On the Feasibility of Memory Sharing

– Content-Based Page Sharing in the Xen Virtual Machine Monitor.

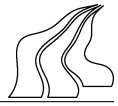
Jacob Faber Kloster
Jesper Kristensen
Arne Mejlholm

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

at

Department of Computer Science,
Aalborg University
June 2006

Department of Computer Science
Aalborg University



Title:
On the Feasibility of Memory Sharing:
–Content-Based Page Sharing in the
Xen Virtual Machine Monitor

Semester:
Dat6
1. February 2006 -
14. June, 2006

Group:
D615a, 2006

Members:
Jacob Faber Kloster
Jesper Kristensen
Arne Mejlholm

Supervisor:
Gerd Behrmann

Copies: 7

Report – pages: 114

Appendix – pages: 18

Total pages: 132

Synopsis:

This thesis evaluates the feasibility of doing page sharing between virtual machines. To do this evaluation we proposed two different designs: One that is transparent to the guest operating system and a paravirtualized one. We implemented one of these based on Potemkin, which is a modification of the Xen virtual machine monitor. In this we find pages eligible for sharing by use of a technique called content-based page sharing. When identical pages are found, the actual sharing of pages is obtained using shadow page tables and Copy-on-Write. Finally the implementation was evaluated and we found no significant overheads except for the use of shadow page tables. The paravirtualized design should mitigate this overhead.

As for the feasibility of page sharing, we carried out a series of experiments. These explored sharing under good and bad conditions as well as under synthetic workloads. We concluded that page sharing is feasible and that most workloads that have similar kernels and applications have something to share.

Keywords: VM, VMM, hypervisor, Xen, Potemkin, VMware ESX Server, virtualization, paravirtualization, memory sharing, overcommitment, flash cloning, content-based page sharing, compare-by-hash, SFH, Copy-on-Write, CoW, shadow page tables, ballooning, reverse mapping, zero pages, benchmarks.

Preface

This thesis continues our work from the last semester[33] and contains all new contents, except where it is explicitly stated else. We would like to thank Jacob Gorm Hansen and the Xen team, in particular Keir A. Fraser, for answering our questions regarding Xen. A great thanks goes to Michael Vrable for providing us with the Potemkin source code and answering our questions; this has been an invaluable inspiration for our implementation effort. Finally we would like to thank Josva Kleist and Gerd Behrmann for providing us with access to the experimental cluster, so we had sufficient machines to implement and evaluate on.

The reader should be aware that we make use of three different ways of doing citations. 1) If a citation is placed directly after a word, then the citation refers to the word. 2) If the citation is at the end of a line, but before the period sign, then the citation refers to the whole line. 3) Finally if the citation is at the end of a paragraph behind the period sign, then the citation refers to the whole paragraph.

It should be noted that we make extensive use of abbreviations (often just the ones used by Xen). By our own experience we know that this can be confusing, so we provide a glossary list with abbreviations in Appendix A on page 115.

Jacob Faber Kloster

Jesper Kristensen

Arne Mejlholm

Contents

Chapter 1

Introduction

The layout of the thesis is as follows: In Chapter 2 we start by introducing the terminology needed for the thesis. Then we argue that it is feasible to explore memory sharing between virtual machine. With these arguments we finally state the goal and limitations of the thesis. In Chapter 3 we analyze related work on doing memory sharing and compare these with each other.

Having explained what the thesis will examine, we use Chapter 4 to introduce important memory management and virtualization techniques. These will be referred to throughout the rest of the thesis. As the necessary techniques have been introduce, we then present our two different designs in Chapter 5. We explain that the implementation will be using one of the approaches explained in Chapter 3, namely content-based page sharing. The two designs differ in the approach needed to create the actual sharing of memory. One is a modification to the operating system running inside the virtual machine, the other uses a transparent approach to change the memory mappings without a given virtual machine knowing about it. In Chapter 6 we first explain the implementation from an overall point of view and then elaborate on selected details.

Having covered the design and implementation, we finally have the means for evaluating memory sharing. This is done in Chapter 7, where we evaluate the feasibility of sharing memory. This is where things get interesting, as we are able to experiment with different workloads and see how much memory can be shared. Finally we compare our results with the results reported by the other memory sharing approaches that were covered in Chapter 3. After experimenting with the implementation we, in Chapter 8, evaluate how efficient the implementation is. This is performed on an overall level and through a series of micro benchmarks.

Finally in Chapter 9 we conclude the thesis and explain how the research can be continued.

Chapter 2

Motivation and Goal

We start this chapter by explaining the terminology and virtualization concepts needed to outline the goal of the thesis. For a more thorough explanation of the terms we refer the reader to our original report[33], which contains a comprehensive summary of much of the virtualization literature.

2.1 Virtualization Terminology

The ability to do *virtualization*[24],[45] is based on two key concepts: A *Virtual Machine Monitor* (VMM) and a number of *Virtual Machines* (VMs) each running a *guest Operating System (OS)*. A system comprised of these concepts is referred to as a *Virtual Machine System* (VMS).

The VMM controls the hardware interface of the underlying machine and exports a virtual abstraction of this to the VMs, or the VMM arbitrates access to the hardware to ensure safety of the VMS to put in another way. The VMs are thus in effect just a safe software representation of the actual hardware. If the VM representation of the hardware differs from the actual hardware, then the differences must be translated at runtime by the VMM, thus creating a performance overhead. On the other hand, the changes in the VM representation can also lead to a performance boost, e.g. when changing the semantics of the `hlt` instruction to yield the VM's control of the processor [60]. It is a balancing act finding the right set of changes to ensure good performance. However as a rule of thumb, it makes sense to keep the virtual representation as close to the underlying hardware as possible without sacrificing the safety of the system.

Key features of VMSes are that the VM abstraction ensures good isolation between different VMs [39]. The very point of the VM abstraction is to ensure that the VM cannot abuse the real machine, so as long as the VMM is correctly implemented, then there is per definition isolation between the VMs. Some developers from the popular open source VMM Xen, even went as far as implying that virtualization achieved many of the goals of microkernels[37],[51], while retaining the Application Binary Interfaces (ABIs) of commodity operating systems [27].

Traditionally there have been two different architectures for a VMM. [23] referred to these as either Type I or Type II VMMs. Figure 2.1 on the next page pictures a Type I VMM, which interacts directly with the hardware within its own protection

domain. A Type II VMM was typically embedded within an existing OS, thus making it able to take advantage of such things as the OSes device drivers. The disadvantage to this approach is that the VMM risks being affected by the failure of the host OS.

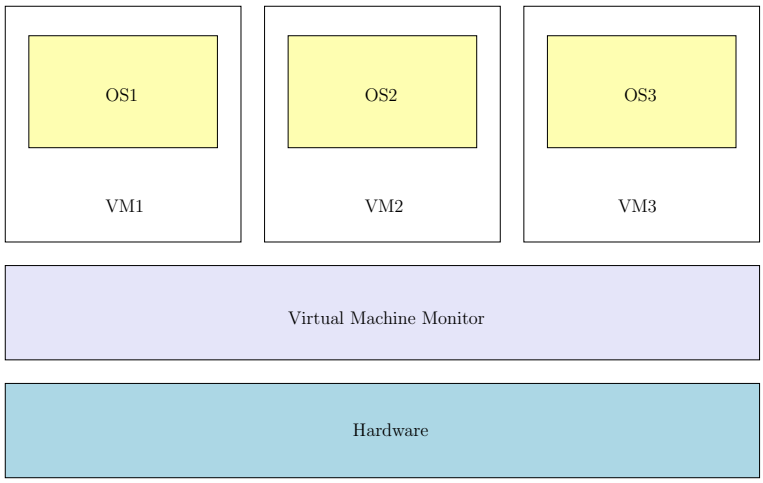


Figure 2.1: Traditional Virtual Machine Monitor Architecture.

Most processor architectures are not easily virtualized [41],[44],[26] because their instruction sets include *sensitive instructions*. The processor distinguishes between a number of modes of privilege (four on the IA-32), which determine which instructions can be executed. Within a normal OS running on a IA-32 processor, the kernel runs in the most privileged mode (ring zero) while applications are run within the least privileged (ring three). To virtualize the processor, the VMs are not allowed to run in the most privileged mode as this would enable them to modify any information and thus sacrifice the safety of the system. Instead the VMs are executed in the second most privileged mode (ring one on IA-32) and the most privileged mode is reserved for the VMM. Executing an OS, that normally assumes to be operating at the most privileged mode in ring one, introduces a number of problems. Some parts of the OS may rely on using certain instructions, which must be run in the most privileged mode. Most of these instructions, when executed at a lesser level of privilege than required, causes a *trap*. If the set of instructions all do this, then there is no problem, because a switch to the VMM can be done and the instructions can be replaced with instructions that are equivalent or the VMM can do the work on behalf of the VM. Virtualization based on this simple scheme is referred to as *full virtualization*. Full virtualization is preferable as it lets OSes run unmodified, but most processors do not have an instruction set that can fulfill the requirements for full virtualization. In most processors there are instructions that instead of causing a trap, fail silently. These operations must be dealt with separately and are referred to as sensitive instructions. The realization of this problem can be directly attributed to [41], which discusses the problem in more detail.

The current trend for hardware manufacturers is to support virtualization at the chip level[55],[3, p. 27-29]. In particular they have eliminated the sensitive

instructions that make the IA-32 architecture non-natively virtualizable [36, p. 12]. Thus these are relying on trapping privileged instructions using full virtualization.

VMware [58],[49],[46] uses an approach called binary translation to detect the sensitive instructions. They parse the binaries at runtime and create a safe representation to be executed. As this is an expensive operation the representation is cached to speed up subsequent executions.

2.1.1 Xen

Xen[5],[42],[19] is a Type I VMM, which uses a different approach to virtualization. This approach is to provide a virtual hardware abstraction to the VMs that is different from the actual machine hardware. Although the approach had been used before, it was [60] which named it paravirtualization. Instead of handling sensitive instructions at runtime, the Xen approach is to modify the guest OS to avoid these instructions, thus avoiding the overheads of evaluating each instruction to replace it or rely on trapping the sensitive instructions.

Xen uses a terminology that is different from the usual terminology. Commonly a VM is referred to as a *domain* and we use the terms interchangeably in the thesis. Therefore we also refer to concepts related to more than one domain as *interdomain*, e.g. interdomain sharing and communication. Although the VMM is fully isolated, it moves much functionality, such as creating a new domain, into a special privileged domain (called domain-0 or often just dom0).

The guest OSes running within Xen are executed from a less privileged mode of execution (ring one instead of the usual ring zero). As ring zero is usually referred to as running in *supervisor* mode, Xen has kept this terminology for guest OSes and applied a new term for the VMM's mode of operation, namely *hypervisor* mode. Therefore the Xen community often refers to Xen as a hypervisor instead of a VMM.

Throughout the thesis we will have to address the operation of switching between the VMM and the VMs. The VMs communicate with the VMM through *hypercalls*, which are clean interfaces exported by the VMM (not unlike system calls within an ordinary OS). When the VMM needs to communicate with the VMs it primarily happens through *events* and upcalls, which in effect are just virtual interrupts forwarded to a VM. Whenever a switch from the VMM to a VM, or the other way around, happens we refer to it as a *hyper switch*. A switch from a VM to another, is referred to as a *world switch*. The task of switching between processes within an OS is, as ordinarily, referred to as a context switch and is only used in this exact meaning.

2.2 Motivation

Having introduced the necessary terminology, we now address the goal of the thesis.

It goes without saying that practically any resource in a computer, virtual or real, is in demand. Everything must be cheaper, larger and faster. Memory is no exception and in a VMS more memory is always applicable. An example of this is UnixShell#[56], an Internet company that is specialized in offering VMs for rent. Their smallest offer is a VM with 64 MB of memory, running on a 8 GB physical machine. This allows as much as 128 machines to run on one server. One of the

most prominent applications of their product, is web hosting, where the customer has entirely free hands to set up the server as he pleases. As a basic service the company provides a number of minimal Linux distribution configurations, where the OS is already set up. Now, as the VMs in their data center run from the same basic configurations and probably some of the same applications, there is a rather significant probability of redundancy, meaning that they have loaded the same data into memory.

If the data is identical, then there is no reason not to reduce multiple private copies into one shared copy. This operation (referred to as *reclaiming* memory) will free up some amount of memory, that can then be used for various purposes, for instance to start additional VMs as well as adjusting the current allocation of memory to each individual VM[1].

This introduces the risk of using more memory than there is available in the machine, this is called *overcommitment* of memory. When the shared copies are altered, a private copy is once again needed. Policies to resolve the situation when the system is overcommitted and there are no free pages, is of course needed. VMware resolves to swapping pages to disk from the VMM [58].

Starting the thesis we had some concerns about whether it would be feasible to share memory. The first concern was if there actually was enough data to share to make it worthwhile. Also a concern was that if the data identified to be shareable changed too fast, then the operation of sharing the pages, just to unshare them soon after would present an overhead. Therefore we carried out a number of initial experiments to examine whether exploring memory sharing actually is worthwhile. We note that the experiments were only conducted to motivate us; they should not be considered representative of general workloads or even be conclusive.

The remainder of this chapter is divided as follows. In Section 2.3 we present the results of the experiments that we have conducted to investigate if there is potential for sharing memory between VMs. Specifically there are three sets of experiments: 1) Changes in memory, 2) shares within a single OS and 3) shares between VMs. The first experiment addresses the concern of whether the memory in an OS changes too rapidly to make it worthwhile to share memory. As for the second, from reading [58], we had a suspicion that even within a single OS there would be redundant copies of memory. The second experiment addresses this. Finally the last experiment examines whether there is anything to share between different VMs. Having presented the results of the experiments, we state the goal of the thesis and a set of limitations.

2.3 Initial Experiments

The non-idle experiments were performed using SPECweb99[18] to generate a workload on the server. SPECweb99 is specifically designed to exercise the web server with both static as well as dynamic web pages. The machine used during the experiments is a 700 MHz AMD Athlon with 64 MB of memory running Debian 3.1 with Apache 2.0.55. We chose to limit the server to 64 MB as we deemed this to

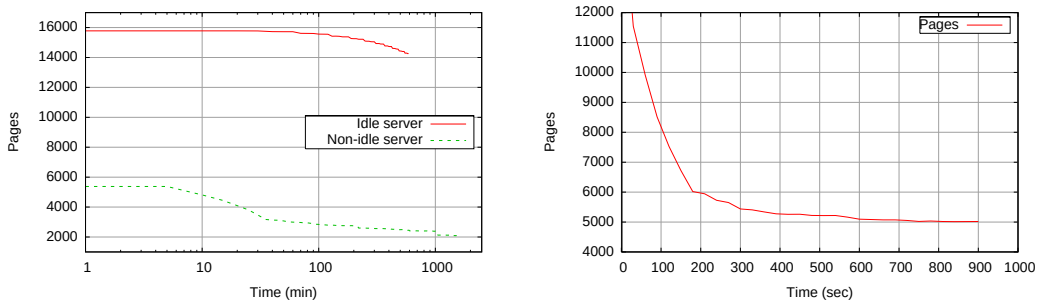
[1] This is however not as simple as it sounds. We explain a technique to do this in a simple manner in Section 4.5 on page 33.

be the minimum realistic size of a web server running in a VM. Furthermore the SPECweb99 benchmark kept a continuous number of 20 connections to the web server.

2.3.1 Changes in Memory

Our first concern was that the contents of memory changes so rapidly that trying to share it, is not feasible. To address this concern we analyzed how memory changes over time on both an idle and a non-idle server.

To ensure that the experiments themselves influenced the memory as little as possible, we took snapshots of the memory using a forensics tool[2] and sent them over a local area network to a remote machine. The reason for doing this is that we do not want the snapshots to end up in the page cache[3], which could happen if it was saved to disk. If simply saved to disk, the next snapshot would reflect the previous snapshot because the memory would contain deferred disk writes that contain parts of the snapshot. Using this approach we cannot ensure that the page cache is not used at all, but we can ensure that it is kept to a minimum. To analyze the snapshots we used a technique called compare-by-hash[4].



(a) Long Run

(b) Short Run

Figure 2.2: Unchanged pages in memory on both idle and non-idle machines. Figure 2.2(a) shows dumps every five minutes for 20 hours. In Figure 2.2(b) we examine the first 15 minutes after the benchmark started, on a non-idle machine with dumps every 30 seconds. Notice that the left graph is in logarithmic scale.

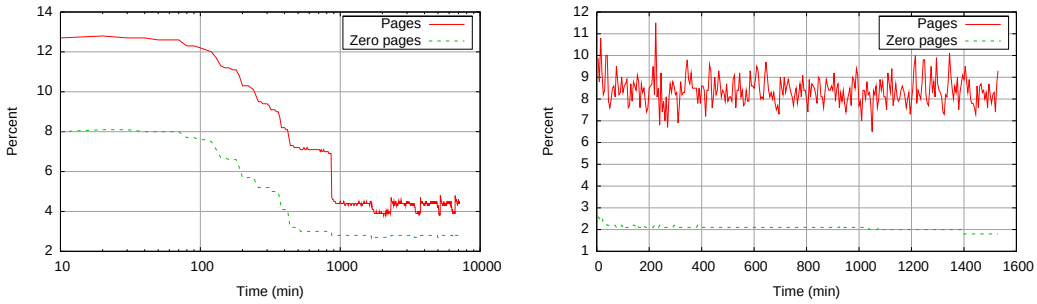
A bit of explanation is in order, as in Figure 2.2(a) we cannot quite tell how fast the memory is changing within the first minutes. As this is a rather interesting detail, we provide an equivalent run in Figure 2.2(b). From the figure we can see that this workload changes roughly 2000 pages per 50 second and that this happens linearly until it starts to flatten out.

The fact that after half an hour 19% of memory still remains unchanged, even with a small memory size and a high workload, suggests that the problem about memory changing too fast described above, not really is a problem to worry about.

[2]Memdump http://www.porcupine.org/forensics/memdump-1.01.README.
[3]The page cache is the general page sized disk cache in Linux [38, cha. 15].
[4]If not familiar with this technique, we explain it in Section 3.1 on page 19.

2.3.2 Shareable Pages Within a Single OS

As explained, we also wanted to see how much memory could be shared inside a single OS. Again in this experiment we also used compare-by-hash to analyze the memory. A *zero page* is a page that is filled with zeros.



(a) Idle (b) Non-idle

Figure 2.3: Possible shares within a single OS. The dotted line indicates how larger a percentage of the possible shares are zero pages.

As can be seen from Figure 2.3, the idle machine steadily uses its zero pages, while the non-idle has consumed its zero pages almost from the start.

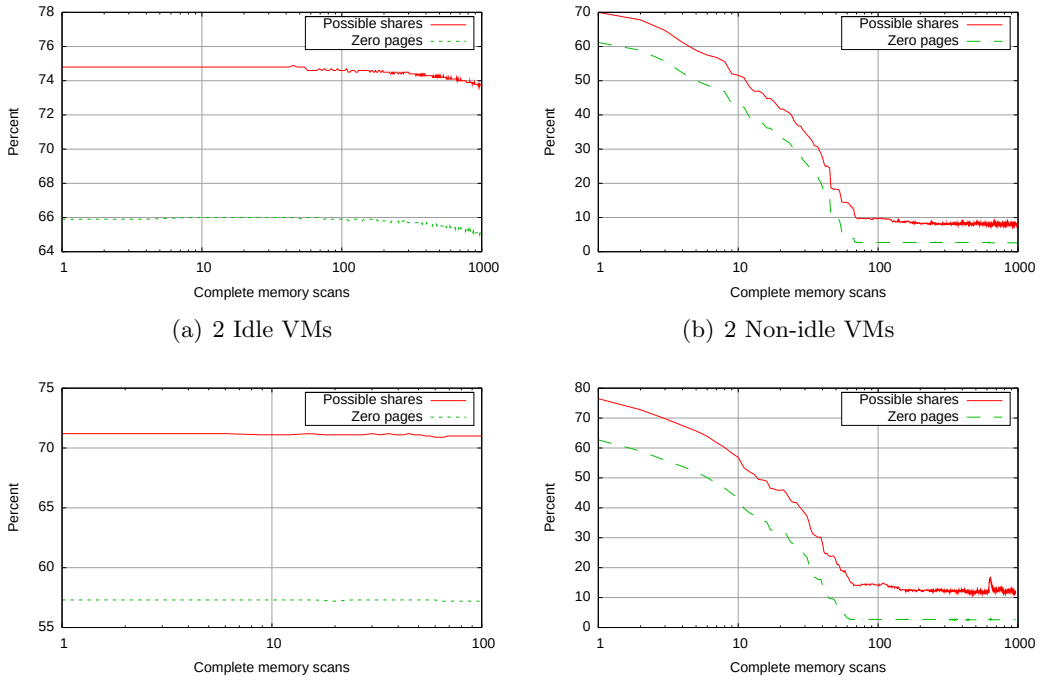
The interesting information in the figures is that there actually is something to share within a single OS. The idle machine starts out with a large percentage of shares, but then as time passes the amount of shares steadily drops. As for the non-idle experiment the number of possible shares is more stable, oscillating between 8-9%. We were surprised that there would be so much to share within a single OS, so this alone almost gave us the motivation we needed.

2.3.3 Interdomain Shareable Pages

Finally to examine how much memory can be shared between VMs, we carried out the following experiment. Contrary to the previous experiments, this was carried out by a modification to Xen[5]. We have constructed a function within the VMM, which hashes the contents of pages belonging to VMs and sends a report of the number of potential shares via a serial line to a remote machine. Contrary to the workloads in the previous experiments, we reduced the load on the web server in this experiment to one continuous connection.

The results of the experiment is presented in Figure 2.4 on the next page. As can be seen a very high percentage of the pages can be reclaimed in the idle experiment, mainly because of the high percentage of zero pages. As for the non-idle experiment we get a more realistic percentage of possible shares with very few zero pages. As can be seen from the figure, about 10% of the memory allocated to the VMs can be

[5]This is actually just a part of the implementation described later in the thesis. Originally we used the VMM to suspend a number of VMs causing the contents of any given VM's memory to be written to a disk image and then compare the VMs based on the saved images. As the implementation of the save function at that time did not seemed reliable, we performed the experiments again with our implementation. The results from both experiment sets do however roughly match each other.



(a) 2 Idle VMs

(b) 2 Non-idle VMs

(c) 3 Idle VMs

(d) 3 Non-idle VMs

Figure 2.4: Possible shares between domains over a number of iterations. In the idle experiments the zero pages account for a very large percentage of the possible shares. In the non-idle the zero pages are quickly consumed.

shared under a workload. Compared to the result of 8-9% in the last experiment, this does not seem like much. We remind the reader that the benchmark run during this experiment only had one connection, so potential shares due to the web server itself could very well be reduced due to the drop in sheer numbers of running processes.

These three initial experiments have convinced us that there is not only something to share between VMs, there is even something to share with a single VM. Furthermore the contents of pages does not seem to change too rapidly to make it feasible to share pages. Thus we got the motivation we needed to explore memory sharing.

2.4 Thesis Goal

As the previous sections indicated, there is a potential for sharing memory between VMs. To fully explore this scenario, we will make an implementation of a memory sharing scheme and conduct a number of experiments on sharing memory. Formalized the goal of the thesis is to:

> Study existing techniques for doing memory sharing between virtual machines. This should be used to design and implement a solution that has little performance overhead. Finally the solution should be evaluated and compared to existing solutions.

2.5 Limitations

The basis for the implementation will be Xen version 3.0.2 and we limit it to modifying Xen and the XenLinux kernel. We will try to keep as much of the implementation architecture independent, but should we face the situation where architecture dependent code is needed, then we will favor the IA-32 architecture without Physical Address Extension (PAE) support. We do this primarily because this is what Xen was primarily developed for and this is the only type of architecture we have access to during the development.

Furthermore we will not in our design be addressing sharing the contents of software managed caches as another subproject of Xen is researching this at the time of writing. We discuss this brifly in Section 3.3 on page 21.

2.6 Summary

This chapter started by explaining general virtualization concepts and gave a quick introduction to Xen. Having established this, we then argued that examining memory sharing between virtual machines would be worthwhile. To convince ourselves and the reader, we carried out a number of initial experiments to investigate if this was the case. In particular we examined whether memory changes too fast to make sharing memory unfeasible. We found that this was not the case. Furthermore we examined whether there would be anything to share, both inside a single virtual machine and between multiple domains. From the experiments we found that the workloads we examined had memory to share. With this insight, we convinced ourselves that making an implementation to experiment with this would be worthwhile. In the next chapter we will be investigating related work on memory sharing to find a suitable technique for the implementation.

Chapter 3

Related Work

This chapter summarizes previous work done on sharing memory between Virtual Machines (VMs). We note that this chapter is almost a direct excerpt from the first thesis and has only been modified lightly.

The chapter is composed as follows, we first introduce necessary concepts, then we summarize related work and finally we evaluate advantages and disadvantages to the different approaches. There are two conceptual approaches to sharing memory between VMs: 1) Use prior knowledge about certain blocks of data being identical or 2) actively compare the blocks by contents to find identical blocks of data.

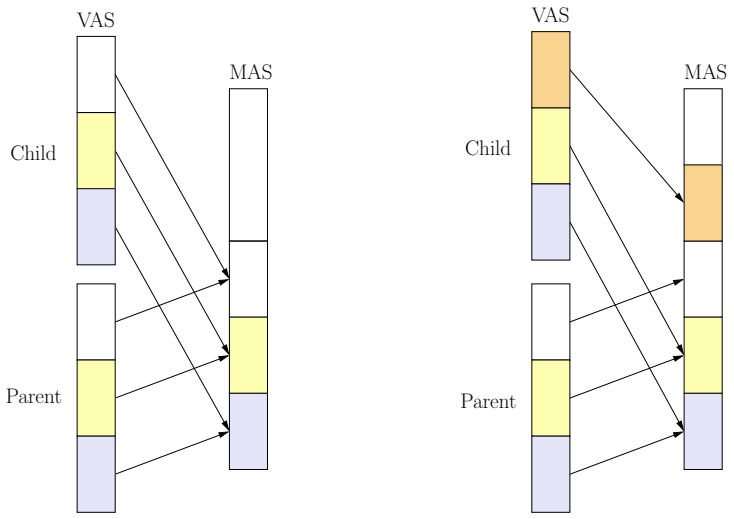
3.1 Compare-By-Hash

A technique called *compare-by-hash*[28],[29] is used to do fast comparisons of blocks of data. Using this technique, a given set of blocks can be compared efficiently by producing a hash value of each data block. The hash values are organised in a hash table and if there are hash values that collide, then there is a good probability that the two involved data blocks are identical. This is efficient compared to a naive approach, which would be to compare each block of data with every other block, thus yielding a complexity of $O(n^2)$.

An example of good use of this technology is rsync[1], which can be used to synchronize entire file directory structures. However the authors of the articles note that the technique can perform worse than the naive approach or cause data loss, if applied incorrectly.

In [29] they list some considerations that could indicate that the use of compare-by-hash is not correctly applied. One of the most important advises are that the technique should not be used as the only means to ensure correctness of data. Furthermore as hash functions are discovered to be unsecure at a large scale, the hash values should only be used temporarily i.e. thrown away after use.

[1] http://rsync.samba.org/.



(a) Sharing (b) Sharing with update

Figure 3.1: Simplified explaintion of Copy-on-Write. On the left picture both the parent and child have identical Virtual Address Spaces (VASes) mapping into the Machine Address Space (MAS). On the right picture an update has been performed by the child and a new page has been created.

3.2 Copy-On-Write

Copy-on-Write (CoW), first introduced for memory sharing in the TENEX system in [7], is a *lazy optimization* technique generally applicable within computer science. As our focus is on memory, we will explain it from this point of view.

The concept of CoW is to manage duplicate data objects by not creating identical copies, but instead giving a pointer to a shared data object. Updates to data objects shared in this manner must be intercepted to make sure that no undesired side-effects occur. When an update occurs then the reference to the shared copy is discarded and a new private object is created. Associated with each shared data object is a reference count, so the object can be removed when it is no longer needed.

The technique is for instance used in the Linux kernel when forking processes as pictured in Figure 3.1. Each process has a Page Table (PT) that translates from a virtual address space into a machine address space. Instead of duplicating the pages of the process, the child process is given a PT where the entries point to pages allocated to the parent process. The pages are marked read-only, so an update will trigger a write-fault. The kernel then intercepts the fault and creates a new page that is not write protected, so that it can be updated by repeating the original write operation. Further details can be found in [50, p. 239-241], [25, p. 87] and [38, p. 31].

Having discussed sharing of pages within an Operating System (OS) as a basis for understanding CoW, we now move beyond a single OS and consider how sharing of memory is done between VMs. Thus from now on, when we discuss sharing pages, unless explicitly said otherwise, we are referring to interdomain shared pages.

3.3 Interdomain Shared Cache

The Disco system[10] introduced the concept of *transparent page sharing*. It provided a level of abstraction over physical memory and was able to share pages by identifying identical pages. It did this by intercepting all access from VMs to block devices and use this to maintain an interdomain shared cache. Thus it was able to share the data from all accesses to a shared devices.

This idea is used in an ongoing research project by Mark Williamson called XenFS[61],[62], which also aims to provide an interdomain shared cache for Xen. The approach is to modify the OS to use a front-end driver to request that a page be brought into memory by a back-end driver. When a different VM request an identical page it is given a reference to the page already in the cache. Currently, as far as we can tell, identical pages are identified solely by the file handle. Whether a content based approach is being considered remains unknown to us.

Finally a content based approach to block devices has been explored. The first [48], was a student project that used hashing to search for identical pages by analyzing all pages brought into memory. Unfortunately they were not able to finish the project and thus no results on sharing were provided.

3.4 Content-Based Page Sharing

VMware introduced[2] *content-based page sharing* in [58]. The concept is based on the compare-by-hash technique as described in Section 3.1 on page 19.

Contrary to Disco's approach, which uses prior knowledge to identify identical data, this approach is based on a service that compares pages in memory at runtime. The comparison is done by using a hash function to index the contents of every page. If the hash value of a page is found more than once, then there is a good probability that the current page is identical with the page that gave the same hash value. VMware ESX server uses a 64 bit hash function[32] to index the pages. To be certain that the pages are identical, the pages are compared bit by bit. If identical pages are found, then the pages are reduced to one page using CoW sharing.

Returning to the concerns raised in [29], as the authors also noticed, content-based page sharing is one of the good applications of compare-by-hash. If hash collisions occur without the pages being identical, then the pages are just not considered for page sharing, thus removing the concern for collisions. As the pages are compared bitwise, we do not rely on the hash values for correctness of the system. Furthermore as hash values are thrown away after use, then the correctness of the system is not coupled to one certain hash function.

The content-based page sharing principle was also implemented as a patch for the Linux kernel in mergemem[43] demonstrating that there are duplicate pages to eliminate within a single OS. Quoting [40], a security issue was discovered in the sharing scenario:

> A hostile process tries to guess the content of a confidential page by creating a set of arbitrary pages containing some guesses. An authorized

[2] Although the author does not directly take credit for the technique in the article, another article by VMware researchers [46] claim that it was VMware that introduced content-based page sharing.

> process merges one of those pages with the confidential page. The sharing of the two pages is not directly visible to the hostile process. But modifying a shared page takes much longer, because it causes a copy-on-write page fault. [40]

The same problem applies to sharing memory between different VMs. Depending on the implementation of the sharing scheme[3], the OS can construct arbitrary pages, wait a sufficient amount of time and write to the page. Exploiting the vulnerability does, however, become more tedious than with the mergemem scheme, as in an OS the attacker is able to tell which processes are running and then direct the attack. In the Virtual Machine System (VMS) scheme this becomes much harder and attacks will have to be launched in the blind.

Therefore we deem that this vulnerability is nothing but a theoretical annotation with no or little practical use. However users should be aware of this vulnerability if they are running a system where security is a concern.

3.5 Flash Cloning

Another approach was taken by Potemkin[57], a Xen based framework, which is able to launch a large number of VMs by introducing new functionality termed *flash cloning* and *delta virtualization*. It uses prior knowledge about pages being identical, but in a different manner than the approaches previously explained.

The concept is, briefly described, to launch one instance of a VM and let it run until it reaches a given state. At this point every memory page in the VM is marked read-only effectively creating a VM in a frozen state, where any modification of a page will trigger a write-fault. At some point the VM is cloned using CoW and another VM that is identical to the original VM is created. The advantage to this approach is that the second VM will not take up much memory until its memory is modified.

In order to implement the scheme a special VM is introduced to keep track of page ownership and how many VMs are sharing a given page.

Using the scheme described, they report impressive results wherein they started a 128 MB referential image and 116 clones of this image. All of the running clones consumed a total of 98 MB, implying that each VM changed less than one MB of its memory.

3.6 Comparison of the Different Approaches

Now we discuss advantages and disadvantages of the content-based page sharing and the flash cloning approaches.

