible shares when the VMM is running its idle loop, it is likely that a lot of opportunities for sharing pages are missed at this point. This is especially the case on the non-disk intensive benchmarks, as reading files from disk often is the cause of the VMM running the idle loop on heavy loads. This explains why we see such a dramatic rise in the number of shared pages right after the kernel compile is done, as the VMM gets sufficient time to analyse all pages. Furthermore this is the only benchmark that does not run with distinct data sets (the source code being the data set in this case).

Starting with the IO-intensive Dbench benchmark, we see that the amount of shares roughly follows a straight line at 4%, but often deviates from this by 1%. This indicates that the memory pressure inside the VM is intensive, meaning that many shareable pages are brought into memory, used briefly, and then discarded. We expect that these shared pages are mostly in-memory representations of the files used for the benchmark.

As expected during the kernel compile, some shareable pages are discovered, but it is not until the compilation is completed that the number of shareable pages sky-rockets. The shared pages here should be due to the same tool binaries being present in memory in the different VMs and perhaps even some pages used to hold compilation metadata during the kernel compilation process.

The two first benchmarks behave pretty much as expected, but we see a more

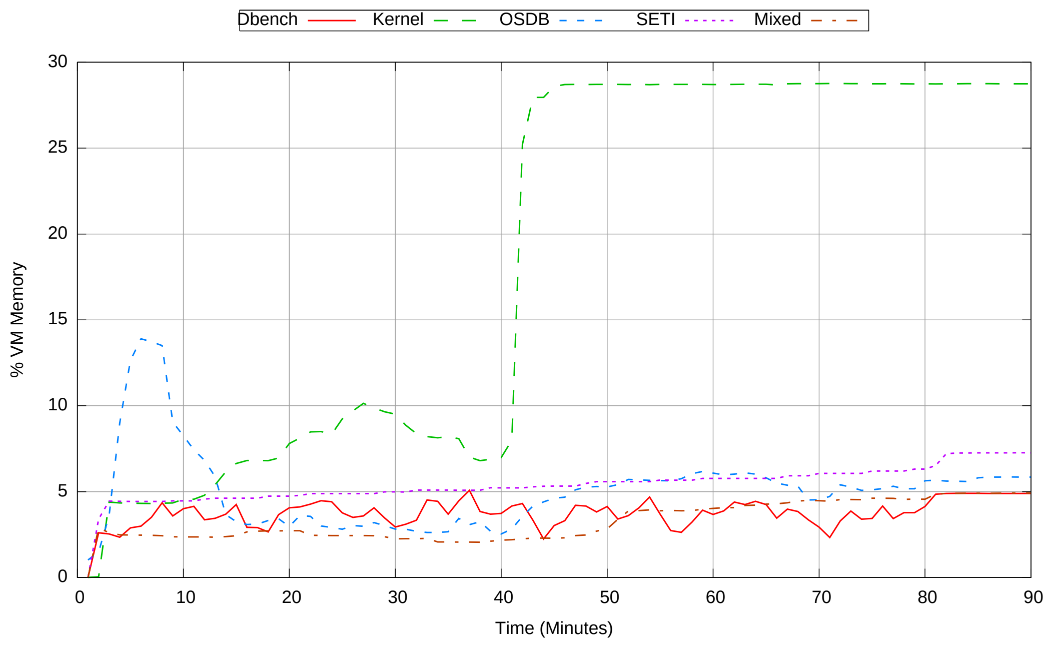


Figure 1.3: The number of reclaimed pages from four concurrently running VMs during various workloads.

interesting occurrence during the OSDB benchmark. It starts by rising to almost 14% and then dropping to $4-5\%$ after some time. As the VMs in this benchmark, in contrast to the other benchmarks, also are running the MySQL daemon, we accredit this initial high level of shared pages to the MySQL and OSDB binaries.

SETI on the other hand is mostly processor intensive without much disk interactions. Interestingly this implies that perhaps the pages shared during this benchmark are mostly due to in memory data used by the SETI application. Furthermore the amount of shared pages increases steadily during this benchmark and reaches a higher level of sharing than most of the other benchmarks, which seems odd due to the nature of the application.

The Mixed workload behaves pretty much as expected. The amount of shared pages is low and seems to increase slightly. These shared pages should be due to the kernel binary and perhaps some OS services.

To summarize, all of these workloads have interesting characteristics, both in terms of the amount of reclaimed pages and how these are used, but also how this changes as the respective benchmarks progress. We remind the reader that the considerations presented during this section are mostly speculations. Investigating whether they are true, is not only interesting, but may also result in possible optimization for the implementation. We therefore deem that it is feasible to proceed with a further investigation of what the pages are used for.

1.2.2 Project Goal

Put formally, the goal of this project is *to further investigate, by empirical means, the usage of pages that are eligible for sharing during various workloads in the virtualized system.*

This goal presents a number of challenges: 1) The semantic gap between the VMM that finds the shareable pages and the VMs that are using the pages. 2) Understanding the use of the pages requires a thorough understanding of the OSes used in order to find the relevant information.

To solve the first challenge, we turn to VM introspection[15] through the kernel to determine what the shared pages are used for. We can make an extension to the paravirtualized driver we are also working on. By modifying this driver we can exchange information between any given VM and the VMM, thus virtually giving us access to all the data structures of the kernel. The second challenge can be solved by persistent work to understand the kernel, given that the source code is available and well-documented.

1.2.3 Categorising Pages

To determine the usage of the interdomain shareable pages we need a basis for the analysis. This must present the results in a way that is easy to inspect. Therefore we categorize the pages in five mutually exclusive categories, which roughly corresponds to how the Linux kernel regards the pages. The categories are as follows:

Inode pages: a page containing data from disk, which is mapped into the virtual address space of one or more processes. These pages would often be used for long running processes, such as system services. Since such services are running on most VMs, this seems probable to cause some sharing.

Anonymous pages: mapped into the virtual address space of one or more processes, but the contents of the pages are not backed on storage. This might be pages that a given process has allocated to contain temporary generated data e.g allocated by calling the `malloc` system call. The probability of finding such pages should be low, but we cannot rule out that some VMs may have identically aligned data, especially when taking into consideration the determinism of memory allocations from many applications.

Cache pages: these pages are only used by the cache, i.e. not currently used by any processes. We suspect that a large amount of the shared pages are due to disk cache.

Kernel pages: only used by the kernel. When running a set of VMs with the same kernels it is naturally to assumed that there might be an amount of redundancy in the pages used by the kernels. This could be the kernel binaries or internal data structures.

Free pages: these pages are marked as free by the guest OS. When pages are freed they are left as is with their old contents. Thus it is probable that we find some pages that have recently been used for any of the previous categories.

Where it is interesting and feasible, we will further explore details about the pages. For example in the case of inode pages and the disk cache, we are interested in what these page represents on the disk.

Furthermore if it is possible, we would like to be able to conclude whether an interdomain shared cache would be just as effective as the full implementation of our paravirtualized design.

1.2.4 Limitations

It may be interesting to examine workloads on OSes other than Linux, but we chose to focus on the Linux kernel as this is the OS best supported by Xen. Furthermore we have previously invested some work into the paravirtualized driver in Linux, so this might as well be reused. Furthermore we will be implementing the extension for Xen 3.0.2 patched with Potemkin and our own CBPS patch.

1.3 Summary

In this chapter the field of virtualization was introduced. We discussed what virtualization has been used for in the past and the new trends that are emerging in the field. The Xen virtual machine monitor has been described and a set of initial experiments has been conducted to show the results from our earlier work, which we have as the motivation for this project. The report will investigate what shareable pages are typically used for in virtual machines. This is non-trivial as the virtual machine monitor does not have direct access to the runtime information of the kernel. This obstacle is also known as the semantic gap. In order to overcome this we will use virtual machine introspection to further analyse the shareable pages from within the virtual machine owning a given shareable page. We found that the pages should be divided into the following five mutually exclusive categories: Inode pages, Anonymous pages, Cache pages, Kernel pages, and Free pages.

In order to actually do this, we need a thorough understanding of the virtual memory subsystem of the Linux kernel, so this will be explored in the following chapter.

Chapter 2

Virtual Memory Related Data Structures in the Linux Kernel

Understanding how the shareable pages are used by the Linux kernel requires an intimate knowledge of two central data structures in the virtual memory subsystem. The first is the page cache, the primary disk cache that contains the in-memory representations of files on storage. The second data structure is reverse mapping, an efficient way of finding all page tables that are using a given page. These two structures contain radix trees and priority search trees respectively, so we discuss these as well.

We advise the reader that this chapter regards only topics within the Linux kernel, not the modified Xen kernel. Any term that may resemble a virtualization term should always be understood as a standard operating system term. An example of this is a shared page. In virtualization terminology we are speaking about interdomain shared pages, i.e. pages that are shared between VMs. In standard OS terminology we are speaking about pages shared between processes in a single OS. So in this chapter we are talking about the latter (in contrast to the rest of the report).

The chapter assumes that the reader is familiar with common virtual memory concepts such as page tables, page descriptors and memory descriptors in the Linux kernel. Any decent book on the Linux kernel should cover these in much more details than needed here, so the reader is advised to consult such a book or the source code itself.

The terminology used to describe virtual memory is regrettably not consistent, so before discussing topics of the rest of the chapter, we start by defining ambiguous terms.

We use the terminology of [17], where a page table in the Linux kernel is a 4-level structure, that consists of a Page Global Directory (PGD), a Page Upper Directory (PUD), a Page Middle Directory (PMD), and a Page Table Entry (PTE) table. A PGD entry points to a PUD entry, which again points to a PMD entry and so on. An entry in the PTE table is referred to as a PTE entry and this points points to a page frame.

A Virtual Memory Area (VMA) is a data structure that describes a contiguous area in virtual memory. A process typically has a number of these, e.g. one for each range of pages mapped into the virtual address space of the process.

2.1 The Linux Page Cache

The section and its subsections is based on [23, cha. 15] and [6, cha. 15]. The use of caches is common in OSes, because it is a lot faster to access memory than physical storage and the speedup gained by caches are significant. This is essentially a trade-off between memory usage and storage access time.

The Linux kernel maintains a set of software managed caches. These are used to store information about Operating System (OS) metadata, such as filesystem information. An example of such a cache is the `dentry` cache, which is used to cache directory lookups and thereby speed up the translation of pathnames to locations on disk.

Another example of caches in the Linux kernel is the slab object caches. Objects cached in these are commonly used data structures such as `dentry_cache` and `vm_area_struct` objects. There exists a slab cache for each of these types of data structures. Object caching is a technique that deals with objects that are frequently allocated and freed again. The overhead incurred by the initialization process of these kernel objects are significantly minimized by the slab caches. [5]

The speedup gained by caches relies on a single entry in the cache being accessed multiple times. If not, the effort of caching data would be wasted. This is known as the principle of temporal locality, which states that: A resource accessed at one point in time, will with high probability be accessed again in the near future.

The primary cache in the Linux kernel is the *page cache*. The page cache is used to minimize access times when performing read or write operations on a block device[1]. When data is read from or written to a block device, it is passed through the page cache. If it is not already in the page cache, a new page is allocated and the data is cached. Read and write operations are handled differently: Write operations are deferred for a period of time, while read operations are performed at once. By deferring the write operation there is a chance that future write operations will obsolete the first operation, thus avoiding expensive disk access.

When a page has been allocated for the page cache, it stays in there until the system reaches a certain threshold and starts reclaiming memory. It is identified by an owner that is an `inode` object, which is an in-memory representation of filesystem metadata. It should however be noted that there is a little twist with this identification method, because two entries in the page cache can contain the same device blocks, e.g. the same device blocks can be added to the cache by reading a file from the filesystem and by reading it from the device file. The page cache is built around a set of different data structures, which are brought together by the `address_space` object.

2.1.1 The Address Space Object

When a file is read into the page cache, a new address space is created for the file. The `address_space` object is the main data structure of the page cache, but it is also

[1]A block device is a device that transfers data in fixed sized blocks.

used in other parts of the kernel e.g. it contains metadata about memory mapped files and is used in reverse mapping.

Figure 2.1 illustrates the interesting parts of the page cache and the relation between these. As it can be seen on the figure, the `address_space` object can be accessed in two ways: 1) Either through the owners (inode) `i_mapping` field or 2) by following the `mapping` pointer from a page descriptor (page struct). Furthermore there is a pointer back to the owner of the `address_space` object, which makes it possible to find the file that the address space represents.

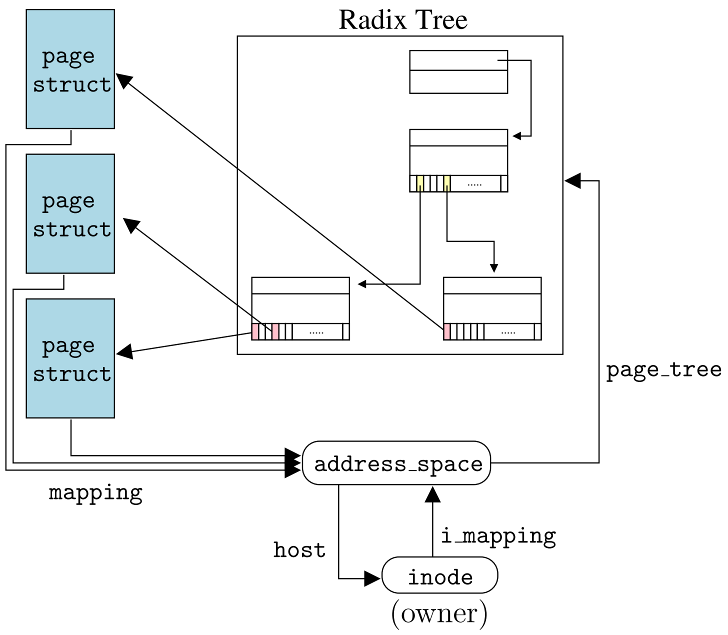


Figure 2.1: The key data structures that constitute the page cache. The main data structure is the `address_space` object, which links the different parts of the cache together.

The `page_tree` field on the `address_space` object contains a pointer to a search tree, a variant of the radix tree data structure, which is used to efficiently organize the page in the page cache.

2.1.2 The Linux Kernel Radix Tree

The memory used for the page cache will increase significantly when accessing large amounts of files from storage. So an efficient data structure is needed to search for a page in the cache. The Linux kernel uses a modified radix tree to resolve this problem.

When searching for a page in the page cache, one first needs to have the right address space. This is obtained by following the `mapping` field on the page descriptor to the `address_space` or the `i_mapping` field on the `inode` object. When the address space has been located the radix tree can be found by following the `page_tree` field on the `address_space` object.

A key is needed to search the tree to determine if a given page is in the page cache. The `index` field on the page descriptor is an offset into the set of disk blocks that represents a given file. This offset can be used as a search key on the tree.

Figure 2.2, is a two level radix tree that point to five page descriptors. The root of the radix tree, `radix_tree_root`, has three fields 1) `height` containing the height of the tree, 2) `gfp_mask` which is used when creating a new node, and 3) `rnode`, which is a pointer to the first node or level in the tree. The tree is structured as in the figure, where nodes points to other nodes on the next level or to leaves, which are page descriptors. The nodes in the radix tree has 64 pointers to other nodes or to leaves. A node, `radix_tree_node`, in the radix tree has three fields: 1) `slots` for the 64 pointers, 2) `count` of how many slots that are not null, and 3) `tags` which is used to indicate that one or more of the child nodes contains either dirty pages or pages that are in the process of being written back to disk.