As both approaches make use of CoW they both potentially have a performance overhead. When faults are triggered due to attempted write operations to read-only pages, there are additional operations that are not present when not using CoW. New operations include a trap to the Virtual Machine Monitor (VMM), finding new

[3] Write faults to shared pages may be more or less transparent to the OS.

space to write in, as well as actually writing the new page. Therefore as updating pages is more expensive than in a non-sharing setup, the number of writes to read-only pages should ideally be minimized. If this is achieved, then the potential performance overhead may be turned into a performance boost, as pages without changing contents should improve locality in caches etc.

The inherent problems with the content-based page sharing approach is that it introduces a new overhead: Finding candidate pages for sharing. Some scheme for scheduling the task must be devised. Content-based page sharing can identify all potentially shareable pages by contents, thus it can achieve as high or higher share percentage than flash cloning[4].

While the flash cloning approach has demonstrated great memory sharing numbers, it should be noted that Potemkin runs an environment that is almost perfect for sharing memory. While their setup is optimal for their needs, it is less applicable to the general user. Especially if the user needs to migrate VMs or run different operating systems. Furthermore we expect memory sharing to decrease proportional to the running time of the VM using flash cloning, as we saw in Section 2.3.1 on page 15.

A drawback of both approaches is that they both require Shadow Page Tables (SPTs) to be easily implemented. We explain SPTs in some detail in Section 4.4 on page 31. SPTs are notoriously known to be expensive in workloads with many processes being started [5, p. 2].

3.7 Summary

In this chapter we examined related work done on memory sharing. In particular we examined three approaches that used prior knowledge to identify duplicate pages. These were: Flash cloning in the Potemkin framework and two interdomain shared cache, one in Disco and one called XenFS. There were two other approaches which used an approach that actively searches for identical pages: Content-based pages sharing in VMware and a student project that examines pages brought into memory by block devices.

The next chapter will examine specific virtualization techniques and concepts that would be useful for the implementation.

[4] Given that there is no significant nondeterminism involved in booting a VM.

Chapter 4

Essential Memory Management and Virtualization Prerequisites

This chapter introduces memory management concepts central to our design and implementation. As consistent terminology about the different constituents of Page Tables (PTs) is non-existent, we will adopt the terminology used in [25]. In the discussions that follow we assume that we are working with the IA-32 architecture.

The first level table is called the Page Global Directory (PGD) and the second level table is called a Page Table Entry (PTE) table. Using this naming scheme a PT refers to the whole structure, consisting of a PGD and a number of PTEs.

Xen adopts a different naming scheme for these tables. It uses a notion of ln tables. On IA-32 without Physical Address Extension (PAE) enabled n is equal to two, thus providing l2 and l1 tables. A l2 table should just be thought of as a PGD and the l1 a PTE table. We found this terminology to be a bit confusing, so we do not adopt it.

The following explanation makes use of the Linux kernel's Memory Area (VMA) notion. A VMA is used to describe a contiguous memory area within a given address space. In order to keep this as efficient as possible, the kernel tries to merge VMAs to reduce the space used to describe regions. [38, p. 255-266],[9, 599-606]

The layout of the chapter is as follows: In Section 4.1 we discuss a recent addition to the Linux kernel that facilitates easy reverse look-up from a given page to the VMAs that are using it. In Section 4.2 and 4.3 we discuss how Xen adapts memory management to ensure safety. In Section 4.4 we give an introduction to Shadow Page Tables (SPTs) and how they are implemented in Xen. In Section 4.5 we describe a technique that allows the Virtual Machine Monitor (VMM) to remove pages from a Virtual Machine (VM) in a simple manner. Finally in Section 4.6 we describe Xen's primitives for doing interdomain and VMM to VM communication.

Before starting, we note that there is no documentation of the SPTs in Xen available, so all our descriptions are based on the implementation, which currently spans 6.000 lines of complex and poorly documented code just for the `shadow32.c` and `shadow.h` files. Therefore the descriptions reflect the way we interpret the code. As for the stability of the code, a member of the Xen development team at Cambridge recently said that the current implementation have, to put it nicely, serious flaws [21].

4.1 Reverse Mapping

This section is based on [9, p. 680-689], [25, p. 48-50], [13], [14], [16] and [15] as well as the kernel source code in `include/linux/mm.h`, `mm/rmap.c` and `include/linux/rmap.h` of the Linux kernel version 2.6.16.

From time to time the kernel needs to relocate the contents of page frames, e.g. when swapping a page out to disk. Moving the contents of a page frame from one place to another would, in a naive implementation (the 2.4 kernel actually did this when all else failed), require examining all virtual addresses spaces, i.e. every virtual address in every process, to see if it points to the given page frame. In order to be able to swap more efficiently, the kernel needed a mapping from physical addresses to virtual addresses. As a solution the 2.5 series of the Linux kernel introduced a reverse mapping (rmap) mechanism that associated a list of PTE pointers for each `page struct`[1]. With this mechanism, all that is needed to find all the virtual addresses that point to a given page is to read the list of PTE pointers of the given `page struct`.

The rmap solution was however deemed to be too costly in terms of memory usage and processor cycles spent on maintenance of the data structure. Therefore an effort was initiated to come up with a more efficient solution. The proposed solutions are referred to as object-based reverse mapping. Instead of associating a list of PTE pointers to pages, another approach is taken. The reasoning is as follows: As the number of pages often is larger than the number of kernel objects (think VMAs) describing the pages, it should be more economic to use these as the links between page frames and virtual addresses. To illustrate the point, remember that a large data file is represented by a single VMA region in the kernel, but spans many pages.

The kernel distinguishes between two types of pages: Inode mapped pages and anonymous pages. A page is inode mapped when it is backed on secondary storage and/or has an entry in the kernels page cache. If this is not the case, the page is referred to as anonymous.

To understand the implementation of reverse mapping, the first observation is that inode mapped pages already contain the needed information, we just have to retrieve it. The lookup hierachy used to locate PTEs for a given page is pictured in Figure 4.1 on the next page.

Each `page struct` contains a `mapping` member that is a pointer to an `address_space` struct. This again has a member `prio_tree_root i_mmap` (a sort of radix tree well suited for describing intervals) that contains pointers to all the `vm_area_structs` that point to the given page. This struct gives the start and end addresses of the area, so we have to search this area for virtual addresses that point to the given Machine Frame Number (MFN). There is however a twist: The page is a memory mapped file and the `page struct` has an associated `index` member, which identifies which block of the file the page is containing. This can be used to index directly into the VMA, so we can reduce the need to search the entire area to a single lookup.

Unfortunately this only applies to inode mapped pages and anonymous pages

[1] There is such a struct for each page frame and it was actually a more compact data structure than described. It is however conceptually just a list of pointers and to us the details are not essential.

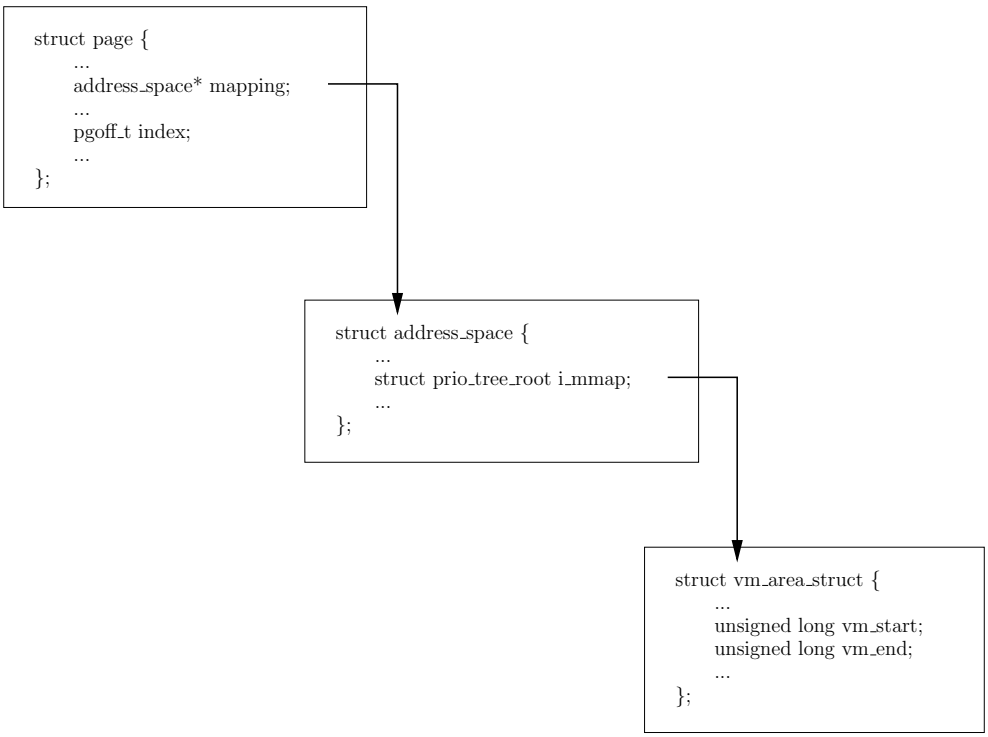


Figure 4.1: Using reverse mapping to resolve the PTEs that are referencing an inode mapped page.

cannot use this lookup scheme. Two different patch sets were devised to remedy this: anon-vma and anonmm.

The first associates a data structure to each page, which gives pointers to the VMAs that are using this page. The latter gives pointers to `mm` structs from each page. Both solutions were included in the kernel and benchmarked. Looking at the source code of the 2.6.16 Linux kernel reveals that the anon-vma patch set was finally chosen. As we make use of the first solution, we will disregard the second and explain the first in the following.

As said before, the anon-vma solution associates a data structure with each `page struct` (pointed to by the `mapping` member), namely an `anon_vma` struct. This essentially is comprised of a lock and a list of related VMAs. This is a rather loose definition and this is done deliberately, as the kernel applies lazy techniques when updating this list. This implies that it may link to VMAs that no more reference the page searched for. Thus the only guarantee we get, is that the VMA once has referenced the page. The VMAs are searched much like the inode mapped case, except there is no way to skip searching the entire VMA. The operation is more expensive, but conceptually it is more simple than the inode mapped case. The lookup hierachy is pictured in Figure 4.2 on the following page.

Furthermore only searching through related VMAs is less costly than running through every process in the system. Finally it should be worth pointing out that using reverse mapping may end up affecting system performance: Extra processor

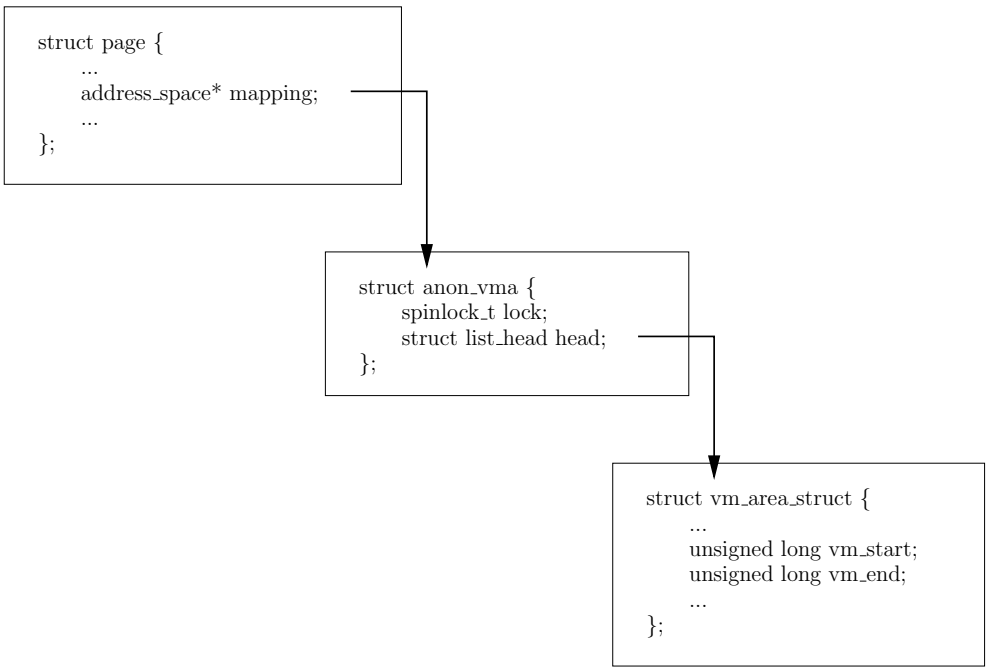


Figure 4.2: Using reverse mapping to resolve the PTEs that are referencing an anonymous page.

cycles spent maintaining the reverse mapping is one way, another is if the VMs are pressured to do a lot of swapping and the reverse mapping keeps the swap performance penalty low.

4.2 Memory Management in Xen

In order to fully virtualize any given Operating System (OS) in a safe and efficient way a number of modifications are required. We describe some of the most significant changes in the following three sections.

Flushing the Translation Look-aside Buffer (TLB) is expensive, because with an empty TLB the system will have to translate every virtual address on demand until the TLB again is filled with a sufficient amount of entries from the working set. Thus flushing the TLB is expensive and should be avoided in order to maximize throughput on the processor. [8, p. 2]

The Linux kernel divides every process address space into two parts, the user space that is different for each process and the kernel space that is common for each address space. Typically the virtual address space is divided so that the first 3 GB part is the user space part and the last GB of the address space is kernel space. [25, p. 53-54]

By doing this the kernel avoids TLB flushes whenever a switch to kernel space is performed, because the pages needed are accessible on any PT. Xen adopts the same strategy: It creates a Xen specific area in the upper 64 MB of the virtual address space of every process [5, p. 3]. The layout of any virtual address space belonging

to a guest domain in Xen is as pictured in Figure 4.3.

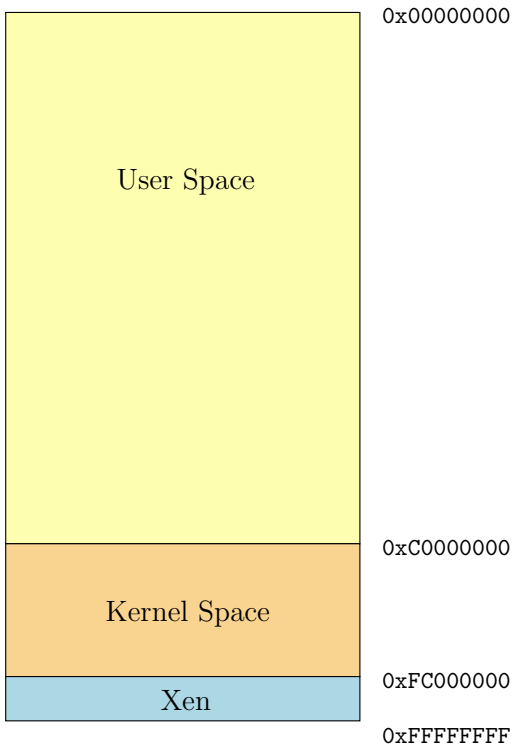


Figure 4.3: Xen mapped into a virtual address space.

Xen needs a fine grained control over memory allocation to ensure isolation between the VMs and therefore all memory allocations are performed at a page level granularity. The VMM keeps track of the ownership and use of each page by associating a `page_info` struct with each page. To us the most important parts of this struct are: Page type, owner, type count and reference count. [53]

Page Type: A page can have one of eight mutually-exclusive types. The first (none) indicates no special use of the page. Then there are four PT types l1 to l4, which are used for different levels of PTs. There are also two types intended for segmentation: Local Descriptor Table (LDT) and Global Descriptor Table (GDT). The last of the eight types is used to indicate that a page is writable (RW), meaning that the guest OS can use it for whatever purposes. The VMM (and not the hardware) regards it as being writable. That is for the paravirtualized case; in shadow mode (as we will explain in Section 4.4 on page 31) the page types change meanings. l1 to l4 have shadowed versions. There are two new types that replace GDT and LDT: hl2 and snapshot. A snapshot page is an internal part of the SPTs, which is used when synchronizing the SPTs. The hl2 table is an optimization that minimizes the need to map pages in from the guest OSes.

Finally the writable page type has changed semantics. In paravirtualized mode, we knew that the page was used as a writable page. In shadow mode we predict

that the page is writable, but it may very well contain one of the OSes PTs.

Owner: The id of the domain that owns the page.

Type Count: This count is used to keep track of how many places the page is used with the page type. The page cannot change type as long as the count is not zero. In shadow mode[2] it counts how many SPTs have been pointed to the page since the last SPT flush.

Reference Count: Is used to keep track of the number of references to the page. This typically reflects any PT that currently has it mapped in and should be thought of as how many processes have access to it. A page may not be freed as long as the count is larger than zero.

In Linux the memory is normally allocated in contiguous blocks of machine memory, but because the VMM allocates at a page level granularity it cannot be guaranteed that it is a contiguous block. This can be a problem because most OSes do not handle fragmented memory well[5, p. 7]. To overcome this problem Xen has introduced a pseudo-physical memory mapping, often referred to as physical memory. We will adapt this terminology; physical memory is a per guest VM abstraction of machine addresses. Using this the paravirtualized OS can create a contiguous range of memory, which may actually be allocated in any given order in machine memory. To make this possible the VMM contains a globally readable machine-to-physical table which contains the mappings from machine addresses to physical addresses. Furthermore each guest VM has a physical-to-machine table with the reverse mapping. [53]

Throughout the rest of this thesis we will use the terminology in Table 4.1. The PT mapping, which is placed in the guest OS, performs a translation from a virtual-to-machine address. The next two mappings, Machine-to-Physical (`M2P`) and Physical-to-Machine (`P2M`), are the ones mentioned before. The last translation, Shadow Page Tables (SPT), is also a mapping from virtual to machine addresses. We will explain SPTs in more detail in Section 4.4 on the next page.

Mapping	Maps address
PT	Virtual to Machine
`M2P`	Machine to Physical
`P2M`	Physical to Machine
SPT	Virtual to Machine

Table 4.1: Memory Translation Mappings.

Finally Xen uses a number of abbreviations for page frame numbers. A Machine Frame Number (MFN) designates a real machine page frame. A Physical Frame Number (PFN) refers to a pseudo-physical frame number (an entry in the `P2M`)[3]. Finally both Guest specific Machine Frame Number (GMFN) and Guest specific

[2] More specifically in ref count mode. This is described in Section 4.4 on the next page.

[3] We choose to view it this way, in the code however it is "a catch-all for any kind of frame number"[6].

Physical Frame Number (GPFN) are used interchangeably depending on the context. In translated shadow mode the latter three are equivalent. [6]

4.3 Handling Page Tables in Xen

The most significant features of Xen's memory management, is the management of PTs for ensuring isolation in the Virtual Machine System (VMS). This section is based on [53]. With ordinary PTs the guest OS has direct read access to the PTs, while updates of the PTs must be validated by the VMM [5]. When a guest VM creates a new process it is expected to allocate and initialize its own PTs from inside its address space and register it with the VMM. When it is registered with the VMM, there are two possible ways to make updates to the PTs: Hypercall and writable page tables.

Hypercall: When a process in a guest OS wants to update one of its PTs it makes a hypercall (`mmu_update`), which transfers the control of the processor to the hypervisor. The hypervisor then checks that the update does not violate the isolation of the guest VM and the VMS as a whole. If it does not violate any constrains, it is allowed to complete the write operation and update the PT.

Writable Page Tables: This method gives the guest OS the illusion that their PTs are directly writable. The VMM traps all writes to memory pages of the l1 type (PTE). If a write operation occurs, then the VMM will allow writes to that page, but at the same time disconnects it from the currently active PT. In this way the guest OS can safely make updates to the page, because the updated entries cannot be used by the Memory Management Unit (MMU) until the VMM validates the page and re-connects it to the PT. The VMM re-connects the page when: 1) The TLB is flushed, 2) a page in the unconnected PTE is accessed or 3) the guest OS modifies another PTE entry in a separate PTE.

The PTs are only handled this way when the guest OS requests it through a `vm_assist` hypercall. It should be noted that writable PTs do not yield full virtualization of the MMU. The memory management code in the guest OS still needs to be aware of Xen, but porting new OSes to Xen should however be easier using this scheme.

4.4 Shadow Page Tables

Full virtualization needs a way to ensure the correctness of updates to PTs. This is necessary because write operations to writable pages are non-trapping operations on x86 and a malicious VM can thus alter its PTs as it pleases. To remedy this, a common trick is to provide a shadow version of every PT to the MMU instead of the OSes own PT. The VMM detects changes in the OSes normal PT and propagates these to the Shadow Page Tables (SPTs). By doing this the VMM can ensure that the VMs cannot access machine frames they are not allowed to access. [46, p. 4-5].

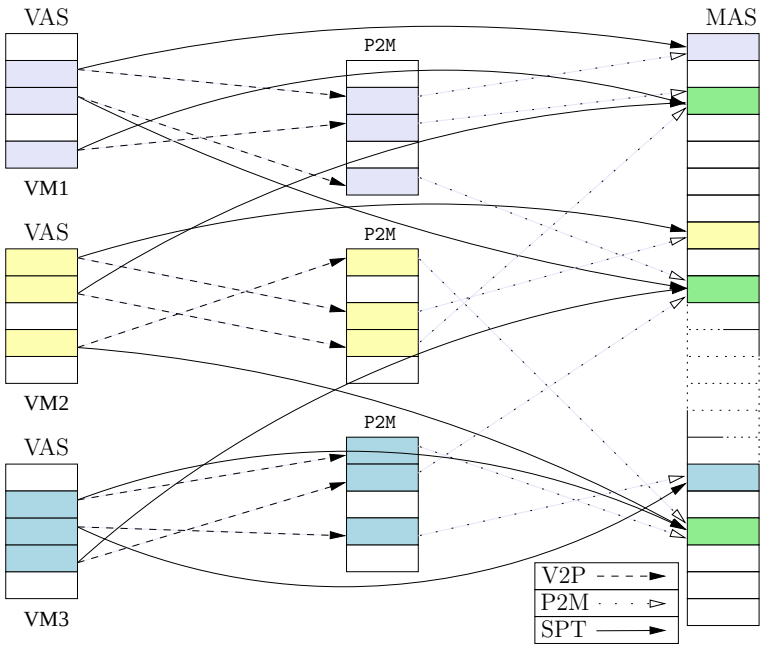


Figure 4.4: The level of indirection obtained by using SPTs in Xen. Leftmost we have the current Virtual Address Space (VAS) of each VM in the system. Each of these consists of "physical" addresses that must be translated by use of the `P2M` table to find the corresponding machine address in the Machine Address Space (MAS). The bold arrow from a virtual address to a machine address indicates a SPT entry.

The main principle behind SPTs is to provide a level of indirection between virtual addresses and machine addresses as pictured in Figure 4.4, thus giving a two level paging lookup. An unmodified OS cannot use this two level lookup, so a hardware specific representation of the two mappings is produced.

At the start of the thesis last semester the Xen SPTs were only used inside VMs to enable them to do live migration [12]. In the mean time the SPT implementation in Xen, necessitated by the need for hardware virtualization support[55], has evolved to support four different states: 1) Disabled, 2) enabled (our guess is that this is the one used for migration, where a read-only SPT is generated from the existing PT in the OS), 3) translated mode, which is transparent to the guest OS and 4) external, used for full virtualization using hardware virtualization chips. The choice of shadow mode is selected when starting a VM at runtime through the configuration file. Furthermore we note that the SPT implementation is architecture dependent, so different versions exist.

A `P2M` table which is organized as a two level PT, is used in both translated and external mode to achieve the extra level of indirection. The hardware specific SPT maps any virtual address onto a machine address as dictated by the `P2M` translation table. A SPT refers to this hardware specific representation while we will refer to the `P2M` as the translation table. One way to regard the SPTs is that they essentially just are a cache of the OSes PTs and the `P2M` tables, so as such it can be discarded at any time [57, p. 8].

In Xen a special mode that can be enabled is ref count mode. If enabled it simply reuses the `type_info` for a given page to keep count of how many SPTs are referencing the page.

Without any optimizations a SPT should be produced every time a world or context switch occurs. As this is a quite large computational overhead, a SPT cache can be used to allow reuse of SPTs. This again raises the concern of spending too much memory storing old SPTs and the task of determining when the SPTs are no longer valid. Xen stores one SPT for every PT, but reserves the right to tear down SPTs if running low on memory. Xen usually defers updates to the SPTs, so that they are done on demand. Sometimes it is however necessary to fully synchronize the SPTs with the OSes PTs.

Using this abstraction it becomes possible for the VMM to relocate a page owned by a given VM if needed, without the VM knowing about it.

4.5 Ballooning

If the VMM for some reason needs to allocate new pages and it has run out of free pages, then one place to turn to is to take pages from a guest OS. When the VMM needs to reclaim memory pages from a guest OS it is put in an awkward position, because it has no knowledge about the usage of pages in the guest OS. It will have to make uninformed choices when selecting pages to reclaim and it would be expensive if the VMM should keep track of or try to guess, which pages in the guest OS could be reclaimed.

The guest OS on the other hand has knowledge about its own working set and memory pressure, so the best way to reclaim pages, is to make the guest OS give the VMM the amount of pages it requests. A technique for doing this, is called ballooning and was introduced in [58].

Ballooning is used to change the memory pressure in the guest OS and force the OS to invoke its native memory management algorithms. The technique is realized by using a driver, the balloon driver, to change the memory pressure inside the guest. When the VMM wants to reclaim memory from the guest, it simply tells the driver to allocate pages, conceptually inflating the balloon, and return the allocated pages to the VMM. When the pages are no longer needed by the VMM it can return them to the guest OS simply by deflating the balloon and thereby lowering the memory pressure.

The balloon driver allocates free pages from the guest OS and inserts them on a linked list to keep track of which pages have been reclaimed by the VMM.

4.6 Events and Event Channels

The section is based on [5] and [53].

Normally an OS communicates with the hardware through interrupts. One of the challenges of virtualization is to intercept interrupts and direct them to the correct VMs. Xen's solution to this is to convert the interrupts into *events* and then propagate these to the VMs instead of the interrupts. The events are passed asynchronously to the VMs through *event channels*.

Each VM has two bitmaps: Event pending and event masked. Together these two are referred to as an event channel. The VMM can set a bit in the bitmap to indicate that a particular event is pending. When this has been done the VMM does an upcall to a given VM, thus scheduling the VM. The VM then checks which events are pending and interprets them as interrupts. If the VM needs to mask out interrupts it sets a bit in the event mask bitmap.

Using events of course requires changes to the OS, to enable it to run using events instead of real interrupts. The event channels should as such be viewed as a means of one-directional communication from the VMM to the VM. To communicate the other way, the VMM exports a hypercall interface to the VMs.

The actual implementation of events in Xen is a bit more fine grained than described above. In fact there are four different kinds of events:

- **pirq:** Used to deliver hardware interrupts to a VM.
- **virq:** Used to deliver virtual interrupts to a VM.
- **ipi:** Used to deliver interrupts from one virtual processor to all other processors.
- **interdomain:** Used to allow interdomain communication.

4.7 Summary

In this chapter we described the techniques necessary to realize and understand our implementation. We first examined reverse mapping, a new feature in the Linux kernel that enables the kernel to do more efficient swapping. It uses a trade-off between memory usage versus processor cycles spent on examining memory areas to find virtual addresses that are referencing a given page.

Then we examined the implementation of memory management in Xen. In particular the most significant parts needed to virtualize an operating system and the associated bookkeeping data structures. We saw that every page has associated a type, type and reference count as well as a pointer to the domain owning it. Then we explained that these were different when using shadow page tables. Shadow page tables enables the virtual machine monitor to provide a level of indirection between virtual and machine addresses in a manner that is transparent to the guest operating system.

Then we described a simple technique that enables the virtual machine monitor to increase and decrease the memory pressure inside a given guest operating system by using a module that allocates and frees pages. This is called ballooning.

Finally we explained how Xen performs virtual machine monitor to virtual machine communication, namely by sending events. These are essentially just software representations of real interrupts. For virtual machine to virtual machine monitor communication the virtual machines can use a hypercall interface exported by the virtual machine monitor.

With these techniques in mind, we can move on to actually proposing a number of designs for the implementation.

Chapter 5

Revised Design

The first part of this chapter sums up the designs from the last semester, though with some alterations. The layout of the chapter is as follows: First we introduce how the designs are split into separate components, which constitutes the basic functionality needed to implement a memory sharing scheme. We choose to use the content-based page sharing approach for the implementation as the flash cloning approach (both discussed in Chapter 3 on page 19) has already been implemented. Furthermore by choosing the content-based page sharing approach we will, at least to some extent, be able to repeat and evaluate the results of VMware.

Having described the components, we then explain the architectures of the two designs. We explain how the two differ from each other and what their advantages and disadvantages are compared to each other.

After that we describe the algorithms needed to create and tear down sharing in both designs. Finally we describe how the designs differ from last semesters report [33].

5.1 Components

The designs are split into components to clearly separate unrelated concepts. They are as follows:

Page Hashing (PH): This component creates a hash value of each memory page accessible from the components location. As the component was chosen only to run in the Virtual Machine Monitor (VMM) itself, this opts for two different approaches: Scan the pages of each domain or every machine page frame[1].

Hash Indexing (HI): Maintains a hash table with open addressing[34, p. 525-541],[17, p. 237-244] based on the hash values generated by the PH component. We will refer to the hash table as the *content index*. The entries in the content index contains: 1) A hash value that reflects the content of a given page and 2) the machine address of that page.

[1] Furthermore we have planned "scrambled" versions of both of these. These should reduce slight tendencies towards starvation.

Copy-on-Write Sharing (CS): This component handles write faults to shared pages and shares pages using Copy-on-Write (CoW). Two implementations of this are required: One using Shadow Page Tables (SPTs) and one running as a module inside the guest OS.

Reference Manager (RM): The RM keeps track of the reference count for a shared page and signals the tearing down of shared pages based on this reference count.

Page Comparison (PC): Compares two memory pages to see if they are bitwise identical.

The HI component is a subcomponent of the PH component and the PC component is a part of the RM component.

5.2 Architectures

The following subsections will outline our different designs. To convey the ideas effectively we try to keep the descriptions on as high a level as possible and return to the details in Section 5.3 on page 39. This means that most optimizations are left out in the descriptions.

Originally we proposed three designs in our DAT5 report[33]. Two of these are deemed usable: The first design, like VMware's design, is completely transparent to the guest Operating System (OS), while the second requires some modifications to the OS. As we are targeting Xen for the implementation, applying modifications to the OS does not seem unreasonable.

As explained in Section 4.4 on page 31, Xen distinguished between shadow modes on a domain basis as dictated by the configuration of the domain. Because of this we are able to choose the appropriate code for the given design at runtime. For the sake of explaining the design, we however distinguish between the two in the following descriptions.

5.2.1 Transparent Design

In this design all the components are, as pictured in Figure 5.1 on the facing page, placed in the VMM. To achieve this transparent solution, we use Shadow Page Tables (SPTs) as introduced in Section 4.4 on page 31.

The Physical-to-Machine (`P2M`) table can be used to share memory pages between Virtual Machines (VMs) without them knowing about it. This is done by changing the translation in the `P2M` tables, so that one or more `P2M` entries point to the same machine address. The CS component in the VMM will carry out this task as well as propagating the changes from the `P2M` tables to the SPTs.

The PH component has direct access to the entire machine address space because of its hypervisor privilege. This makes the PH component capable of hashing the memory pages for the entire system, as opposed to if it was placed in the individual VMs, which are only able to access the pages belonging to them. Furthermore placing the PH component in the guest VMs would require us to trust the values calculated by the module. As we generally cannot trust a VM to be friendly, the trust is hard

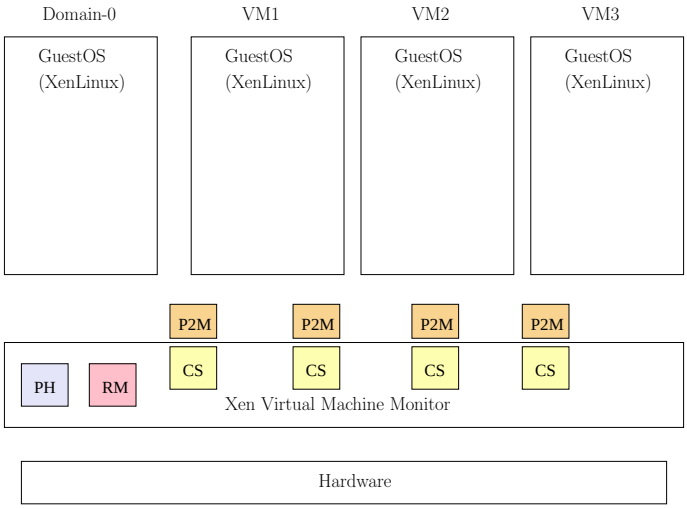


Figure 5.1: The architecture of the transparent design.

to achieve and typically comes at the expense of performance [22]. Furthermore placing the PH component in the guest VM could possibly introduce a new Denial-of-Service attack, where the guest VMs are sending arbitrary hashes to the VMM, thus overloading the VMM and disrupting the service of the other VMs. A third option would have been to place the PH component in domain-0, which would have the advantage that we keep the VMM simple.

While it is possible to enable a VM to be able to access the memory of the other VMs, we expect that it is expensive to do this. The expense paid is primarily that a hyperswitch is needed to map each page into the domains virtual address space. The latter argument also applies to placing the RM component in domain-0.