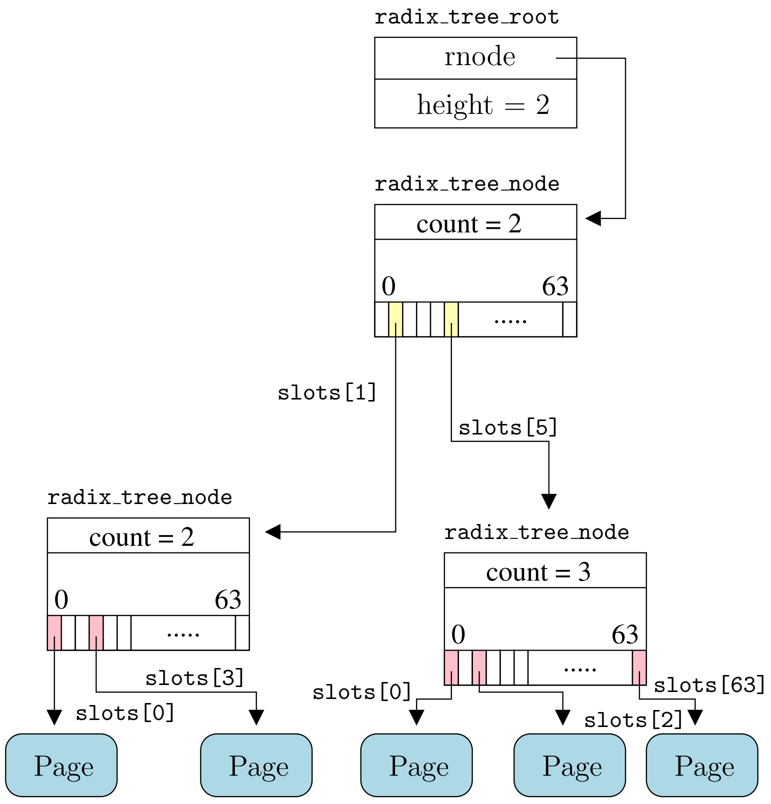


Figure 2.2: A Linux kernel radix tree with two levels plus the root node.

2.1.3 Searching in the Radix Tree

The height of a radix tree in the Linux kernel is restricted to 6 levels on a 32 bit architecture and thus the file size that the cache can handle is also restricted. To understand this restriction, one has to look at Table 2.1 on the next page and on how lookups are performed in the radix tree.

When performing a lookup in a Linux kernel radix tree, the 32 bit key, retrieved from the `index` field on the page descriptor, and the height of the tree is used. A

Height	**Number of indices**	**Maximum file size**
1	$2^6 = 64$	265 Kilobytes
2	$2^{12} = 4.096$	16 Megabytes
3	$2^{18} = 262.144$	1 Gigabytes
4	$2^{24} = 16.777.216$	64 Gigabytes
5	$2^{30} = 1.073.741.824$	4 Terabytes
6	$2^{32} = 4.294.967.296$	16 Terabytes

Table 2.1: The number of indices at different levels in a radix tree and the maximum file size at these levels. The file size is calculated by multiplying the number of indices with 4 KB, as each can point to a 4 KB page. The table is based on Table 15-3 in [6, p. 606].

search or lookup in the tree is performed in two different ways depending on the height of the tree.

1. If the tree has height 1, it can contain a maximum of 64 entries and these can be indexed with 6 bits ($2^6 = 64$). So when the height is 1, we simply use the 6 less significant bits of the key as an index into the `slots` array on the first node. If the tree has more than 1 level, another 6 bits are used at each level in the tree. Starting with the 6 less significant at level 1 and the next 6 less significant bits on the next level until one reaches the leaves.

2. If the tree has 6 levels and we used the same technique as before it would require a 36 bit key. To resolve this problem, the 2 most significant bits are used as an index into the first level `slots` array instead of 6 bits, which means that it can have only four entries ($2^2 = 4$). The next 6 bits of the key are then used at level 2 in the tree and so on until level 6. At the last level the last 6 and less significant bits are used.

The tree is restricted to only 6 levels, because the 32 bit key can only represent 2^{32} different keys. So further modifications to the tree would not make it able to represent large files. But on the other hand 16 Terabytes should be sufficient for most workloads.

2.2 Priority Search Tree

The Priority Search Tree (PST) data structure was originally used to represent overlapping intervals and was introduced by Edward McCreight in 1985[25]. The data structure is a clever combination of heaps and balanced search trees, which can be queried with sets of intervals. The PSTs are used by Linux to organize memory regions and nonlinear intervals, thus they can be used to find overlapping intervals in VMAs. This section is based on [6, s. 686-689] and `Documentation/prio_tree.txt` in the Linux kernel version 2.6.16.

As with the radix trees in the last section, the PST can be found on the `address_space` object, which means that there can be only one PST for each `address_space` object.

The reader should remember from Section 2.1.1 on page 18, that there exists an `address_space` object for each file that is mapped into memory.

There are some differences between McCreight's PST and the once used in the Linux kernel e.g. the Linux PSTs are not always kept balanced, because the balancing algorithms is costly in memory usage and processor time.

Every node in a PST represents a memory region, which contains pages with data that are read from block devices. So the PST is used to organize VMAs that are file backed i.e. memory mapped files. To identify a node in a PST a three tuple of indices is used. This contains the following: 1) radix index, which represents the beginning of the interval (memory region) that this node covers. 2) size index, which is the size of the interval. 3) heap index, which is the end of the interval. In the actual implementation the size index is calculated on demand ($heap - radix = size$).

Figure 2.3 illustrates a PST that is organized based on the three indices in the tuple. The following rules are used when creating a PST: 1) The heap index on a given node, must not be smaller then the indexes of its children, also known as the heap property. 2) The radix index of the right child is always larger than that of the left child for a given node. 3) If the radix index of two children are equal, then they are indexed based on the size index.

The original data structure by McCreight, does not support intervals starting at the same position (radix index), which memory regions tend to do. So a partial solution to this problem is the size index, which makes it possible to distinguish two nodes having the same radix index. So the tree is extended with overflow subtrees. These are the blue subtrees in the figure and they are indexed by the heap and size index.

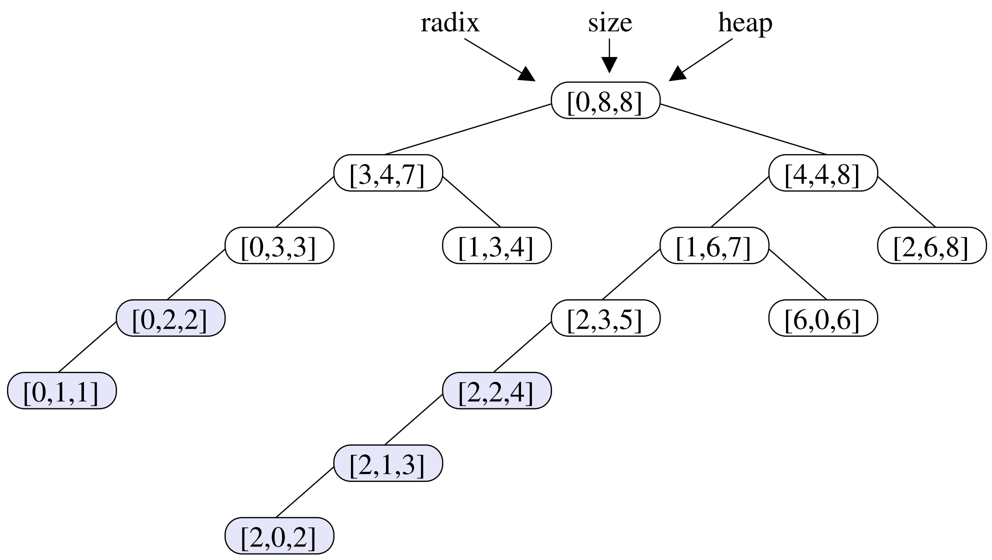


Figure 2.3: An example of a priority search tree as used by the Linux kernel. The white node is index by the three indices: radix, size, and heap. The blue nodes are overflow subtrees, which are indexed by the heap and size index and have the same radix index.

Any process can map in pages that represents the same part of a file. If this occurs, different memory regions will have the same radix, size, and heap indices, which would force the kernel to place nodes at the same position in the tree. To resolve this issue the nodes are extended with a circular doubly linked list, that links the memory regions with the same indices together.

To better understand how the PST works, the next section will explain how a given interval is found in a PST.

2.2.1 Retrieving Information from a PST

We use Figure 2.3 on the facing page as an example of how one can retrieve information about which memory regions a given interval covers. We will in this example use a simple interval that only consists of one page, as this is how reverse mapping uses it. Thus we search for a single page in the all VMAs on the system.

We will be searching for the interval $[7, 0, 7]$, which means that we are searching for the page at offset 7 in the file that this particular PST represents. The following algorithm retrieves the nodes that contains this interval:

1. The first node that is retrieved from the PST is the root node ($[0, 8, 8]$), because the interval $[7, 0, 7]$ is included in the interval.

2. We then take the left child ($[3, 4, 7]$) and compare the heap index (7) with the heap index of the interval (7) that we are searching for. If the heap index is less or equal to the heap index of the interval the node is retrieved. It is, so the node is retrieved.

3. Again the left child ($[0, 3, 3]$) is selected, but this time the heap index (3) is smaller than the offset of the page, so it does not contain the page. We now known from the heap property, that the children of this node do not contain any intervals with the offset of the page. So we jump to the right node ($[1, 3, 4]$) of the parent, which does not contain the interval. The node is childless, so we return to the root node and investigate the right child $[4, 4, 8]$ of the root node. The procedure performed in these steps are then carried out in the same manner for the right child of the root node.

The result of running the algorithm on our example tree, is that we retrieve the nodes with the intervals: $[0, 8, 8], [3, 4, 7], [4, 4, 8], [1, 6, 7], [2, 6, 8]$. The nodes do however not reference any VMAs, instead each VMA has a reference to a node. So one has to find the VMAs, that have a reference to a given node, in order to find the set of VMAs represented by the given node.

2.3 Reverse Mapping

Reverse mapping is a new feature in the 2.6 series of the Linux kernel. The main reason behind this addition, is to efficiently find all page tables that are mapping a given page. This can then be used to alter mappings when a given page is swapped out to disk.

Reverse mapping significantly reduces the time used on finding these mappings compared to the naive approach, which is to search through the page tables of all processes. But there is an overhead both in the memory footprint and the maintenance of the data structures used to create reverse mapping. It is a classic trade-off between memory usage and CPU usage.

This section takes a rather practical approach by reflecting how it is used to find a given page. It is based on [6, p. 680-689] and the following source files: `include/linux/mm.h`, `mm/rmap.c`, and `include/linux/rmap.h` of the Linux kernel version 2.6.16. The Linux kernel contains a relative new feature called reverse mapping[2]. The history on how the reverse mapping became, can be found in our master thesis[21]. For a more theoretic description see [6, p. 680-689], [17, p. 48-50], [24], [10], [11], [13] and [12].

The implementation of reverse mapping that will be discussed in the remainder of this section is called object-based reverse mapping, as it is found in the 2.6.16 Linux kernel. We start by having a page descriptor. In reverse mapping context we are interested in three fields on the page descriptor:

`mapcount`: Is a counter of how many PTE entries points to the page[3]. It can be used to indicate when to stop the search for PTE entries or be used as a sanity check of whether we have found all PTE entries or not.

`mapping`: This field has a double usage: 1) If the least significant bit is set, then the other bits are used to point to an `anon_vma` object, which means that the page is related to an anonymous region. 2) If this bit is not set, then the field points to an `address_space` object, which means that the page is related to a file.

`index`: If the `mapping` field above points to 1) an `address_space` object, then it gives us the position in a file. 2) An `anon_vma` object, then the index gives us a position in the anonymous memory region.

We explain the rest of the reverse mapping process by the use of an example, where we will illustrate the reverse lookup process for a given page. This page can, as described, either be a anonymous or a mapped page. So we describe each case in separate sections.

2.3.1 Reverse Mapping for Anonymous Pages

Remember that anonymous pages are pages that the kernel cannot relate to a file, hence the name anonymous pages. Often anonymous pages only have one virtual mapping, but they can have several mappings. A reason why there are more mappings may be that a process using an anonymous page is forked, so its page table is duplicated for the child process. The consequence of this is that the anonymous page is referenced by two page tables.

Figure 2.4 on the next page illustrates the different parts of the reverse mapping process for anonymous pages.

In this subsection we discuss the page of the example as an anonymous page. The next step is to follow the mapping pointer from the page descriptor to an

[2]Often in the Linux source code and on malling lists called rmap.

[3]Note that mapcount is called `_mapcount` in the actual code and the same goes for page count. These counts are actually, in some versions of the Linux kernel, one below the correct value, but this is encapsulated in macros, so it is no concern to us.

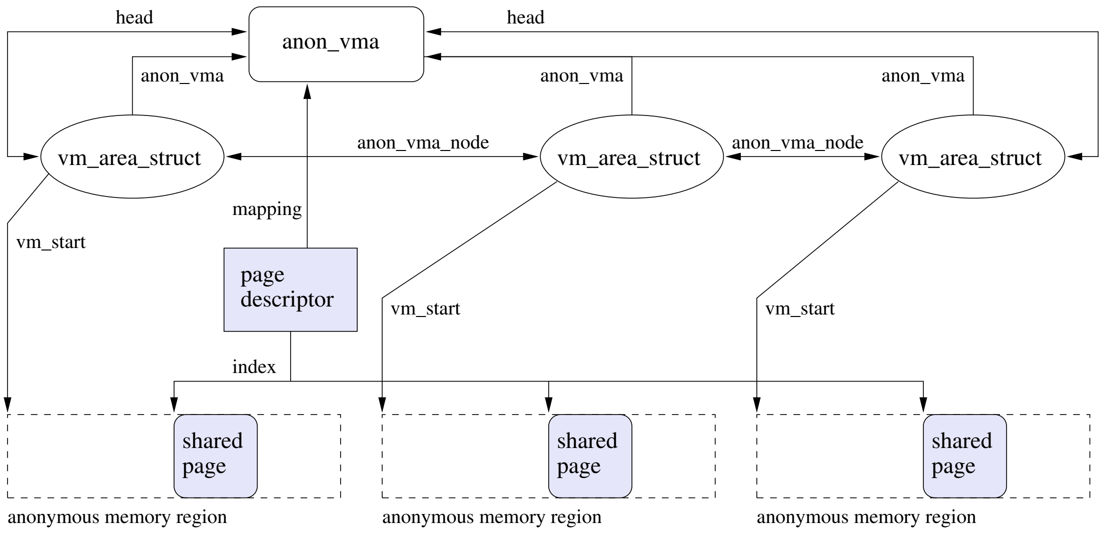


Figure 2.4: Object-based reverse mapping for anonymous pages.

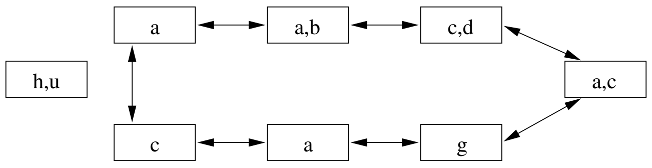


Figure 2.5: An unrelated example of a VMA chain, where the characters represent different pages.

`anon_vma` descriptor, this corresponds to following the arrow in the figure from the page descriptor to `anon_vma`. The `anon_vma` object contains a lock and is part of a doubly linked circular list of the VMAs (`vm_area_struct`) that has the page in common.

This doubly linked circular list is represented by the arrows called `head` and `anon_vma_node` in the figure. Furthermore as seen in the figure each VMA (`vm_area_struct`) has one pointer to the `anon_vma` object. This pointer is used when removing a VMA from the list, since the `anon_vma` object has the `lock` for the list.

An important note here is that there are no guarantees that the page, we are looking for, is part of each VMA in the chain of VMAs. There are two reasons for this: 1) The page is no longer part of the VMA, but the VMA chain might not be updated to reflect this, since lazy updating is used. 2) That the page was never part of the VMA, but another page in the VMA has been or still is related to another VMA in the chain.

To illustrate this, we will now make use of Figure 2.5. It shows a VMA chain where each box is a VMA and the characters inside each box represents pages, meaning two different characters represent different pages. We will now give an explanation of why the different VMAs in the figure are in the chain and why not:

(a), (a) These are in the chain since they contain the page **a**, which other VMAs also do. This is an example of reason number 2.

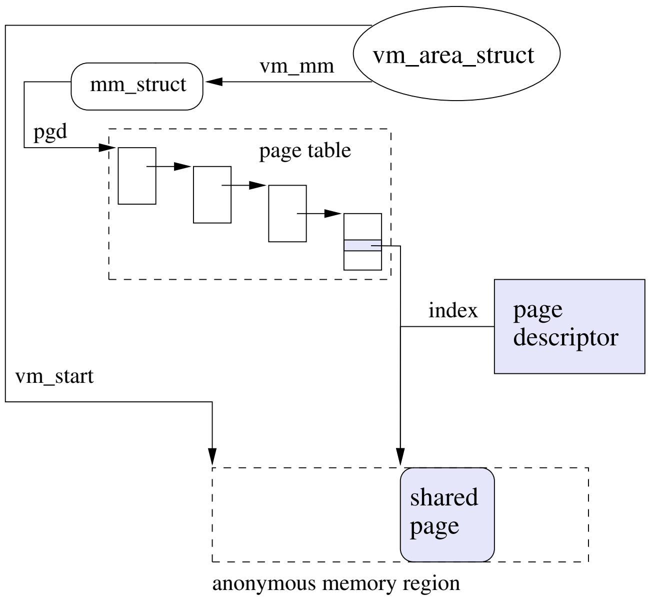


Figure 2.6: Focus on a single memory region object and the page table lookup in object-based reverse mapping.

(a,b), (a,c) These both contain **a** so reason number 2 again.

(c,d), (c) Contains **c**, which relates it to **(a,c)** so again reason number 2.

(g) This VMA has contained **b**, but now it does not. So it should be removed from the chain, since it is no longer related with the other VMAs. This however has not happened yet and this is allowed since lazy updating is used. Therefore this is an example of reason number 1.

(h,u) This is not in the chain since it has no pages in common with the other VMAs.

Returning to the example, we now have a `anon_vma` object, from which we can start traversing the list of VMAs. For each of the VMAs in the list, we perform a lookup to see if the VMA contains the page we are searching for. As mentioned before the `index` field tells us where in the VMA to find the page and the `vm_start` holds the address of where in the virtual memory of the process this anonymous memory region is located.

We add `vm_start` from the `vm_area_struct` to the `index` of the page descriptor. The sum of these two gives us the virtual address in which we should be able to find the page if it is mapped in by this process.

The process of finding a virtual address is illustrated in Figure 2.6. Where the `vm_start` and `index` variables are showed as arrows into the anonymous memory region.

With the virtual address, which we have just calculated, we can now make a lookup in the page table of that process. This is done by following the `vm_mm` pointer from the `vm_area_struct`, which again gives us a pointer to the page table (page global directory (pgd)). As we traverse the page table of the process, we either end up with a PTE entry or encounter a “not present” entry. If we encountered a not present entry, it can be concluded that the shared page was not part of this VMA. The same can be concluded if we reach a PTE entry and it does not contain the address of our shared page. Otherwise if we reached the PTE entry and it contained the address of the shared page, then we have found one of the PTE entries we are looking for.

What has happened so far will now be summarized both to make the pattern clear and to indicate which part can be repeated on the next VMA in the chain. We started with a shared page, which we determined was an anonymous page. As a consequence of it being an anonymous page the `mapping` field pointed to an `anon_vma` object, which is the head of the circular doubly linked list of VMAs that might have pages in common. By following the next pointer on the `anon_vma` we got to the first element in the list.

We then used information from both the page descriptor and the `vm_area_struct` to give us a virtual address. From this we were able to make a lookup in the page table of the process, to see if it contains the shared page. The page table we were able to find by using the `vm_mm` field on the memory region object, which led us to the memory descriptor containing a pointer to the page table of the process.

The last paragraph is what we can simply repeat on the other VMAs in the chain. We proceed in this manner until we have reached the number of PTE entries that we know references the shared page from the `mapcount` variable as explained in the start of this section.

This concludes how reverse mapping works on anonymous pages. We now proceed with the other case where the page was a mapped page.

2.3.2 Reverse Mapping for Mapped Pages

Remember that a mapped page is a page that contains part of a file. In this context, shared pages are quite common. An example of this is the standard C library, which many processes use. This of course only applies if the applications are dynamically linked, since if statically linked, the binary itself will contain the library needed and it will not result in shared pages.

Since sharing of pages is common on mapped pages, there is an efficient method of finding the VMAs, that might contain a given page. It has already been described in section 2.2 on page 21 and is called a PST. Its primary feature is its ability to give us each VMA that has mapped in a specified part of a file.

We now return to the example, but this time we have a mapped page. To use the PST we need two basic pieces of information:

1. The root of the tree. This is found by following the `mapping` field on the page descriptor, which leads us to an address space object. Here we have a field called `i_mmap` which points to root of the PST.

2. An interval for searching in the PST. In our example where we only want to know which VMAs reference a given page we can make use of the `index` variable on the page descriptor.

As mentioned before, the `index` variable gives us the position within a file. The `index` is used as both the beginning and the end of the interval, since we are searching for a single page. We can now make use of the PST with these two pieces of information. The end result is a set of VMAs, which contain the page of our interest. These VMAs can then be further investigated to see if they contain the shared page.

The process of investigating each VMA to see if it contains the shared page, is similar to what we did with the anonymous pages. With one important exception being that anonymous pages would just use the `index`, from the page descriptor, and the `vm_start`, from the VMA object, to get the virtual address as described in the previous section. But since when dealing with a mapped page, the `index` field contains the position of the page in a file, not within a VMA. So to get the virtual address of a mapped page, we need a third variable in the calculation of the virtual address. The extra variable is `vm_pgoff`, which can be found on the VMA object. It tells us the offset of the VMA object within the file that the VMA is mapping. Without it we might get an incorrect virtual address, since the VMA might not start from the start of the file, hence giving us a wrong offset into the VMA. The calculation to get the address then becomes: `vm_start` + (`index - vm_pgoff`).

With this virtual address we can proceed as with the anonymous page, by doing a lookup in the page table of the process owning the VMA to see if it contains the shared page. This is done on all the VMAs returned from the lookup in the PST. When this is completed the number of found PTE entries should correspond to `mapcount`.

2.4 Summary

In this chapter we explained the two essential data structures in the virtual memory subsystem of the Linux kernel. These structures include: 1) The page cache and its role in the way the kernel handles read and write operations to block devices. 2) The radix tree data structure as it is implemented in the kernel. 3) Priority search trees as used by reverse mapping. 4) Finally there was an in-depth description of reverse mapping for anonymous and mapped pages.

These are necessary to understand in order to build an implementation of virtual machine introspection that is able to analyse shareable pages.

Chapter 3

Design and Implementation

Having found and presented the data structures in the Linux kernel that are important to us, we are now able to begin constructing an implementation to fulfill the project goal. The first obstacle is to overcome the semantic gap by enabling the VMM to send the addresses of the shareable pages to the VMs owning the pages. The second obstacle is to select the right pieces of information to accurately place any given page within the five categories presented in Chapter 1. With the information presented in the previous chapter, this is possible.

Where it is possible we enable the implementation to investigate the usage of the shareable pages in further details. The implementation described in this chapter is available from our homepage[1].

We start the chapter with a quick presentation of how the existing platform works, then explain how we extend this to perform the actual analysis and finally evaluate whether this implementation is sufficient.

3.1 The Existing Platform

The existing platform consists of Xen version 3.0.2 patched with the Potemkin and our Content Based Page Sharing (CBPS) patch. Our patch provides us with a large part of what is needed to start analysing the pages.

To understand what the existing platform provides us with, we will provide an overview of it. We use Figure 3.1 on the next page to describe the set of actions that are involved in analysing pages; these actions are the numbered arrows in the figure.

1. The page hashing component identifies identical pages by the fact that they produce the same hash value, hence there is are high probability that the pages are identical. The addresses of the identical pages are then sent to the reference manager component.

2. Pages found in the last step are compared bit by bit, to see if they in fact are identical. If so, it is determined which VM each page belongs to and the machine address of the page is put into a buffer that is shared between the VM owning the page and the VMM.

[1]`http://www.cs.aau.dk/~mejlholm/dat7/introspection/`

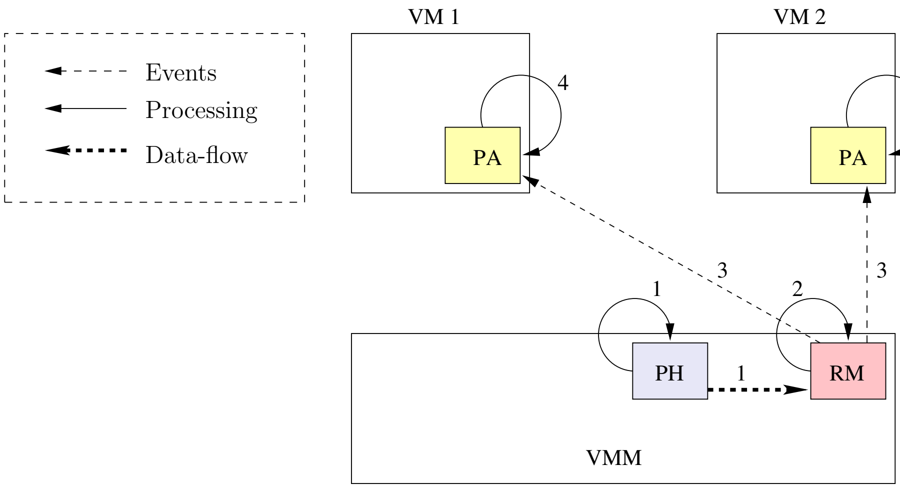


Figure 3.1: The overall steps involved in analysing identical pages. It consist of the following components: Page Hashing (PH), Reference Manager (RM) and the Page Analysis (PA) component.

3. When a predefined number of addresses in the shared buffer is reached, an asynchronous event is sent to the VM owning the buffer. Furthermore each time a page scanning round has been completed, we also send the event.

These three steps were more or less already present in CBPS as presented in our master thesis[20]. The reader is referred to the thesis for a more detailed description and the reasoning behind the design. The actual analysis of the pages is carried out by the page analysis component, which corresponds to step 4. The remainder of this chapter explains this.

3.2 Extending the Platform for Page Analysis

The page analysis component is implemented as a driver for the Linux kernel. This analysing driver is triggered by an interrupt handler, which receives the event sent in step 3 by the VMM. The interrupt handler defers the work, as it is customary in the Linux kernel, simply by scheduling a work queue handler. This calls a function we want triggered on each event. In this section we will make use of the `pg` abbreviation to denote a specific page descriptor.

To give an overview on how we analyse each page, we show a flow chart in Figure 3.2 on the facing page. This represents the process of dividing pages into the different mutually exclusive categories, as presented in Section 1.2.3 on page 14. We will now give a textual description of this process, where each step depends on the steps taken before the current step:

1. We start by having a new page to analyse. The first check determines if it is a free page. This is a simple test on whether the page count is above zero. The kernel always uses this count to determine if the page is free. As long as the count is above zero, then the page is per definition in use. Therefore this is a reliable way to tell that the page is free.

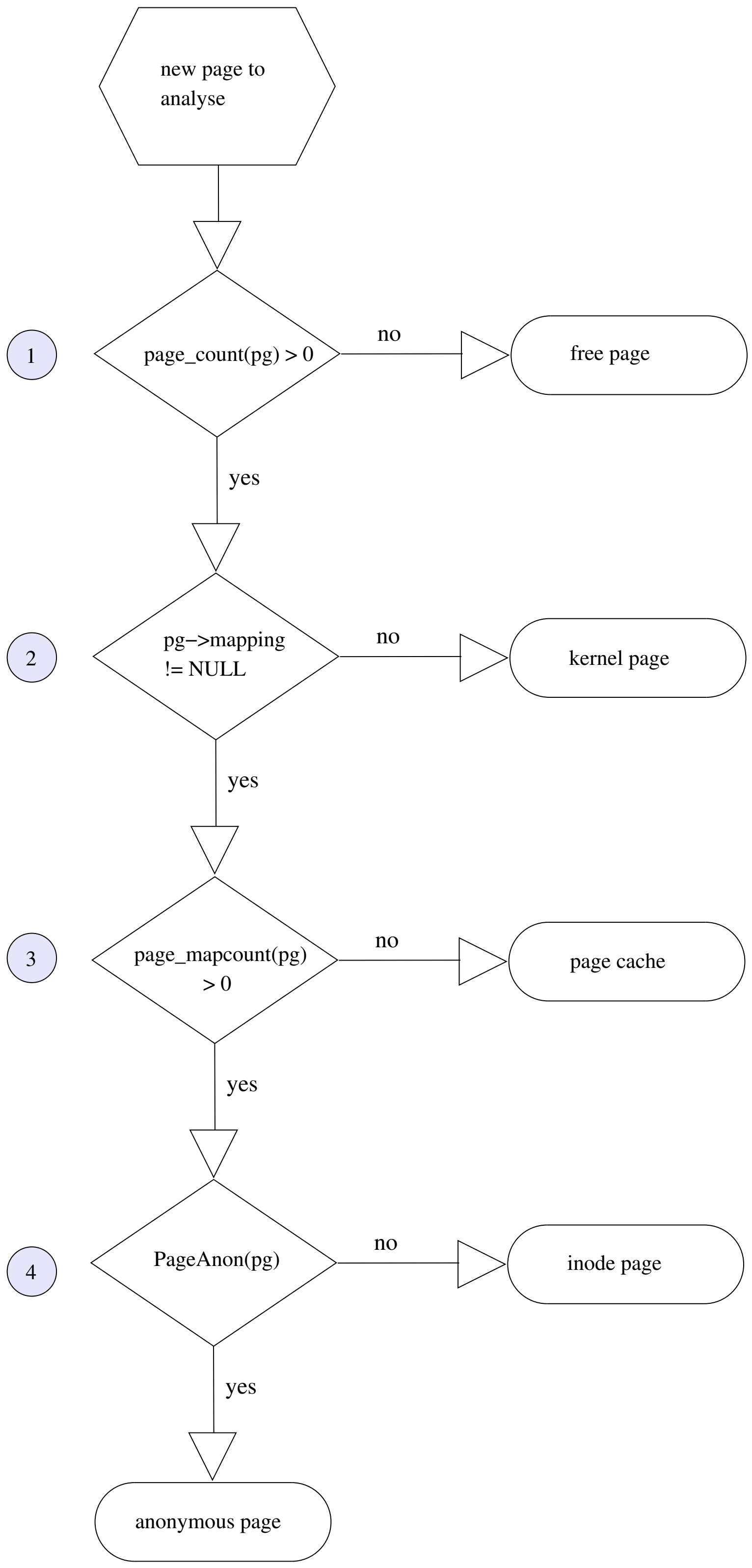


Figure 3.2: Flow chart of how a given page (pg) is categorized.