The advantages to the transparent design are that it can be done completely transparent to the VMs, hence no modifications to the guest OS. The disadvantage is that most of the complexity is placed in the VMM, which is contrary to the goal of keeping the VMM as simple as possible. Additionally the `P2M` tables and SPTs introduce both a memory usage overhead and a performance overhead in keeping the tables updated.

5.2.2 Paravirtualized Design

The design is as illustrated in Figure 5.2 on the following page and is essentially just a modification of the transparent design, where all the components, except for the CS component, are placed in the VMM. Placing the CS component in the VMs will allow us to avoid the overheads of using `P2M` tables and SPTs at the price of additional complexity. We explain the complexities in some detail in this section, but most will not be apparent until we discuss the algorithms in the next section.

The actual sharing is handled by the CS component in the VMs. The reason why we are able to trust the CS component in this case is that we are able to use the functionality in Xen that validates updates to the Page Tables (PTs), thus making us able to ensure that the CS component actually updates the PTs to valid addresses.

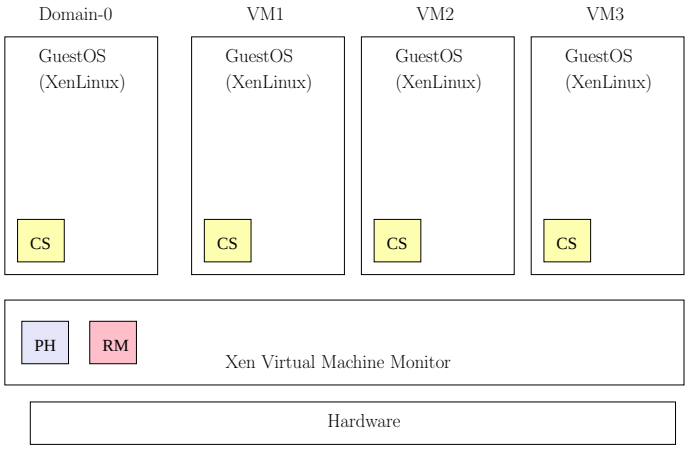


Figure 5.2: The architecture of the paravirtualized design.

To be able to realize the sharing of pages in the CS component, we need to be able to find all references to a given page and change them. This task resembles the task of swapping a page to disk, so we might as well take advantage of the reverse mapping solution, as described in Section 4.1 on page 26.

To actually reclaim pages in this design we use ballooning as described in Section 4.5 on page 33. One thing is freeing shared pages within an OS, another thing is reclaiming the pages and turning them over to the VMM. The balloon interface does not allow us to reclaim a given page directly, but we can free the page from inside the guest OS and balloon an arbitrary free page out. It is not necessarily the same page that is reclaimed by the balloon driver, but this does not really matter to us. In fact we might not even be interested in ballooning the pages out, as there might be better use for the freed page inside the OS.

In order to let a VM handle write faults to shared pages, the VMM needs to: 1) Trap the write faults, 2) detect they were caused by write operations to a shared page and 3) notify the VM of this. This kind of communication is normally done through events (as described in Section 4.6 on page 33), therefore we need to introduce new events in this design. Events are also used to signal the CS component in the VM that shares are pending. It should be noted that the event mechanism does not provide for data exchange, it only signals that an event has occurred. Therefore a buffer is needed, so that the VMM can exchange data with the VM.

A secondary page comparison is needed in this design, because from the point where we identified two identical pages until the PTs are updated, the pages may very well have changed. This could be done as a part of the VMM's PT validation. If the pages have changed, then a rollback to the old addresses is needed.

Finally it should be noted that the theoretically possible attack we described in Section 3.4 on page 21, becomes more feasible in this design. The main reason being that we inform the VM of exactly where shares are located in memory.

It should be noted that we expect the paravirtualized design to be the most efficient solution, because the transparent design has the SPT overheads. The paravirtualized design is thus the preferred approach, but for OSes other than Linux and

full virtualization we fall back to using SPTs.

5.3 Algorithms

Having introduced the architectures, we will now describe the details of the designs. To realize our designs we need two algorithms: One to share pages and one to remove the sharing of pages. We start by presenting the paravirtualized design as this contains the most interesting details.

5.3.1 Algorithms in the Paravirtualized Design

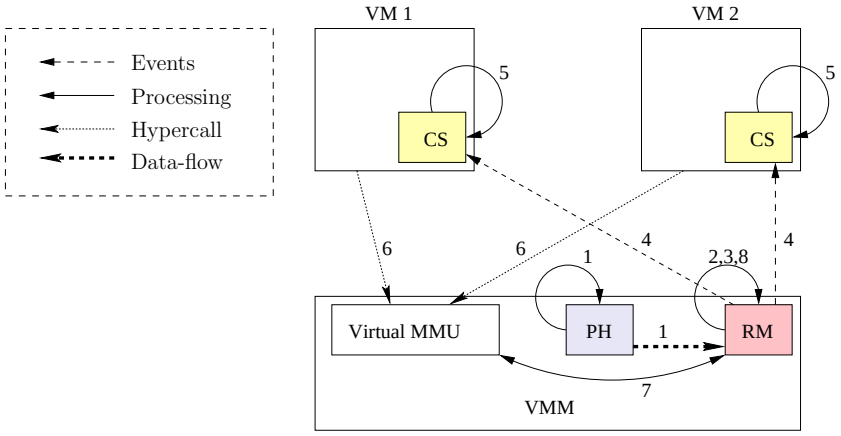


Figure 5.3: Algorithm of how sharing is created in the paravirtualized design.

We will now give a stepwise description of how the sharing of pages is carried out. This is illustrated in in Figure 5.3, where the numbers corresponds to the steps of the algorithm.

1. **Hashing and Content Index Lookup:** The PH component hashes a page and inserts the hash value into the content index through the HI component. When the hash value is inserted there can be two different outcomes: 1) The hash value has not been seen before and is added to the content index or 2) the hash value has been seen before and we have a sharing opportunity.

2. **Page Comparison:** If the pages are actually identical, then send an event to the involved VMs that they have to share the page. This can be done asynchronously as long as the VMs are not run between the point of sending the event and setting up the share. If they are not identical then we have a hash collision or one of the pages has changed since hashing it.

3. **Page Allocation:** A new page is allocated and the contents of one of the pages to be shared is copied into the new page frame.

4. **Share Pending Notification:** An event with two machine addresses is sent to the VM, indicating which machine address to remove and which to insert in its place.

5. **Page Table Update:** The VMs that should share the pages update their PTs to point to the same page and mark it as read-only in the PT entries.

6. **Commitment:** The updates to the PTs are validated as always by the VMM. As the pages may have changed since step 2, we need to ensure that the pages are still identical. Therefore we do a page comparison.

7. **Reference Count Update:** The RM updates the reference count for the shared page. In order to ensure the correctness of the system, it needs to validate all changes to PTs. They must either point to 1) the VMs own address space or 2) a shared page.

8. **VMs Relinquish Pages:** The two pages that were to be shared are now obsolete and the VMs therefore free and possibly balloon out the pages.

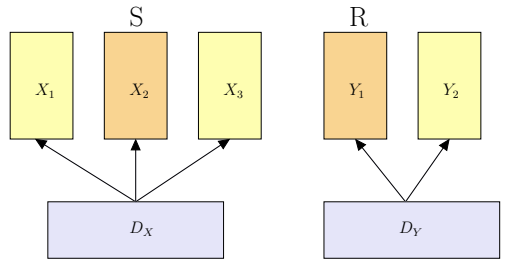
There is a subtle, but crucial detail to this algorithm. In order to avoid introducing new vulnerabilities, we employ a preemptive strategy. In order to share two pages, 1) we allocate a new page, 2) copy the contents of one of the pages to be shared to the new page, 3) point the two mappings to the new page and 4) free the two old pages. Why this is necessary is perhaps best explained through one of the problems avoided by the algorithm.

As a thought experiment, consider the following scenario: To share two pages we select one of the pages to be the one to be reclaimed, R, and the other to become CoW-shared, S (see Figure 5.4(a) on the facing page).

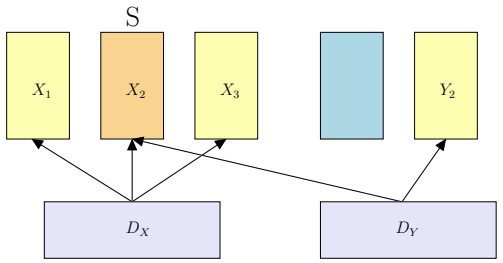
As it can be seen in Figure 5.4(b) page S, which is owned by domain D_X, gets shared between D_X and D_Y. We update both PTs to reflect this. After a while domain D_X attempts to write to the shared page, thus triggering a CoW page fault. A private copy of the page is created and the PT is updated as in Figure 5.4(c). The private page needs to be allocated from somewhere. Let us for simplicity say that the page is allocated from within the OS. If the D_Y domain (the non-owning domain) is malicious, it may hold onto the page forever. This situation forces domain D_X to use two pages instead of one for the data. If this happens at a large scale, then domain D_X may very well end up being crippled because most of its pages are shared to other domains and it is not in possession of enough free pages to keep all of its working set in memory. Alternatively it could allocate the page from Xen's domain heap[2]. This however has the drawback that a malicious domain may end up claiming all the free pages and we are not confident that Xen will be able to resolve such a situation.

By leaving the concerns of page allocation and ownership of shared pages up to the VMM, we get a simple solution to the problems just outlined. Furthermore we use a pseudo-domain called the share domain to be the actual owner of the shared pages. The reasoning behind this is that Potemkin uses this approach for handling shared pages, hence we gain compatibility. Thus when step 3 in the algorithm above allocates a new page, this is allocated to the share domain. As both domains relinquish the pages that should be shared, the VMM can free (through a ballooning

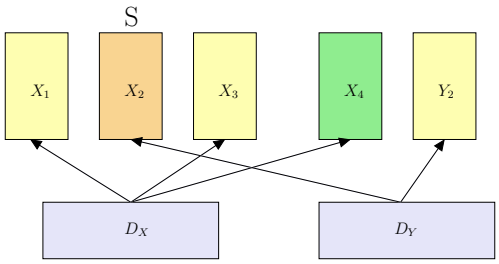
[2]To even realize this, new hypercalls would be needed to enable a VM to request more memory at runtime. Furthermore it would presumably require extensive modification to the guest OS to be able to use this for anything useful.



(a) S and R are shareable pages which belongs to domain D_X and D_Y



(b) S is shared between the two domains D_X and D_Y



(c) D_X allocates a new private page as a result of a CoW-fault

Figure 5.4: Possible new vulnerability. When sharing pages between two domains D_X and D_Y. First the page S can be shared with R, so it gets reclaimed by the VMM. Then the CoW-share is broken and domain D_X needs to allocate a new private page X_4, but D_Y still references domain D_X's page X_2.

process in the paravirtualized design or directly if in the transparent design) the pages.

It should be noted that this approach does have some drawbacks as we have an additional task of having to copy an entire page. We do however deem that the simplicity obtained is worth this extra operation. In the transparent design, this could however easily be avoided. Instead of allocating a new page, one page is chosen and page ownership is altered to the special domain. This is possible because of the extra level of indirection obtained by the `P2M` table. Doing the same thing in the paravirtualized design could easily prove to be a complicated task, mainly because the OS is not used to losing a page.

We will now stepwise explain the algorithm to handle write faults on shared pages:

1. The VMM receives a write fault from the Memory Management Unit (MMU). If it is a write fault to a shared read-only page, then it is converted to an event and sent to the corresponding VM. If not, it is handled as a normal page fault and propagated to the VM.

2. The VM receives the event of a shared read-only write fault. It makes a copy of the shared page and updates its current PT to point to its new own private copy.

3. The PT updates are validated as normal by the VMM. If a reference to a shared page has been removed, then the reference count is decremented. If the reference count reaches zero, then the page is freed.

As it can be seen we have to introduce new events and the VMM has to check for shares when validating PT updates. Now that we have presented the paravirtualized design we will continue with the transparent design.

5.3.2 Algorithms in the Transparent Design

As explained the difference in this design, compared to the previous, is that the CS component is placed in the VMM and there is a level of abstraction between virtual and machine addresses provided by the `P2M` tables and SPTs. The CS component just updates the `P2M` tables with the new mappings, which simplifies the algorithms, because the VMs are never involved. As with the other design we now present the algorithm for sharing pages:

1. **Hashing and Content Index Lookup:** The PH component hashes a page and inserts the hash value into the content index through the HI component. When the hash value is inserted there can be two different outcomes: 1) The hash value has not been seen before and is added to the content index or 2) the hash value has been seen before and we have a sharing opportunity.

2. **Atomic Phase Begins:** The following operations are critical. Therefore we need to ensure that there are no switches from the VMM to the VMs involved during the following steps, as doing this may leave the system in an inconsistent state.

3. **Page Comparison:** If the pages are actually identical, then proceed to step 4. If they are not identical then we have a hash collision or changes has happened to one or both of the pages since they where hashed. Then we have to bail out and end the atomic phase.

4. **Mappings are Updated:** The VMM changes the mappings in the `P2M` tables.

5. **Shadow Page Table Synchronization:** The SPTs are invalid as the mappings they contain have been changed, so they must be synchronized.

6. **Reference Count Update:** The VMM updates the reference count for the involved pages.

7. **Atomic Phase Ending:** The updates are done and we can resume normal mode of operation.

The SPTs will be synchronized in step 5, therefore in this design it would be preferable to batch the operations, as synchronizing the SPTs is expensive if performed each time a single page is shared.

The last thing we need to show is the stepwise description of a write fault to a shared read-only page. It is presented here:

1. The VMM receives a write fault from the MMU. If it is a write fault to a shared read-only page, then we continue with step 2. If not it is handled as a normal page fault and propagated to the VM.

2. We copy the content of the page to a free page and update the `P2M` entry to point to this new page. Finally the current SPT is synchronized to ensure that the changes are propagated.

3. The reference count for the given shared page is decremented. If the count reaches zero, then the page is freed.

5.4 Changes to the Original Design

For those familiar with our original design in [33], we now explain most of the changes we have made since then. If not familiar with the old designs, you may skip this section.

The designs were originally split into three components: PH, CS and RM. The RM component was further divided into two subcomponents: HI and PC. This separation proved a little inconvenient, so we decided to move the HI subcomponent from the RM to the PH. This allows the PH component to insert the hash values directly into the content index, instead of placing it in a buffer and waiting for the HI component to service it. It should be noted that if the PH component was not located in the VMM, this was not possible. Finally we note that the paravirtualized design was called the "hypervisor level design" and the transparent design was called the "transparent hypervisor level design".

Last semesters report suggested that we could choose to either do continuous hashing or flush the content index after each scan. We have chosen to flush the index after each scan, as this keeps the implementation as stateless as possible. Furthermore doing continuous hashing incurs the overhead of doing reverse lookup to maintain the content index, so flushing might even be more efficient. For more details see [33, p. 60-62]. A final note on the continuous hashing option, is that VMware's optimization where they always check if a page can be shared before swapping it out (from the VMM) to disk [58, p. 6], is not possible or at least difficult without continuous hashing. Furthermore flushing seems reasonable, because Xen as of this writing is not able to do swapping to disk from the VMM.

Furthermore the report discussed whether it was worth to resolve conflicting hash values. As we concluded in the report, this was not feasible as the probability of a collision with the hash function was too low to bother. The worst possible consequence is that we might miss a page sharing opportunity.

As for the hash function itself, as we concluded in the report, we chose to use the SuperFastHash[30] hash function. We did a number of experiments regarding the function and found it to be excellent in terms of low collision rate, uniform distribution, speed and code complexity. Our only concern was that the experiments were carried out on random data and as such this could not reveal any potential pattern related flaws in the function. Successively we tested the function on real data and found still no flaws.

5.5 Summary

In this chapter we presented two different designs: Transparent design and paravirtualized design. The first uses shadow page tables to share pages between virtual machines in a manner that is transparent to the virtual machines. The second uses common operating system functionality to share pages with the cooperation of the virtual machine.

First we presented the components needed to explain the architectures of the designs. After having presented these and the actual architectures, we gave algorithms for creating shares and tearing down shares when private copies were needed. From these descriptions we quickly found that the transparent design was the most simple design to implement. It did however have the disadvantage of having to use shadow page tables.

Finally we explained how the designs had evolved since the report was written last semester.

Chapter 6

Implementation

This chapter describes how we constructed our implementation and explains design choices made during the process. We start by accounting for the status of the implementation. From this we move onto an overall description of the implementation and finally we explain selected parts of the implementation in details. Our implementation and instructions on how to install it, can be found on http://www.cs.aau.dk/~mejlholm/dat6/cbps/.

6.1 Implementation Status

The transparent design is fully implemented. The paravirtualized design is as of yet only able to receive events about pending shares and set them up. None of the code in the Virtual Machine Monitor (VMM) that is supposed to check that the shares are set up correctly has been written. As this design has not been fully carried out and tested we acknowledge that it may still contain undiscovered design flaws. In the descriptions that follow, whenever talking about this design, we are explaining how we expect the implementation to be.

As for the actual implementation we started out trying to implement the entire scheme ourselves, mainly because the Potemkin code seemed unstable. We were not interested in fighting bugs that the Potemkin developers could not find. Later in the semester we however received an updated version of the Potemkin code, which seemed stable. At that time we were fighting bugs due to our own Copy-on-Write (CoW) implementation and decided to try to make use of the Potemkin code to see if that would eliminate the bugs. This proved more stable, so we decided to streamline our implementation with the Potemkin code. The combination of both content based page sharing and forking (as described in Section 3.4 and 3.5 on page 22) should presumably form a powerful combination. The Potemkin patch set from April, as we used for the implementation is built upon the Xen unstable repository changeset 9515.

As for the amount of code written/modified for our implementation, the total amount of changes to Xen was roughly 3600 lines of code. This included the Potemkin patchset, which alone changed roughly 1200 lines of code. Our part of the changes was thus roughly 2400 lines of code.

One thing that should be noticed is that the privileged Virtual Machine (VM)

cannot partake in the sharing, as translated Shadow Page Tables (SPTs) are not supported for this VM by Xen as of this writing. This could very well change, either by Xen getting support for a translated privileged VM or by use of the CoW Sharing (CS) module for the paravirtualized design.

6.2 Overall Description

The heart of the implementation is the `rm_main_reference_manager` function, which is called whenever the VMM is running its idle loop. This way we ensure that we only use processor cycles that would otherwise be wasted. Figure 6.1 serves as an overview of what functionality is called from within this important function.

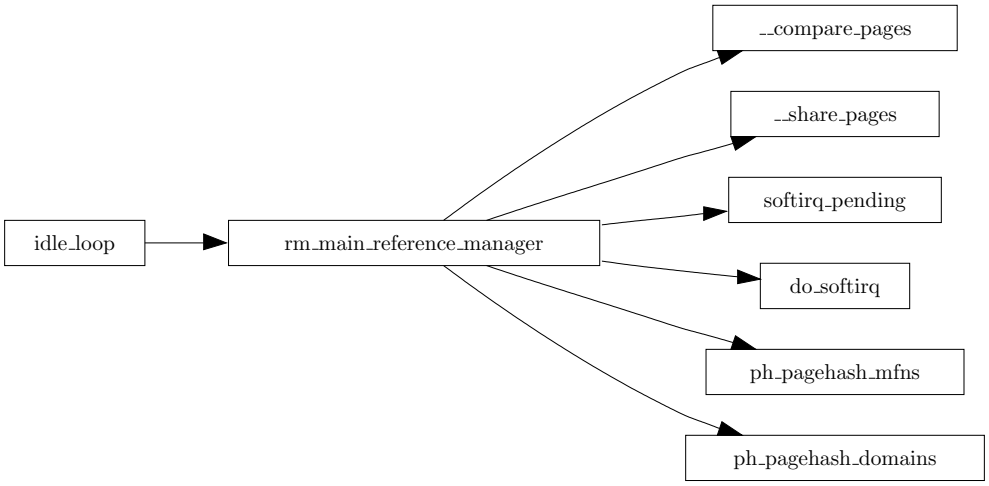


Figure 6.1: Call graph for the most significant functions in the reference manager.

The function has three different responsibilities: 1) Activate page hashing to identify possible shares, 2) page comparison of potential shares and 3) setting up confirmed shares by activating the CS components.

Normally the VMM uses `softirqs` to signal that the idle loop should terminate. Therefore we check if there are any `softirqs` pending from multiple points in the implementation. This ensures that the system still remains responsive.

The component keeps a timestamp of the last successful page hashing round and uses this to wait a predefined period (currently one minute, but can easily be changed) between each page hashing.

The page hashing has two different modes: Scan machine frames or scan domains. In either case we make sure to scan the share domain before looking for new shares. There is a subtle detail to this choice. Any pair of identical pages can have any of the following three combinations, which require different actions:

Both pages are unshared: As of this writing the implementation copies the contents of one of the pages to a new page and sets up the sharing. We are working on another version which simply converts one of the pages to be the shared

page, this is however not implemented completely. The latter should be preferable as it avoids the page copy, but will only be possible in the transparent design.

One page is shared: As one of the pages is already shared, all we have to do is to point the unshared copy to the shared and update the reference count. Simple and efficient.

Both pages are shared: An unlikely scenario, but it does happen from time to time. We regard it as unlikely because this means that somehow the content index logic failed and two shared copies of the page has been created. In this case the page with the lowest reference count should be converted to point to the page with the larger reference count. This includes searching all Physical-to-Machine (`P2M`) entries, to find entries that point to the page with the lower count or keeping a separate data structure. We are not interested in doing any of these, so we simply ignore this case. A better solution is to try to avoid the case completely and further work will be to determine exactly why this occurs.

The reason why we scan the share domain first is in anticipation that we hit the case with one shared page and one unshared, and avoid the last case.

To optimize we avoid certain page types, which we deem unfit for sharing, to speed up the page hashing. We elaborate on this in Section 6.6 on page 62. Furthermore to reduce the size of the content index, we divide the hash value space into intervals and scan all pages for each interval, we will explain this in more detail in Section 6.7 on page 65.

Whenever a page hashing round has been successfully completed, the content index is flushed. While this sounds expensive, the alternative (to always keep the content index up-to-date) is more expensive. The expenses are both in terms of space usage and processing, because to be able to remove old entries from the content index we need to keep a reverse content index or otherwise search through the entire data structure. The reverse content index would consume 4 MB with 4 GB of memory as we need one pointer for each element in the content index. Therefore we gladly sacrifice some of our idle processor cycles to avoid the extra data structure.

Whenever the page hashing component finds potentially identical pages, the addresses of the two pages are inserted into a buffer shared between the Reference Manager (RM) and Page Hashing (PH) components. If the buffer is filled, then the page hashing component is terminated. The first thing done in the next idle loop is to service the contents of the buffer. If the page hashing component stops due to pending `softirqs` then the contents of the buffer is always serviced first.

When examining any pair of candidate pages for sharing within the buffer, the pages are mapped into the Xen part of the Page Table (PT) of the last process to run before entering the idle loop (Xen does not change PT before a switch to the idle loop). This caused some difficulties for us, so we explain this in some detail in Section 6.3 on page 50. When the pages are mapped in, we do a page comparison. If they are found to be identical then we can safely share them. No matter the outcome before, both pages are unmapped from the PT.

As for the actual comparison we use the `memcmp` functions. On processors that support this instruction this is used, otherwise a string version is used. The string

version searches through both pages in character sized chunks, searching for the first set of characters that differ from each other. If any are found, then it returns the difference. If no difference is found it returns zero.

According to [4] the processor supported version of `memcmp` is rather slow. Consequently he wrote another version that is supposed to be 4-6 times faster than the processor supported version on the IA-32 architecture. As the page comparison is a central part of our implementation, investigating another implementation of the `memcmp` function may very well prove feasible.

From this point on the CS components are activated, so the proceeding actions differ. The transparent version is the simpler, so we start with this.

The transparent solution requires that nothing changes within the two domains affected by the share. Therefore we pause both domains (standard Xen functionality) to ensure that they are not run while we change the mappings.

The updating of the page mappings are done as described above by the use of Xen's Shadow Page Table (SPT) implementation. In particular we need to update both the `P2M` and Machine-to-Physical (`M2P`) tables. The latter is also necessary because of the shadow mode used. While the domain is running in shadow mode, it is still a kernel modified by Xen using hypercalls, event channels etc. This mode is highly dependent on the `M2P` mapping, so we need to ensure the correctness of this table also. The SPTs are then resynchronized and the domains are again unpaused, free to resume operation unaware of the changes. We explain this in more details in Section 6.4 on page 52.

The paravirtualized solution takes another approach. Instead of simply updating the mappings, we turn to the Operating System (OS) internals to update the mappings. The CS module running inside an OS is sent an event signaling that shares are pending. A pair of addresses (to and from) are placed in a buffer shared between the module and the VMM. The module then reads the addresses and updates the mappings. We do not describe this in further details as it is not yet fully implemented. Finally the VMM is automatically notified about the PT updates as a side effect of the hypercall used to update PTs.

The hypercall can be used to register that a given share has been committed by the involved OS. In order to guarantee the safety of the system, it becomes necessary to track whether the shares, that the CS modules have been instructed to create, actually have been carried out. In order to keep track of this, a separate data structure is needed, unless we can find an existing structure that may be modified for the purpose. Given this structure, it would be needed to periodically check if any of the pending shares have been outdated.

To keep track of how many references there are to any given page we use Xen's type and reference counts, as explained in Section 4.2 on page 28. As the type count is a 16 bit value, there is a possibility of overflowing it if a certain type of page content is frequently shared. VMware noted the same problem with sharing zero pages [58, p. 6]. Sharing zero pages is a little bit of a dilemma, so we save the discussion of whether this is actually feasible until Section 7.3 on page 74. For now we just note that sharing zero pages is optional in the implementation. If zero pages are shared, then a workaround is needed. VMware's solution was to use pointers to create an arbitrarily large reference count. Ours is to always allocate one zero page and point all others to this. Recall that the type count is used to prevent a page

from changing its type or being freed. As we alone have control of the allocated zero page, we do not need to track the type count of the page.

Instead we can suffice with updating the reference count, which is a 29 bit value that takes a while longer to overflow. In fact as the 29 bits can represent a value up to $2^{29} - 1$, which would require 100 VMs to have approximately 5368709 zero pages (2.1 GB of zeros) each, to have the value overflow. This of course is a hack and it does not protect us from overflow on other page contents, which will overflow on $2^{16} - 1 = 65535$ values.

The pages freed by the sharing operation are returned to different pools depending on the CS module used. If SPTs are used, then the page is simply freed to the Xen domheap. This is Xen's primary source of free pages, which is used for almost all purposes e.g. when allocating pages to create a new domain. If the CS module implemented within the OS is used, then the page should be freed to the free pool of the OS. The VMM can then decide to balloon pages out depending on the memory pressure of the VM and the other VMs.

The description so far accounts for what happens when shares are being set up. All that is left is a description of what happens when a page fault to a shared page occurs. This is an event that is handled by the different CS components.

In the shadow version we rely on the functionality that is provided by the Potemkin code. When this event takes place we need to create a private copy for the faulting domain. This can be done in two ways: 1) Copy the contents of the faulting page to a new page and update the mapping to point to this or 2) if the reference count is exactly one, then the shared page can be converted to a private copy. The first has the overhead of having to do a page copy, while the latter simply removes the shared status, thus avoiding the page copy. It should be noted that the convert option is currently disabled in the implementation as it seems unstable. One could be surprised that this part of the Potemkin code is unstable, but the code has probably not been thoroughly tested because when a domain is forked in Potemkin the parent VM is suspended. From the suspended state it is not possible to resume operation, so the pages will reside in memory until either the VM is destroyed or the machine is halted. Thus as the pages retain unmodified and always with a count of more than one, it seems unlikely that the convert operation has been thoroughly tested.

Handling page faults in the paravirtualized design is another case. Normally in Xen, page faults to a paravirtualized VM are propagated to the OS. We expect to do just the same, as the OS should normally determine that the page fault was due to a write operation to a shared read-only page. In the context of a page fault there should be no difference between a page shared CoW inside a single OS and interdomain. As for the reference count, this should be automatically handled by Xen when the OS tries to update the current PT. Again, this has not been fully implemented and tested, so we might not have discovered problems with this approach.

This concluded the overall description of the implementation, the following sections address noteworthy topics that deserve extra explanation.

6.3 Super Page Problem

In this section we will outline an obstacle that took us a lot of effort to overcome. So much effort that we deem it is worth mentioning in the thesis.

The IA-32 is not restricted to using page frames with a size of 4KB, it can also use 4 MB pages frames. These are referred to as *super pages* or *huge pages*. Page frames of both sizes can coexist. A virtual address using the 4 MB page frames consists of 32 bit as normal, but instead of dividing it into a 10 bit PGD entry, a 10 bit PTE and a 12 bit offset, it is only split into a 10 bit PGD entry and a 22 bit offset[1]. [9, p. 49-50]

To determine whether a given page is a super page or ordinarily sized, the Linux kernel uses one bit, called the `_PAGE_PSE` flag, in the PTE entry for the given page. For the curious PSE is an abbreviation for Page Size Extension.

Normally the Translation Look-aside Buffer (TLB) has an entry for each of the pages recently used. By using super pages a larger virtual addresses area can be kept in the TLB, because the pages each cover a larger amount of the physical memory. Thus the benefits of using super pages is that the translation of virtual addresses may be sped up on systems with large amounts of memory.

Xen has its own heap, accessible to tasks within the hypervisor itself. To reduce the need to do TLB flushes, Xen uses super pages to map the individual page frames of the heap. As these are only mapped into the PGD, replacing these are somewhat difficult as we will explain shortly.

Whenever a Page Table (PT) is created (governed by the hypervisor), it is built from a predefined PT called the idle PT. In particular what happens is that the contents of the idle PTs PGD is copied onto a new page. As the idle PT contains entries pointing to every PTE spanning the Xen part of the address space (as we described in Section 4.2 on page 28), the newly created PT contains the Xen mappings by default, as pictured in Figure 6.2 on the facing page. A direct consequence of this, is that the last 16 entries (which spans the Xen area of the virtual address space), are identical on all PTs, unless updates to them are done. Normally Xen does not do this.

Both the task of page comparison and page hashing needs to be able to access the contents of any given page frame. Because the IA-32 does not allow direct access to the contents of page frames[25, p. 153],[9, p. 69] after paging has been enabled on the processor, we need to map the page frame into the Xen specific part of the current PT.

To have a number of virtual addresses to which it is safe to map the pages onto, our first approach was to allocate a number of pages at boot time. At that time we were not aware that the Xen heap was mapped using super pages. The pages to map onto were allocated from the Xen heap and were thus mapped into the virtual address space as super pages as pictured in Figure 6.2 on the next page.

Now, we wanted to use the virtual address to map any given page onto, thus we were trying to replace an entry in the PGD (remember super pages are mapped only into the PGD) as pictured in Figure 6.3 on page 52. As the PGDs in the system are not shared (although once copies of the idle PT's PGD), the changes made to the

[1] 22 bits are necessary to address each byte within the 4 MB page frame.

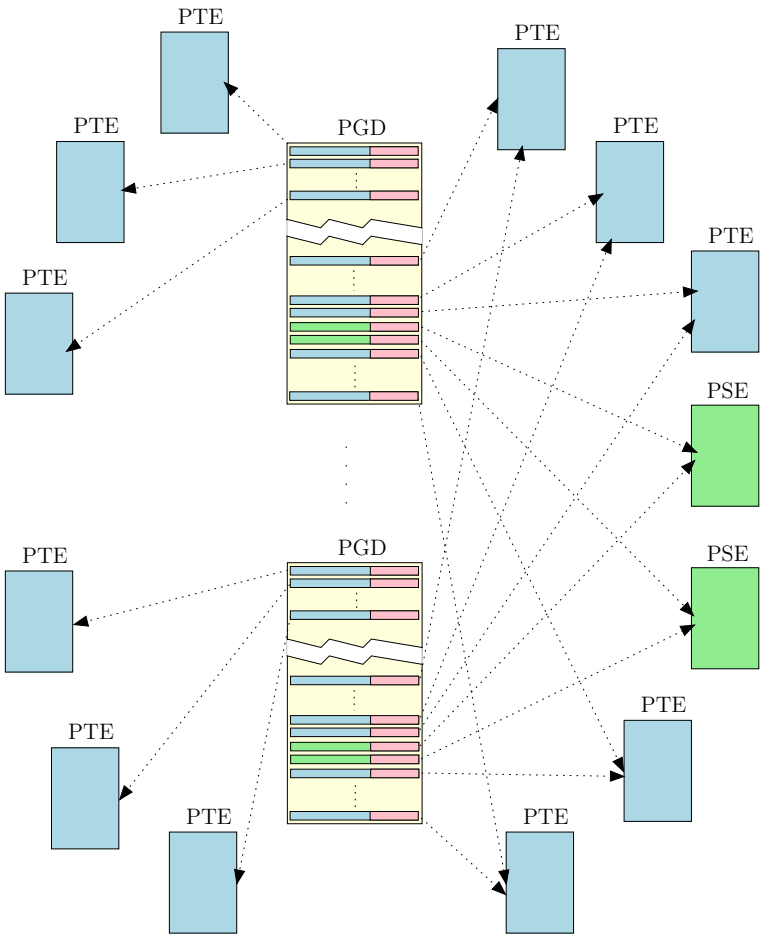
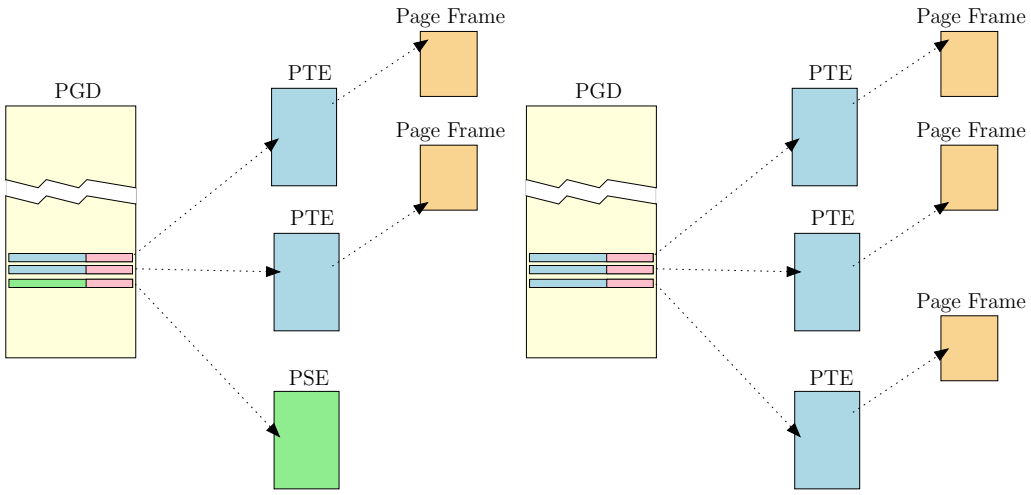


Figure 6.2: Page Table (PT) layouts in Xen. On the left are ordinary PTE tables, which are private to each PT. On the right are the PTEs representing the Xen address space, which are shared between all PTs. Green PSE pages indicate the super pages of the Xen heap.