2. Next we determine if the page belongs to the kernel page category. This is done by testing if the `mapping` field on the page descriptor is `NULL`. We know that the page is in use, because the page count was above zero in the first check. If `mapping` is null, then it is not in the page cache or used as an anonymous page, because the `mapping` field would then either point to an `address_space` or `anon_vma` object. This rules out that a process is using it, so it is by definition used only by the kernel.

3. At this step it is determined if the page only belongs to the page cache. This is simply done by evaluating if the `mapcount` field on the page descriptor is above zero. Again we know that the page is in use, but no page tables are referencing it because the `mapcount` field is zero. The `mapping` field is used, so there must be either an `anon_vma` og `address_space` object associated with it. If it was an `anon_vma` object, then there should also be a process using it, but then this is not the case (because of the `mapcount`). Thus it must be a page that is used only by the page cache if the `mapcount` is zero.

4. In this last step it is determined if the page is an inode or anonymous page. This can be determined by use of the `PageAnon()` macro, which indicates just this. It returns true if the least significant bit on the `mapping` field on the page descriptor is set. Which, as we described in the previous chapter, means that the page is an anonymous page. Otherwise, it returns false meaning that the page is an inode page.

Having categorized a page, the sequence of events that follows depends on which category the page was in. Because of this we will describe them separately. A set of counters are used to keep track of the number of pages in each of the categories, thus giving us an overview of the pages in the shared buffer.

Free page: A free page should also satisfy, both `page_mapcount(pg) == 0` and `pg->mapping == NULL`. If this is not the case then the page is in "bad state" and an error count is incremented. Otherwise we can increment our free page counter.

Kernel page: First the kernel page counter is incremented and `pg->flags` is read, so we later can analyse more specifically what the kernel page is used for.

Cache page: Here the first step is to validate that the page in fact is part of the page cache. This is done by making a lookup in a radix tree as described in Section 2.1.2 on page 19. If so, then the page cache counter is incremented.

Next we proceed by making an in-depth examination of what the page is used for in the OS. We do this by retrieving which file it is related to. This is done by following the `mapping` field on the page descriptor. In the case of a page belonging to the page cache it points to an `address_space` object, which again has a `host` field that points to the owning inode of that `address_space` object. This corresponds to the following code: (`pg->mapping->host`). On the `inode` object, we can follow the pointer in the `i_dentry` field, to get the `dentry` object. Such a `dentry` object contains metadata about the file. With this `dentry` object we can get an absolute file name of the file, that the page is backed by.

On the first `dentry` object we get the file name from `dentry->d_name.name`. Then we get the path by traversing to the root of the file hierarchy by following the `d_parent` pointer, which points to the parent directory of the `dentry` object. On each `dentry` object the `d_name.name` is retrieved, which in this case contains a directory name. The root `dentry` object is identified by the fact that its `d_parent` pointer points to itself.

There might be more than one `dentry` object for each inode, because of filesystem hardlinks etc. We handle this by reading the absolute file names for all the `dentry` objects linked to the inode.

With this we have the information needed to analyse this category of pages.

Anonymous page: First the anonymous page counter is incremented. After this we proceed by making an in-depth analysis of the page. The information we want to retrieve from an anonymous page is which processes it is used by. We use reverse mapping to retrieve any VMAs containing the page to find these processes, as described in Section 2.3 on page 23. On each of these VMAs we use its `vma->vm_mm` to give us the memory descriptor of the process it belongs to. With this we can relate each VMA to a specific process[2]. So we can identify the set of processes that are using this page.

Inode page: As with the other cases, we start by incrementing the counter of this type of page. The in-depth analysis on this category of pages consist of relating them to files and processes. First we relate the page to processes, this is done as we did with anonymous pages by making use of reverse mapping to get the VMAs that contains the page and on each of these VMAs get the process that was related to the VMA. Then we relate the pages to files, which is done as we did with the cache pages.

This ends the description on how the pages in the different categories are analysed. Next we will evaluate the usefulness of the implementation.

3.3 The Usefulness of the Extension

The implementation has two disadvantages, which both are consequences of the fact that the implementation is built upon our existing platform. This was also the reason why it was a feasible implementation, since we only had to implement the analysing driver in order to bridge the semantic gap.

The first disadvantage is that we do not have any guarantees that the pages, we receive from the VMM, have not changed it usage or contents since it was determined that it was shareable. The second issue is that we only scan pages when a processor is idle, which then means that during workloads where there are no idle processor time (no IO waits etc.), it can not be expected that any pages are scanned. That being said the implementation is more than sufficient to give us the indication of what the shared pages are actually used for.

[2]Though it actually requires iterating over the set of processes to determine which process the memory descriptor belongs to.

3.4 Summary

In this chapter we have presented the design and implementation of a page analysing driver for the Xen modified Linux kernel. Based on the description of the virtual memory subsystem from the previous chapter, we were able to identify several pieces of key information to correctly categorize any given page within the five mentioned categories. To do an in-depth analysis of the pages, we found that it is worthwhile to lookup the file names of pages found in the page cache. When a page is used by a process, i.e. it is a mapped or anonymous page, then we find the name of the process. Finally when a page is used by the kernel, we can use the flags field on the page descriptor to further determine the use of the page in the kernel.

We evaluated the implementation and found that it may not be ideal for the purpose, but it is sufficient to give good indications about what the pages are used for.

Chapter 4

Experimental Results

Having constructed what is needed to analyse the shareable pages, we can finally start the really interesting part of the project, namely to investigate what the pages, found shareable in virtualized systems are used for. To do this, we return to the workloads from the initial experiments in Section 1.2.1 on page 11. We run the experiments as before, but with our analysing driver activated. The experiments were conducted under the same conditions as the experiments in the first chapter. Therefore we expect the amount of shareable pages to be roughly the same and the results are approximately equivalent to the initial experiments.

It is important to note that the analysis in this chapter is based on a given time interval rather than a single scanning round. It should also be noted that we do not setup shareable pages, so the same pages are found multiple times. Thus the more often we encounter a given page, the more static the contents of the page probably is. Therefore the frequency of finding a given page is of interest to us, since pages with static contents are ideal to share.

Another note worth pointing out is that the pages present in the page cache is the sum of two categories, namely the inode pages and cache pages. Inode pages are pages in use by processes, while cache pages are the pages that are only present in the page cache. Both are as such present in the page cache, but our view reflects the fact that we are also implementing a paravirtualized driver to obtain the sharing (and also to some extent how the kernel views the pages). Inode pages are plainly more interesting than the cached pages, because there are more virtual mappings to alter for the pages in this category, which is why we have made two separate categories.

Throughout the experiments we collected 397 MB of runtime data from the analysing driver and the rest of this chapter presents the interesting results from these. This data is also available from our homepage along with scripts to analyse it.

The chapter starts with an overview of how the shareable pages are generally used, in particular how the analysed pages are distributed over the five categories. Afterwards, we present the in-depth details that were also collected during the page analysis. Finally we summarize the chapter and discuss the consequences of the results of the page analysis.

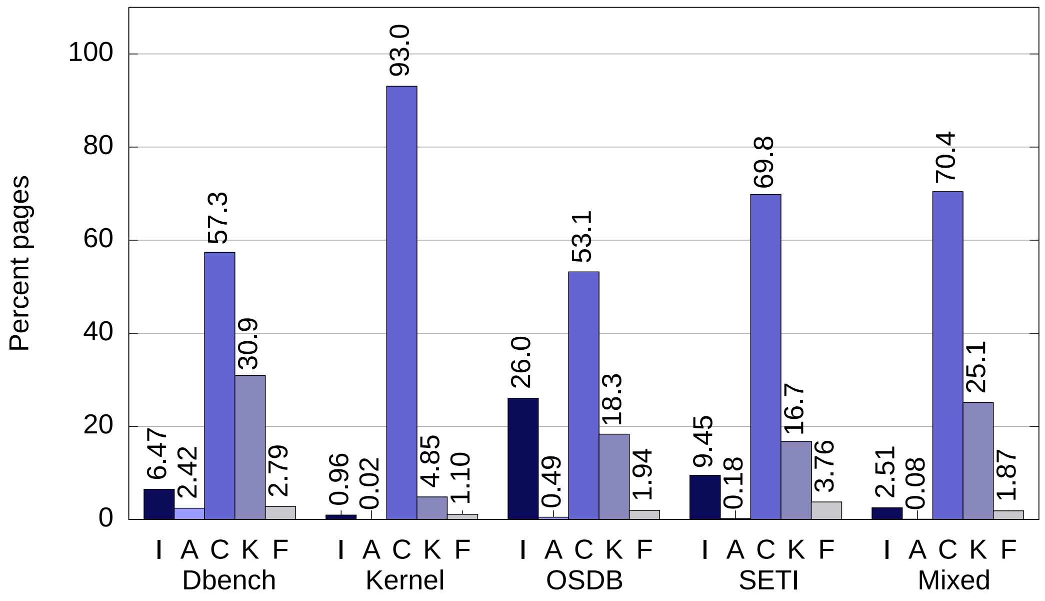


Figure 4.1: The frequency distribution of shareable pages in the time frame from 2 to 90 minutes in the experiments. The letters in the graph represent: (I) Inode pages, (A) Anonymous pages, (C) Cache pages, (K) Kernel pages, and (F) Free pages.

4.1 Categorizing Pages

To give a overview on how the shareable pages are distributed across the different categories, we present Figure 4.1, which shows the frequency distribution of the shareable pages throughout the experiments. This figure is based on information collected from the beginning of the experiments, which is from 2 minutes after boot, until 92 minutes have passed. So information is collected while the system is both idle and busy. The results in the graph is the average of all four VMs during each experiment. If the reader is interested, then the results for the individual VMs can be found in Appendix C.

The amount of pages in the inode pages category varies between 0.96%-26%. Inode pages represents in-memory files and application code used by running processes. The amount of anonymous pages found during the different experiments is low as these only represent 0.08%-2.42% of the pages. This type of page is also used by running processes and is typically created when a process needs to store volatile information. Generally the cache and the kernel categories are responsible for the most shareable pages. The amount of free pages is low because the load on the VMs is high and the consequence is that the amount of shareable free pages is correspondingly low (1.1%-3.76%).

For the readers convenience we have chosen to reintroduce the graph from page 13 as Figure 4.2 on the facing page, as it shows the amount of shared pages during the different experiments. This will be used in the following combined with Figure 4.1.

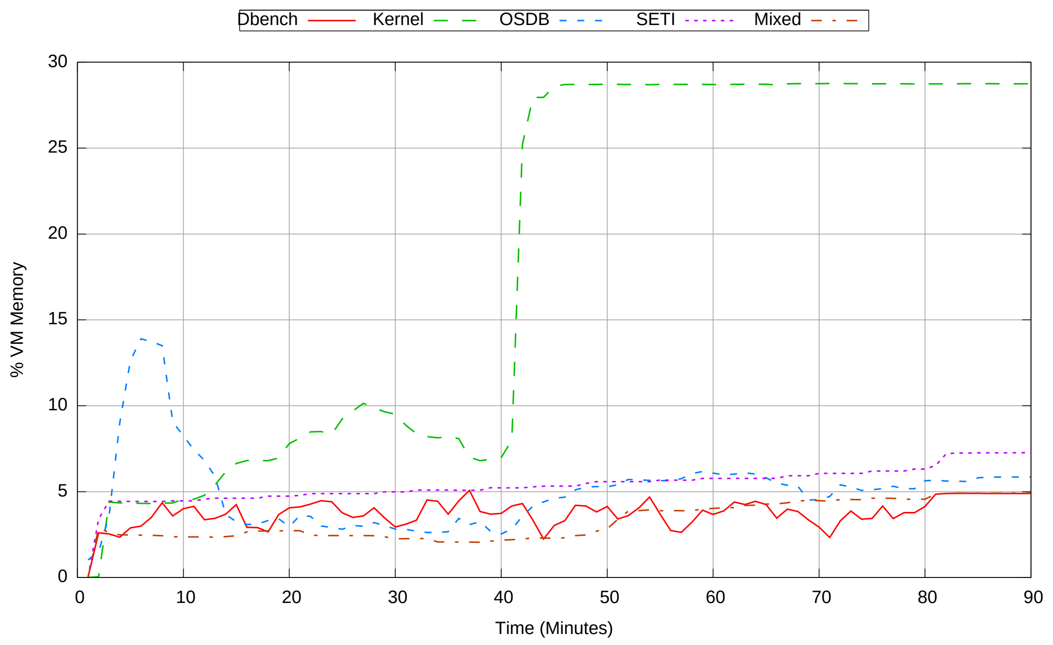


Figure 4.2: The number of reclaimed pages from four concurrently running VMs during various workloads.

From the graph in Figure 4.2 we observe that the amount of shared pages was roughly 4% while running the Dbench benchmark, which we accredited to memory representations of files used by Dbench. This assumption seems to be correct, because 63.77% of the pages found during this benchmark belong to the page cache[1]. We did however not expect that 30.9% of the pages where due to pages used internally by the Linux kernel.

During the kernel compilation workload, up to 93% of the pages found also belong to the cache pages category, which was what we expected. As each VM is compiling the same kernel sources the contents of the page cache in the VMs are almost identical when the compilations are completed. This is why the number of shareable pages sky-rockets in Figure 4.2 after the compilations are finished. The fact that the cache pages category is overrepresented compared to the inode pages category clearly shows that once the processes used during the compilation are terminated, they are left in the page cache.

The OSDB benchmark rises to almost 14% shared pages in the beginning of the run and drops to 4-5% after some time. Our first assumption was that the shareable pages were due to the MySQL and OSDB binaries. As there are unusually many pages in the inode pages category, this seems reasonable. The MySQL and OSDB processes run for the entirety of the benchmark, so they should be found on each scanning round.

In the beginning of this report we expected SETI to be processor intensive and allocate some amounts of anonymous pages. But as Figure 4.1 illustrates, SETI

[1]Remember that the actual amount of pages from the page cache consists of both the inode pages and cache pages categories.

has almost no anonymous pages. Instead the cache page category is dominant on this workload. Furthermore the memory footprint of the SETI application is limited to 30 MB as default, which explains why this workload has the largest amount of shareable free pages (3.76%).

The Mixed workloads behave as expected, with a low amount of shareable pages that increases slightly over time. As it can be seen in Figure 4.1 the largest amount of shareable pages is again due to the page cache.

The next section investigates what the shareable pages in the page cache are used for. This is interesting as the page cache represents such a large percentage of the pages that we find.

4.1.1 Files Found in the Page Cache

As explained in Section 2.1 on page 18, the page cache is responsible for caching data fetched from block devices. In this section we investigate which files end up in the page cache during the different experiments. Table 4.1 on the next page contains a top five of the most commonly found shareable files in the page cache. For further information see Appendix D.

During the Dbench experiment we expected that the cache would be filled with data generated by Dbench, but as the table shows the `client.txt` file is dominating the page cache. This file has a size of 26 MB and it contains metadata with instructions about how Dbench should behave. So it is not strange that this file is dominating the page cache, as it is used intensely by the Dbench processes.

The page cache contains, not surprisingly, files belonging to the kernel source code during the kernel compilation experiments. More precisely the files found belongs to the kernel source code and the result of the compilation e.g. the newly built kernel images. This is because the compilations need to read and write a large number of files to and from disk and all of these operations go through the page cache.

The page cache, during the OSDB experiments, contains the MySQL binaries and temporary data files as expected. During the SETI experiment the page cache contains application binaries and libraries as expected. In the Mixed experiment the kernel compilation dominates the results, hence most of the files we find in the top five are files that are part of the kernel source code. At first this might seem surprising, because the purpose of the page cache is to ensure that the files only end up in memory once, so why is the kernel compilation the cause of so much sharing? The truth however is that during the compilation, the produced object code is actually copied to new files. New files imply new entries in the page cache, so this is a good example of how identical pages end up in memory and result in internal sharing due to inner workings of the kernel.

The page cache only shows us what files the different VMs have read from or written to disk. It do not tell us which files are used by the processes currently in the system. So the next section will take a closer look at what the inode and anonymous pages are used for by the different processes.

	Files	**Count**
Dbench		
	/usr/share/dbench/client.txt	179037
	/bin/bash	1758
	/lib/tls/libc-2.3.6.so	734
	/lib/libncurses.so.5.5	675
	/lib/libc-2.3.6.so	374
Kernel compile		
	/usr/src/linux-2.6.16/.tmp_vmlinux2	189245
	/usr/src/linux-2.6.16/vmlinux	145534
	.../arch/i386/boot/compressed/vmlinux.bin	122929
	/usr/src/linux-2.6.16/.tmp_vmlinux1	85434
	.../arch/i386/boot/vmlinux.bin	56412
OSDB		
	/var/lib/mysql/ibdata1	102658
	/root/data/asap.uniques	22765
	/tmp/MLxnR9uL	22612
	/var/log/mysql/mysql-bin.000029	20567
	/var/log/mysql/mysql-bin.000028	12095
SETI		
	.../setiathome-5.12.i686-pc-linux-gnu	12791
	/usr/lib/i686/cmov/libcrypto.so.0.9.8	4259
	/usr/lib/libstdc++.so.6.0.8	3330
	/usr/lib/i686/cmov/libcrypto.so.0.9.7	2982
	/bin/bash	2968
Mixed		
	/usr/src/linux-2.6.16/.tmp_vmlinux2	20382
	/usr/src/linux-2.6.16/vmlinux	13265
	.../arch/i386/boot/compressed/vmlinux.bin	11365
	/var/lib/mysql/ibdata1	8426
	.../arch/i386/boot/vmlinux.bin	4460

Table 4.1: A top five over most commonly found files in the page cache. The count reflects how many times pages representing a given file is found.

4.1.2 Shareable Pages Used by Processes

Now we investigate which processes typically are responsible for shareable pages. Table 4.2 lists the ten most commonly found processes, that are using anonymous and inode pages, during the Dbench experiment. A top ten for each of the experiments can be found in Appendix E, if the reader is interested in more information about how anonymous and inode pages are used by processes.

Anonymous pages		**Inode pages**	
Process	**Count**	**Process**	**Count**
dbench	9416	dbench	30894
bash	353	sshd	9945
sshd	150	bash	8102
inetd	120	cron	7768
getty	106	syslogd	6393
init	75	init	5313
dhclient	62	getty	5071
cron	50	klogd	3970
klogd	12	inetd	3391
rm	3	dhclient	3267

Table 4.2: The number of shareable anonymous and inode pages found during the Dbench experiment and the names of the processes that are using them.

As the table shows, the dbench process is using a fairly large amount of both anonymous and inode pages compared to the other processes. Surprisingly the dbench process uses many anonymous pages. They may be used to generate temporary data, that the benchmark uses during its execution. Furthermore the tables show, not surprisingly, that the different system services executing are also responsible for anonymous and inode pages.

As we have no knowledge about the internal workings of the processes, we can not investigate further what the processes are using the anonymous pages for. But we can make an in-depth analyse on the usage of inode pages during the experiment. To do this we have investigated what files the inode pages represent in the filesystem. A top ten of the most commonly observed files can be found in Table 4.3 on the facing page. The binaries of the dbench application and the most common system applications are found in the files listed. But the investigation also reveals, that the inode pages is mainly used to contain libraries rather than the binaries of the processes.

Note that all the same libraries appears during all the experiments conducted, which can be seen in Appendix F on page 67. In fact the dynamic library `libc-2.3.6.so` is observed the highest number of times during every experiment that we have conducted. So it is interesting to further investigate the usage of these types of libraries, as they generally are shareable on almost any workload in a homogeneous system. Table 4.4 on the facing page lists the three most commonly observed standard libraries throughout all the experiments and the number of times they have been observed.

The two first libraries are the standard C library and the linker library, which

Files	**Count**
/lib/tls/libc-2.3.6.so	40110
/lib/libc-2.3.6.so	20226
/lib/ld-2.3.6.so	6670
/bin/bash	5766
/usr/lib/i686/cmov/libcrypto.so.0.9.7	2307
/usr/sbin/sshd	1994
/usr/bin/dbench	1702
/sbin/dhclient	904
/usr/sbin/cron	808
/lib/tls/libnsl-2.3.6.so	619

Table 4.3: Top ten of the most frequently found file names gathered from the shareable pages used by processes during the Dbench workload.

	Files	**Count**
1.	/lib/libc-2.3.6.so	124770
2.	/lib/ld-2.5.6.so	20968
3.	/usr/lib/i686/cmov/libcrypto.so.0.9.7	9439

Table 4.4: The three most commonly observed standard libraries throughout all the experiments.

are used by almost all dynamically linked applications. So it is not surprising that it is these files we find. We can take the investigation even further by following the inode pages back to the process that owns them. This is shown in Table 4.5.

	Files	**Process**
1.	libc-2.3.6.so	init, syslogd, inetd, sshd, cron, getty, dbench, klogd, bash, dhclient, sleep, rm, run-parts, make, sh, gcc, cc1, as, mysqld_safe, mysqld, logger, osdb-my, bonic_client, bonic, setiathome-5.12, dbench
2.	ld-2.5.6.so	init, dhclient, syslogd, klogd, inetd, sshd, cron, getty, bash, dbench, rm, sleep, sh, run-parts, make, gcc, cc1, as, mysqld_safe, mysqld, logger, osdb-my, bonic_client, bonic, setiathome-5.12
3.	libcrypto.so.0.9.7	sshd

Table 4.5: Processes using the three most frequently shared standard libraries, based on the pages in the inode pages category.

As the table illustrates, these standard libraries are used by quite a few processes throughout the experiments. All the observed processes subjected to our workloads use these standard libraries, so they are shareable between the VMs in all the experiments.

As every process in Linux has the init process as its ancestor[6, s. 91], we can take our investigation a step further. The output below shows the currently mapped memory region for the init process.

```
sh-3.1$ cat /proc/1/maps

08048000-0804f000 r-xp 00000000 08:01 32160      /sbin/init
0804f000-08050000 rw-p 00007000 08:01 32160      /sbin/init
08050000-08071000 rw-p 08050000 00:00 0          [heap]
b7e0c000-b7e0d000 rw-p b7e0c000 00:00 0
b7e0d000-b7f35000 r-xp 00000000 08:01 32029      /lib/libc-2.3.2.so
b7f35000-b7f3d000 rw-p 00127000 08:01 32029      /lib/libc-2.3.2.so
b7f3d000-b7f40000 rw-p b7f3d000 00:00 0
b7f42000-b7f43000 rw-p b7f42000 00:00 0
b7f43000-b7f59000 r-xp 00000000 08:01 32026      /lib/ld-2.3.2.so
b7f59000-b7f5a000 rw-p 00015000 08:01 32026      /lib/ld-2.3.2.so
bfd40000-bfd56000 rw-p bfd40000 00:00 0          [stack]
bfffe000-bffff000 r-xp bfffe000 00:00 0
```

It contains virtual address regions used by the init process with information about: the virtual memory range, permission, file offsets, the device(s) the file is located on, the files inode, and the pathname. The init process have mapped in the two standard libraries (`libc, ld`) that we have observed during all our experiments. It is plausible that this is why such a large amount of the shareable inode pages, that we find, can be followed back to these particular libraries.

The next section investigates what the large amount of kernel pages are used for within the VMs.

4.1.3 Pages Used by the Linux Kernel

The second largest amount of pages in a category are due to pages that are used internally by the Linux kernel. Remember from the previous chapter that kernel pages are pages that have their `mapping` field set to `NULL` and are not free pages. When we find such a page, we use the flags on its page descriptor to determine what the kernel uses it for. A complete list of these flags can be found in Appendix B.

Experiment	**Reserved**	**slab**	**Others**
Dbench	77.26%	22.71%	0.03%
Kernel compile	76.09%	23.60%	0.31%
OSDB	75.44%	24.48%	0.08%
SETI	78.36%	21.63%	0.01%
Mixed	74.89%	25.09%	0.02%

Table 4.6: The distribution of kernel pages according to their flags.

As it can be seen in Table 4.6 most of the kernel pages that we find has only the `reserved` flag set, which means that the page is reserved for kernel use or is unusable. These pages are not allowed to be swapped out to disk, because the kernel uses them or they may even be corrupted memory pages[2].

[2]This information can be found in `/include/linux/page-flags.h` in the Linux 2.6.16 kernel source code.

The other large group of pages are used by slab[5] caches. Remember that slab objects are preinitialized commonly used kernel objects, such as `vm_area_struct` and `dentry_cache` objects. They are stored in slab caches to minimize the overhead of initializing and freeing these common structures every time the kernel needs them [23, p. 194-195]. Our implementation probably finds these structures, because the different VMs run the same kernel version, hence needs the same basic objects in their slab caches.

4.2 Discussion

This chapter presented the results of the project, namely what shareable pages are used for on a number of workloads. We used the analysis driver, presented in the previous chapter, to categorize shareable pages and used category specific information to examine the usage of a given page in-depth. We showed which processes were responsible for the pages and how their use of libraries was the cause of some shareable pages. We argued that all the processes used the same standard libraries, because they are forked from the init process. Finally we saw that the shareable pages used by the kernel contained runtime data, e.g. slab caches.

Generally we found that the page cache, on all workloads, was responsible for the most shareable pages. In this respect it does not matter whether the pages in the page cache are due to the pages we categorize as inode pages or as cache pages, both are in the cache; inode pages just happen to be used by processes. We saw examples of how easily identical pages ended up several times in memory. The second largest amount of shareable pages was due to pages in use by the kernel. Finally free pages and anonymous pages presented a non-significant amount of the shareable pages.

The in-depth analysis found that on systems with the same degree of homogeneity in the system, as the workloads presented here, it can be expected that the set of system services, that are running on all VMs, will be shareable. A part of the shareable pages due to these contain the standard libraries.

The remainder of this chapter discusses how these results are affected by changes in the parameters, i.e. the degree of homogeneity. The degree of homogeneity can be reduced in a number of ways: 1) the amount of system services is reduced, 2) binaries and libraries are not identical, and 3) different versions of the kernel is used. If the amount of system services is reduced, then the amount of shared pages should decrease, primarily due to pages in the inode pages category. In fact it goes for all of these that the amount of shareable pages would decrease. If the VMs were using distributions with different versions of the binaries and libraries, then the amount of shared pages would probably be reduced to the pages that are shareable due to the kernel. Finally if different kernel versions are applied, then we might not see any sharing at all.

On the other hand the degree of homogeneity could also increase. If more system services are run, then the amount of shareable pages should rise, primarily due to pages in the inode pages category. This seems plausible as we used virtual machines that ran only a minimal set of system services. RedHat Enterprise Linux 4 for instance per default runs a much larger set of system services than those present on our virtual machines running Debian.

Another parameter to consider is the type of workload. In general we expect the relation between the amount of pages due to the page cache and the pages due to the other categories to be consistent on general server room grade workloads. However highly specialized workloads may tip the scale further in the direction of anonymous pages. Examples of such could be applications running state space exploration or even common desktop usage. It is however questionable whether such workloads would be worth virtualizing.

We could also consider what happens if OSes that are different from Linux are applied. Most commodity OSes have disk caches that work in a manner that is equivalent to the page cache in Linux, so the results will probably also be valid on other OSes.

All these considerations lead us to conclude that generally the greater the degree of homogeneity in a virtualized system, the greater the level of redundancy in memory.

Chapter 5

Conclusion

This report examined what pages eligible for sharing in virtualized systems are typically used for. Generally we can conclude that the greater the degree of homogeneity in the virtualized system, the greater the level of redundancy. We found that the main factor causing this redundancy is applications, namely their binaries and the libraries they use as well as the files they open. All of these end up in disk caches and are left there when they are no longer needed. Furthermore some redundancy is caused by the use of identical kernels, as VMs in homogeneous virtualized systems often appear to have the exact same data structures.

In order to bridge the semantic gap between the VMM, that identifies shareable pages, and the OS in a VM, that is responsible for the use of the pages, we turned to virtual machine introspection. This made all the data structures used internally in the kernel accessible for an analysis of the shareable pages. As the kernel is a complex piece of software, we needed a way to find the most relevant pieces of information about the pages. Therefore we examined the details of the virtual memory subsystem of the Linux kernel and established five mutually exclusive categories to give an overview of how the pages are used. The five categories roughly corresponds to how the Linux kernel views the pages.

The functionality needed to categorize pages and examine a given page further based on the category, was implemented as an analysing driver for the Xen modified Linux kernel. The functionality needed to find identical pages was reused from our CBPS patch for Potemkin.

This driver was subjected to a number of experiments with synthetic workloads, which exercise the kernel in different ways. These workloads are not representative of real-world workloads, but they still illustrate how the virtual memory mechanisms of the kernel work when subjected to these. We expect that real-world workloads will exhibit the same characteristics, because the workloads make use of real-world applications (e.g. MySQL and source code compilation).

The results of the experiments clearly show that the most shareable pages are due to cached files. Interestingly a significant amount of shareable pages are due to data structures inside the kernel. e.g. slab caches. As could be expected, at least on the workloads presented here and probably also on typical server workloads, shareable pages due to anonymous pages were scarce. We expect that very specialized workloads, such as applications doing long running state space exploration, may show otherwise. It is however questionable whether such workloads would be worth virtualizing. Finally the amount of shareable pages due to free pages was low, as the page cache typically takes up most of main memory.

When examining which processes were typically responsible for the shareable pages, we found that the system services and dormant daemons, such as sshd, init and cron, were shareable on all the examined workloads. This was primarily due to the binaries of the applications and the libraries used by these. In particular we found that the standard C libraries always are shareable on workloads as homogeneous as those examined here. In our experiments we used VMs that only ran minimal services, so the amount of shareable pages due to services is lower than what could be expected by real-world setups. Furthermore the task of compiling a kernel exemplified how easily identical pages end up in the page cache.

Generally we can conclude that the page cache in Linux is responsible for most of the shareable pages on everyday workloads. Most commodity OSes have disk caches that work in a manner that is equivalent to the page cache in Linux, so this result will presumably also be valid in regards to OSes different from Linux. A direct result from this is that the interdomain shared cache approach, currently being implemented as XenFS and used in Disco, principally should be close to equivalent to CBPS in regards to the ability to share as many pages as possible, because a very large percentage of the shareable pages is due to disk cache. This approach will however still miss the sharing opportunities we identified in the kernel.

Future work may include investigating real world workloads. In particular internal sharing on a single VM exercised by different workloads should prove interesting, as this should identify the sources of redundancy in everyday applications.

Bibliography

[1] MySQL AB. Mysql: The world's most popular open source database. `http://www.mysql.com/`.

[2] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*, October 2006.

[3] AMD. Amd secure virtual machine architecture reference manual. `http://www.cs.utexas.edu/~hunt/class/2005-fall/cs352/docs-em64t/AMD/virtualization-33047.pdf`.

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.

[5] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer*, pages 87–98, 1994.

[6] Daniel Bovet and Marco Cesati. *Understanding the Linux Kernel, Third Edition.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2005. ISBN 0596005652.