PGD will not be propagated to the all the other processes PGDs. In fact changes made to any PGD using the `map_pages_to_xen()` function, which we used, are only applied to the idle PT's PGD.

Because of this behavior we experienced some odd results: Sometimes the page scanning reported that all pages were zero pages and sometimes it worked just as expected. Our guess is that the wrong results were caused by obsoleted PT entries that pointed into the Xen heap through super pages, thus pointing to some arbitrary place in the page, while the normal results were obtained on PTs that were created after the idle PT was updated. To be fair, we note that the function works fine when updating the PTE tables, because these are in effect shared between all processes in the system.

A result of this is that, when we map a given page onto the Xen address space, the hardware most likely will not see the change. A quick solution is to compile the hypervisor with the debug compile option enabled, which forces the Xen heap to use



(a) Before (b) After

Figure 6.3: When replacing a super page entry in the idle PT's PGD, the function allocates a new page to use as PTE table and points the PGD entry to point to the new page. It then populates the new PTE table with entries corresponding to the part of the virtual address space covered by the super page. Finally the TLB entry for the super page is flushed.

normal page mappings instead of super pages, thus solving the problem for the time being. [20]

Another and better solution to the problem is to use the `map_domain_page()` function instead of the `map_pages_to_xen()` function. This looks through a 4 MB reserved part of the Xen address space to locate a vacant PTE entry and maps the page into that slot. The reserved part of the Xen address space is represented by a bitmap (one bit for each entry in the corresponding PTE) and a cursor to the next predicted vacant slot. For those familiar with bitmap memory management[50, p. 199-200], this technique resembles it a lot. The difference being that instead of using the bitmap to keep track of vacant pages, the bitmap is used to keep track of vacant virtual addresses.

The function needs to flush the TLB when the cursor exceeds the number of entries in the bitmap, because this triggers some fixup code. This will therefore happen every 1024 time we map a page in. Using this function removes the need to manually find a set of virtual addresses that are safe to map onto at all times, thus solving our problem.

6.4 Sharing Pages

We will now explain the details for the part of the implementation that is responsible for creating the actual sharing. In particular we will explain the code that is responsible for changing the mappings in the `P2M` and the `M2P` tables. This is carried out by our `cs_change_mapping` function, which is called by the `__share_pages` from within

the reference manager. To provide an overview of the call hierarchy in this function we provide Figure 6.4. To better understand the code in this section, we advice the reader to keep referring to the figure while reading the code. The code referred to in this section and the following is based on our own repository, specifically changeset 9779.

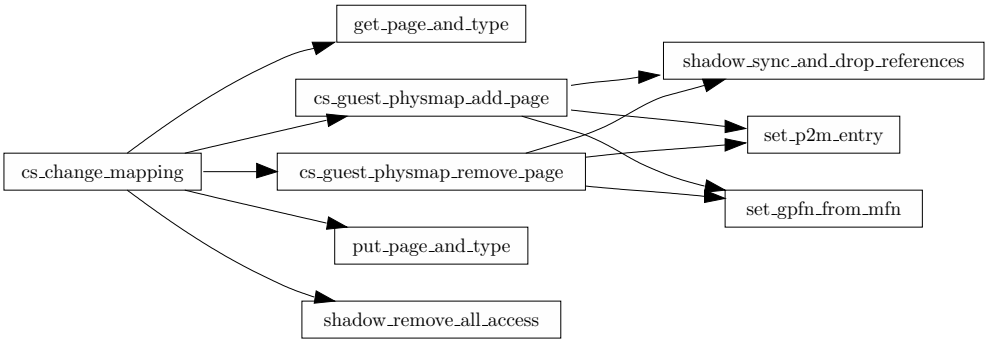


Figure 6.4: Call graph of the functions used when setting up shared pages.

We will explain the internals of this function by following the logical flow of the code as it would be executed in a normal run. For the sake of simplicity, we omit such things as debugging and sanity checking etc. For all the details we refer the reader to the source code. Our approach is to show the code in appropriately sized parts and immediately after explain the code. This means that we will skip large portions of code that is irrelevant to a basic run.

As said, we start with the `cs_change_mapping` function, which can be found in `xen/common/cs_sharing.c` starting on line 49. It should be mentioned that the VM owning the page to be shared has previously been paused, meaning that it will not be allowed to run on any processor. This was a direct consequence of beginning the atomic phase as described in the transparent design in Section 5.3.2 on page 42.

```
int cs_change_mapping(
    struct page_info *from, struct page_info *to )
{
    unsigned long gpfn = 0;
    unsigned long mfn = 0;
    struct domain *owner = NULL;

    mfn = page_to_mfn(from);
    owner = page_get_owner(from);
    gpfn = get_gpfn_from_mfn(mfn);
```

The parameters taken by the function are *from page* and *to page*, which means that we want to change the mappings from the *from page* to the *to page*.

We get the Machine Frame Number (MFN) and owner by using functions that looks up the information in Xen's data structures. The Guest Physical Frame Number (GPFN) is retrieved by a function that collects it from the `M2P` table. This is the entry that we want to update, so it will point to the new address (the *to page*) as a

result of the function call. The difference between MFN and GPFN was explained in Section 4.2 on page 28.

```
get_page_and_type(to, dom_cloned, PGT_writable_page);
```

Here a type and page reference is taken to the page we want to change the mapping to. This is done since we are about to change our mapping to point to this page, so taking a reference to it is necessary.

Shared pages are identified by the fact that they are owned by the cloned domain. Originally we named it the share domain, but in our effort to synchronize with the Potemkin source code we switched to use the domain already used there. Therefore in this code, the reference is taken on behalf of the cloned domain, since this page is going to be a shared page. The type of the page is `PGT_writable_page`, which is the same as the `RW` page type that was explained in Section 4.2 on page 28.

```
shadow_lock(owner);

/* Unmap from old location, if any. */
if ( gpfn != INVALID_M2P_ENTRY ){
    cs_guest_physmap_remove_page(
    owner, gpfn, page_to_mfn(from) );
}
```

The next interesting bit is when the shadow lock is taken. This is done since we are about to change the mapping in the `P2M` and `M2P` tables. The approach used to do this is first to remove the old entries and then add new entries that reference the shared page. This part only shows the removing part and we return to the adding part later.

First the GPFN is checked to see if it is a valid GPFN. If this is the case, then we need to remove it from the `M2P` table[2]. This is performed by a call to our `cs_guest_physmap_remove_page` function, which can be found in `xen/common/cs_sharing.c` on line 17. It is structured as follows:

```
static inline void cs_guest_physmap_remove_page(
    struct domain *d,
    unsigned long gpfn,
    unsigned long mfn)
{
    struct domain_mmap_cache c1, c2;

    domain_mmap_cache_init(&c1);
    domain_mmap_cache_init(&c2);

    shadow_sync_and_drop_references(d, mfn_to_page(mfn));

    set_p2m_entry(d, gpfn, -1, &c1, &c2);
    set_gpfn_from_mfn(mfn, INVALID_M2P_ENTRY);

    domain_mmap_cache_destroy(&c1);
    domain_mmap_cache_destroy(&c2);
}
```

[2] Actually it is not removed entirely, it is replaced by a special INVALID_M2P_ENTRY entry.

First c1 and c2 are initialized. These are used by the set_p2m_entry function, which again uses map_domain_page_with_cache to map in the P2M table. Actually the first function does not use the caches, it only passes them onto the latter function. The latter function is similar to the map_domain_page, which we explained in Section 6.3 on page 50. The only difference is that a "cache" is kept outside of the function, so the virtual address used to map into can be reused if the MFN is the same.

Then shadow_sync_and_drop_references is called, which checks whether the page mappings are out of sync. If so, then all pages that are out of sync are synchronized. Furthermore the function also removes all references to the *from page* in the SPTs by a call to shadow_remove_all_access. We will return to this latter function later in the section.

Finally its time for actually updating the mappings: First set_p2m_entry updates the P2M mapping with a value of -1, which indicates a none existing entry. Then the M2P table is updated in the same manner with an INVALID_M2P_ENTRY. Lastly the caches are destroyed.

Returning to the cs_change_mapping, the next step is to add the *to page* instead of the removed entries.

```
/* Map at new location. */
cs_guest_physmap_add_page(owner, gpfn, page_to_mfn(to));
```

The cs_guest_physmap_add_page function updates the P2M and M2P tables with the *to page*. This function can be found in xen/common/cs_sharing.c on line 34 and resembles the cs_guest_physmap_remove_page a lot, the difference being the arguments to the functions that are called by the function.

```
static inline void cs_guest_physmap_add_page(
        struct domain *d,
        unsigned long gpfn,
        unsigned long mfn)
{
    struct domain_mmap_cache c1, c2;

    domain_mmap_cache_init(&c1);
    domain_mmap_cache_init(&c2);

    shadow_sync_and_drop_references(d, mfn_to_page(mfn));

    set_p2m_entry(d, gpfn, mfn, &c1, &c2);
    set_gpfn_from_mfn(mfn, gpfn);

    domain_mmap_cache_destroy(&c1);
    domain_mmap_cache_destroy(&c2);
}
```

set_p2m_entry updates the P2M table with the MFN of the *to page*, such that GFPN now maps to it. Then the M2P table is updated by the set_gpfn_from_mfn function, so that the MFN of the *to page* points to the GPFN, The reader might wonder why it is nessesary to call the shadow_sync_and_drop_references. The reason why this is done is more as a precaution than as a necessity. For example if a page

is not freed correctly, then it might still have SPTs pointing at it or not be fully synchronized.

Having explained how to add and remove pages, we now return to the main function `cs_change_mapping`, where we call:

```
shadow_remove_all_access(owner, mfn);
```

This removes all of the entries referencing the given MFN in the SPTs of the faulting domain. When these have been removed, then access to one of these obsolete entries will result in a page fault. The SPT implementation checks if the fault was due to a missing entry in the SPT and thus updates it on demand with the missing entry. This way we do not have to fully synchronize the SPTs every time we setup a share.

```
    shadow_unlock(owner);

    put_page_and_type(from);
    domain_unpause(owner);

    return 0;
}
```

We have now changed the mapping we wanted. Both the type and reference counts are decremented on the *from page*, since we have updated a pair of mappings to use another page. If the reference count reaches zero then the page is freed as a side effect. Lastly we unpause the domain on which we have updated the mappings, which corresponds to ending the atomic phase, as described in Section 5.3.2 on page 42.

Finally it is important that the shared pages are only mapped into the SPTs as read-only. This is achieved by removing read-write flags on pages owned by the clone domain when the guest OSes PTs are propagated to the SPTs.

One important note is that by using Xen's reference counting mechanisms and always keeping the system in a consistent state, we automatically ensure the correctness of the system. A consistent state involves always either fully creating the share or rolling back the changes as well as ensuring that the shared pages are only mapped read-only. Fully creating the share involves such things as keeping both the `P2M` and `M2P` tables as well as the SPTs consistent. To ensure this, we make sure to explicitly synchronize the tables as we create the shares or tear them down.

6.5 Handling Page Faults to Shared Pages

Having covered how pages are shared, we now explain how the shares are torn down again. Recall that this takes place when a write operation is performed to a shared read-only page. The typical call trace for this operation is illustrated in Figure 6.5 on the next page. Again, as with the previous section, we only explain the path that leads to the CoW break handled by the `__cow_break_sharing` function. We remind the reader that this function is a part of the Potemkin framework.

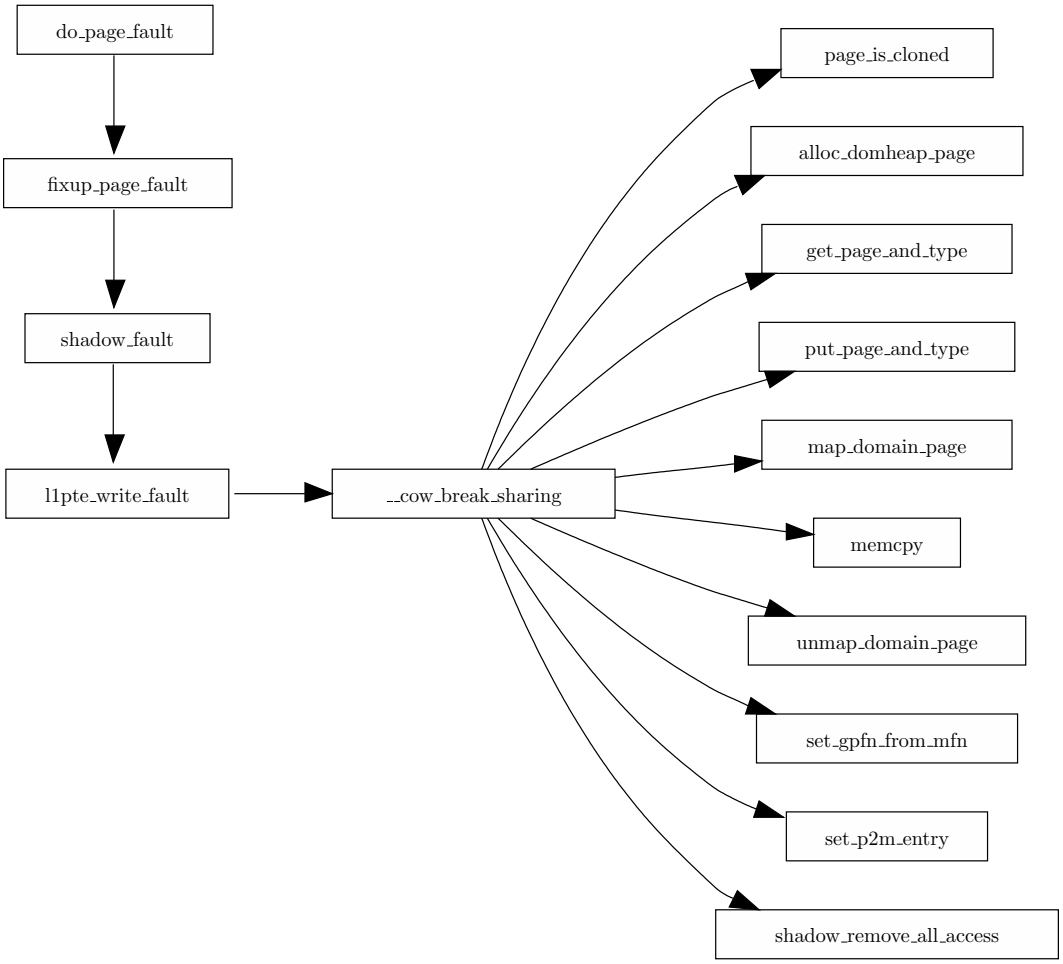


Figure 6.5: Call graph for a CoW break operation. A series of filtering logics leads to calling the `__cow_break_sharing` function.

Determining what kind of page fault we are dealing with is a long sequence of events, which starts with a page fault being raised by the hardware. The `do_page_fault` function is registered as the handler for this event.

The input parameter to this function is `struct cpu_user_regs *regs`, which is a pointer to the CPU registers. From this, we are able to retrieve the address that caused the fault by reading the `cr2` registry. The function is located on line 689 in `xen/arch/x86/traps.c`.

```
asmlinkage int do_page_fault(struct cpu_user_regs *regs)
{
    unsigned long addr;
    int rc;

    __asm__ __volatile__ ("mov %%cr2,%0" : "=r" (addr) : );

    if (unlikely((rc = fixup_page_fault(addr, regs)) != 0))
        return rc;
```

The read value is placed in the `addr` variable and can now be used to call the `fixup_page_fault`, as can be seen in the call graph. This function can be found in `xen/arch/x86/traps.c` on line 587:

```
static int fixup_page_fault(
    unsigned long addr,
    struct cpu_user_regs *regs)
{
    struct vcpu   *v = current;
    struct domain *d = v->domain;
```

First `v` is pointed to the current virtual CPU, which essentially just is a software representation of the state of the VM's processor. Since the page fault happens in the context of the current virtual CPU, the domain that caused the fault is the one currently running on this virtual CPU. Hence we can determine which domain caused the page fault.

In the process of determining which kind of page fault caused the fault, a lot of logics are evaluated. The logic that we end in is the following:

```
    else if ( unlikely(shadow_mode_enabled(d)) )
        return shadow_fault(addr, regs);
```

This leads to the call of `shadow_fault`. To understand the next code snip we first introduce the possible error codes that the page fault could have. An error code about what caused the fault can be retrieved from the processor registers, which can contain the following values:

```
/*
 * #PF error code:
 *  Bit 0: Protection violation (=1) ; Page not present (=0)
 *  Bit 1: Write access
 *  Bit 2: User mode (=1) ; Supervisor mode (=0)
 *  Bit 3: Reserved bit violation
 *  Bit 4: Instruction fetch
```

```
*/
```

Listing 6.1: Page Fault (PF) error codes.

We are interested in the page fault that happened due to a write access, hence the second bit is the one of interest (named "Bit 1" in the listing above).

Now lets see the relevant parts of the function `shadow_fault`, as it can be found in `xen/arch/shadow32.c` on line 3041.

```
int shadow_fault(unsigned long va, struct cpu_user_regs *regs)
{
    l1_pgentry_t gpte, spte;
    struct vcpu *v = current;
    spte = l1e_empty();

    /* Write fault? */
    if ( regs->error_code & 2 ){
        if (unlikely(!l1pte_write_fault(v, &gpte, &spte, va)))
```

As the second bit is set[3], we enter the if statement in the second last line. From this statement the `l1pte_write_fault` function is called. This function can be found in `xen/include/asm-x86/shadow.h` on line 849. The function mostly contains sanity checks before we reach the point where `__cow_break_sharing` is called.

```
static inline int l1pte_write_fault(
    struct vcpu *v,
    l1_pgentry_t *gpte_p,
    l1_pgentry_t *spte_p,
    unsigned long va)
{
    struct domain *d = v->domain;
    l1_pgentry_t gpte = *gpte_p;
    unsigned long gpfn = l1e_get_pfn(gpte);

    if ( __cow_break_sharing(d, gpfn) )
```

It is called with the parameters `d` and `gpfn`, where `d` is a pointer to the domain the page fault happened in. The second parameter, `gpfn`, is the page we got a write fault on.

We have now reached the function where the actually CoW break takes place. This is a rather large function so we will take it piece by piece. The function `__cow_break_sharing` can be found in `xen/arch/x86/shadow32.c` on line 173.

```
int __cow_break_sharing(struct domain *d, unsigned long gmfn)
{
    unsigned long mfn;
    struct page_info *page;

    /* Shared pages are only permitted
     * for translated domains. */
    if ( !shadow_mode_translate(d) )
```

[3]Remember that 2 is 10 in binary, so we get the bitmask we need.

```
        return 0;

    mfn = gmfn_to_mfn(d, gmfn);
    if ( !VALID_MFN(mfn) )
        return 0;
    page = mfn_to_page(mfn);

    if ( !page_is_cloned(page) )
        return 0;
```

As parameters it takes the two variables `d` and `gmfn`, as explained in the text above. Note that what we before called `gpfn` are now named `gmfn`.

So far only a set of checks have been evaluated, which ensures that 1) the faulted domain is in shadow translated mode, 2) a valid MFN can be found in the `P2M` table and 3) that the page faulted on is owned by domain cloned.

```
/* For pages which are truly shared, we must make a copy.
 * It's possible that we race with another domain which is
 * calling __cow_break_sharing on the same page, but the
 * worst-case scenario is that both domains make a copy of
 * the page and the original page is freed (instead of
 * being converted to an unshared page). */

if ( (page->u.inuse.type_info & PGT_count_mask) > 1 )
{
    struct page_info *p = NULL;
    unsigned long omfn, nmfn;
    char *parent_page, *child_page;
    struct domain_mmap_cache l1cache, l2cache;

    p = alloc_domheap_page(d);

    omfn = gmfn_to_mfn(d, gmfn);
    nmfn = page_to_mfn(p);
```

Based on the type count, we choose either to create a private copy by copying or by converting. If the count is larger than one, then we must make a copy. If not, then we can convert the page to point to the domain. We start with the code for the first outcome.

The first thing done in the copy approach, is that a page is allocated and its owner is set to the domain which caused the page fault. Then the original MFN is retrieved by calling `gmfn_to_mfn`, which uses the `P2M` table of the domain given.

```
    parent_page = map_domain_page(omfn);
    child_page = map_domain_page(nmfn);

    memcpy(child_page, parent_page, PAGE_SIZE);

    unmap_domain_page(child_page);
    unmap_domain_page(parent_page);
```

In order to copy the page, the two pages are mapped in. Then the actual page copy is done and the pages are unmapped again. Having created the private copy,

this needs to be taken into use.

```
        set_gpfn_from_mfn(nmfn, gmfn);

        domain_mmap_cache_init(&l1cache);
        domain_mmap_cache_init(&l2cache);

        set_p2m_entry(d, gmfn, nmfn, &l2cache, &l1cache);

        domain_mmap_cache_destroy(&l2cache);
        domain_mmap_cache_destroy(&l1cache);
```

First the M2P mapping is updated much like when sharing the pages, so that the newly allocated frame points to the Guest specific Machine Frame Number (GMFN) in the M2P table. Next the P2M table is updated, such that the GMFN points to the newly allocated frame.

```
        shadow_remove_all_access(d, omfn);
```

Then all access to the original machine frame number is removed. As stated before this gives the effect that the SPTs will be updated on demand.

```
        /* Drop our reference to the original page. */
        put_page_and_type(mfn_to_page(omfn));

        return 1;
    }
```

Now the CoW break is almost complete, the last thing done are that the type and reference counts are decremented. That completes the operation.

To sum up what has taken place, first a page fault happened. After it was detected to be a write fault on a shared page and the involved VM was in shadow mode, we carried out the actually CoW breaking. This copied the contents of the faulting page frame to a newly allocated frame. Finally the P2M and M2P mappings where updated to reflect the changes, so that they use the newly allocated frame, instead of the frame faulted on.

We will now move on to explaining what happens if the type count is one. This indicates that there is only one VM using the shared page and it can therefore be converted instead of copied.

```
    else {
        spin_lock(&dom_cloned->page_alloc_lock);
        spin_lock(&d->page_alloc_lock);

        list_del(&page->list);
        dom_cloned->tot_pages--;
        page_set_owner(page, d);
        list_add_tail(&page->list, &d->page_list);

        d->tot_pages++;
        put_page_and_type(page);
```

```
        spin_unlock(&d->page_alloc_lock);
        spin_unlock(&dom_cloned->page_alloc_lock);

        return 0;
        }
    }
```

We are going to manipulate the page lists, so as the first thing we take the locks on the page lists of the involved VMs. Next the faulted page is removed from the cloned domains page list and the total number of pages for the domain is decremented. Then the owner of the page is changed to be the faulting domain and the page is inserted into that domains page list. Then the total number of pages of the faulting domain is incremented, since it now has a page more.

Lastly the locks are released. This accounts for the CoW break sharing using the convert approach. Instead of copying the page it was simply converted. This concludes how we share and break pages.

As explained in the beginning of the chapter, we now describe how we sort out pages that we deem it is unfeasible to share.

6.6 Filtering Pages

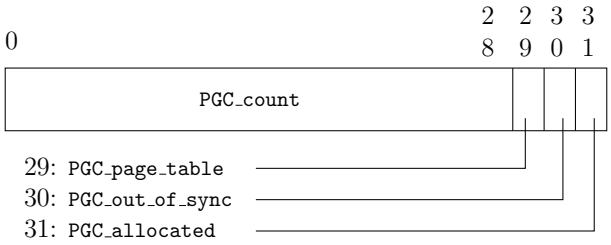
From VMware's point of view, the concept behind content-based page sharing was to identify shareable pages solely by their contents. Mostly because they had to provide a fully transparent solution[58] and consequently they did not have much information about the pages at hand. Xen on the other hand has to keep track of pages (as described in 4.2 on page 28) in order to validate PT updates. Therefore there is a lot more information available about each page. In this section we first describe this information in more detail than in Section 4.2 and then explain how we use this information to avoid scanning pages that we have no benefit from sharing.

VMware argued that all pages that had been found to be identical, could be shared per definition because they are identical [58]. There is no doubt about this. The filtering we apply is solely to reduce the amount of CoW breaks shortly after the sharing is created. Therefore we filter out pages that we deem to have a high probability of rapidly being modified.

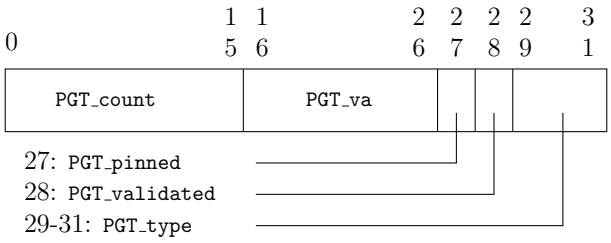
The information provided in this section is based on the `xen/arch/x86/mm.c` and `xen/include/asm-x86/mm.h` Xen files. Associated with each page is a `page_info` struct. This struct groups the three of the four interesting pieces of information described in Section 4.2 together. `count_info` contains the reference count and `type_info` contains the type count as well as the actual type of the page. As is customary in kernel code, information is usually compacted into bitmaps. This is also the case with these two counts in Xen. The layout of these are pictured in Figure 6.6 on the next page.

Listing 6.2 shows the different types a given page can have. Three bits (29-31 in `type_info`) are used to represent the type.

```
/* The following page types are MUTUALLY EXCLUSIVE. */
#define PGT_none            (0U<<29) /* no special use */
#define PGT_l1_page_table   (1U<<29) /* L1 page table? */
#define PGT_l2_page_table   (2U<<29) /* L2 page table? */
```



(a) `count_info`



(b) `type_info`

Figure 6.6: Contents of the `page_info->count_info` and `page_info->u.type_info` bitmaps.

```
#define PGT_l3_page_table (3U<<29) /* L3 page table? */
#define PGT_l4_page_table (4U<<29) /* L4 page table? */
#define PGT_gdt_page      (5U<<29) /* GDT? */
#define PGT_ldt_page      (6U<<29) /* LDT? */
#define PGT_writable_page (7U<<29) /* writable mappings */

#define PGT_l1_shadow     PGT_l1_page_table
#define PGT_l2_shadow     PGT_l2_page_table
#define PGT_l3_shadow     PGT_l3_page_table
#define PGT_l4_shadow     PGT_l4_page_table
#define PGT_hl2_shadow    (5U<<29)
#define PGT_snapshot      (6U<<29)
#define PGT_writable_pred (7U<<29) /* predicted writable */
```

Listing 6.2: Definition of the different page types. As can be seen, the types are reused in shadow mode.

This per page information is used to filter out pages that we deem it is not feasible to share. In particular we filter based on the following four arguments:

1. **Writable Page:** We require that the page must be of type `PGT_writable_page` or `PGT_writable_pred` in shadow mode. By requiring this we sort out PTs in normal paravirtualized mode. In shadow mode this sorts out the SPTs as well as the `hl2` and snapshot pages. We deem that all other pages than writable pages have a high probability of being changed, so we avoid them. Here we of course have no guarantee that a predicted writable page is not one of the other page types. Furthermore in shadow mode, the PT status is available elsewhere: `PGC_page_table` on `count_info` is used to indicate that a page is a PT. Finally

PGT_va on type_info is used to keep the eleven most significant bits of a virtual address, if the page is a PT.

2. **Type Count:** In paravirtualized mode this reflects how many places this page is used with its current type. This could be used to ensure that we do not share pages that are not mapped in by any OS. Because they are not mapped in, there is a good probability that they will be used for time critical operations, such as creating a private copy from a CoW shared page within the OS. Our view on this class of pages are much like zero pages. In shadow mode, the type count is only available in ref count mode. Here it reflects the number of SPTs that have mapped the page in since the last flush. Therefore we cannot use this for filtering.

3. **Validated Page:** If PGT_validated is not set, then the page has not been validated by the VMM for its current type. Therefore there is a good chance that it will change subsequently, e.g. a page in the process of being converted to a PT, so we might as well not scan it.

4. **Pinned Page:** The paravirtualized OS has a hypercall to pin a page to indicate that the type must not be changed, even if the type count reaches zero. This should indicate that the guest OS has special use of the page, so we might as well avoid scanning it.

The actual filtering rules are compacted into two different logics: One for paravirtualized mode (Listing 6.3) and one for the shadow mode (Listing 6.4).

```
#define PGT_count_mask      ((1U<<16)-1)
#define PGT_filter_mask     ~PGT_count_mask

if ( !(
    ( ( pg->u.inuse.type_info & PGT_filter_mask ) ==
    (PGT_writable_page | PGT_validated ) )
    &&
    ( ( pg->u.inuse.type_info & PGT_count_mask ) != 0 )
) ) {
    return FILTERED;
}
```

Listing 6.3: Optimized filter for paravirtualized mode.

```
if ( !(
    ( ( pg->u.inuse.type_info & PGT_filter_mask ) ==
    ( PGT_writable_page | PGT_validated ) )
    &&
    !(pg->count_info & PGC_page_table)
) ) {
    return FILTERED;
}
```

Listing 6.4: Optimized filter for shadow mode.

The filtering is applied both before hashing pages, but also again right before creating any shares. The main reason is that we have no guarantee that the pages have not changed in the period from before hashing the pages until the sharing is carried out. Furthermore if zero pages are chosen not to be shared, then these are filtered out immediately after hashing them in the page hashing component.

6.7 Size of the Content Index

As the size of the content index should be proportional to the amount of machines pages and we must be able to support up to 4 GB of memory, we have come up with a way to limit the size of the content index in return for more work while finding pages to share. Thus what we explain in this section is a classical time versus space trade-off.

It should be mentioned that we assume worst case here, where no page are identical, so each page uses a slot in the content index. This means that it can be assumed that inserts into the content index, usually is faster than our calculations done in [33, p. 56-58], since all the identical page (which also includes the zero pages), will only use one slot for each identical set of pages.

Recall that a single entry in the content index contains a machine address and a hash value representing the contents of the page at that address. Each of these values occupy 32 bit. Open addressing requires an additional 10% size of the content index to ensure good performance. All in all this makes it a total size of 8.8 MB on a 4 GB machine, which would seem reasonable to use to share pages. The total size of the Xen heap is however only 12 MB, so requiring 8.8 MB from the Xen heap is unreasonable. The Xen heap is used to store VM meta-data on and without any modifications the Xen heap is exhausted by instantiating 116 VMs[57, p. 11]. This means that any space we use from the heap will affect Xen's ability to instantiate concurrent VMs, so space allocated from this heap should really be kept to a minimum.

In order to remedy this we have decided to split the hash value space into intervals. To explain this we have to look at the SuperFastHash (SFH) function hash value space. It generates a 32 bit hash value, which can contain 2^{32} different hash values. This space is much larger than the number of entries in the content index. This is however not a problem because we are not trying to make room for all possible outcomes of this function, as we only need one hash value for each page in the system. As is common with hash tables, what we do is that we compute a key or index into the content index, where we can store the SFH value and corresponding machine address. We do this by using a secondary hash function (a simple modulus) to reduce the address space of the SFH function. This hash function takes the SFH value modulus the size of the content index. This will reduce the address space so it can fit into the content index.