[7] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. pages 143–156, 1997. ISBN 0-89791-916-5.

[8] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, page 133. IEEE Computer Society, Washington, DC, USA, 2001.

[9] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, May 2005.

[10] Jonathan Corbet. The object-based reverse-mapping vm. `http://lwn.net/Articles/23732/`.

[11] Jonathan Corbet. Reverse mapping anonymous pages - again. `http://lwn.net/Articles/77106/`.

[12] Jonathan Corbet. The status of object-based reverse mapping. `http://lwn.net/Articles/85908/`.

[13] Jonathan Corbet. Virtual memory ii: the return of objrmap. `http://lwn.net/Articles/75198/`.

[14] Kemal Ebcioglu, Erik R. Altman, Michael Gschwind, and Sumedh W. Sathaye. Dynamic binary translation and optimization. *IEEE Transactions on Computers*, 50(6):529–548, 2001.

[15] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium*, February 2003.

[16] Robert Philip Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, June 7(6):34–45, 1974.

[17] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall PTR, 2004. ISBN 0131453483.

[18] Judith S. Hall and Paul T. Robinson. Virtualizing the vax architecture. In *ISCA '91: Proceedings of the 18th annual international symposium on Computer architecture*, pages 380–389. ACM Press, 1991.

[19] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Geiger: Monitoring the buffer cache in a virtual machine environment. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*, October 2006.

[20] Jacob Faber Kloster, Jesper Kristensen, and Arne Mejlholm. Efficient memory sharing in the xen virtual machine monitor. `http://www.cs.aau.dk/library/cgi-bin/detail.cgi?id=1136884892`, January 2006.

[21] Jacob Faber Kloster, Jesper Kristensen, and Arne Mejlholm. On the feasibility of memory sharing: Content-based page sharing in the xen virtual machine monitor. Master's thesis, Department of Computer Science, Aalborg University, June 2006. `http://www.cs.aau.dk/library/cgi-bin/detail.cgi?id=1150283144`.

[22] Joshua LeVasseur, Volkmar Uhlig, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. Pre-virtualization: Slashing the cost of virtualization. Technical Report 2005-30, Fakultät für Informatik, Universität Karlsruhe (TH), November 2005.

[23] Robert Love. *Linux Kernel Development*. Novell Press, 2005. ISBN 0131453483.

[24] Dave McCracken. Object-based reverse mapping. In *Proceedings of the 2004 Ottawa Linux Symposium*, July 2004.

[25] Edward M. McCreight. Priority search trees. 14(2):257–276, May 1985. CODEN SMJCAT. ISSN 0097-5397 (print), 1095-7111 (electronic).

[26] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel Virtualization Technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3):167–177, August 10, 2006.

[27] Gerald J. Popek and Robert Philip Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974. ISSN 0001-0782.

[28] John Scott Robin and Cynthia E. Irvine. Analysis of the intel pentiums ability to support a secure virtual machine monitor. In *USENIX Security Symposium*, pages 129–144, 2000.

[29] Mendel Rosenblum. The reincarnation of virtual machines. *Queue*, 2(5):34–40, 2004.

[30] Mendel Rosenblum and Tal Garfinkel. Virtual machine monitors: Current technology and future trends. *Computer*, 38(5):39–47, 2005.

[31] Stephen Soltesz, Marc E. Fiuczynski, Larry Peterson, Michael McCabe, and Jeanna Matthews. Virtual doppelgänger: On the performance, isolation and scalability of para- and paene- virtualized systems. `http://www.cs.princeton.edu/~mef/research/paenevirtualization.pdf`.

[32] The Open Source Database Benchmark Team. The open source database benchmark. `http://osdb.sourceforge.net`.

[33] Andrew Tridgell. The dbench benchmark. `http://samba.org/ftp/tridge/dbench/README`.

[34] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.

[35] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2005.

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

[37] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. In *Proceedings of the USENIX Annual Technical Conference*, 2002. `http://denali.cs.washington.edu/pubs/distpubs/papers/denali_usenix2002.pdf`.

[38] Mark Williamson. Xen wiki: Xenfs. `http://wiki.xensource.com/xenwiki/XenFS`.

Appendices

Appendix A

Setup of the Experiments

This part of the appendix describes the setup used during the different experiments and the software used to perform the experiments.

All the experiments were carried out on the same hardware configuration. The machine used was a Intel P4 2.6 Ghz Northwood with hyperthreading enabled and with 2GB of memory. The Virtual Machines (VMs) ran Debian on a modified 2.6.16 Xen kernel with a bare minimum of system services. The VMs had a memory allocation of 128 MB each and used shadow page tables under the initial experiments in Section 1.2.1 on page 11. The introspection experiments used normal paravirtualized page tables and our page analysis driver as described in Section 3 on page 29. The changesets numbers of our repository used for the experiments were 9894:ffa043a2170d and 9906:f7e1020e15b8 respectively.

The best case experiments allowed sharing of zero pages, while this was disabled in all the other experiments.

A.1 The Different Benchmarks

The benchmarks used during the experiments were the following:

Kernel compile: A simple test where we spend CPU cycles on compiling the Linux 2.6.16 kernel. This workload should exercise the potential shares thoroughly, thus ensuring variation in the pages. Furthermore it should make a moderate load on the disk due to read and write operations to the disk under the compilations. This takes approximately 42 minutes to complete.

OSDB: The Open Source Database Benchmark[32] exercises a database system of choice. For these experiments we have chosen the MySQL[1] 5.0.21 database, as this combined with the Apache web server appears to be a frequent combination. We used version 0.21 of the benchmark. OSDB executes two different types of tests which both are multi-user tests: 1) Information Retrieval (IR) and 2) On-line Transaction Processing (OLTP) tests. The data set for this benchmark is 40 MB. The benchmark runs for approximately 20 minutes, so we run it three consecutive times.

Dbench: The Dbench benchmark[33] emulates the workload of a number of network clients (we have tested with 8 clients) by generating a large workload consisting of disk IO operations. More specifically it emulates the behavior of the NetBench benchmark, which is used to evaluate file servers such as Samba and Windows NT [33]. Because of its intensive number of IO operations, the benchmarks generate a significant workload for the processor as well as the page cache. Dbench version 3.04 was used in the experiments. The Dbench application runs for approximately 10 minutes, so we run it 6 consecutive times with 1 minute of sleeping between each.

SETI: SETI is a CPU intensive application, that downloads a work unit through the internet and then spends a long time analysing this. This is activated by the boinc client, which is a general distributed computing services specifically designed for making use of home PCs for large computational tasks. Version 5.12 was used for the experiments. On this setup the SETI application is per default limited to 30 MB memory usage. The application is terminated after 80 minutes.

Mixed: VM1 is running OSDB, VM2 runs SETI, VM3 is performing a kernel compilation, and finally VM4 is running the Dbench benchmark.

It is arguable whether these are actually benchmarks. The benchmarking point is however not essential to us, as we only use them to generate load on the VMs.

Appendix B

Page Descriptor Flags

This table can be found in [6, p.296-297]. The table is extended with the `PG_uncached` flag, as found in `include/Linux-flags.h` in the Linux kernel source code version 2.6.16.

Bit	**Name**	**Comment**
0	PG_locked	The page is locked; for instance, it is involved in disk I/O operation.
1	PG_error	An I/O error occurred while transferring the page.
2	PG_referenced	The page has been recently accessed.
3	PG_uptodate	This flag is set after completing a read operation, unless a disk I/O error happened.
4	PG_dirty	The page has been modified.
5	PG_lru	The page is in the active or inactive page list.
6	PG_active	The page is in the active page list.
7	PG_slab	This page frame is include in a slab.
8	PG_checked	Used by some filesystems such as Ext2 and Ext3.
9	PG_arch_1	Not used on 80x86 architecture.
10	PG_reserved	The page frame is reserved for kernel code or is unusable.
11	PG_private	The `private` field of the page descriptor stores meaningful data.
12	PG_writeback	The page is being written to disk by means of the `writepage` method.
13	PG_nosave	Used for system suspend/resume.
14	PG_compound	The page frame is handled through the extended paging mechanism.
15	PG_swapcache	The page belongs to the swap cache.
16	PG_mappedtodisk	All data in the page frame corresponds to blocks allocated on disk.
17	PG_reclaim	The page has been marked to be written to disk in order to reclaim memory.
18	PG_nosave_free	Used for system suspend/resume.
19	PG_uncached	Page has been mapped as un-cached

Table B.1: A list over the different flags found in the `flags` field of the page descriptor in the Linux kernel.

Appendix C

Categorizing Pages

This appendix contains detailed information about the results used to create Figure 4.1 on page 36. The information is presented in Table C.1. It contains the frequency distribution of the pages into the categories defined in Section 1.2.3 on page 14.

These tables contain the distribution of pages within a 90 minute time frame, while executing the five experiments. The time frame is more precisely 2 minutes after boot-up, until 92 minutes has passed. The Mixed experiment is extended with information about the workloads on the four virtual machines. Note that the results from the different virtual machines cannot be compared directly with each other because the sample size differ.

	VM	Inode	Anon	Cache	Kernel	Free	Errors
Dbench							
	1	6.68%	2.65%	53.8%	33.3%	3.48%	0.00%
	2	9.15%	2.23%	54.8%	31.0%	2.69%	0.01%
	3	3.35%	1.49%	63.6%	30.1%	1.37%	0.0%
	4	6.14%	2.84%	62.7%	25.7%	2.47%	0.00%
Kernel compile							
	1	0.99%	0.01%	90.9%	6.85%	1.22%	0.0%
	2	1.44%	0.02%	93.1%	4.50%	0.90%	0.0%
	3	0.81%	0.03%	93.8%	4.27%	1.00%	0.0%
	4	0.74%	0.04%	95.0%	2.98%	1.16%	0.0%
OSDB							
	1	29.9%	0.54%	47.4%	20.1%	1.80%	0.00%
	2	27.0%	0.61%	53.7%	16.9%	1.66%	0.0%
	3	26.1%	0.43%	54.7%	16.1%	2.53%	0.01%
	4	15.3%	0.31%	65.0%	17.2%	2.04%	0.01%
SETI							
	1	11.4%	0.12%	63.6%	21.9%	2.88%	0.0%
	2	11.0%	0.25%	69.9%	16.6%	2.14%	0.0%
	3	9.04%	0.14%	69.6%	15.2%	5.96%	0.0%
	4	6.00%	0.24%	78.4%	10.7%	4.47%	0.0%
Mixed							
OSDB	1	6.04%	0.01%	29.5%	59.9%	4.35%	0.0%
SETI	2	0.09%	0.00%	68.3%	31.4%	0.13%	0.0%
Kernel compile	3	1.14%	0.04%	88.3%	10.0%	0.43%	0.0%
Dbench	4	14.2%	0.97%	22.3%	47.0%	15.3%	0.0%

Table C.1: In this table the aggregated results are presented.

	VM	Inode	Anon	Cache	Kernel	Free	Errors	Total
Dbench								
	1	11283	4486	90811	56233	5871	13	168697
	2	5874	1432	35166	19940	1731	7	64150
	3	2003	893	37998	18015	819	0	59728
	4	4400	2033	44893	18456	1772	2	71556
Kernel compile								
	1	6320	98	580067	43709	7785	0	637979
	2	4579	76	294102	14228	2874	0	315859
	3	3004	132	347666	15828	3721	0	370351
	4	3785	236	484890	15205	5941	0	510057
OSDB								
	1	63870	1165	101221	43012	3841	8	213117
	2	23658	536	46974	14844	1454	0	87466
	3	21327	354	44652	13195	2070	10	81608
	4	13315	275	56600	14986	1781	13	86970
SETI								
	1	7034	77	39216	13513	1777	0	61617
	2	3063	71	19488	4625	596	0	27843
	3	2835	44	21837	4767	1871	0	31354
	4	2640	107	34504	4738	1966	0	43955
Mixed								
OSDB	1	1774	4	8690	17616	1279	0	29363
SETI	2	30	2	21744	10004	43	0	31823
Kernel compile	3	1070	42	82848	9414	412	0	93786
Dbench	4	1237	84	1932	4070	1329	0	8652

Table C.2: In this table the absolute results are presented and a total column is added, which shows the sample size.

Appendix D

Files Found in the Page Cache

The files listed in this appendix are files found in the page cache during the experiments. More accurately, it is files that can be traced back to pages that we have found shareable. The lists are however limited to only showing the ten most commonly found files. The reader should furthermore be aware that the results are based on a 90 minute execution time frame.

	Files	**Count**
Dbench		
	/usr/share/dbench/client.txt	179037
	/bin/bash	1758
	/lib/tls/libc-2.3.6.so	734
	/lib/libncurses.so.5.5	675
	/lib/libc-2.3.6.so	374
	/bin/rm	256
	/lib/tls/libm-2.3.6.so	205
	/lib/libm-2.3.6.so	182
	/lib/tls/libpthread-2.3.6.so	174
	/usr/sbin/sshd	131
Kernel compile		
	/usr/src/linux-2.6.16/.tmp_vmlinux2	189245
	/usr/src/linux-2.6.16/vmlinux	145534
	.../arch/i386/boot/compressed/vmlinux.bin	122929
	/usr/src/linux-2.6.16/.tmp_vmlinux1	85434
	.../linux-2.6.16/arch/i386/boot/vmlinux.bin	56412
	/usr/lib/gcc/i486-linux-gnu/4.0.4/cc1	51311
	.../arch/i386/boot/compressed/vmlinux	50987
	/usr/src/linux-2.6.16/.tmp_kallsyms2.S	37277
	/usr/src/linux-2.6.16/drivers/built-in.o	35138
	/usr/src/linux-2.6.16/.tmp_kallsyms1.S	33097
OSDB		
	/var/lib/mysql/ibdata1	102658
	/root/data/asap.uniques	22765

Continued on the next page

Table D.1 – **continued from last page**

	Files	**Count**
	/tmp/MLxnR9uL	22612
	/var/log/mysql/mysql-bin.000029	20567
	/var/log/mysql/mysql-bin.000028	12095
	/tmp/MLI1FImI	8246
	/tmp/MLv94789	7313
	/tmp/MLfDbN5r	7300
	/lib/libc-2.3.6.so	1665
	/usr/sbin/mysqld	636
SETI		
	.../setiathome-5.12.i686-pc-linux-gnu	12791
	/usr/lib/i686/cmov/libcrypto.so.0.9.8	4259
	/usr/lib/libstdc++.so.6.0.8	3330
	/usr/lib/i686/cmov/libcrypto.so.0.9.7	2982
	/bin/bash	2968
	/usr/sbin/sshd	1236
	/usr/lib/i686/cmov/libssl.so.0.9.8	1167
	/usr/lib/libidn.so.11.5.19	1095
	/lib/tls/libc-2.3.6.so	1090
	/usr/lib/libkrb5.so.3.2	936
Mixed		
	/usr/src/linux-2.6.16/.tmp_vmlinux2	20382
	/usr/src/linux-2.6.16/vmlinux	13265
	../arch/i386/boot/compressed/vmlinux.bin	11365
	/var/lib/mysql/ibdata1	8426
	.../linux-2.6.16/arch/i386/boot/vmlinux.bin	4460
	.../arch/i386/boot/compressed/vmlinux	4460
	/usr/src/linux-2.6.16/.tmp_vmlinux1	4086
	/usr/src/linux-2.6.16/.tmp_kallsyms1.S	2499
	/usr/src/linux-2.6.16/.tmp_kallsyms2.S	2497
	/usr/src/linux-2.6.16/.tmp_System.map	1554

Table D.1: A top ten over most commonly found files in the page cache. The count reflects how many times pages representing a given file is found.

Appendix E

Usage of Inode and Anonymous Pages by Processes

Inode and anonymous pages can be traced back to the processes that uses them. So this appendix lists detailed information about the number of times a shareable page, of either page type, can be traced back to a given process. The tables are limited to show the ten most commonly found processes during the experiments. The information has been placed into two Tables E.1 and E.2. The reader should notice that the results in this appendix is based on an execution time of 90 minutes.

	Anonymous		Inode	
Dbench				
	Process	**Count**	**Process**	**Count**
	dbench	9416	dbench	30894
	bash	353	sshd	9945
	sshd	150	bash	8102
	inetd	120	cron	7768
	getty	106	syslogd	6393
	init	75	init	5313
	dhclient	62	getty	5071
	cron	50	klogd	3970
	klogd	12	inetd	3391
	rm	3	dhclient	3267
Kernel compile				
	Process	**Count**	**Process**	**Count**
	sshd	98	sshd	10067
	init	97	cron	7273
	dhclient	90	syslogd	6066
	cron	89	init	4962
	cc1	72	getty	4636
	inetd	35	klogd	3859
	as	25	dhclient	3218
	make	18	inetd	3180
	sh	5	cc1	607
	gcc	2	make	520
OSDB				
	Process	**Count**	**Process**	**Count**
	osdb-my	6624	mysqld	97274
	mysqld_safe	653	osdb-my	58459
	sshd	413	mysqld_safe	16791
	dhclient	377	sshd	12901
	cron	209	cron	10168
	klogd	41	syslogd	9655
	mysqld	14	logger	6689
	getty	1	init	6609
			getty	6540
			klogd	4769

Table E.1: The table contains a top ten from each experiment of the processes responsible for the most shareable pages. Notice that it goes for all the experiments, that the benchmarking application shows up as the process with the most shareable pages.

	Anonymous		Inode	
SETI				
	Process	**Count**	**Process**	**Count**
	syslogd	102	boinc_client	11234
	inetd	37	sshd	2669
	cron	37	cron	1837
	sshd	34	syslogd	1613
	dhclient	32	setiathome	1609
	init	20	boinc	1425
	boinc	20	init	1237
	klogd	10	getty	1160
			klogd	996
			dhclient	891
Mixed				
	Process	**Count**	**Process**	**Count**
	dhclient	53	sshd	1439
	init	36	mysqld_safe	1217
	cron	18	cron	1153
	dbench	13	syslogd	1006
	cc1	4	init	755
	syslogd	2	getty	748
	make	2	klogd	381
	as	2	inetd	338
			dhclient	333
			mysqld	189

Table E.2: The table contains a top ten from each experiment of the processes responsible for the most shareable pages. Notice that it goes for all the experiments, that the benchmarking application shows up as the process with the most shareable pages.

Appendix F

File names from Shareable Pages used by Processes

The inode pages found during the different experiments have been traced back to the files that they represent in the filesystem. This appendix contains information about the number of times a shareable inode page has been traced back to a given file. It should however be noted that this is only a top ten of those files. The reader should again notice that the results are based on a 90 minute execution time frame.

	Files	**Count**
Dbench		
	/lib/tls/libc-2.3.6.so	40110
	/lib/libc-2.3.6.so	20226
	/lib/ld-2.3.6.so	6670
	/bin/bash	5766
	/usr/lib/i686/cmov/libcrypto.so.0.9.7	2307
	/usr/sbin/sshd	1994
	/usr/bin/dbench	1702
	/sbin/dhclient	904
	/usr/sbin/cron	808
	/lib/tls/libnsl-2.3.6.so	619
Kernel compile		
	/lib/tls/libc-2.3.6.so	17715
	/lib/libc-2.3.6.so	11942
	/lib/ld-2.3.6.so	4221
	/usr/lib/i686/cmov/libcrypto.so.0.9.7	2599
	/usr/sbin/sshd	2022
	/sbin/dhclient	942
	/usr/sbin/cron	805
	/lib/tls/libnsl-2.3.6.so	540
	/usr/lib/gcc/i486-linux-gnu/4.0.4/cc1	487
	/lib/tls/libpthread-2.3.6.so	408
OSDB		

Continued on the next page

Table F.1 – **continued from last page**

	Files	**Count**
	/lib/libc-2.3.6.so	85521
	/usr/sbin/mysqld	78620
	/usr/lib/libmysqlclient.so.12.0.0	26581
	/lib/ld-2.3.6.so	7630
	/bin/bash	6666
	/usr/local/bin/osdb-my	5588
	/usr/lib/libstdc++.so.6.0.8	5516
	/usr/lib/i686/cmov/libcrypto.so.0.9.7	3515
	/lib/libpthread-0.10.so	2370
	/usr/sbin/sshd	2154
SETI		
	/lib/tls/libc-2.3.6.so	5337
	/lib/libc-2.3.6.so	3960
	/usr/lib/i686/cmov/libcrypto.so.0.9.8	2949
	/usr/lib/libstdc++.so.6.0.8	2574
	/usr/bin/boinc_client	2121
	.../setiathome-5.12.i686-pc-linux-gnu	1425
	/lib/ld-2.3.6.so	1052
	/usr/lib/libcurl.so.3.0.0	726
	/usr/lib/i686/cmov/libcrypto.so.0.9.7	712
	/usr/sbin/sshd	572
Mixed		
	/lib/libc-2.3.6.so	3121
	/lib/ld-2.3.6.so	1395
	/bin/bash	1111
	/usr/sbin/sshd	625
	/usr/lib/i686/cmov/libcrypto.so.0.9.7	306
	/usr/sbin/cron	255
	/lib/libnss_files-2.3.6.so	241
	/sbin/dhclient	127
	/lib/libresolv-2.3.6.so	123
	/sbin/getty	90

Table F.1: Top ten of the most frequently found file names gathered from the shareable pages used by processes during the different experiments.

Appendix G

Files used by Processes

This appendix lists the processes that are using a given file e.g. application binaries and standard libraries. To obtaining this information the following steps are performed: 1) The names of the processes using the shareable inode page in question, is located. 2) The file name that the inode page represents is found. 3) These two pieces of information is combined for the finale result. The results of this process are shown in the tables.

File	**Processes**
/lib/tls/libc-2.3.6.so	init, dhclient, syslogd, klogd, inetd, sshd, cron, getty, bash, dbench, rm, sleep, run-parts, sh
/lib/ld-2.3.6.so	init, dhclient, syslogd, klogd, inetd, sshd, cron, getty, bash, dbench, rm, sleep, sh, run-parts
/lib/libc-2.3.6.so	init, syslogd, inetd, sshd, cron, getty, dbench, klogd, bash, dhclient, sleep, rm, run-parts
/lib/tls/libresolv-2.3.6.so	syslogd, sshd
/lib/tls/libpthread-2.3.6.so	sshd, sleep
/lib/tls/libnsl-2.3.6.so	cron, sshd
/lib/tls/libcrypt-2.3.6.so	sshd, cron
/lib/libpthread-0.10.so	sshd, sleep
/lib/libpam.so.0.79	sshd, cron
/lib/libnss_files-2.3.6.so	syslogd, cron
/lib/libncurses.so.5.5	bash, sh
/bin/bash	bash, sh
/usr/sbin/sshd	sshd
/usr/sbin/inetd	inetd
/usr/sbin/cron	cron
../i686/cmov/libcrypto.so.0.9.7	sshd
/usr/bin/dbench	dbench
/sbin/syslogd	syslogd
/sbin/klogd	klogd
/sbin/init	init
/sbin/getty	getty
/sbin/dhclient	dhclient
/lib/tls/libutil-2.3.6.so	sshd
/lib/tls/librt-2.3.6.so	sleep
/lib/tls/libnss_dns-2.3.6.so	syslogd
/lib/tls/libnss_compat-2.3.6.so	cron
/lib/tls/libm-2.3.6.so	sleep
/lib/security/pam_env.so	cron
/lib/librt-2.3.6.so	sleep
/lib/libresolv-2.3.6.so	syslogd
/lib/libnss_nis-2.3.6.so	cron
/lib/libnss_compat-2.3.6.so	cron
/lib/libnsl-2.3.6.so	cron
/lib/libm-2.3.6.so	sleep
/bin/sleep	sleep
/bin/rm	rm

Table G.1: The names of the different processes that are using a given file during the Dbench experiment.

File	Processes
/lib/tls/libc-2.3.6.so	init, syslogd, klogd, sshd, cron, inetd, gcc, cc1, as, dhclient, getty, make, sh
/lib/libc-2.3.6.so	init, syslogd, klogd, sshd, cron, make, sh, dhclient, inetd, getty, gcc, cc1, as
/lib/ld-2.3.6.so	init, syslogd, cron, getty, make, sh, gcc, cc1, as, dhclient, klogd, inetd, sshd
/lib/tls/libresolv-2.3.6.so	syslogd, sshd
/lib/tls/libpthread-2.3.6.so	sshd, make
/lib/tls/libnsl-2.3.6.so	cron, sshd
/lib/libpthread-0.10.so	sshd, make
/lib/libpam.so.0.79	sshd, cron
/lib/libnss_files-2.3.6.so	syslogd, cron
/usr/sbin/sshd	sshd
/usr/sbin/inetd	inetd
/usr/sbin/cron	cron
/usr/lib/libbfd-2.17.so	as
/usr/lib/libbfd-2.16.91.so	as
../i686/cmov/libcrypto.so.0.9.7	sshd
../gcc/i486-linux-gnu/4.0.4/cc1	cc1
/usr/bin/make	make
/usr/bin/gcc-4.0	gcc
/usr/bin/as	as
/sbin/syslogd	syslogd
/sbin/klogd	klogd
/sbin/init	init
/sbin/getty	getty
/sbin/dhclient	dhclient
/lib/tls/libutil-2.3.6.so	sshd
/lib/tls/librt-2.3.6.so	make
/lib/tls/libnss_dns-2.3.6.so	syslogd
/lib/tls/libnss_compat-2.3.6.so	cron
/lib/tls/libcrypt-2.3.6.so	sshd
/lib/librt-2.3.6.so	make
/lib/libresolv-2.3.6.so	syslogd
/lib/libnss_compat-2.3.6.so	cron
/lib/libnsl-2.3.6.so	cron
/lib/libncurses.so.5.5	sh
/bin/bash	sh

Table G.2: The names of the different processes that are using a given file during the Kernel compile experiment.

File	**Processes**
/lib/libc-2.3.6.so	dhclient, syslogd, init, klogd, inetd, mysqld_safe, mysqld, logger, sshd, cron, getty, osdb-my
/lib/ld-2.3.6.so	init, syslogd, mysqld_safe, mysqld, logger, cron, getty, osdb-my
/lib/libnss_files-2.3.6.so	syslogd, mysqld_safe, mysqld, cron, osdb-my
/lib/libnsl-2.3.6.so	mysqld_safe, mysqld, cron, osdb-my
/lib/libdl-2.3.6.so	mysqld_safe, mysqld, sshd, cron
/usr/lib/libz.so.1.2.2	mysqld, sshd, osdb-my
/lib/libnss_compat-2.3.6.so	mysqld, cron, mysqld_safe
/lib/libcrypt-2.3.6.so	mysqld, osdb-my, cron
/lib/libpthread-0.10.so	mysqld, sshd
/lib/libpam.so.0.76	sshd, cron
/lib/libm-2.3.6.so	mysqld, osdb-my
/usr/sbin/sshd	sshd
/usr/sbin/mysqld	mysqld
/usr/sbin/inetd	inetd
/usr/sbin/cron	cron
/usr/local/bin/osdb-my	osdb-my
/usr/lib/libstdc++.so.6.0.8	mysqld
/usr/lib/libmysqlclient.so.12.0.0	osdb-my
../i686/cmov/libcrypto.so.0.9.7	sshd
/sbin/syslogd	syslogd
/sbin/klogd	klogd
/sbin/init	init
/sbin/getty	getty
/sbin/dhclient	dhclient
/lib/security/pam_env.so	cron
/lib/libwrap.so.0.7.6	mysqld
/lib/librt-2.3.6.so	mysqld
/lib/libresolv-2.3.6.so	syslogd
/lib/libnss_nis-2.3.6.so	cron
/lib/libncurses.so.5.5	mysqld_safe
/lib/libgcc_s.so.1	mysqld
/etc/ld.so.cache	sh
/bin/bash	mysqld_safe

Table G.3: The names of the different processes that are using a given file during the OSDB experiment.

File	**Processes**
/lib/tls/libc-2.3.6.so	init, dhclient, syslogd, klogd, inetd, sshd, boinc_client, cron, getty, boinc, setiathome-5.12
/lib/libc-2.3.6.so	boinc_client, boinc, setiathome-5.12, init, syslogd, klogd, inetd, sshd, cron, getty, dhclient
/lib/ld-2.3.6.so	init, dhclient, syslogd, klogd, boinc_client, inetd, sshd, cron, getty, boinc, setiathome-5.12
/lib/tls/libresolv-2.3.6.so	syslogd, boinc_client, sshd, boinc
/lib/tls/libpthread-2.3.6.so	boinc_client, sshd, boinc, setiathome-5.12
/lib/tls/libnsl-2.3.6.so	boinc_client, cron, sshd, boinc
/lib/libpthread-0.10.so	boinc_client, sshd, boinc, setiathome-5.12
/lib/libnss_files-2.3.6.so	syslogd, cron, boinc_client, boinc
/lib/tls/libnss_dns-2.3.6.so	syslogd, boinc_client, boinc
/lib/tls/libm-2.3.6.so	boinc_client, boinc, setiathome-5.12
/lib/libresolv-2.3.6.so	syslogd, boinc_client, boinc
/lib/libm-2.3.6.so	boinc_client, boinc, setiathome-5.12
/usr/lib/libz.so.1.2.3	boinc_client, boinc
/usr/lib/libstdc++.so.6.0.8	boinc_client, boinc
/usr/lib/libkrb5.so.3.2	boinc_client, boinc
/usr/lib/libk5crypto.so.3.0	boinc_client, boinc
/usr/lib/libidn.so.11.5.19	boinc_client, boinc
/usr/lib/libgssapi_krb5.so.2.2	boinc_client, boinc
/usr/lib/libcurl.so.3.0.0	boinc_client, boinc
/usr/lib/i686/cmov/libssl.so.0.9.8	boinc_client, boinc
../i686/cmov/libcrypto.so.0.9.8	boinc_client, boinc
/usr/bin/boinc_client	boinc_client, boinc
/lib/libpam.so.0.79	sshd, cron
/lib/libnsl-2.3.6.so	boinc_client, cron
/usr/sbin/sshd	sshd
/usr/sbin/inetd	inetd
/usr/sbin/cron	cron
/usr/lib/libkrb5support.so.0.0	boinc_client
../i686/cmov/libcrypto.so.0.9.7	sshd
/sbin/syslogd	syslogd
/sbin/klogd	klogd
/sbin/init	init
/sbin/getty	getty
/sbin/dhclient	dhclient
../setiathome-5.12.i686-pc-linux-gnu	setiathome-5.12
/lib/tls/libutil-2.3.6.so	sshd
/lib/tls/libnss_compat-2.3.6.so	cron
/lib/tls/libcrypt-2.3.6.so	sshd
/lib/libnss_compat-2.3.6.so	cron
/lib/libcom_err.so.2.1	boinc_client

Table G.4: The names of the different processes that are using a given file during the SETI experiment.