Our solution to this size problem is to split the SFH's hash value space into smaller intervals and focus on one of these at each scan. This of course requires that the pages are scanned more than once as the pages that hashes to a value outside the interval currently of interest are discarded. As there is an overhead in scanning all the pages more than once, it is important to find a acceptable size for the content

No. of scans	Entries	Size in MB	Entries + 10%	Size with overhead
3	349.525	2.6	384.478	2.9
4	262.144	2.0	288.359	2.2
5	**209.715**	**1.6**	**230.687**	**1.7**
6	174.763	1.3	192.240	1.4
7	149.782	1.1	164.761	1.2
8	131.072	1.0	144.180	1.1
9	116.508	0.8	128.159	0.9

Table 6.1: The size of the content index relative to the number of scans with 4 GB of memory.

Memory size (MB)	Scans
512	0.625
1024	1.250
2048	2.500
4096	5.000

Table 6.2: The number of scans required to scan a given amount of memory.

index.

To calculate the threshold or number of entries in the content index, we use the total number of pages in the system and divide it with the number of scans we wish to conduct ($\frac{totalpages}{scans} = entries$). As we need to fill the content index x times with x scans we split the SFH's value space into x intervals.

To find a threshold that we deem is acceptable, we have looked at the size of the content index relative to the number of scans required to search SFH's value space, which can be seen in Table 6.1. The values in the table are calculated relative to the largest amount of memory that we can support at the moment, which on the x86 architecture is 4 GB without Physical Address Extension (PAE) support.

From these calculations we choose to say that five scans are acceptable, which gives us 230.687 entries totaling 1.7 MB. This amount of memory is statically allocated at boot time, unless the total number of pages can fit into these entries. If this is the case, then we only allocate the amount of space needed for the pages and thereby keep the memory usage to a minimum. Also as the number of entries is fixed, the number of scans are reduced on a machine with less than 4 GB. For instance on a 2 GB machine, the number of scans will only be 2.5, which we round up to three. In Table 6.2 we list the different number of scans required. As can be seen, as soon as there is more than approximately 800 MB available, more than one scan is needed.

It should be noted that this solution is based on the assumption that the SFH functions is uniformly distributed, which we have tested to some extent in [33]. If the hash function used to hash the pages is not uniformly distributed then the content index will overflow, because this calculation is based on the assumption that the amount of hash values in each interval can fit into the content index.

It should be noted that it would be possible to reduce the size of the content index further, by only storing the MFN instead of the machine address. The MFN

only spans 22 bits (on a 4 GB machine of course) compared to the 32 bits that the machine address takes up. This means that we could reduce each entry in the content index by 10 bits, which is a 15.6% reduction.

6.8 Summary

This chapter explained how our implementation was realized. We started with a quick status of the implementation. Here we explained that the implementation is based on Potemkin's Copy-on-Write code. Furthermore the transparent design is fully implemented, while the paravirtualized design is only partly implemented. Then we gave a high-level description of the implementation, about the little subtleties that are taken into account during the page sharing process.

Having covered the implementation from an overall point of view, we moved onto describing selected details. First we started with a description of how we ran into trouble when mapping pages into the Xen specific part of any virtual address space. Then we showed the most important code segments of how sharing and tearing down Copy-on-Write shared pages is done. Then we explained how we can use Xen bookkeeping data structures to avoid scanning pages that we deem unfeasible to share because they have a high probability of being changed subsequently after sharing them. Finally we explained how we split the hash value space of our selected hash function into intervals so that we can reduce the size of the hash table needed to store the hash values and machine addresses.

With a working implementation we could finally start experimenting with memory sharing, as we will present in the next chapter.

Chapter 7

Sharing Evaluation

In this chapter and the following we carry out a number of experiments. These can be categorized into the following: Sharing experiments and performance experiments. The first set of experiments evaluate how much there is to be shared on a number of workloads. The second set of results examine the overhead in the implementation. Specifically the experiments and results that will be presented in the following sections are as follows:

Best and Worst Case Experiments: Two simple experiments to see how much can be reclaimed under almost optimal and worst conditions.

Synthetic Workload Sharing Experiments: Unfortunately we do not have access to real-world workloads, so in this experiment we construct a number of workloads that are intended to emulate real workloads. Under these circumstances we measure how much memory can be shared in a number of different configurations.

Phone Booth Experiment: To satisfy the inner children, we perform an experiment to see how much we can overcommit the system with.

Performance Evaluation: In order to determine how well the implementation performs, we conduct a number of experiments. The main purpose of this experiment is to determine how expensive using Shadow Page Tables (SPTs) are compared to the normal Xen paravirtualized approach. When this has been determined we compare our performance with that of Xen running with SPTs.

Micro Benchmarks: To determine which parts of the implementation has the highest overheads, we conduct a number of micro benchmarks.

Besides explaining the results of the experiments, we also evaluate 1) whether it is feasible to share zero pages, 2) how much using the same binaries affects the share rate and 3) how much the allocation of memory to the Virtual Machine (VM) affects the sharing rate.

7.1 Benchmarks

Before presenting the experiments, we start by introducing the tools and benchmarks used for the experiments:

Kernel Build Time: A simple test where we measure the time spent on compiling the Linux 2.6.16 kernel. This is not really an industry standard test, but it is guaranteed to generate a large workload as well as demand for memory. This workload should exercise the potential shares thoroughly, thus ensuring variation in the pages. Apart from the following benchmarks this is the only benchmark where high numbers signify low throughput, thus low times are better.

SPECweb99: The SPECweb99 benchmark[18] is designed to exercise a web server. The workload at the server is a mixture of different requests, where 30% are dynamic, 16% are static and 0.5% are due to execution of CGI scripts, thus exercising both network interface as well as the processor. Given the example of UnixShell# in Chapter 2 on page 11 it should be interesting to examine how feasible it is to share memory in such a setting. In the experiments we used the Apache web server version 2.0.55.

OSDB: The Open Source Database Benchmark[52] exercises a database system of choice. For these experiments we have chosen the MySQL[2] 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.

dbench: The dbench benchmark[54] emulates the workload of a number of network clients (we have tested with 4 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 WindowsNT [54]. Because of its intensive number of IO operations, the benchmarks generate a significant workload for the processor as well as the page cache. In the experiments version 3.04 was used.

AIM: AIM Independent Resource Benchmark exercises the different parts of the Unix system. The benchmark consist of three subsystems: I/O transfers, function calls and Unix system calls. It measures the throughput of the different subsystems over a given time. We have chosen to run each benchmark for 60 seconds. In the tests the version used is aim-suite9. We note that the benchmark is run from user space, so the numbers often vary due to the workloads of the machine. Therefore the numbers produced by this benchmark should be taken lightly and with some considerations.

MWG: Medium Workload Generator is a script written by us. It exercises a number of common open source tools. Specifically it decompresses and compiles a number of open source packages. After one package has been compiled there is a predefined waiting period, so we ensure a medium workload. The compiles,

performed by gcc, are a number of packages packed with either the gzip or bzip2 applications. The test is as such not designed to make a real workload, it is just a quick mock up to generate a moderate workload.

TioBench: This benchmark is designed to benchmark a file-system by a mix of random and sequential read/write operations. Tiobench is especially designed to test I/O performance on a file system by using multiple threads. [35]

Stress: This tool imposes a configurable amount of I/O, CPU and memory stress on the Operating System (OS). Stress is not designed to be a benchmarking tool, it has been written to expose massive workloads to a system and thus expose weak points and bugs. [59]

Crafty: Crafty[31] is a chess engine with high demands for processing power. Although the memory footprint is minimal, the application happily spends any processor cycles it can get.

Freebench: This benchmark exercises many parts of the system with processor and memory intensive operations. It is split into two main groups: Integer and floating point operations. [47]

Having introduced the overall structure of the evaluation and the benchmarks used, we now proceed with presenting our results and the actual experiments.

This first part of the evaluation tries to answer the following questions:

- How much memory can be shared under nearly optimal conditions?
- Is it feasible to share zero pages?
- How much can be shared between different distributions of Linux or even other OSes?
- How much memory can be shared under the worst conditions?
- If we estimate what a real workload looks like, then how much can be shared?
- Can Xen do overcommitment?
- How much impact does the memory allocation of each VM affect the overall share rate?

7.2 Best Case Experiments

In this section we explore how much memory can be shared under good conditions.

The experiments were carried out on a 2.6 GHz P4 Northwood with hyperthreading, but hyperthreading was disabled. The machine had 2 GB of memory and besides the VMs reported in the individual tests, there was the privileged VM with an allocation of 64 MB. This was idle and not allowed to participate in the sharing. All VMs in these experiments ran Debian Sarge on lvm partitions that were originally identical.

We note that the experiments in this first section should not be considered the general case, as the experiments were conducted on artificially created workloads. These workloads seek to maximize the number of pages we can reclaim.

The setup used in the experiments is as follows: A number of VMs were started incrementally. Right before starting a machine we collected the number of currently shared, reclaimed and zeroed pages for later use.

7.2.1 Idle Virtual Machines

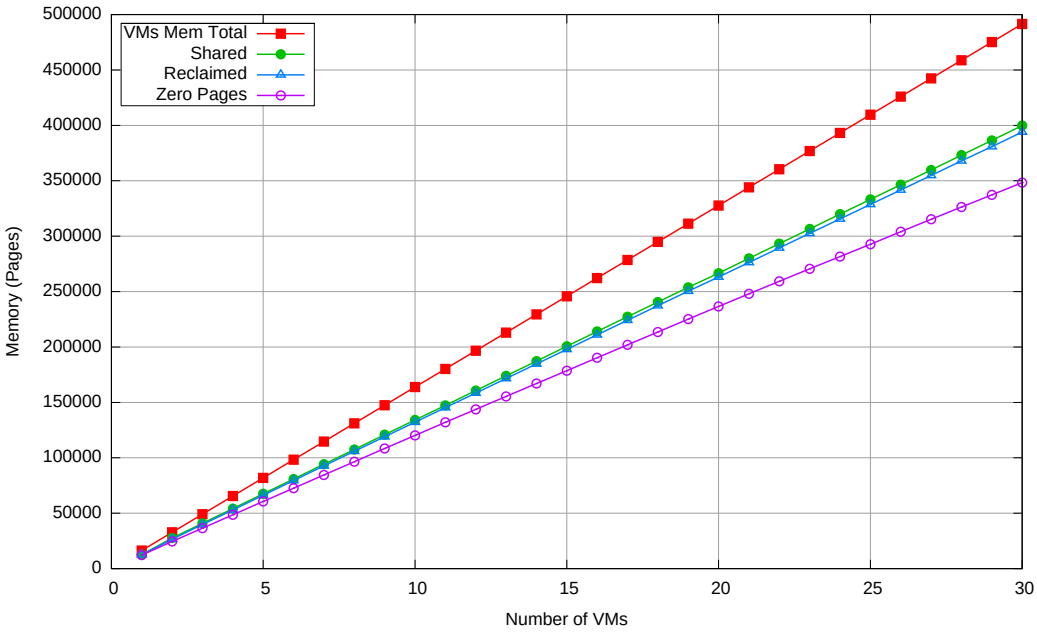
The first experiment consists of 30 idle VMs with a memory allocation of 64 MB. The VMs are started with a three minute interval, thus ensuring that we have plenty of time to reclaim pages before the next one is started.

The result is presented in Figure 7.1 on the facing page. The top graph shows the amounts of shared, reclaimed and zero pages in absolute numbers. The bottom graph shows the shared and reclaimed pages as a percentage of the currently running VM's initial memory allocation. From the figure we can see that approximately 80% of each VM's memory were shared and that approximately 70% of these are due to zero pages. It should be noted that this amount of reclaimed pages are highly unusual on other than idle workloads as we shall see in the following experiments.

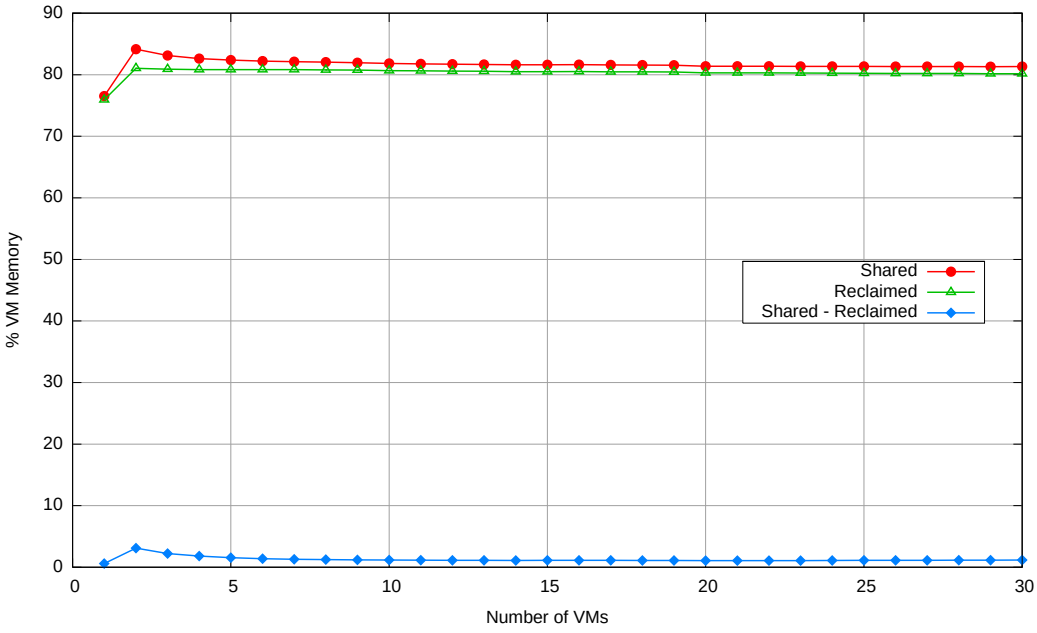
7.2.2 Virtual Machines running Kernel Compiles

The second experiment examines the sharing potentials under a non-idle workload. We execute the same tasks on every VM, namely the compilation of the 2.6.16 Linux kernel. There was a delay of 30 minutes between starting each VM in this experiment to ensure that one compilation was finished before starting another. This ensures that data left in memory should be almost identical after the compilation on most VMs and that there were plenty of time to ensure that most shares were found and set up.

The results are presented in Figure 7.2 on page 74, where we notice that most of the zero pages have been used for the compilation process. Furthermore the sharing percentage has significantly decreased compared to the idle experiment, but is still good. The set of shared pages now mostly reflect the pages used for the compilation process instead of zero pages. Worth noticing is that the difference between shared pages and reclaimed pages have increased significantly. This difference reflects the number of *shared copies*. Before doing sharing, there are a number of pages that are identical. When sharing is created a number of pages becomes superfluous, thus they are reclaimed. Some of the pages must however remain in memory to serve as shared pages for the domains. In our implementation these are all the pages that are owned by the share domain. We refer to these pages left in memory as shared copies. In the idle VM experiment above, the number of shared copies was low, because most of the shares were zero pages that were removed by pointing them to a single page in memory.



(a) Absolute



(b) Aggregate

Figure 7.1: Absolute and aggregate graphs for 30 idle VMs. Due to zero pages the sharing percentage is very high.

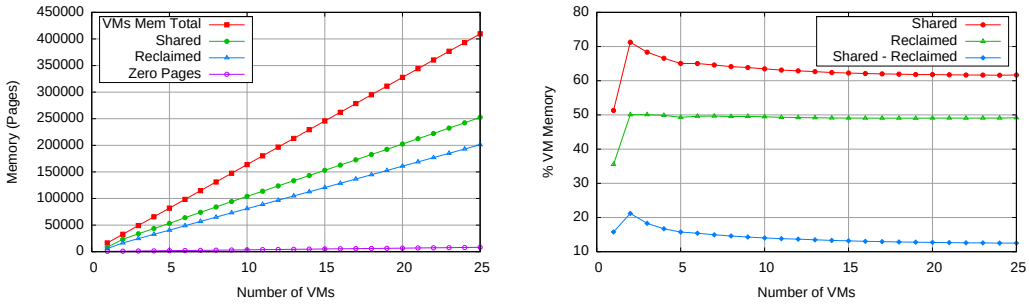


Figure 7.2: Absolute and aggregate graphs for 25 concurrent VMs, each having performed one kernel compile. The share percentage is high, while there is a very low amount of zero pages.

7.3 Feasibility of Sharing Zero Pages

When Xen boots up, one of the things done during the process is to scrub all unused pages, i.e. fill them with zeros. It is optional to scrub pages when starting a VM. If a VM is started without scrubbing the pages that it allocates, then potentially the VM has the possibility to examine the contents of the pages and thus pick up leaked data from a previous run VM. On the other hand scrubbing every page is an expensive operation, so there is a trade-off.

Anyways, there is a good probability that a high number of pages within a newly booted VM are zeroed, either due to the initial Xen boot up scrubbing or due to scrubbing when starting the VM. We could easily share these pages, thus freeing a large number of pages. Our experiences however tell us that most free pages within a VM are quickly used when the VM becomes busy. To examine just how quickly, we carried out an experiment with a single VM first sharing all its zero pages and then starting a kernel compile. The result is presented in Figure 7.3 on the facing page. As can be seen all zero pages are used within approximately 70 seconds. This is consistent with the results in Figure 7.2, where we saw that nearly all the zero pages were consumed at the end of the kernel compiles.

Our main concern about sharing zero pages is that as soon as there is a workload, then the pages are rapidly consumed. The tasks of sharing all the zero pages, just to tear them down again is not only superfluous, it is even a performance concern. It is a concern because a write operation to a shared read-only page will trigger the tear down code, which is an expensive sequence of actions that include at least a page fault and page copy as well as updating the corresponding Shadow Page Table (SPT). Furthermore this will happen in time critical code (as opposed to the code executed while in the idle loop).

To investigate whether this problem outlined is only a theoretical overhead or if it really does make a different, we timed a number of kernel compiles on a 700 MHz AMD Athlon machine running our implementation. Ten runs were made where we share zero pages and correspondingly without sharing zero pages. The results of this experiment is presented in Table 7.1 on the facing page.

On average the kernel compiles done while sharing zero pages were consistently slower than the compiles without sharing zero pages. So from this empirical data we

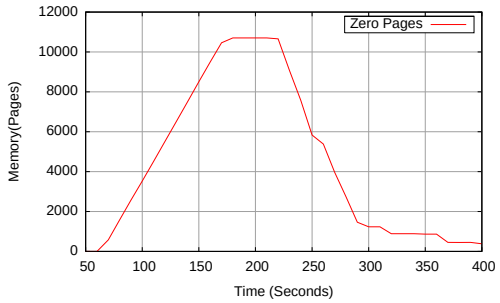


Figure 7.3: Zero page consumption on an 700 MHz AMD Athlon machine. After roughly 175 seconds all zero pages have been shared. After 220 seconds we activate a kernel compile and consequently the number of zero pages drop rapidly. After 290 seconds most of the zero pages have been used, so most of the zero pages are consumed within 70 seconds.

	Mean	Standard deviation	Skewness	Kurtosis	Minimum	Maximum	Sample size
Sharing Zero Pages	**1565.46**	12.11	-0.25	1.83	1546.46	1580.446	10
Filtering Zero Pages	**1515.33**	6.50	0.859	2.857	1506.61	1528.69	10

Table 7.1: Statistical analysis of time spent (seconds) on a number of kernel compiles. As can be seen, on average it takes 50 seconds more to compile the kernel when sharing zero pages compared to when not sharing zero pages.

can conclude that our hypothesis is real and that the problem outlined is a concern.

Sharing zero pages poses a dilemma, because on the one hand there are large amounts of pages that can be shared, but on the other hand tearing them down poses an overhead. When the VM containing a lot of zero pages is idle (and is expected to stay that way) then the pages may just as well be reclaimed, thus freeing pages to dynamically reassign or distribute to more busy VMs. On the other hand if the VM is expected to have a decent workload, then the best solution is not to share the zero pages.

So in order to answer our question as to whether sharing zero pages is feasible, the answer must be that unless the VM containing the zero pages is idle, then it is not feasible. Therefore we disregard zero pages by filtering them out in the rest of the experiments unless explicitly stated else.

7.4 Impact of using Different Binaries

Before moving on we are a bit curious as to how much sharing is obtained by using the same Linux distribution for the experiments. The reasoning behind this question is that different distributions have different binaries. Using different binaries should

Distributions	VM nr.	**Mean**	Standard deviation	Skewness	Kurtosis	Minimum	Maximum	Sample size
Only Gentoo								
	VM 1	**11275.2**	735.87	-0.31	1.40	10352	12064	5
	VM 2	**11264.0**	729.19	-0.30	1.60	10296	12088	5
Only Debian								
	VM 1	**10332.0**	150.60	-0.47	1.95	10112	10500	5
	VM 2	**10362.4**	151.88	-0.96	2.59	10112	10504	5
One of each								
	VM 1	**3490.4**	83.07	-0.95	2.33	3356	3556	5
	VM 2	**3605.6**	168.80	-0.076	1.46	3400	3800	5

Table 7.2: Reclaimed memory (in KB) from VMs running combinations of homogeneous and heterogeneous distributions.

cause the shares that are due to shared application code to be eliminated.

In order to answer this we ran three different setups: 1) Two VMs running Gentoo, 2) two VMs running Debian and 3) one VM with Gentoo and one with Debian. The results are presented in Table 7.2.

As can be seen in the table, as long as we are running VMs with homogeneous distributions, then the number of reclaimed pages is always higher. In fact one could be surprised that two distributions running their own version of the kernel, all binaries compiled by their own version of gcc and with different versions of services could have anything to share. First of all it should be mentioned that in this experiment all the VMs ran the domU kernel provided by Xen, so this should be shareable no matter what. This takes up 2.6 MB uncompressed. As only 3.5 MB is shared between VMs running heterogeneous distributions, this leaves a remainder of approximately 1 MB, which may possibly contain kernel structures, page cache and read-only data pages from services.

As for our expectations for sharing pages between different OSes we have not investigated this further. We do however deem that the probability of sharing anything besides zero pages between different OSes is low. Mainly because there is so little to share between different distributions of the same OS. Further work could explore this question.

7.5 Worst Case Experiment

While a workload with many VMs running the same applications is near ideal conditions for sharing, it is interesting to explore the opposite scenario. Thus in this section we examine what we consider to be one of the worst cases. The worst case scenario is one where all the VMs are running different applications within different operating systems, where the applications have a high demand for memory and the contents of the memory changes rapidly.

Figure 7.4 shows the results of incrementally starting ten VMs that all run different benchmarks. Each benchmark is allowed to run for five minutes after the VM has been started, after that it is terminated. The benchmarks run in the different VMs are: Crafty, OSDB, SPECweb99, stress, tiobench, freebench, dbench, AIM, MWG and kernel build.

We did however deem that using ten different OSes or even ten different distributions was not feasible to set up. Furthermore we expect that people interested in sharing memory would use the same distributions to ensure as much sharing as possible. Therefore we also used Debian for this experiment and note that this only is a nearly the worst case.

Using different distributions would presumably lower the sharing rates, as shares due to the same applications being run are more unlikely to be found. Again using different kernels/OSes would probably remove the shares that are caused by sharing the same kernel.

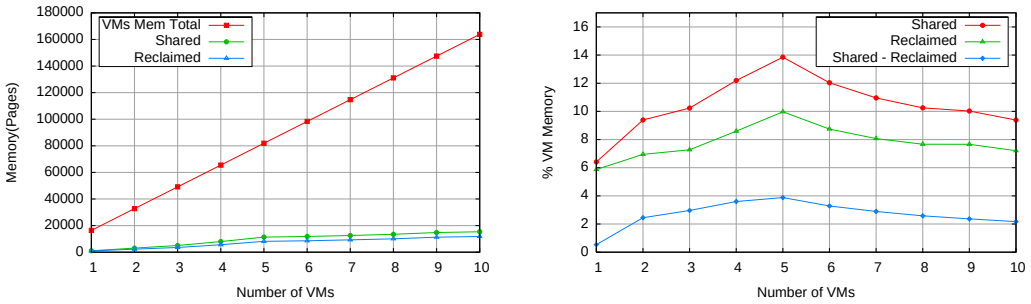


Figure 7.4: Absolute and aggregate graphs showing the results of the worst case experiment. The VMs (64 MB memory) are all running different applications, most with a very high processor load.

As can be seen from the graphs, the reclaim rate is somewhere between 6-10%. 6% of a VM running with 64 MB is roughly 3.6 MB, which seems to be consistent with the results from the previous section.

7.6 Synthetic Workload Experiments

Having examined the sharing potential under the best and the worst conditions, we now move onto some more interesting workloads. While the kernel compilation experiment was interesting as an example of sharing potential under similar conditions, we would much rather investigate how sharing behaves under real world workloads. As we do not have access to such workloads all we can do is to try to emulate one. So we do this by examining a number of different workloads in this section.

7.6.1 Virtual Machines running the Medium Workload Generator

The first experiments consists of a number of VMs with an allocation of 64 MB memory, running our MWG script. The VMs randomly uncompress and compiles

different open source packages, so shares due to the page caches should be coincidental. Most of the tools (bzip2, gzip, gcc, ld and make) used to compile the packages are however the same, so a minimum of sharing could be expected due to this.

Figure 7.5 shows the results of the experiments. Unlike the previous experiments this experiment was over a number of unrelated runs. The sharing numbers are somewhat more diverse and dependent on which packages were compiled in a particular run.

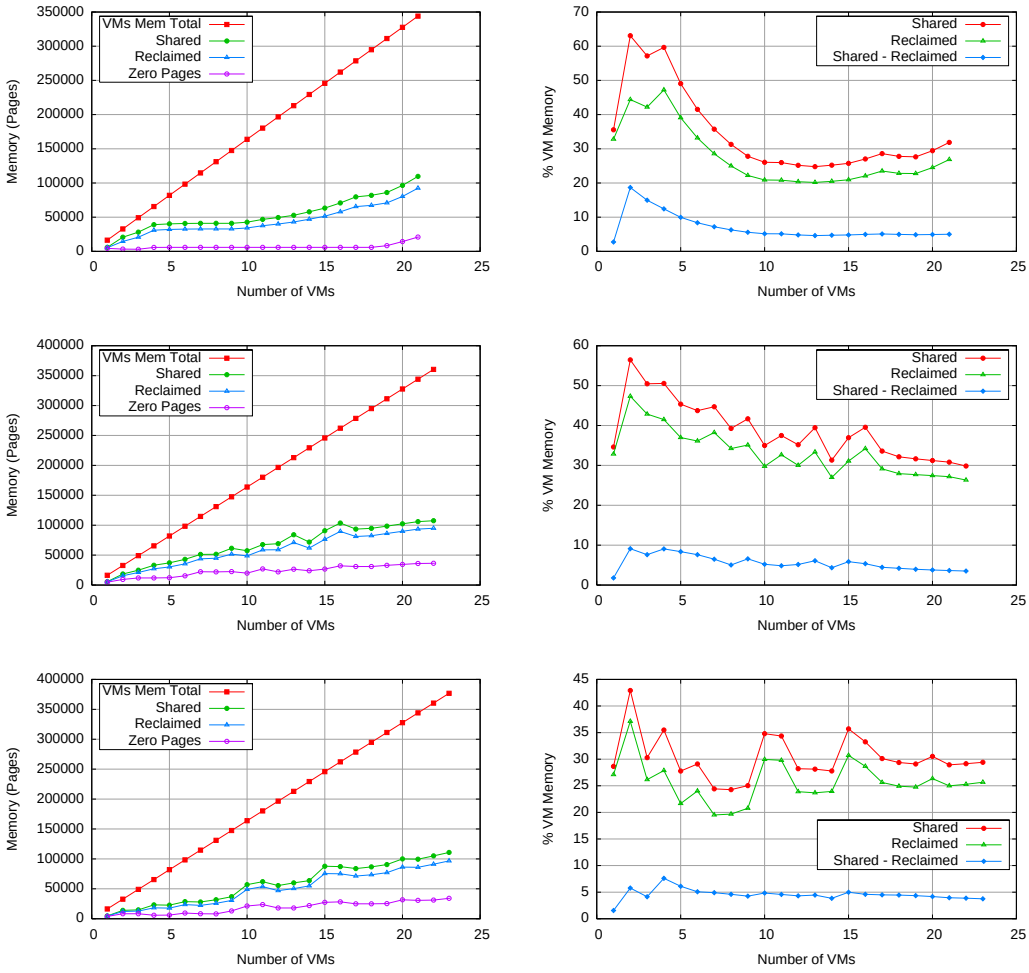


Figure 7.5: Absolute and aggregate graphs from three runs. Each represents a number of concurrently running VMs (64 MB memory) executing the MWG workload emulating script. It is interesting to see that regardless of the individual runs, the shares seem to climb up to roughly 100.000 reclaimed pages.

As can be seen from the figures, the number of shares are much more dependent on the particular workload during a run. We note that a share percentage ranging between 20-40% must be said to be a good percentage.

7.6.2 Virtual Machines running Mixed Workloads

Instead of looking at incrementally started VMs, we now move onto looking at how workloads behave over time. So the experiments in this section starts eight VMs concurrently, then wait five minutes and then start the individual benchmarks. Each VM run in the experiment had a memory allocation of 192 MB, amounting to a total of 1536 MB for the eight VMs run.

The individual setups for the eight VMs in the tests were: 1) OSDB, 2) SPECweb99, 3) MWG and finally a 4) mixed setup with VMs running different benchmarks. Specifically the benchmarks used in the latter setup were: SPECweb99, kernel compile, dbench, freebench, AIM, stress and one idle VM. We note that distinct datasets were generated for both OSDB and SPECweb99 for this experiment.

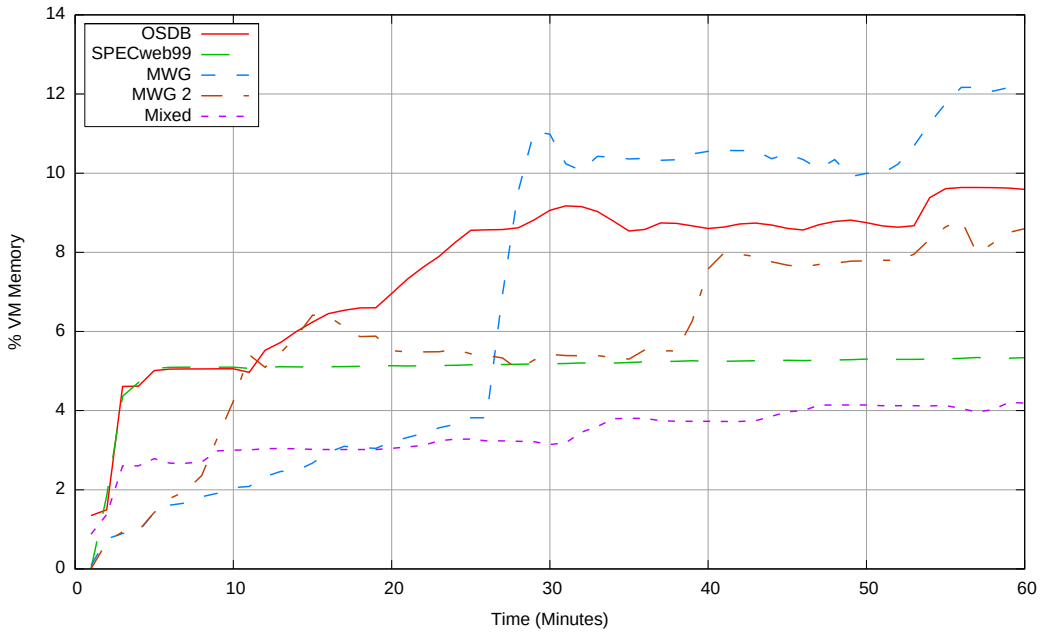


Figure 7.6: Reclaimed pages from eight VMs (192 MB memory each) under different workloads. The number of reclaimed pages is minimal on the SPECweb99 and Mixed workloads, while the other workloads find up to as much as 12%.

The results of the experiments are presented in Figure 7.6. It should be noted that the percentages are not directly comparable to those of the previous experiments because the VM memory size has changed.

Surprisingly the figure shows that the shares in the SPECweb99 experiment are rather low. Based on the results in the previous sections, a good guess would be that after the VMs are booted up, the applications and kernel binaries are shared between the VMs. We would have expected that there would be some sharing due to the execution of the benchmark. This however seems to be minimal.

As for the OSDB experiment, it was run on the same disk images as the SPECweb99 experiment. So this explains the common starting point in the shares. Contrary to the SPECweb99 test, there is actually found something to share during the benchmark.

The MWG script experiment ran as expected. In the beginning the sharing rates are low because the probability of the VMs are compiling the same packages is low. After some time the VMs encounter the same packages and because the VMs have a quite large memory allocation, the most recent compilations are allowed to stay in memory. Therefore it is no surprise that after roughly 25 minutes, the number of reclaimed page rises drastically. The second run of the experiment produced similar results.

Finally the mixed application experiment, much like the worst case experiment above, only finds a minimum of sharing. We ran this once more and saw roughly the same numbers.