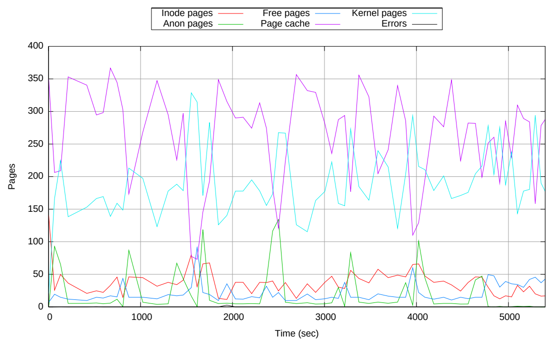
File	**Processes**
/lib/ld-2.3.6.so	init, mysqld_safe, mysqld, logger, getty, osdb-my, syslogd, cron, make, sh, gcc, cc1, as
/lib/libc-2.3.6.so	init, syslogd, klogd, inetd, cron, sshd, getty, dhclient, dbench
/lib/libnss_files-2.3.6.so	syslogd, mysqld_safe, mysqld, cron, osdb-my
/lib/libnsl-2.3.6.so	mysqld_safe, mysqld, cron, osdb-my
/usr/lib/libstdc++.so.6.0.8	mysqld, boinc_client, boinc
/lib/libnss_compat-2.3.6.so	mysqld_safe, mysqld, cron
/bin/bash	mysqld_safe, sh, bash
/lib/librt-2.3.6.so	mysqld, make
/lib/libm-2.3.6.so	mysqld, osdb-my
/lib/libcrypt-2.3.6.so	mysqld, osdb-my
/usr/sbin/sshd	sshd
/usr/sbin/inetd	inetd
/usr/sbin/cron	cron
../lib/i686/cmov/libcrypto.so.0.9.7	sshd
/sbin/syslogd	syslogd
/sbin/klogd	klogd
/sbin/init	init
/sbin/getty	getty
/sbin/dhclient	dhclient
/lib/libresolv-2.3.6.so	syslogd
/lib/libpthread-0.10.so	sshd
/lib/libncurses.so.5.5	mysqld_safe

Table G.5: The names of the different processes that are using a given file during the Mixed experiment.

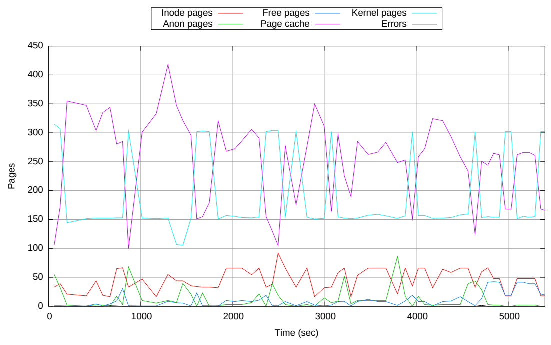
Appendix H

Page Scanning Intervals

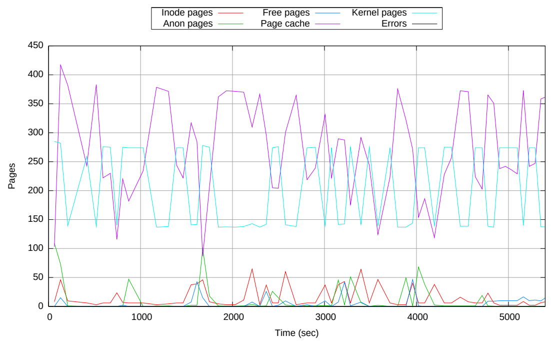
This appendix contains graphs that shows the frequency of page addresses being put in the shared buffer during the different experiments. If the time between two scans are large, the graphs will have straight lines. The interval between scans during the experiments influences the quality of the evaluation, because the data set collected is small. This however does not mean that the data collected is useless. Each figure in this appendix is divided into four graphs one for each Virtual Machine (VM) that are running during the experiments. These graphs contains information about the number of: inode pages, anonymous pages, free pages, pages in the page cache, kernel pages, and the number of errors found during the analysis.



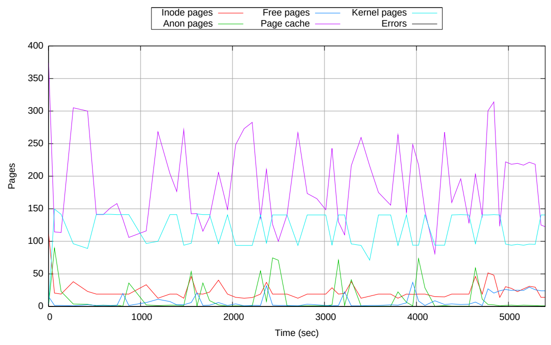
(a) VM1



(b) VM2

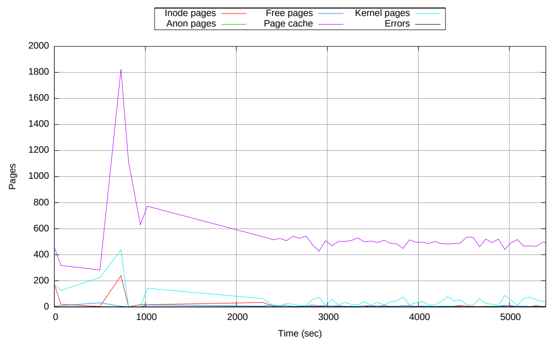


(c) VM3

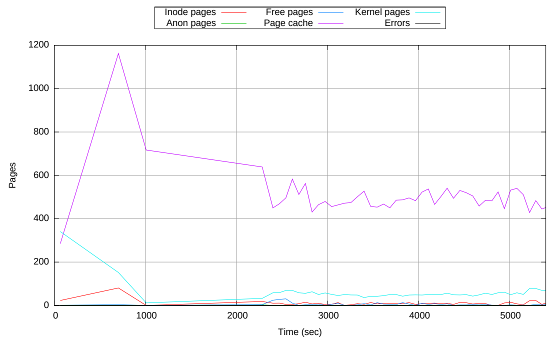


(d) VM4

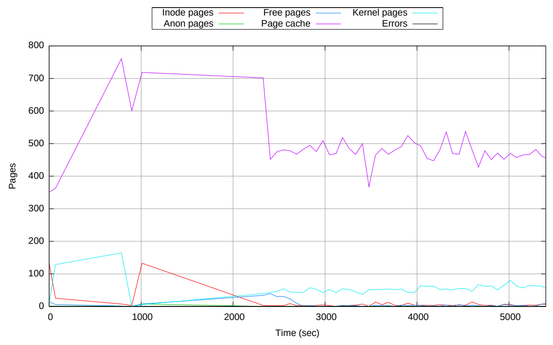
Figure H.1: The scanning interval, for the four different virtual machines, during the Dbench experiment.



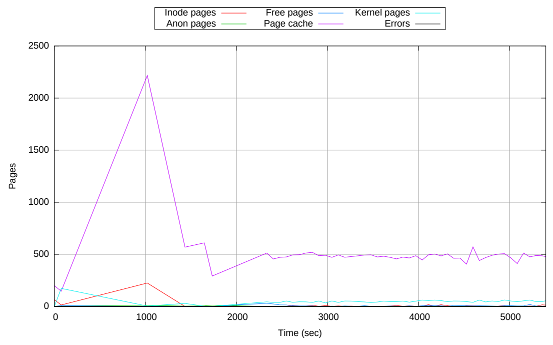
(a) VM1



(b) VM2

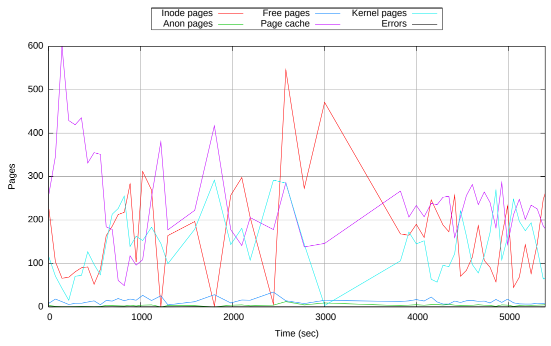


(c) VM3

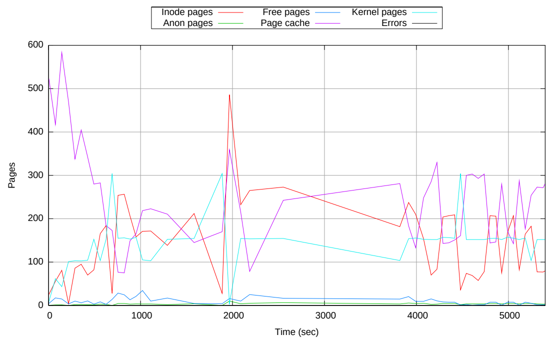


(d) VM4

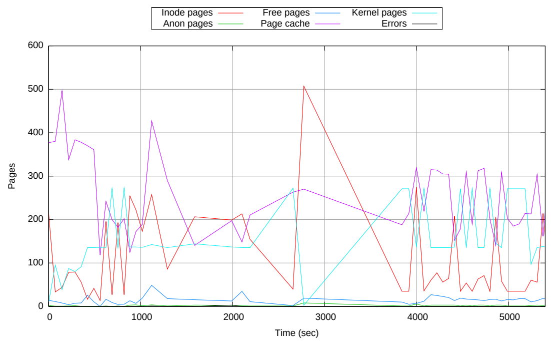
Figure H.2: The scanning interval, for the four different virtual machines, during the Kernel compilation experiment.



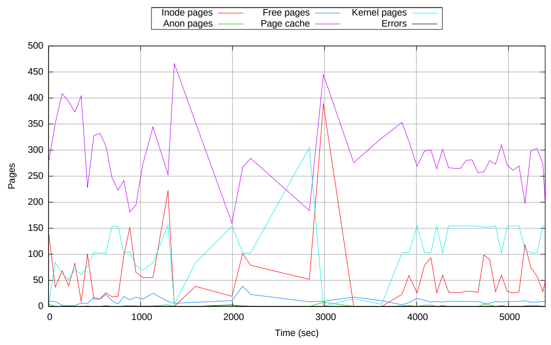
(a) VM1



(b) VM2

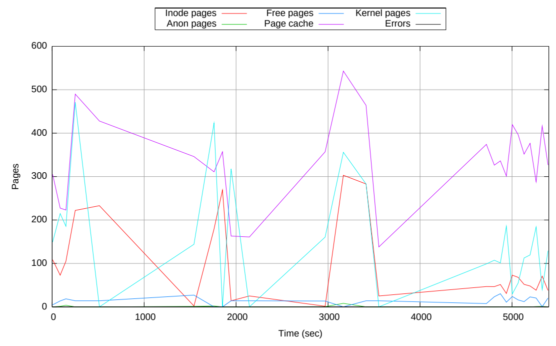


(c) VM3

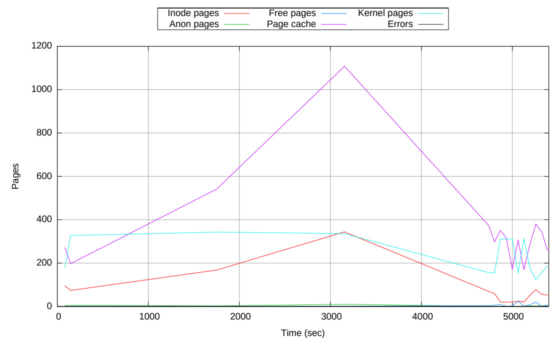


(d) VM4

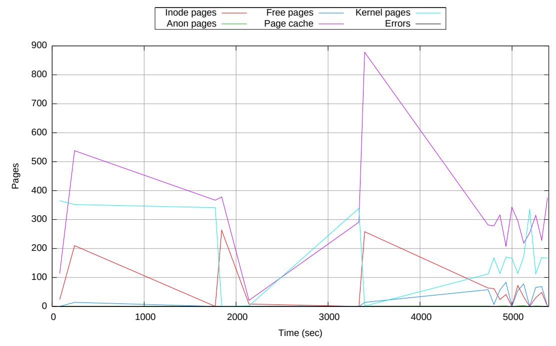
Figure H.3: The scanning interval, for the four different virtual machines, during the OSDB experiment.



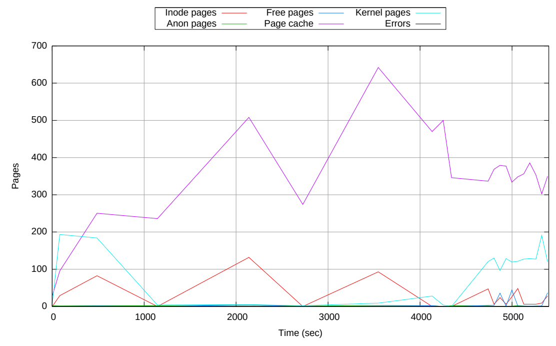
(a) VM1



(b) VM2

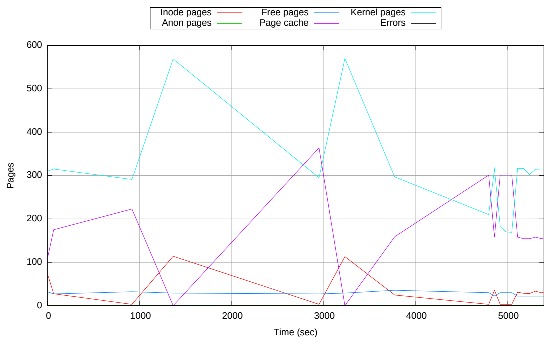


(c) VM3

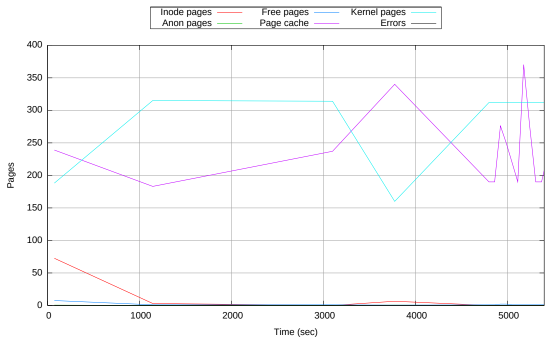


(d) VM4

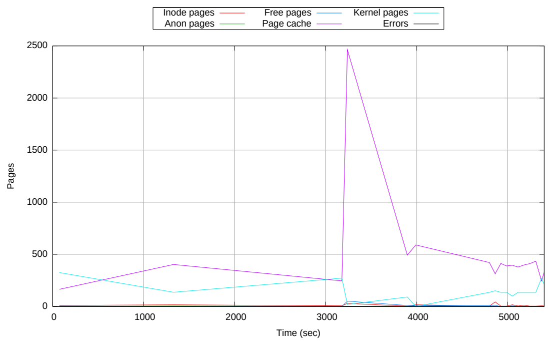
Figure H.4: The scanning interval, for the four different virtual machines, during the SETI experiment.



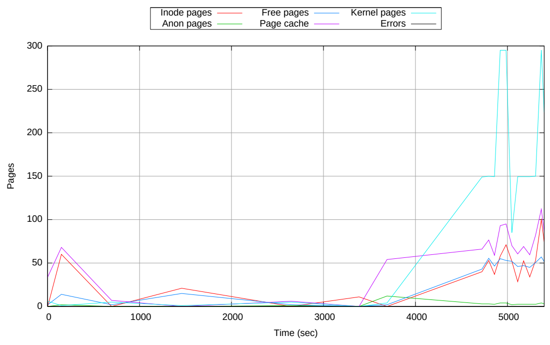
(a) VM1 is running OSDB



(b) VM2 is running SETI



(c) VM3 is running Kernel compile



(d) VM4 is running Dbench

Figure H.5: The scanning interval, for the four different virtual machines, during the Mixed experiment.