The workloads with the best results were able to share up to 12%, which is roughly 180 MB. As can be seen in the figure, the percentage of shares is highly dependent on the workloads and the memory allocation of each VM. Furthermore it seems probable that if the VMs were allowed to run for longer periods of time, then the number of shares would have been larger as the page caches got filled. It seems evident that the more similar the workloads on the involved VMs are, the more we find to share.

This concludes our experiments with synthetic workloads.

7.7 Overcommitment

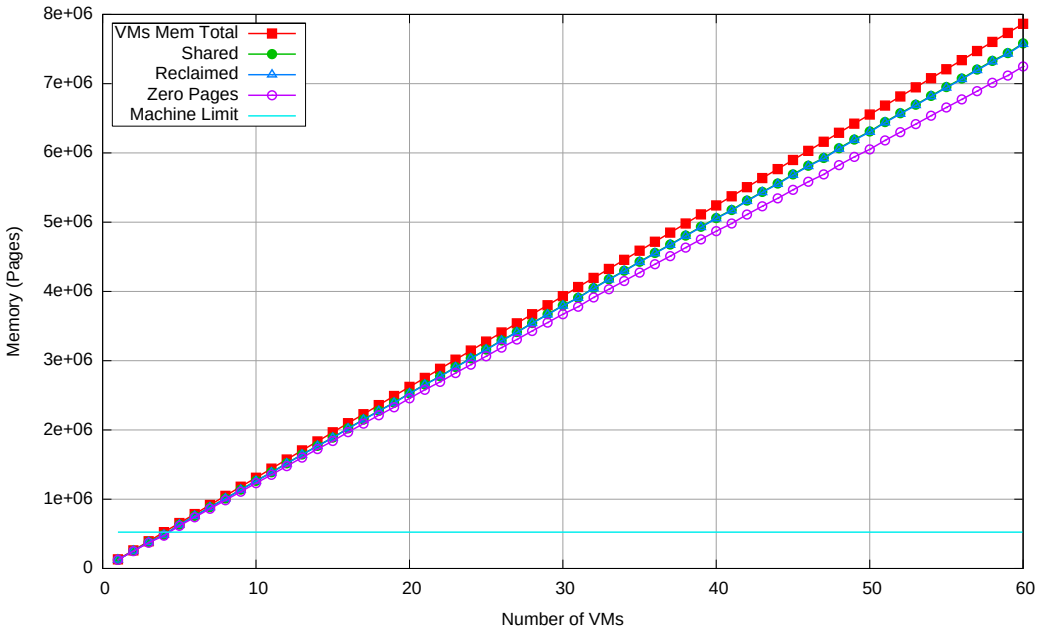


Figure 7.7: Overcommitment on a machine with 2 GB of memory. When 60 VMs are reached then the system is overcommitted by approximately 29 GB.

In this section we explore whether Xen can do overcommitment. Figure 7.7 shows up to 60 VMs running concurrently, each with a memory allocation of 512 MB. The VMs are idle, so a very large percentage of the pages are zero pages. The

horizontal line in the bottom of the graph indicates the memory limit of the physical machine, which is exceeded after starting only four machines. After that point we incrementally start 56 additional VMs, thus exceeding the machines memory limit by approximately 29 GB. If we had pursued the limit for concurrent machines in Xen (approximately 116 due to the limited size of the Xen heap[57, p. 11]) further, this may easily be doubled. Furthermore had we used 1.5 GB instead of 512 MB for each VM this could theoretically have overcommitted the machine by approximately 172 GB.

While the experiment has little practical use, it illustrates one important point. By sharing zero pages on idle machines there can be reclaimed large amounts of pages, but as soon as the VMs do any work there is a large probability that the VMs will need more private copies than is available. We note that the experiment could easily be done by allocating the same amount of pages and ballooning the pages out instead.

7.8 Impact of the Memory Allocation of Virtual Machines

To examine how the memory allocation of the VMs affect the sharing rate we carried out the following experiment: Eight VMs were started concurrently, each running a MySQL and Apache server. After ten minutes the SPECweb99 benchmark was started (one connection per VM). This was executed for the following VM memory allocations: 64, 128 and 192 MB.

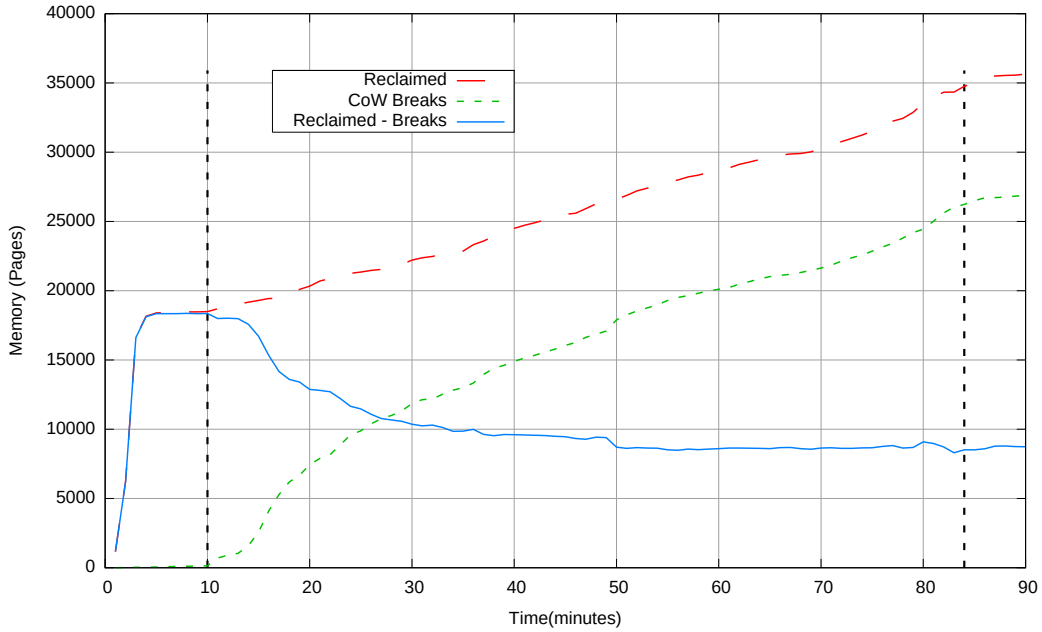
The results of the experiment are presented in Figure 7.8 on the following page. As can be seen from the figure, the 64 MB VM has a slightly lower share rate at the beginning of the experiment. As the benchmark starts, the VMs are forced to write to the shared pages and thus reuse them. The 128 MB VM on the other hand had sufficient memory free to avoid writing to the shared pages, so the shares are preserved in that experiment. There was no significant difference between the 128 and 192 MB VMs, so we omit the last graph.

In order to examine this scenario more thoroughly we carried out another set of experiments. Instead of using SPECweb99, we turned to a scenario we knew would result in the VMs exercising their page caches and thus hopefully pick up a larger amount of shares than in the first experiment in this section.

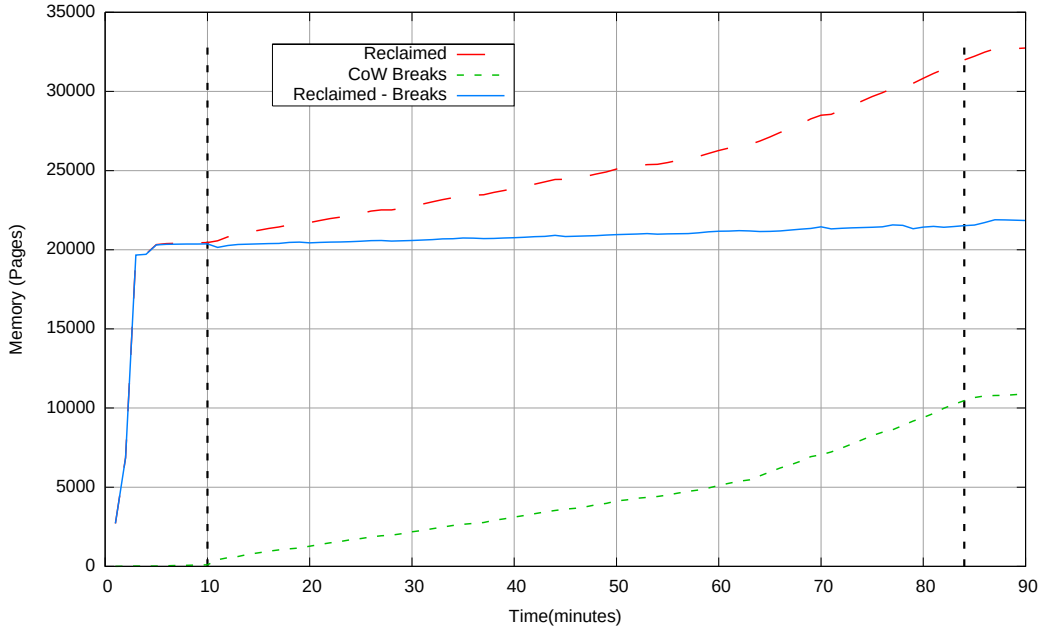
This next set of experiments ran four VMs with memory allocations ranging from 64 MB to 448 MB. These VMs were subjected to a script that chose a package randomly from Gentoo's package system[1] and compiled this (including dependencies). Once a compilation was done the VMs had a five minute break. The results of the experiments are presented in Figure 7.9 on page 83. We ensured that the different runs started from the same starting point, e.g. the run with 64 MB used disk images that were identical to those running with 128 MB etc.

As can be seen from the figure there seems to be a tendency, where the number of reclaimed pages rises with the amount of available memory. To better illustrate this point we provide Figure 7.10 on page 84, which shows the peaks of the graphs for each memory allocation size.

[1] The Gentoo distribution can be found at www.gentoo.org.

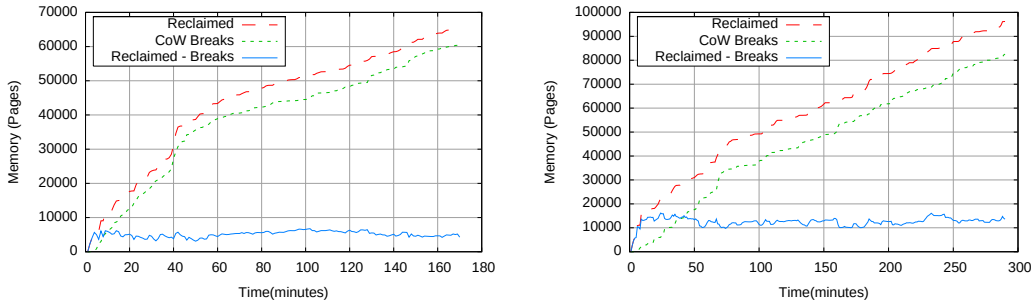


(a) VM with 64 MB memory



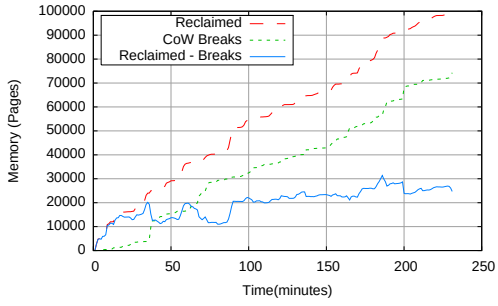
(b) VM with 128 MB memory

Figure 7.8: Aggregated reclaimed and Copy-on-Write (CoW) breaked pages for eight VMs running the SpecWEB99 benchmark. The vertical dotted lines indicate the beginning and end of the benchmark.

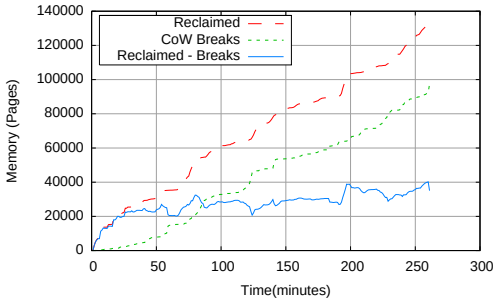


(a) 64 MB

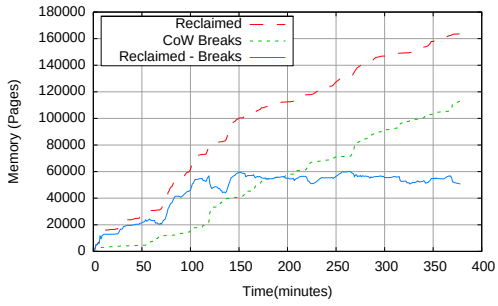
(b) 128 MB



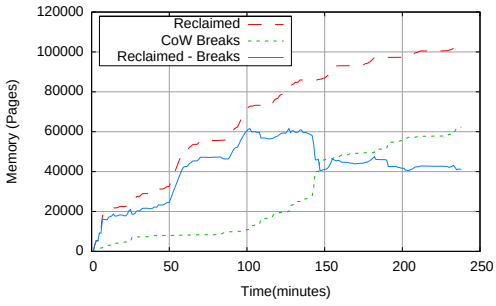
(c) 192 MB



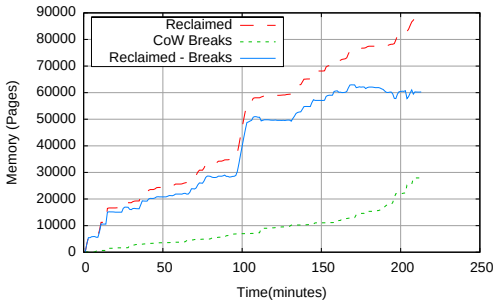
(d) 256 MB



(e) 320 MB



(f) 384 MB



(g) 448 MB

Figure 7.9: Four VMs with different memory allocation sizes subjected to randomly chosen compiles using Gentoo's Portage package system.

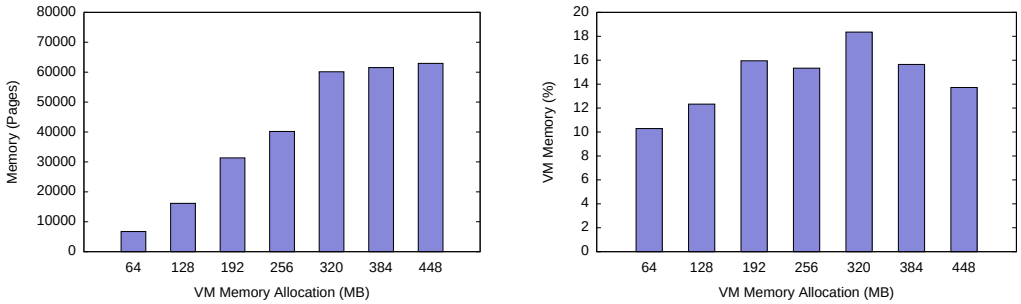


Figure 7.10: Maximum number of reclaimed pages from each memory allocation in Figure 7.9 on the preceding page. The graph on the left shows absolute numbers, while the graph on the right shows the numbers as a percentage of the VMs total memory.

As can be seen from the absolute graph in the figure, the tendency seems to be that the number of reclaimed pages rise linearly until the VM size of 384 MB. At this point it seems to flatten out. If we however turn to the other graph, which shows the number of reclaimed pages as a percentage of the total number of pages in all four VMs, then we see that roughly the same percentages are shared on all sizes.

So to answer the question of how the memory allocation of the VMs affect the overall share rate, then we can conclude that with a workload that uses some of the same applications, the sharing rate rises linearly with the size of the VMs. If we however look at the increase in the number of reclaimed pages as a percentage of the VMs memory size, then it is roughly consistent.

7.9 Comparison with Other Approaches

To conclude the experiments about page sharing, we now sum up our own results and compare them to those of VMware[58, p. 6-7] and Potemkin[57, p. 11].

We start with VMware's best case results. In their experiment they ran up to ten VMs with 40 MB memory, which were subjected to the SPEC95 benchmark. This ran for 30 minutes on all VMs, where after the number of shared and reclaimed pages were noted. The SPEC95 benchmark is a mix of processor intensive applications that are executed in the same order. Therefore it should leave the VMs in roughly the same state.

Our kernel compile experiments in Section 7.2.2 on page 72 should be roughly equivalent to the VMware setup. Both the kernel compile and the SPEC95 benchmark are processor intensive and leave the VMs in roughly the same states. Therefore their number of shared and reclaimed pages should be comparable with our setup. With ten VMs they were able to reclaim 55% of the VMs pages. With our setup we were able to reclaim 50% of the pages, so not only does VMware's results sound credible; we were actually able to recreate them rather precisely. Finally it perhaps is worth noticing that they used 40 MB VMs where we used 64 MB VMs. This means that the OS binaries for the VMs account for a larger percentage of the shares in the VMware setup and may very well account for the last 5% difference.

As for their real world workloads we were of course not able to reproduce these in an equivalent manner. Therefore we used synthetic workloads consisting of different benchmarks. VMware reported three setups: One Windows NT and two Linux. As we are not able to run Windows on Xen (at least not without hardware virtualization chips), we could not investigate the first result. We will however discuss the two Linux results.

The first setup consisted of nine VMs with memory allocations ranging between 64 and 768 MB. Their total allocation was 1846 MB. The applications run on the VMs were a mix of web and mail servers. In this setup they were able to reclaim 345 MB of which 70 MB were due to zero pages. With zero pages this constitutes 18.7% and 15% without zero pages.

The second setup consisted of five VMs with memory allocations ranging between 32 and 512 MB, with a total of 1658 MB. The servers ran applications such as web proxy, mail server and ssh. In this setup they were able to reclaim 120 MB of which 25 MB were due to zero pages. That amounts to 7.2% with zero pages and 5.7% without.

Our experiments in Section 7.6.1 and 7.6.2 on page 79 match these numbers, but also show potential for lower and higher shares based on the specific workload. The synthetic workloads seem to indicate that VMware's real world results are realistic, but the nature of our experiments predetermines that they remain inconclusive. The only way to fully determine it is to subject our implementation to a real workload with the same applications in the same environment.

Finally we present a simple little experiment carried out to compare content-based page sharing with forking in Potemkin. In the article they reported a setup where a 128 MB VM was booted. This was then suspended and subsequently 116 VMs were cloned from the initial VM. The clones took up 98 MB together, which roughly means that each cloned VM took up only 0.85 MB.

The first part of our experiment started one VM with 64 MB of memory and then another VM was cloned from this. Initially the child VM took up only 1064 KB, which matches Potemkin's number rather well. We tried to replicate this using content-based page sharing, we replicated this using two 64 MB VMs. The VMs were left idle and allowed to share zero pages. This way we were able to reduce the memory footprint of the VM to 4416 KB.

To return to our initial expectation about content-based page sharing being able to reclaim more pages than forking in Section 3.6 on page 22, we can actually conclude that we were wrong. The forking solution is able to keep the initial footprint smaller. We guess that the difference lies either in the pages that we filter out (as explained in Section 6.6 on page 62) or the nondeterminism involved in booting two different VMs compared to booting one VM and then forking it. Potentially such things as the OS random pool and other kernel structures that are dynamically changed during the boot process might account for the difference in the number of reclaimed pages.

Subsequently we wondered how much a workload would change these results. Therefore we started a kernel compile in both the cloned VM and the one replicating a cloned VM. After the compilations were done the memory footprint of the forked VM was 57572 KB in Potemkin and 56520 KB using content-based page sharing.

7.10 Chapter Conclusion and Summary

Contrary to the previous chapters we use this last section of the chapter to both summarize and conclude, as the results of the previous sections are best analyzed together.

In this chapter we performed a number of experiments to examine sharing under a number of workloads. Specifically we started with a couple of tests intended to see how much we could reclaim under good conditions. We saw that as long as the virtual machines are running the same tasks, then sharing up to 50% on 64 MB virtual machines is no problem. If we on the other hand examined how feasible the opposite situation were, we found that the amount of shares was reduced to around 6%. This corresponds to only a little more than the uncompressed binary of the Xen modified Linux kernel, which all the virtual machines were running. Assuming that most workloads would exhibit values between these two extremes, we carried out a number of synthetic workloads.

Our Medium Workload Generator script was designed to make a medium workload on the virtual machines and ensure that the they, after a period of time, would encounter the same tasks, only in a random order. Generally this specific workload resulted in a reclaim percentage of 20-40%. Finally to conclude the series of synthetic workloads we presented results from a number of different benchmarks. These ran on 192 MB virtual machines and generally the shares were between 4-12%. Again we note that the percentages are not directly comparable between virtual machines of different sizes, because the shares that are due to sharing the kernel (which are certain to be shared on all workloads) constitutes a lower percentage on virtual machines with a higher memory allocation.

As for the contents of the pages we can share, we only found indications of what the pages are typically used for. As explained it seems quite probable that all the virtual machines in the experiments were able to share the same binary kernel image, as this was used on all the experiments. The fact that when we tried running two virtual machines with different Linux distributions, the shares dropped to roughly the size of the kernel, seems to support this theory. As for the experiments that were carried out using the same distributions but with different applications, they also found roughly the same amount of shares. So we can conclude that as long as the virtual machines are not set up the same way and not running the same applications with the same binaries, then the probability of sharing anything is low. As for the experiments carried out when the virtual machines were running similar workloads, this was, as expected, the cases where the number of shared pages increased quite a bit. Our conclusions are that the shares under these circumstances are due to shared application data left in memory by dead processes and the page cache. We did however not examine this in detail, so this conclusion is only based on indications.

Furthermore we found that sharing zero pages on active workloads only presents an overhead, but on idle virtual machines it is feasible to share them as a means of dynamically distributing the memory allocation. This was illustrated by an experiment that succeeded in overcommiting a system with 16 times its actual memory.

Chapter 8

Performance Evaluation

In this chapter we will investigate whether there are possible performance penalties inflicted on Xen by our implementation. We start by examining the overall performance of Xen relative to our implementation. Our approach is to use standardized benchmarks, so we get a feel for how much any overheads affect ordinary user space applications. We will then dive into the code and perform a series of micro benchmarks to investigate the actual functions used in the implementation to determine if there is room for optimizations and improvements.

8.1 Evaluation using Benchmarks

In this section we perform a series of benchmarks to evaluate the overhead of our implementation. In particular we want to answer the following questions:

- Is there a performance overhead in sharing pages under different conditions and workloads?

- Is there a performance penalty in using shadow page tables?

Initially we carried out the experiments on one machine, but encountered some strange results, particularly in the form of larger overheads than those presented in [5], [19] and [11], so we redid the experiments on another (larger) machine. Our main concern was that we expected the poor performance on the first machine to be hardware bound, e.g. in the form of too low processing power.

The first machine is a 700 MHz AMD Athlon with 340 MB of memory and a single 100 Mbit network interface. The other is a Sun Fire X4100 server with two 2200 MHz dual core AMD Opteron processors and 4 GB of memory with a single 1 Gbit network interface. It should be noted that the experiments carried out in native Linux on the Sun Fire had 4 GB of memory, while all other experiments were conducted with only 64 MB of memory. Furthermore all the Virtual Machines (VMs) ran using disk partitions for the guest Operating Systems (OSes) instead of using disk images to guarantee that there is no extra overhead in disk access from inside the VMs. For all test setups there is one VM running plus the privileged VM, except for the experiments which involves sharing pages. In these setups we run two

VMs plus the privileged VM to ensure that some level of sharing were obtained, thus exercising both the Copy-on-Write (CoW) sharing and breaking mechanisms.

For the experiments we have used five of the tools mentioned in Chapter 7 on page 69, namely: Kernel build times, OSDB, dbench, SPECweb99 and AIM. The benchmarks were carried out on a number of different setups: Debian/Ubuntu Linux (L), Xen[1] (X), Xen with Shadow Page Tables (SPTs) (S), Xen with Potemkin[2] (P), our implementation without sharing (D) and our implementation with sharing[3] (C). Out of curiosity we chose to include results for native 2.6.8 Linux kernel in Debian Sarge and on a 2.6.16 Linux kernel in Ubuntu, so we knew what to expect from a non-virtualized setup compared to the virtualized setups.

In particular we chose these setups for three purposes: 1) To evaluate the overhead in using SPTs, 2) to determine possible overheads of the Potemkin framework and 3) to examine the overheads of our implementation with and without sharing.

The results of the experiments in this section is the mean of three runs and the complete set of results can be found in Appendix C on page 127. We note that the results vary significantly in some of the results of the OSDB benchmarks, so the comparison between results from different setups should be taken lightly. The results of the experiments are presented in Figure 8.1(a) for the 700 MHz AMD Athlon machine and for the Sun Fire in Figure 8.1(b) on the next page.

Generally the results deviates from those presented in the before mentioned articles: [5], [19] and [11]. [11] reported that they had problems obtaining the same results as [5] until they got help, so the impact of the setups may be significant. So we cannot rule out that more attention to the individual setups could have yielded better performance. Furthermore remember that the native Linux experiments were conducted with 4 GB of memory on the Sun Fire, which most probably accounts for the anomalies in the graphs regarding the performance of Linux versus Xen. Finally we note that the SPECweb99 benchmark run on the 700 MHz AMD machine kept only one continuous connection, while the Sun Fire machine kept 20.

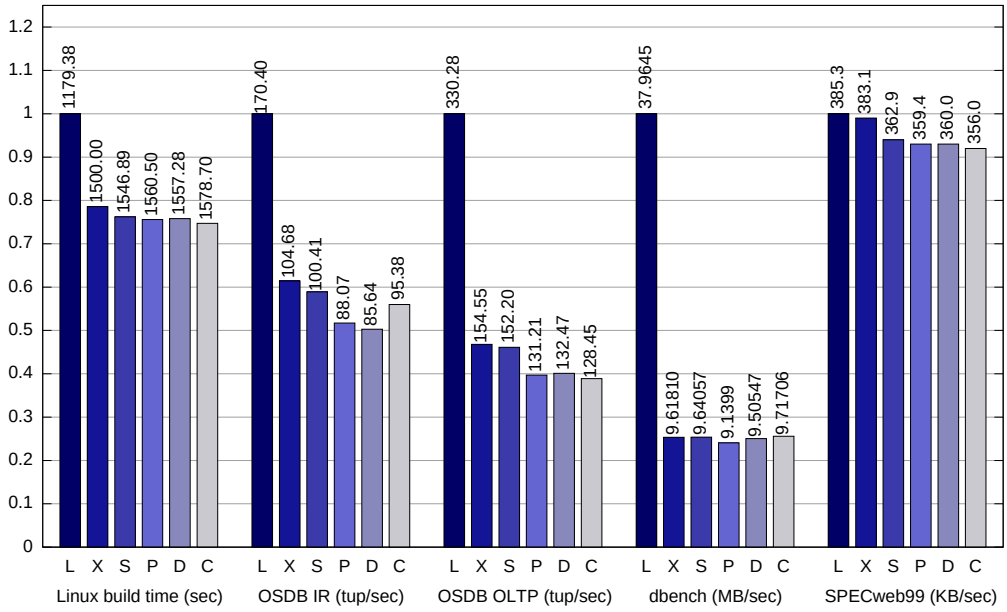
In an effort to further investigate the anomalies we tried changing the amount of memory during the kernel build and dbench benchmarks. As it can be seen in Figure 8.2 on page 90 this has a high impact on the performance scores in the benchmarks. When the memory size is 128 MB in the dbench benchmark, the Linux score is almost equal to that of Xen and our implementation. Increasing the memory allocation, further shows that Xen does not seem to be able to take advantage of the increase in memory. As for the kernel compiles the difference between ours and the others seems minimal. Based on these results we attribute the strange results in the first graph to the VM memory size and perhaps poor setups on our behalves. Finally we note that determining the real causes of the performance anomalies between native Linux and Xen, though they may be very interesting, are not central to this thesis.

As for the performance of our implementation and possible overheads, when we compare the overall results in Figure 8.1 on the next page, we find that there are only minor differences between Xen with SPTs and our implementation. In some

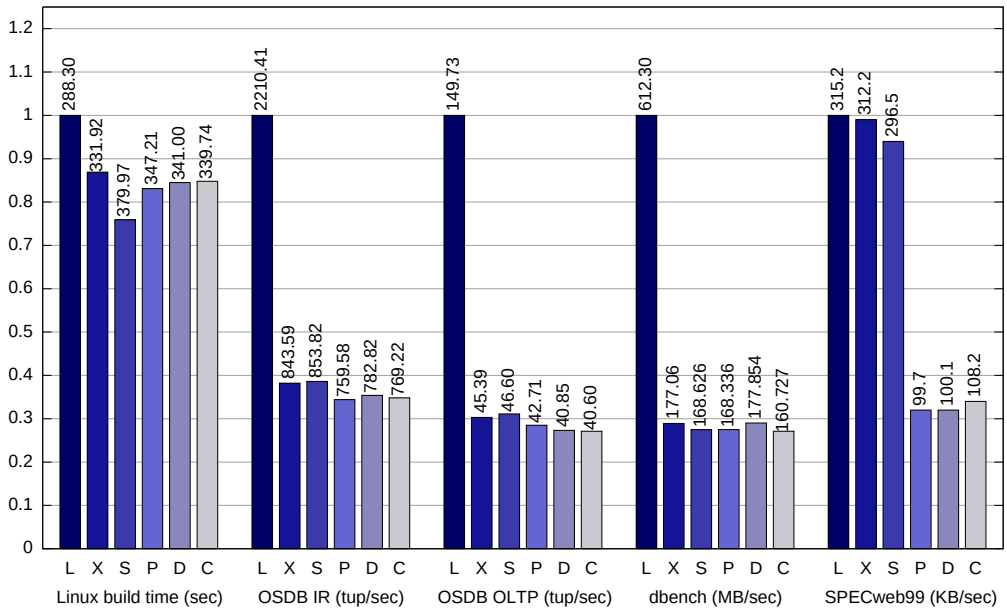
[1]The version used in the experiments is changeset 9883 from Xen's unstable repository.

[2]The Potemkin implementation was obtained by patching Xen unstable changeset 9515 with the Potemkin patch set from April.

[3]The version used under these experiments were changeset 9785.

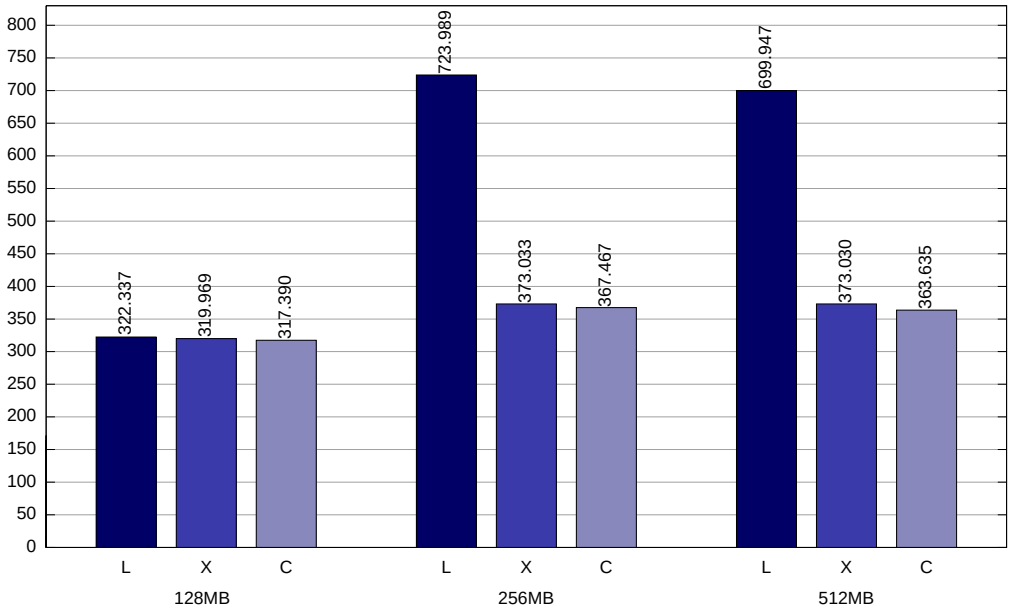


(a) The performance results for the 700 MHz AMD Athlon machine running Debian.

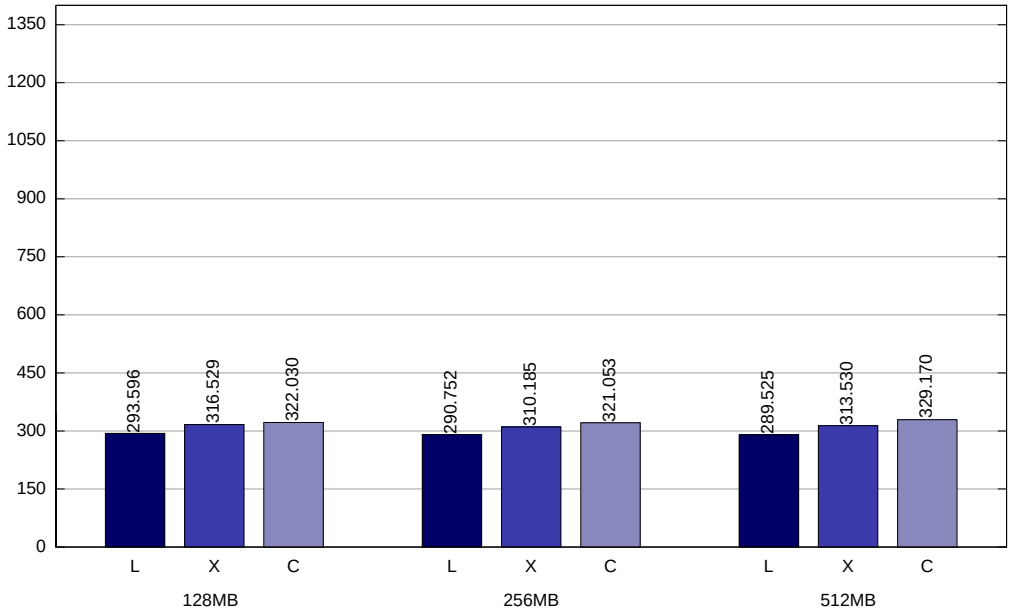


(b) The performance results for the Sun Fire X4100 server running Ubuntu.

Figure 8.1: Overall performance benchmarks. The results in the graphs is relative to the performance of Linux (L). Xen (X), Xen with SPTs (S), Potemkin (P), Xen with memory scans (D) and Xen with CBPS (C). High columns indicates good performance.



(a) dbench



(b) Kernel build times.

Figure 8.2: Memory allocations impact on benchmarks under Ubuntu Linux (L), Xen (X) without SPTs and Content-based page sharing (C). The graphs illustrate the performance results of dbench and kernel builds with a memory allocation of 128-512 MB of memory. These experiments were conducted on the Sun Fire X4100 machine. It should be noticed that in 8.2(b) lower scores is better.

700 MHz AMD

Operation	XenU	XenU SPT	Unit/second
exec_test	149.65	117.96	Program Loads
fork_test	835.27	592.05	Task Creations
shell_rtns_1	35.31	27.13	Shell Scripts
shell_rtns_2	35.64	26.60	Shell Scripts
shell_rtns_3	36.26	27.15	Shell Scripts

Sun Fire X4100

Operation	XenU	XenU SPT	Unit/second
exec_test	828.50	565.58	Program Loads
fork_test	3554.98	2013.00	Task Creations
shell_rtns_1	182.44	134.13	Shell Scripts
shell_rtns_2	182.65	135.13	Shell Scripts
shell_rtns_3	182.77	134.83	Shell Scripts

Table 8.1: Results from AIM benchmark for XenU with and without Shadow Page Tables (SPTs). See Table B.2 and Table B.6 on page 123 for the unabridged result sets.

results we score higher, in some results we score a little lower. In general we can conclude that at least on an overall level our implementation does not seem to have any large overheads.

As for whether there is an overhead in using Potemkin as a basis for our implementation we notice that there is a sudden drop of performance in the SPECweb99 benchmark starting with the run of Potemkin. As Potemkin is still under development and it is not yet officially released, we attribute this to an issue that may very well be resolved in future versions.

We were also interested in how Xen using SPTs performs compared to Xen without SPTs. As we can see from the results, there seems to be an overhead. To investigate the matter further, we first turned to the AIM benchmark. A complete list of results can be found in Appendix B on page 117. In the experiment we found that there were significant overheads in the `exec` and `fork` tests when performed in the VMs, as presented in Table 8.1.

The overhead of using SPTs lies between $21 - 31\%$ when executing a program and $29 - 43\%$ when forking a program. When AIM runs its `exec_test` it 1) executes a new shell, 2) waits until it exits and 3) repeats this for the entire duration of the test. The fork test is done in the same way, but it exits immediately after a fork operation has been performed.

Finally we found overheads in the AIM tests that starts shell scripts. The tests simply count how many times a shell script can be executed. The three shell tests are slightly slower when using SPTs, which should be a direct consequence of the slower `exec_test`. We remind the reader that the AIM experiments are run from user space, so the potential error margin may be large. Furthermore as the AIM tests are rather simple, we wanted to exhibit a more diverse workload to the SPTs.

We choose to use the SPECweb99 benchmark to examine this further. In this benchmark the total throughput rises with the number of concurrent connections.

Connetions	4	8	16	32	64
Shadow L1	141	152	173	196	245
Shadow L2	21	23	27	27	30

Table 8.2: Number of SPTs after 40 minutes during the benchmark in Figure 8.3. L1 tables are Page Table Entry (PTE) and L2 are Page Global Directory (PGD) tables.

Therefore as we increase the number of simultaneous connections, we also exercise the memory more as more data is brought into memory.

The experiment was carried out on a 2.6 GHz P4 Northwood with 2 GB of memory and the VMs run had 512 MB of memory. We ran an Apache web server on a VM with a varying number of connections. This is done in two different setups, one with SPTs and one without. The reader should notice that on this benchmark run we enabled performance counters, so the results in this experiment cannot be directly compared to the previous SPECweb99 results. The results of the experiment is pictured in Figure 8.3.

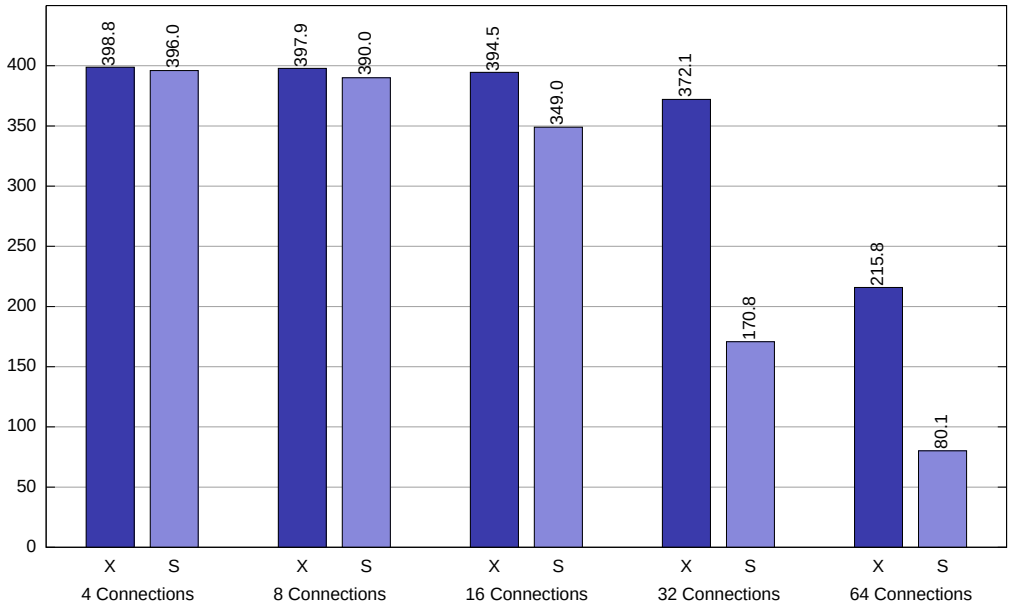


Figure 8.3: Benchmark results from experiment on the overhead of Shadow Page Tables (SPTs). The figure shows the throughput in KB/s for the SPECweb99 benchmark with 4-64 continuous connections for Xen (X) and Xen with SPTs (S).

As can be seen from the figure, the throughput of the run with SPTs degrades extensively compared to the run without SPTs, as the number of connections rise. It seems that the processor time needed to keep the SPTs up-to-date takes too many processor cycles and as a consequence the throughput falls.

During the benchmarks we examined how many SPT there were and how large they were. The results of this is presented in Table 8.2. As can be seen the number

of L2 tables (one for each process) does not increase significantly as the throughput rises. The number of L1 tables (indicating roughly the number of entries used in the L2 tables) rises significantly more. So this explains why the throughput drops, the amount of SPTs to keep synchronized simply gets too high.

So to conclude the question of whether there is an overhead in using SPTs, there is a noticeable overhead in executing/forking processes. Also the performance seems to degrade as the number of PTE tables increase. This seems logical as there are more operations required to create and maintain the SPTs.

To conclude the overall benchmark, we found that there are no apparent performance penalties in our implementation, except in the use of SPTs. Though they are expensive during forks, it should be noted that this is not that common an operation. However on forking intensive workloads the use of SPTs might incur large performance penalties.

In order to fully examine how expensive the individual operations in our implementation are, we have performed a series of micro benchmarks, which will be presented in the next section.

8.2 Micro Benchmarks

In this section we investigate our implementation and try to identify any overheads through a number of micro benchmarks. Therefore we start by regarding the most frequently used of the basic functions in the implementation. As we uncover expensive operations we narrow in on those operations and examine them in more detail. Thus in this section we are trying to answer the following questions:

- How expensive are the most commonly used operations in our implementation ?

- Can we optimize anything with this knowledge?

The micro benchmarks are performed by using a high resolution timer, which measures the number of clock ticks occurred on the processor since it was last reset. The timer is a 64 bit counter, which means that we will not get in any problems with overflow, in the small intervals we will measure.[1][p. 403],[9][p. 228]

We have used the timer to measure the number of processor cycles a given function spends when executed and thereby obtain a measure for the cost of the function. For each function we collect a large number of readings to avoid statistical deviations.

It should be noted that the number of processor cycles used on a function might differ quite significantly on different CPUs, since one CPU might be able to execute a single instruction while another CPU may have to execute a series of instructions to accomplish the same operation. Hence the results represented in this section might not give a correct picture of the processor cycles used for an operation on a different processor than the processor we ran the benchmarks on. Since we in this benchmark want to compare the different functions in the implementation with each other, this method fits the purpose.

We note the results in the following sections are results from different runs. Therefore the number of and size of the SPTs most probably vary and as a consequence only times from the same run can be compared directly. The latter two

tables in this section each contain results from a single run. So the numbers within one of those tables can be compared against the other numbers in that table.

The micro benchmarks are all conducted on the same machine, which is a 700 MHz AMD Athlon Processor with 512 KB cache and 320 MB memory. The front side bus is running at 100 MHz. For all the experiments there are two VMs running plus the privileged VM. The non-privileged VMs were configured with 64 and 96 MB of memory and the privileged VM had 64 MB of memory.

Having covered the method used in the micro benchmarks, we now present the actual results.

8.2.1 Benchmark of Frequently Used Functions

We start the micro benchmarks by investigating the cost of the more interesting functions. The results of these benchmarks can be seen in Table 8.3 on the facing page. The functions in the table are listed in the same order as they are explained in the following:

SuperFastHash (SFH): This function is called each time a page is hashed. It is called from the page hashing component, which means that it is only run when the CPU is idle, hence the performance of this function is not critical for the systems responsiveness. If it is slow, then it only affects the performance of the page hashing component.

Copy page: This operation copies the content of one page into another as the name implies. As it can be read in the design it is used rather often both in the setup of shared pages and in breaking them down.

Compare pages: This function is used each time we have a pair of pages that potentially can be shared. It uses the function `memcmp`, which can be seen just below compare pages in Table 8.3, where it also can be seen that it accounts for most of the cycles spent in the compare pages function. The `memcmp` function stops comparing two pages, when it finds a difference in them. So the worst case scenario for this function is when two pages are in fact identical, because it has to run through the whole page. The table only includes the results where the pages where bitwise identical, meaning the worst case results.

Zero page check: A simple check of whether a given page is a zero page. This operation is not used for anything other than sanity checks in the implementation. As it is much faster than the SFH function, there might be an opportunity for optimization of the identification of zero pages in the page hashing component. So instead of always hashing any given page, we first check it with a specialized zero page checking function. Besides being faster than hashing a page, it also has the advantage of being able to return as soon as it finds a part of the page that is not zero. This means the check will not use most time when called on a non zero page, unless the first part of the page contains only unset bits.

(Un)Map domain page: These two functions are generally very fast and they are used very often, e.g. each time the content of a page is read. The maximum

Operation:	**Mean**	Standard deviation	Skewness	Kurtosis	Minimum	Maximum	Sample size
Super-FastHash	**22000**	1626	18.40	470.55	13141	69760	93267
Copy page	**9936**	1903	0.73	12.25	3982	22789	3063
Compare page	**35955**	14791	8.27	80.75	21563	228697	34476
memcmp (identical)	**31379**	13799	8.65	86.78	17380	207113	33298
Zero page check	**3087**	805	52.19	2779.22	2998	48582	29368
Map domain page	**339**	287	51.61	6576.58	197	42687	145524
Unmap domain page	**306**	278	102.02	14739.06	197	42687	72762
Alloc dom-heap page	**2575**	996	1.27	5.07	895	8182	7886
Break CoW (copy)	**2422795**	339927	-0.54	3.36	725921	3300204	15566
Break CoW (convert)	**995**	564	2.94	26.28	120	6246	563
Change mapping	**7232671**	2872469	-0.73	2.18	251133	11364587	49947

Table 8.3: Statistical analysis of the results from micro benchmarks of the most frequently used functions in the implementation. Each row in the table were collected from separate runs.

values of this operation are, as explained in the latter part of Section 6.3 on page 50, presumable due to TLB flushes every 1024 time the function is used.

Allocate from domheap: This function allocates a page from the domain heap and is called on each CoW break (copy) as well as each time a new shared page is created. Compared to the other operations it is not expensive, so we will not dig further into it.

Break CoW (copy): In contrast to the SFH function the break CoW sharing function is run when the system is working on a task and has tried to write to a page that is CoW shared. This write operation will result in a write fault on the shared page and a private copy is created. This operation calls a function named `shadow_remove_all_access`, which as we shall see later on, is responsible for most of the processor cycles used in the break CoW function.

Break CoW (convert): This function is an optimization of the Break CoW (copy) above. It should be used instead of the above function when the page being CoW broken has a type_count of 1, which indicates that only one VM is using this shared page. In this case it can safely be converted to a normal page, that is owned by the one VM that still has a reference to it, instead of making a private copy of it.

Change mapping: This operation covers the changing of mappings in the Physical-to-Machine (`P2M`) and Machine-to-Physical (`M2P`) tables and the necessary synchronization after these updates. It is called on each setup of a shared page. As can be seen from the table, it is by far the most expensive operation in the implementation, but as with the SFH function, it is only called when the system is idle, so it should not interfere with the responsiveness of the system.

We were a bit surprised that a page comparison is actually more expensive than a page copy, which in fact is not as expensive as we expected. As our operations need to map in a given page to read the contents of it, we were relieved that this operation is as cheap as it is. Finally the two most expensive operations are, as could be expected, the operations that operate on the SPTs. Because these are so expensive, the following sections investigate exactly why these functions are so costly.

8.2.2 Investigation of the Expensive Operations

As we uncovered in the last section the change mapping and break CoW operations are very expensive compared to the other operations in our implementation. Therefore we explore the costs of the sub-functions used within these functions. These are presented in Table 8.4 on the facing page. It should be mentioned that the `set entries` operation in the table actually covers two functions in the actual code, both used for setting entries in a mapping.

guest_physmap_remove_page: This function invalidates the mapping of an old invalid Machine Frame Number (MFN) in both the `P2M` and `M2P` tables. It primarily uses the `free_shadow_page`, `set_entries` and `shadow_sync_and_drop_references` functions.

Operation:	**Mean**	Standard deviation	Skewness	Kurtosis	Minimum	Maximum	Sample size
Change mapping	**6938934**	1577294	0.85	2.79	1438681	11346164	34664
Guest phys-map remove	**2009855**	516399	0.70	2.30	355527	3298847	34664
Guest phys-map add	**1695540**	379101	0.83	2.90	345185	2823024	34664
Shadow remove all access	**1622055**	361458	0.75	2.92	342352	2703164	34664
Put page and type	**1603276**	360921	0.71	2.95	347731	2646831	34664

Table 8.4: Statistical analysis of the subfunctions of the change mapping operation. Unlike the first table, the data in this table originates from the same run, so they can be compared directly. The four lower rows can be added together to roughly give the value of the change mapping operation.

guest_physmap_add_page: This function creates the new mappings to a MFN in both the `P2M` and `M2P` tables. It primarily uses the `shadow_sync_and_drop_references` and `set entries` functions.

shadow_remove_all_access: This function removes all access to a given MFN. It does so by traversing all SPTs of a given VM, while replacing all Page Table (PT) entries that contain the given MFN with an empty entry. This is rather expensive operations. Exactly how expensive varies depending on the number and size of the SPTs currently in the domain.

put_page_and_type: This function decrements the reference and type count of a given page. The function is not really expensive until the situation where it ends up freeing a page. This happens when the page has a count of one when the function is called. This means that the page should be freed to the the Xen domheap. When this happens, a number of things will take place, but of special interest is that the `shadow_drop_references` function is called which again calls the `shadow_remove_all_access` function.

We provide a call graph to illustrate the connections between the functions with the highest costs in Figure 8.4 on the next page. As can be seen they all rely on the `shadow_remove_all_access` function. It should however be noted that the call graph has been simplified by removing uninteresting details.

As can be seen from the table the `shadow_remove_all_access` function is expensive and the `put_page_and_type` ends up being expensive when a page is freed because it calls the first function. The `guest_physmap_add_page` and `guest_physmap_remove_page` functions do however require some more investigation.

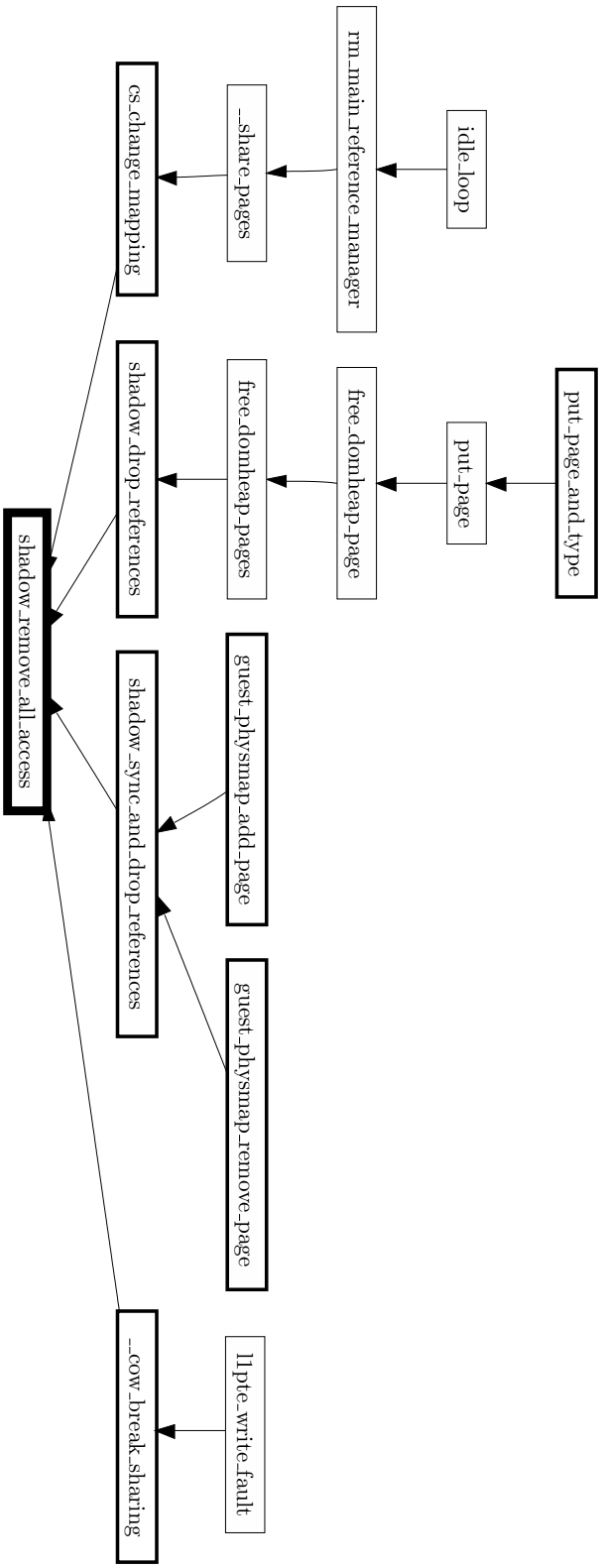


Figure 8.4: Simplified call graph showing the functions that call the expensive `shadow_remove_all_access` function. The set of functions that we have benchmarked have a thicker edge in the figure.

Operation:	**Mean**	Standard deviation	Skewness	Kurtosis	Minimum	Maximum	Sample size
Shadow sync and drop references	**1271309**	213864	1.23	3.96	299417	2154620	31019
Free shadow page	**320**	184	1.16	5.24	70	1657	31019
Set entries	**1780**	876	32.81	1611.48	906	47076	31019

Table 8.5: Statistical analysis of the functions called by the functions being examined. All the numbers are from the same run. The interesting operation is the first, as this is the expensive one.

This accounts for the functions in the change mapping operation. As for the break CoW operation, we also found that the `shadow_remove_all_access` function is the single most expensive function called by this operation. As this function is called, this means that the cost of the break CoW operation is determined by both the number of SPTs and their sizes.

8.2.3 Further Investigation of the Expensive Operations

The previous section showed that there were some expensive functions that could be investigated further, namely `guest_physmap_add_page` and `guest_physmap_remove_page`. They are quite similar in the way that they are constructed in terms of the functions they call. In this section we investigate the costs of these functions and the results can be seen in Table 8.5. The three interesting functions are as follows:

shadow_sync_and_drop_references: If the page given as argument is out of sync, then it synchronizes the SPTs. Then it drops all references to the page, which is done by calling the `shadow_remove_all_access` function. As we saw this function is rather expensive and is possibly responsible for almost all of the processor cycles used.

free_shadow_page: This function as the name suggest frees a shadow page and is not really expensive compared to the other functions in this section, so we will not investigate it further.

set_entries: This operation includes the following two functions: `set_p2m_entry` and `set_gpfn_from_mfn`. These two functions update the `P2M` and `M2P` mappings. As it can be seen in the table these updates are not really expensive, but the side effects of updating them are. This is primarily because we need to call the `shadow_sync_and_drop_references` function as we explained above.

Most of the processor cycles used in the implementation end up being used in the `shadow_remove_all_access` function. This is true for both the time critical parts of

the implementation and the not as critical idle loop context. Because this function is so expensive, further optimization efforts could easily start with this knowledge.

A set of minor possible optimizations were discovered during these micro benchmarks: 1) We found one unnecessary call to `shadow_remove_all_access` that can be removed, since it is done implicitly by another function on the same MFN. 2) There might be a small performance benefit in having a specialized function for the filtering of zero pages instead of identifying them by their hash value. As explained in Section 7.3 on page 74, as soon as there is a decent workload all the zero pages are used, so this optimization presumable is futile when the VMs are busy and even becomes an overhead. 3) Some of the `shadow_remove_all_access` calls could probably be reduced into one call, which takes more than one MFN as argument. Our guess is that traversing the SPTs is the most expensive part of the operation, so looking for more than one MFN to replace during each traversing might be more efficient.

8.3 Summary

This chapter examined possible overheads in our implementation. The approach was first to examine this at an overall level by using common benchmarks to see how good the performance is at the application level. In particular we applied the benchmarks to a number of setups, ranging from an unmodified Linux kernel to our implementation of content-based page sharing within the Potemkin patch set for Xen. As for our own implementation, we found that as the part of the code that searches for identical pages is only called when a processor is idle, then this part should not influence overall performance. Contrary to this was the task of breaking the sharing, which is time critical. This is dependent on manipulating the shadow page tables, so might not scale well. We did however find that at least on the workloads we examined, it did not have a noticeable impact.

As one of our designs uses shadow page tables, we examined how large the overhead of using shadow page tables was. Generally we found that on most workloads the penalty is minor, but workloads with frequent forking and large processes might pay a larger penalty.

Finally we examined our implementation through micro benchmarks and found that the most expensive functions used were so expensive due to the cost of updating the shadow page tables.

Chapter 9

Conclusion and Future Work

The recently reemerged popularity of virtualization has brought attention to server consolidation as a means of reducing the costs of running a business grade server room, while gaining better fault tolerance through isolation. Server consolidation by using virtual machines as a software abstraction that arbitrates the access to the actual hardware however often entails an increased amount of redundancy within a system. This redundancy consists of multiple instances of the same kernels as well as applications. By using virtualization we get the chance to eliminate the redundancy in a manner that was not possible without consolidation. This thesis approached reducing the redundancy in a system by searching for identical pages in memory to share and found that it is not only feasible, but also possible to implement without much loss of performance.

In particular we started by examining related work done on sharing memory. During this search we encountered content-based page sharing in VMware and mergemem, flash cloning in Potemkin and an emerging shared page cache in XenFS. While Potemkin's flash cloning approach is able to keep the initial memory footprint to a minimum, as soon as it is subjected to a general non-specialized workload, the memory footprint will grow and the approach will not be able to identify additional shares. The content-based page sharing approach on the other hand is able to do this at all times, but the nondeterminism involved in booting an operating system keeps this approach from reaching as little a memory footprint as flash cloning. Therefore a combined approach should form the optimal solution.

For our implementation we chose the content-based page sharing approach, but chose to make several changes to the design applied by VMware. We suggested to use a paravirtualized approach and an approach that was transparent to the guest operating systems. The latter makes use of notoriously expensive shadow page tables to provide an address space abstraction over the pages used by a given virtual machine. The first avoids this expense at the cost of having to modify the guest operating system running inside the virtual machine by use of reverse mapping and ballooning.

The actual implementation is a set of modifications to the Xen virtual machine monitor aided by the Copy-on-Write mechanisms implemented within the Potemkin framework. By using Potemkin as the basis for our implementation and by applying certain design decisions used in Potemkin, we automatically get compatibility with

Potemkin's flash cloning. This is however not something we have tested thoroughly. As for the status of the designs, the transparent design is fully implemented, while the paravirtualized design is only partly implemented.

With a working implementation we were able to examine several interesting questions about the sharing of memory. First and foremost we wanted to determine whether it was even feasible to share memory. We found that the amount of shareable pages is highly dependent on the specific workload, but generally there is something to share on almost any workload. This is mainly based on an assumption where we run the same distributions as well as kernels. Our argument is that this is most likely also the case on real consolidated servers as companies tend to keep to the same distributions and operating systems to reduce the cost of having to be familiar with a multitude of different distributions. We do though note that without this assumption, the amount of pages to share could very well be zero. Furthermore during this evaluation we also had the opportunity to verify the memory sharing results presented by VMware and Potemkin. We found that both were credible and some of our experiments actually came very close to their actual results. Finally we successfully used the pages we reclaimed to do overcommitment in Xen.

As for the contents of the shareable pages, we found that most likely we always share the pages that contain the Xen modified kernel binary. Any other shares are dependent on the individual applications run in the different virtual machines.

We found that page sharing is very feasible in terms of performance, given that our implementation only uses processor cycles, that would otherwise be wasted, in the process of finding identical pages. The only parts that are time critical is the handling of page faults to shared pages. These are however bounded by the implementation of shadow page tables in Xen. The expenses of the shadow page tables are mainly the overhead of synchronizing the tables on workloads with many and/or large processes. To avoid these overheads, we avoid sharing pages that we deem are likely to be broken down momentarily without doing any good.

As for the general applicability of our implementation in Xen, the answer is two-fold. The advantages gained by using content-based page sharing cannot alone justify the use of shadow page tables, as these degrade performance too much. Our hope is however that a completed implementation of the paravirtualized design can justify content-based page sharing in Xen, by omitting the overheads. Hardware supported virtualization technology on the other hand is dependent on shadow page tables, so using this technology there should only be advantages to gain by using content-based page sharing as the expenses have already been paid. The benefits of sharing pages are freeing memory and perhaps even a slight increase in performance due to locality. Furthermore AMD's Pacifica chip supports shadow page tables at the hardware level, so there may not even be any overheads on this architecture.

Finally to conclude the thesis, we return to the goals presented in the first part of the thesis. These were put shortly to examine existing techniques to do sharing between virtual machine, implement one of these in an efficient solution and compare this with existing solutions. We conclude that we succeeded in accomplishing these goals.

9.1 Future Work

Finally we explain interesting research and implementation issues that could be investigated further.

An in depth analysis of what the shared pages actually contain, could prove interesting. This has several aspects, first analyzing the uses of shared pages from a statistical point of view, could very well be interesting to gain a better knowledge of exactly what the pages are used for. In particular we would like to examine how large a percentage of the shared pages are due to the page cache. This could be used to evaluate the feasibility of an interdomain shared cache.

Another interesting subject could be to investigate the scalability of shadow page tables in depth. As we showed the shadow pages tables are vulnerable to forking intensive workloads, but we have no indication that these are typical workloads. Based on the findings, classifications of workloads could be proposed, which determine what workloads are suitable to be run on shadow page tables and which are not.

The implementation also leaves a number of issues. As explained one concern is preparing the implementation for hardware virtualized chip support. Another is completing the implementation of the paravirtualized design. Furthermore the implementation could benefit from using a randomized permutation version of the page hashing algorithms. As of this writing they always start from the same point, so to ensure that the virtual machines are scanned equally, randomizing the order of scanning should make it more fair. Finally the implementation should be prepared for inclusion in the main Xen repository and adapted to the newest changes.

Another topic could be investigating dynamically distribution and reallocation of free memory dictated by policies to ensure the maximum throughput of the system as a whole. One approach could be to start the virtual machines with pages mapped Copy-on-Write to a single zero page. This way we could ensure that the virtual machines only occupy the pages that they actually need. This of course has the overheads of breaking the Copy-on-Write sharing, so it may only be feasible on idle workloads.

Finally making Xen fully capable of handling overcommitment, perhaps by using the privileged domain to swap pages to disk as a last resource. Another approach could be to suspend entire virtual machines to disk to free pages for the system.

Bibliography

[1] *Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference*. 1997. Intel Document Order Number: 243191.

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

[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] Kevin Atkinson. Slow memcmp for aligned strings on pentium 3. http://gcc.gnu.org/ml/gcc/2003-04/msg00166.html.

[5] 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.

[6] Hollis Blanchard. Xen wiki: Xen terminology. http://wiki.xensource.com/xenwiki/XenTerminology?highlight=%28terminology%29.

[7] Daniel G. Bobrow, Jerry D. Burchfiel, Daniel L. Murphy, and Raymond S. Tomlinson. Tenex, a paged time sharing system for the pdp - 10. *Commun. ACM*, 15(3):135–143, 1972.

[8] James Bottomley. Improving kernel performance by unmapping the page cache. In *Proceedings of the 2004 Ottawa Linux Symposium*, pages 103–111, July 2004.

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

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

[11] Bryan Clark, Todd Deshane, Eli Dow, Stephen Evanchik, Matthew Finlayson, Jason Herne, and Jeanna Neefe Matthews. Xen and the art of repeated research. In *USENIX Annual Technical Conference, FREENIX Track*, pages 135–144, 2004.

[12] 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.

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

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

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

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

[17] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms second edition*. The MIT Press, 2003. ISBN 0-262-03293-7.

[18] Standard Performance Evaluation Corporation. Specweb99 benchmark. http://www.spec.org/web99.

[19] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the xen virtual machine monitor. Technical report, 2004. http://www.cl.cam.ac.uk/netos/papers/2004-oasis-ngio.pdf.

[20] Keir A. Fraser. Xen developers archives: Odd mapping behavior with map_pages_to_xen. http://lists.xensource.com/archives/html/xen-devel/2006-03/msg00673.html.

[21] Keir A. Fraser. Xen developers archives: Re: Overcommitting memory (was: Disable auto-balloon on ia64). http://lists.xensource.com/archives/html/xen-devel/2006-05/msg01104.html.

[22] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th Symposium on Operating System Principles(SOSP 2003)*, October 2003.

[23] Robert Philip Goldberg. *Architectural principles for virtual computer systems*. PhD thesis, Division of Engineering and Applied Physics, Harvard University Cambridge Massachusetts, Febuary 1972.

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

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

[26] 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.

[27] Steven Hand, Andrew Warfield, Keir Fraser, Evangelos Kotsovinos, and Dan Magenheimer. Are Virtual Machine Monitors Microkernels Done Right? In *Proceedings of the 10th USENIX Workshop on Hot Topics in Operating Systems (HotOS-X), Santa Fe, NM*, June 2005.

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

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

[30] Paul Hsieh. Hash functions. http://www.azillionmonkeys.com/qed/hash.html.

[31] Bob Hyatt. Crafty. http://www.limunltd.com/crafty/.

[32] Bob Jenkins. A hash function for hash table lookup. http://burtleburtle.net/bob/hash/doobs.html.

[33] Jacob Faber Kloster, Jesper Kristensen, and Arne Mejlholm. Efficient memory sharing in the xen virtual machine monitor. Technical report, Department of Computer Science, Aalborg University, January 2006. http://www.cs.aau.dk/library/cgi-bin/detail.cgi?id=1136884892.

[34] Donald Ervin Knuth. *Sorting and Searching*, volume 3. Addison Wesley Longman, 1998. ISBN 0-201-89685-0.

[35] Mika Kuoppala. Tiobench - threaded i/o bench for linux. http://directory.fsf.org/all/tiobench.html.

[36] 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.

[37] Jochen Liedtke. Toward real microkernels. *Commun. ACM*, 39(9):70–77, 1996.

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

[39] Stuart E. Madnick and John J. Donovan. Application and analysis of the virtual machine approach to information system security and isolation. In *Proceedings of the workshop on virtual computer systems*, pages 210–224, 1973.

[40] Ulrich Neumerkel. Mergemem project. http://www.complang.tuwien.ac.at/ulrich/mergemem/.

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

[42] Ian Pratt, Keir Fraser, Steven Hand, Christian Limpach, Andrew Warfield, Dan Magenheimer, Jun Nakajima, and Asit Mallick. Xen 3.0 and the art of virtualization. In *Proceedings of the 2005 Ottawa Linux Symposium*, pages 65–77, July 2005.

[43] Philipp Richter and Philipp Reisner. Mergemem. http://mergemem.ist.org/.

[44] 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.

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

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

[47] Peter Rundberg. Freebench. http://www.freebench.org/.

[48] Selvamuthukumar Senthilvelan and Murugappan Senthilvelan. Study of content-based sharing on the xen virtual machine monitor. http://www.cs.wisc.edu/~remzi/Classes/736/Spring2005/Projects/Muru-Selva/cs736-report.pdf.

[49] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 1–14. USENIX Association, 2001.

[50] Andrew S. Tanenbaum. *Modern Operating Systems (2nd ed.)*. Prentice Hall PTR, 2001. ISBN 0130313580.

[51] Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos. Can we make operating systems reliable and secure? *Computer*, 39(5):44–51, 2006.

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

[53] The Xen team. Xen interface manual. http://www.cl.cam.ac.uk/Research/SRG/netos/xen/readmes/interface/interface.html.

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

[55] 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.

[56] unixshell#. Xen. http://www.unixshell.com/xen.html.

[57] 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.

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

[59] Amos Waterland. Stress project page. http://weather.ou.edu/~apw/projects/stress/.

[60] 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.

[61] Mark Williamson. 1st year progress report. http://www.cambridge.intel-research.net/~mwilli2/proposal_final.pdf.

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

Summary

This thesis examines the feasibility of sharing pages in a virtual machine system. The first part of the thesis is concerned with introducing terminology necessary for understanding virtualization as well as arguing that sharing memory between virtual machines is worthwhile investigating. We found that this was the case, so we decided that building an implementation to have first hand experiences with memory sharing, was the right approach. As a matter of inspiration we examine related work and find that there are two noteworthy approaches to finding identical pages in memory: Actively searching for identical pages in the system or by use of prior knowledge about their existence. We examined two examples of use of this: The first example, called content-based page sharing, finds pages through a technique called compare-by-hash. The second example called flash cloning uses prior knowledge and takes an entirely different approach. It starts a virtual machine, suspends it and then clones more virtual machines from the first virtual machine. In both cases the pages are shared using Copy-on-Write. This means that when the pages are attempted updated, a Copy-on-Write page fault will be triggered. This then initiates a sequence of actions, that creates a private copy, which can then be modified.

We choose to implement content-based page sharing for the Xen virtual machine monitor. To accomplish this we propose two different designs: One that is transparent to the guest operating system and a paravirtualized version. The first uses notoriously expensive shadow page tables to achieve a level of indirection between virtual and machine addresses. The latter design avoids this overhead by a series of modifications to the guest operating system. The latter is however clearly the most expensive design to implement and therefore we focus on the first, as Xen already has a working shadow page table implementation. Our actual implementation is based on the Potemkin framework, which introduced flash cloning as described above, as this has a working Copy-on-Write implementation.

Having produced a working implementation, we are able to explore memory sharing in a virtual machine system. We do this by a series of synthetic as well as best and worst case experiments. We find that the best case experiments are able to share up to 80% on idle machines. This is due to zero pages. Another best case experiment runs a number of virtual machines that are doing kernel compiles and on this workload roughly 50% of all the memory allocated to the virtual machines can be reclaimed. The worst case test examines virtual machines running completely different applications and we find that the sharing percentage drops to 6% on this workload. A further investigation shows that the 6% are mostly due to the fact that the virtual machines are running on the same Xen modified kernel image.

As for the synthetic workload experiments, we examine virtual machines doing

a wide range of tasks, such as web servers, databases, compilation of source applications, etc. On these workloads the percentage of pages that can be reclaimed typically range between 4-12%. The reader should though note that in these experiments the memory allocations of the virtual machines have been increased, so the kernel images take up a lower percentage of the total memory. We conclude the conditions for sharing pages are best on workloads running similar applications, but no matter what there seem to be something to share on almost every workload. Furthermore we note that we did not examine the impact of running different operating systems, but based on other experiments we estimate that the number of shares between different operating systems probably will be minimal. Finally we examine the feasibility of sharing zero pages. We find that zero pages are only worthwhile sharing on idle workloads, as breaking the Copy-on-Write sharing incurs an overhead. This is intensified by the fact that zero pages are usually free pages within the operating system. When pages are needed in the operating system, the zero pages are used. Thus we have set up shared pages only to tear them down. Consequently we choose not to share zero pages, but do note that it is optional in the implementation. Finally we used the memory we freed due to reclaimed pages to overcommit the system. In an experiment we successfully overcommitted the system by 16 times its capacity.

In order to determine whether it is feasible to share the pages we examine the overhead of our implementation. Through a series of benchmarks we found that the overhead at the overall level is minimal. As the performance of the design implemented however is bounded by the use of shadow page tables, we examine the overhead of using these further. We found that the shadow page tables incur significant overheads on workloads with large processes and forking intensive operations.

Finally we concluded that sharing pages between virtual machines is feasible if the virtual machines are doing similar tasks. The use of shadow page tables in the implementation incurs that the application of content-based pages sharing is too inefficient on contemporary personal computer hardware. However as hardware virtualization supported chipsets depend on the use of shadow page tables, the feasibility of doing content-based page sharing can be reevaluated. Using these chips, the expenses of using shadow page tables are mandatory and therefore sharing might as well be done. Furthermore as a side effect of having implemented content-based page sharing we were able to verify the results of VMware and Potemkin during the experiments.

Appendices

Appendix A

Glossary

- **ABI:** Application Binary Interface. The low level interface between the operating system and any application program.
- **CoW:** Copy-on-Write. A technique commonly used for sharing memory. When a page shared in this manner is attempted modified, then it triggers a page fault. A direct consequence of this is to create a private copy and carry out the modifications to this copy.
- **CS:** CoW Sharing. A part of our design. This component handles creating shares and tearing them down.
- **GDT:** Global Descriptor Table. Equivalent of a page table when using segmentation instead of paging.
- **GMFN:** Guest specific Machine Frame Number. See the latter part of Section 4.2 on page 28 for the details.
- **GPFN:** Guest specific Physical Frame Number. See the latter part of Section 4.2 on page 28 for the details.
- **HI:** Hash Indexing. A part of our design. This is responsible for inserting hash values into the content index and reporting whether there are conflicts.
- **LDT:** Local Descriptor Table. Equivalent of a page table when using segmentation instead of paging.
- `M2P`**:** Machine To Physical. See the latter part of Section 4.2 on page 28 for the details.
- **MFN:** Machine Frame Number. See the latter part of Section 4.2 on page 28 for the details.
- **MMU:** Memory Management Unit. Part of the processor that is responsible for the translation of virtual addresses to machine addresses.
- `P2M`**:** Physical To Machine. See the latter part of Section 4.2 on page 28 for the details.
- **PAE:** Physical Address Extension. A modification to paging that enables the processor to make use of up to 64 GB of memory instead of 4 GB on the IA-32 architecture.

- **PC:** Page Comparison. A part of our design. This is responsible for comparing two pages to ensure that they are bitwise identical.
- **PFN:** Physical Frame Number. See the latter part of Section 4.2 on page 28 for the details.
- **PGD:** Page Global Directory. The first level part of a Page Table (PT).
- **PH:** Page Hashing. A part of our design. Responsible for reading the contents of a page and producing a hash value.
- **PSE:** Page Size Extension. A modification to normal paging, which enables the processor to use pages of size 2/4 MB. Such a page requires only one page table entry and thus only one entry in the Translation Look-ahead Buffer (TLB), so a larger percentage of the memory can be kept in the TLB.
- **PT:** Page Table. Used as a general term for a Page Global Directory and a number of Page Table Entry (PTE) tables.
- **PTE:** Page Table Entry. The second level part of a Page Table (PT).
- **RM:** Reference Manager. A part of our implementation. This is responsible for keeping count of reference count to shared pages.
- **SFH:** SuperFastHash. The name of the hash function we make use of.
- **SPT:** Shadow Page Table. A hardware specific representation of an operating systems page table subjected to another level of paging. See Section 4.4 on page 31 for a more detailed explanation.
- **TLB:** Translation Look-aside Buffer. A cache used for the translation of virtual addresses.
- **VM (domain):** Virtual Machine. A software abstraction of the actual hardware.
- **VMA:** Virtual Memory Area. The Linux kernels descriptor of a memory area.
- **VMM:** Virtual Machine Monitor (hypervisor). The part of a virtual machine system that administers the actual hardware, thus ensuring the safety of the system.
- **VMS:** Virtual Machine System. A common term for a system comprised of a Virtual Machine Monitor (VMM) and a number of Virtual Machines (VMs).
- **ln:** Level n Page Table (PT). Xen terminology for the constituent of PTs. On IA-32 the l1 table refers to the PTE table, while the l2 refers to the PGD table.
- **rmap:** revers mapping. A means of finding all the virtual addresses that are pointing to a given page. See Section 4.1 on page 26 for the details.

Appendix B

AIM Benchmarks

The tables in this appendix shows the results of the AIM Independent Resource Benchmark experiments conducted on the two test machines. Table B.1, B.2 B.3 and B.4 is from the 700 MHz AMD Atlhon with 64 MB of memory. The results from the Sun Fire X4100 server can be found in Table B.5, B.6, B.7 and B.8. The highlighted rows in the tables are used in Section 8.1 on page 87.

No.	Operation	Debian	Domain 0	Unit/second
1	creat-clo	94750.00	83486.09	File Creations and Closes
2	page_test	74888.37	56005.67	System Allocations & Pages
3	brk_test	882152.97	950866.67	System Memory Allocations
4	jmp_test	5880866.67	6207182.14	Non-local gotos
5	signal_test	95900.68	167788.70	Signal Traps
6	exec_test	63.44	53.46	Program Loads
7	fork_test	2577.90	844.30	Task Creations
8	link_test	33524.40	43115.26	Link/Unlink Pairs
9	disk_rr	46212.16	39351.85	Random Disk Reads (K)
10	disk_rw	40649.80	33946.16	Random Disk Writes (K)
11	disk_rd	192927.54	187841.39	Sequential Disk Reads (K)
12	disk_wrt	62220.74	57627.71	Sequential Disk Writes (K)
13	disk_cp	44847.96	37363.55	Disk Copies (K)
14	sync_disk_rw	475.39	352.19	Sync Random Disk Writes (K)
15	sync_disk_wrt	161.67	137.67	Sync Sequential Disk Writes (K)
16	sync_disk_cp	160.33	132.52	Sync Disk Copies (K)
17	disk_src	19872.50	24250.00	Directory Searches
18	fun_cal	45516800.00	46481853.02	Function Calls (no arguments)
19	fun_cal1	48008533.33	48691618.06	Function Calls (1 argument)
20	fun_cal2	74577437.09	75609798.37	Function Calls (2 arguments)
21	fun_cal15	36172800.00	36655514.83	Function Calls (15 arguments)
22	sieve	4.67	5.01	Integer Sieves
23	num_rtns_1	63411.67	64239.29	Numeric Functions
24	new_raph	186456.67	188215.30	Zeros Found
25	trig_rtns	305000.00	309511.91	Trigonometric Functions
26	matrix_rtns	636825.00	646045.66	Point Transformations
27	array_rtns	159.35	160.56	Linear Systems Solved
28	string_rtns	1341.22	1363.18	String Manipulations
29	mem_rtns_1	883352.77	933844.36	Dynamic Memory Operations
30	mem_rtns_2	262741.67	211714.43	Block Memory Operations
31	sort_rtns_1	332.00	325.95	Sort Operations
32	misc_rtns_1	5840.17	3133.81	Auxiliary Loops
33	dir_rtns_1	987333.33	1934177.64	Directory Operations
34	shell_rtns_1	23.35	18.97	Shell Scripts
35	shell_rtns_2	23.35	19.07	Shell Scripts
36	shell_rtns_3	23.39	19.12	Shell Scripts
37	series_1	1973886.67	1994602.57	Series Evaluations
38	shared_memory	198761.67	214166.94	Shared Memory Operations
39	fifo_test	247381.67	244703.33	FIFO Messages
40	stream_pipe	232378.33	250783.20	Stream Pipe Messages
41	dgram_pipe	225051.67	241321.45	DataGram Pipe Messages
42	pipe_cpy	297168.33	291254.79	Pipe Messages
43	ram_copy	1106419428	1117391699	Memory to Memory Copy

Table B.1: Results from the AIM benchmark running in Debian and domain-0.

No.	Operation	XenU	XenU SPT	Unit/second
1	creat-clo	99866.69	94700.88	File Creations and Closes
2	page_test	55949.01	51824.09	System Allocations & Pages
3	brk_test	599900.03	892485.84	System Memory Allocations
4	jmp_test	4566127.96	4624479.25	Non-local gotos
5	signal_test	169088.49	164072.65	Signal Traps
6	**exec_test**	**149.65**	**117.96**	**Program Loads**
7	**fork_test**	**835.27**	**592.05**	**Task Creations**
8	link_test	36498.22	34412.21	Link/Unlink Pairs
9	disk_rr	39148.43	35639.63	Random Disk Reads (K)
10	disk_rw	33116.96	31180.03	Random Disk Writes (K)
11	disk_rd	211506.08	210567.57	Sequential Disk Reads (K)
12	disk_wrt	54208.76	54576.95	Sequential Disk Writes (K)
13	disk_cp	37411.45	40118.45	Disk Copies (K)
14	sync_disk_rw	315.17	310.96	Sync Random Disk Writes (K)
15	sync_disk_wrt	127.64	75.49	Sync Sequential Disk Writes (K)
16	sync_disk_cp	130.06	127.79	Sync Disk Copies (K)
17	disk_src	21852.61	22502.50	Directory Searches
18	fun_cal	37190601.57	36653091.15	Function Calls (no arguments)
19	fun_cal1	44698683.55	45150874.85	Function Calls (1 argument)
20	fun_cal2	53187935.34	52437127.15	Function Calls (2 arguments)
21	fun_cal15	27809396.87	27686052.32	Function Calls (15 arguments)
22	sieve	5.16	5.24	Integer Sieves
23	num_rtns_1	64918.36	62942.84	Numeric Functions
24	new_raph	180646.56	182596.23	Zeros Found
25	trig_rtns	300516.41	289618.40	Trigonometric Functions
26	matrix_rtns	415744.04	413921.01	Point Transformations
27	array_rtns	275.57	276.06	Linear Systems Solved
28	string_rtns	936.62	932.87	String Manipulations
29	mem_rtns_1	922846.19	937500.00	Dynamic Memory Operations
30	mem_rtns_2	327040.00	325397.43	Block Memory Operations
31	sort_rtns_1	328.17	334.72	Sort Operations
32	misc_rtns_1	2833.03	2338.11	Auxiliary Loops
33	dir_rtns_1	1889036.99	1904182.64	Directory Operations
34	**shell_rtns_1**	**35.31**	**27.13**	**Shell Scripts**
35	**shell_rtns_2**	**35.64**	**26.60**	**Shell Scripts**
36	**shell_rtns_3**	**36.26**	**27.51**	**Shell Scripts**
37	series_1	1641574.74	1634409.27	Series Evaluations
38	shared_memory	204305.95	190511.50	Shared Memory Operations
39	fifo_test	215292.45	219056.82	FIFO Messages
40	stream_pipe	228511.91	233617.73	Stream Pipe Messages
41	dgram_pipe	219883.37	223266.12	DataGram Pipe Messages
42	pipe_cpy	252954.51	270388.27	Pipe Messages
43	ram_copy	970223567	986497198	Memory to Memory Copy

Table B.2: Results from the AIM benchmark running in XenU without SPTs and with SPTs.

No.	Operation	Page scanning	CBPS	Unit/second
1	creat-clo	92952.35	94167.64	File Creations and Closes
2	page_test	51710.81	51444.76	System Allocations & Pages
3	brk_test	851982.67	886402.27	System Memory Allocations
4	jmp_test	4549241.79	4597300.45	Non-local gotos
5	signal_test	168305.28	166438.93	Signal Traps
6	exec_test	118.00	122.61	Program Loads
7	fork_test	598.50	610.75	Task Creations
8	link_test	34862.59	31674.27	Link/Unlink Pairs
9	disk_rr	37673.38	34889.70	Random Disk Reads (K)
10	disk_rw	31446.07	31984.01	Random Disk Writes (K)
11	disk_rd	207461.51	211591.40	Sequential Disk Reads (K)
12	disk_wrt	54500.75	54047.29	Sequential Disk Writes (K)
13	disk_cp	36078.84	40008.00	Disk Copies (K)
14	sync_disk_rw	311.29	300.60	Sync Random Disk Writes (K)
15	sync_disk_wrt	105.05	77.79	Sync Sequential Disk Writes (K)
16	sync_disk_cp	120.23	113.07	Sync Disk Copies (K)
17	disk_src	21992.58	22347.53	Directory Searches
18	fun_cal	36339886.70	36753066.67	Function Calls (no arguments)
19	fun_cal1	44476853.86	44963172.80	Function Calls (1 argument)
20	fun_cal2	53896083.99	53077020.50	Function Calls (2 arguments)
21	fun_cal15	28343009.50	28756281.24	Function Calls (15 arguments)
22	sieve	4.86	5.32	Integer Sieves
23	num_rtns_1	61991.33	62697.88	Numeric Functions
24	new_raph	178606.90	181603.07	Zeros Found
25	trig_rtns	300116.65	288903.70	Trigonometric Functions
26	matrix_rtns	407853.69	411694.72	Point Transformations
27	array_rtns	272.53	277.10	Linear Systems Solved
28	string_rtns	896.52	929.38	String Manipulations
29	mem_rtns_1	798866.86	851858.02	Dynamic Memory Operations
30	mem_rtns_2	319945.02	323699.38	Block Memory Operations
31	sort_rtns_1	330.33	332.06	Sort Operations
32	misc_rtns_1	2166.50	2150.31	Auxiliary Loops
33	dir_rtns_1	1905349.11	1913847.69	Directory Operations
34	shell_rtns_1	26.40	27.16	Shell Scripts
35	shell_rtns_2	26.37	26.73	Shell Scripts
36	shell_rtns_3	26.38	27.12	Shell Scripts
37	series_1	1585117.48	1591479.75	Series Evaluations
38	shared_memory	179928.33	187578.33	Shared Memory Operations
39	fifo_test	214784.20	208701.88	FIFO Messages
40	stream_pipe	227820.36	227910.35	Stream Pipe Messages
41	dgram_pipe	217887.02	218360.27	DataGram Pipe Messages
42	pipe_cpy	276207.30	272766.21	Pipe Messages
43	ram_copy	966326934.5	949645127.8	Memory to Memory Copy

Table B.3: Results from the AIM benchmark running in Xen with page scanning and content-based page sharing.

No.	Operation	Potemkin	Unit/second
1	creat-clo	99216.67	File Creations and Closes
2	page_test	52540.82	System Allocations & Pages
3	brk_test	856797.73	System Memory Allocations
4	jmp_test	4041441.89	Non-local gotos
5	signal_test	167972.00	Signal Traps
6	exec_test	118.04	Program Loads
7	fork_test	556.94	Task Creations
8	link_test	37044.13	Link/Unlink Pairs
9	disk_rr	38289.14	Random Disk Reads (K)
10	disk_rw	35360.29	Random Disk Writes (K)
11	disk_rd	211812.06	Sequential Disk Reads (K)
12	disk_wrt	54482.60	Sequential Disk Writes (K)
13	disk_cp	41110.11	Disk Copies (K)
14	sync_disk_rw	314.50	Sync Random Disk Writes (K)
15	sync_disk_wrt	122.70	Sync Sequential Disk Writes (K)
16	sync_disk_cp	117.11	Sync Disk Copies (K)
17	disk_src	22045.08	Directory Searches
18	fun_cal	36490984.84	Function Calls (no arguments)
19	fun_cal1	42761939.68	Function Calls (1 argument)
20	fun_cal2	54826062.32	Function Calls (2 arguments)
21	fun_cal15	28833055.65	Function Calls (15 arguments)
22	sieve	5.14	Integer Sieves
23	num_rtns_1	63057.31	Numeric Functions
24	new_raph	172481.25	Zeros Found
25	trig_rtns	300899.70	Trigonometric Functions
26	matrix_rtns	413737.71	Point Transformations
27	array_rtns	274.48	Linear Systems Solved
28	string_rtns	933.75	String Manipulations
29	mem_rtns_1	945342.44	Dynamic Memory Operations
30	mem_rtns_2	325144.14	Block Memory Operations
31	sort_rtns_1	334.50	Sort Operations
32	misc_rtns_1	2237.13	Auxiliary Loops
33	dir_rtns_1	1935854.72	Directory Operations
34	shell_rtns_1	26.26	Shell Scripts
35	shell_rtns_2	27.54	Shell Scripts
36	shell_rtns_3	27.44	Shell Scripts
37	series_1	1623942.68	Series Evaluations
38	shared_memory	189480.09	Shared Memory Operations
39	fifo_test	221936.34	FIFO Messages
40	stream_pipe	232566.24	Stream Pipe Messages
41	dgram_pipe	222981.17	DataGram Pipe Messages
42	pipe_cpy	283306.12	Pipe Messages
43	ram_copy	980988678	Memory to Memory Copy

Table B.4: Results from the AIM benchmark running Potemkin.

No.	Operation	Ubuntu	Domain 0	Unit/second
1	creat-clo	200599.90	288672.33	File Creations and Closes
2	page_test	277280.45	287328.33	System Allocations & Pages
3	brk_test	2574933.33	2599716.71	System Memory Allocations
4	jmp_test	21937143.81	21376270.62	Non-local gotos
5	signal_test	585000.00	574587.57	Signal Traps
6	exec_test	151.20	146.48	Program Loads
7	fork_test	5564.07	3399.43	Task Creations
8	link_test	77588.37	119799.73	Link/Unlink Pairs
9	disk_rr	101615.06	174221.63	Random Disk Reads (K)
10	disk_rw	86513.58	152585.04	Random Disk Writes (K)
11	disk_rd	442049.98	758060.32	Sequential Disk Reads (K)
12	disk_wrt	130879.52	248407.86	Sequential Disk Writes (K)
13	disk_cp	97641.57	180730.97	Disk Copies (K)
14	sync_disk_rw	183.43	203.11	Sync Random Disk Writes (K)
15	sync_disk_wrt	51.19	51.59	Sync Sequential Disk Writes (K)
16	sync_disk_cp	51.25	51.93	Sync Disk Copies (K)
17	disk_src	53539.83	80359.11	Directory Searches
18	fun_cal	242048000.00	236726412.26	Function Calls (no arguments)
19	fun_cal1	290227200.00	283839626.73	Function Calls (1 argument)
20	fun_cal2	253918213.63	268448058.66	Function Calls (2 arguments)
21	fun_cal15	121147733.33	112297017.16	Function Calls (15 arguments)
22	sieve	52.05	56.24	Integer Sieves
23	num_rtns_1	225695.00	220848.19	Numeric Functions
24	new_raph	N/A	N/A	Zeros Found
25	trig_rtns	1102149.64	1074487.59	Trigonometric Functions
26	matrix_rtns	3486272.29	3403424.43	Point Transformations
27	array_rtns	1961.00	1916.35	Linear Systems Solved
28	string_rtns	4337.61	4174.30	String Manipulations
29	mem_rtns_1	3694768.41	3595400.77	Dynamic Memory Operations
30	mem_rtns_2	N/A	N/A	Block Memory Operations
31	sort_rtns_1	1172.67	1144.81	Sort Operations
32	misc_rtns_1	13946.18	11137.14	Auxiliary Loops
33	dir_rtns_1	5731166.67	6343942.68	Directory Operations
34	shell_rtns_1	44.95	43.01	Shell Scripts
35	shell_rtns_2	44.90	42.97	Shell Scripts
36	shell_rtns_3	44.94	43.03	Shell Scripts
37	series_1	11117683.33	10794248.33	Series Evaluations
38	shared_memory	757230.00	737902.02	Shared Memory Operations
39	fifo_test	412884.52	724400.00	FIFO Messages
40	stream_pipe	557710.00	729576.74	Stream Pipe Messages
41	dgram_pipe	517641.67	690988.17	DataGram Pipe Messages
42	pipe_cpy	495640.73	873411.10	Pipe Messages
43	ram_copy	4233480327	4163076966	Memory to Memory Copy

Table B.5: Results from the AIM benchmark running in Ubuntu and domain-0.

No.	Operation	XenU	XenU SPT	Unit/second
1	creat-clo	329861.69	318363.61	File Creations and Closes
2	page_test	283966.01	273201.13	System Allocations & Pages
3	brk_test	2695467.42	3064589.24	System Memory Allocations
4	jmp_test	16152157.97	16157997.33	Non-local gotos
5	signal_test	623746.04	621616.67	Signal Traps
6	**exec_test**	**828.50**	**565.58**	**Program Loads**
7	**fork_test**	**3554.89**	**2013.00**	**Task Creations**
8	link_test	144096.88	143688.50	Link/Unlink Pairs
9	disk_rr	210668.00	208946.51	Random Disk Reads (K)
10	disk_rw	182467.18	180304.51	Random Disk Writes (K)
11	disk_rd	982961.51	972296.62	Sequential Disk Reads (K)
12	disk_wrt	292729.88	293045.91	Sequential Disk Writes (K)
13	disk_cp	219611.40	219282.36	Disk Copies (K)
14	sync_disk_rw	197.53	199.94	Sync Random Disk Writes (K)
15	sync_disk_wrt	56.69	56.79	Sync Sequential Disk Writes (K)
16	sync_disk_cp	56.23	57.44	Sync Disk Copies (K)
17	disk_src	99193.75	98359.86	Directory Searches
18	fun_cal	136920113.31	136996900.52	Function Calls (no arguments)
19	fun_cal1	246196833.86	243927345.44	Function Calls (1 argument)
20	fun_cal2	296782536.24	273550141.64	Function Calls (2 arguments)
21	fun_cal15	108534444.26	108577103.82	Function Calls (15 arguments)
22	sieve	61.99	59.46	Integer Sieves
23	num_rtns_1	233974.34	245853.33	Numeric Functions
24	new_raph	N/A	N/A	Zeros Found
25	trig_rtns	1062822.86	1064333.33	Trigonometric Functions
26	matrix_rtns	3480578.24	3159135.14	Point Transformations
27	array_rtns	985.50	984.67	Linear Systems Solved
28	string_rtns	3245.04	3246.13	String Manipulations
29	mem_rtns_1	3818363.61	3812864.52	Dynamic Memory Operations
30	mem_rtns_2	N/A	N/A	Block Memory Operations
31	sort_rtns_1	1018.16	1079.67	Sort Operations
32	misc_rtns_1	15187.64	12283.95	Auxiliary Loops
33	dir_rtns_1	6567572.07	6557907.02	Directory Operations
34	**shell_rtns_1**	**182.44**	**134.13**	**Shell Scripts**
35	**shell_rtns_2**	**182.65**	**135.13**	**Shell Scripts**
36	**shell_rtns_3**	**182.77**	**134.83**	**Shell Scripts**
37	series_1	10351338.11	10414860.86	Series Evaluations
38	shared_memory	750048.33	732023.00	Shared Memory Operations
39	fifo_test	803922.68	831164.81	FIFO Messages
40	stream_pipe	796352.27	786045.66	Stream Pipe Messages
41	dgram_pipe	752160.00	710951.67	DataGram Pipe Messages
42	pipe_cpy	935089.15	948145.31	Pipe Messages
43	ram_copy	4068490901.18	4178402133	Memory to Memory Copy

Table B.6: Results from the AIM benchmark running in XenU without SPTs and with SPTs.

No.	Operation	Page scanning	CBPS	Unit/second
1	creat-clo	328873.71	330994.83	File Creations and Closes
2	page_test	266900.00	261586.40	System Allocations & Pages
3	brk_test	2745325.78	3072237.96	System Memory Allocations
4	jmp_test	16157557.07	16164589.24	Non-local gotos
5	signal_test	611433.33	593067.82	Signal Traps
6	exec_test	598.83	600.48	Program Loads
7	fork_test	2503.33	2083.61	Task Creations
8	link_test	137949.11	143861.90	Link/Unlink Pairs
9	disk_rr	195007.00	205141.33	Random Disk Reads (K)
10	disk_rw	169216.72	175130.96	Random Disk Writes (K)
11	disk_rd	1036285.95	1044903.18	Sequential Disk Reads (K)
12	disk_wrt	263550.74	277287.12	Sequential Disk Writes (K)
13	disk_cp	206096.63	210191.27	Disk Copies (K)
14	sync_disk_rw	201.10	200.25	Sync Random Disk Writes (K)
15	sync_disk_wrt	56.54	52.97	Sync Sequential Disk Writes (K)
16	sync_disk_cp	55.93	52.64	Sync Disk Copies (K)
17	disk_src	95854.02	98082.40	Directory Searches
18	fun_cal	137011200.00	137039560.07	Function Calls (no arguments)
19	fun_cal1	245761706.38	246358940.18	Function Calls (1 argument)
20	fun_cal2	264037060.49	225447225.46	Function Calls (2 arguments)
21	fun_cal15	108534444.26	108611231.46	Function Calls (15 arguments)
22	sieve	64.08	62.80	Integer Sieves
23	num_rtns_1	246053.33	246000.67	Numeric Functions
24	new_raph	N/A	N/A	Zeros Found
25	trig_rtns	1063656.06	1066488.92	Trigonometric Functions
26	matrix_rtns	3484872.52	3493346.11	Point Transformations
27	array_rtns	985.50	985.00	Linear Systems Solved
28	string_rtns	3245.04	3235.05	String Manipulations
29	mem_rtns_1	3720379.94	3815364.11	Dynamic Memory Operations
30	mem_rtns_2	N/A	N/A	Block Memory Operations
31	sort_rtns_1	1014.66	1022.83	Sort Operations
32	misc_rtns_1	12550.91	12231.46	Auxiliary Loops
33	dir_rtns_1	6577403.77	6595900.68	Directory Operations
34	shell_rtns_1	135.60	137.22	Shell Scripts
35	shell_rtns_2	136.01	138.40	Shell Scripts
36	shell_rtns_3	136.35	137.81	Shell Scripts
37	series_1	10177747.04	10461494.75	Series Evaluations
38	shared_memory	714750.00	672392.93	Shared Memory Operations
39	fifo_test	810473.25	827640.39	FIFO Messages
40	stream_pipe	775215.00	774587.57	Stream Pipe Messages
41	dgram_pipe	727472.09	722294.62	DataGram Pipe Messages
42	pipe_cpy	944410.00	980906.52	Pipe Messages
43	ram_copy	4180793628	4188221253.12	Memory to Memory Copy

Table B.7: Results from the AIM benchmark running in Xen with page scanning and content-based page sharing.

No.	Operation	Potemkin	Unit/second
1	creat-clo	343118.96	File Creations and Closes
2	page_test	256190.00	System Allocations & Pages
3	brk_test	3016997.17	System Memory Allocations
4	jmp_test	16166872.19	Non-local gotos
5	signal_test	571016.67	Signal Traps
6	exec_test	575.40	Program Loads
7	fork_test	2041.97	Task Creations
8	link_test	140945.31	Link/Unlink Pairs
9	disk_rr	204748.83	Random Disk Reads (K)
10	disk_rw	174562.91	Random Disk Writes (K)
11	disk_rd	1024938.51	Sequential Disk Reads (K)
12	disk_wrt	277240.92	Sequential Disk Writes (K)
13	disk_cp	207546.82	Disk Copies (K)
14	sync_disk_rw	200.56	Sync Random Disk Writes (K)
15	sync_disk_wrt	56.52	Sync Sequential Disk Writes (K)
16	sync_disk_cp	55.86	Sync Disk Copies (K)
17	disk_src	96091.25	Directory Searches
18	fun_cal	137176070.65	Function Calls (no arguments)
19	fun_cal1	246565278.24	Function Calls (1 argument)
20	fun_cal2	264779336.78	Function Calls (2 arguments)
21	fun_cal15	108646400.00	Function Calls (15 arguments)
22	sieve	58.77	Integer Sieves
23	num_rtns_1	246016.67	Numeric Functions
24	new_raph	N/A	Zeros Found
25	trig_rtns	1035000.00	Trigonometric Functions
26	matrix_rtns	2832324.61	Point Transformations
27	array_rtns	986.50	Linear Systems Solved
28	string_rtns	3247.25	String Manipulations
29	mem_rtns_1	3847217.59	Dynamic Memory Operations
30	mem_rtns_2	N/A	Block Memory Operations
31	sort_rtns_1	1124.83	Sort Operations
32	misc_rtns_1	12405.53	Auxiliary Loops
33	dir_rtns_1	6583000.00	Directory Operations
34	shell_rtns_1	131.24	Shell Scripts
35	shell_rtns_2	132.28	Shell Scripts
36	shell_rtns_3	135.38	Shell Scripts
37	series_1	10195615.73	Series Evaluations
38	shared_memory	681706.38	Shared Memory Operations
39	fifo_test	835113.33	FIFO Messages
40	stream_pipe	793701.67	Stream Pipe Messages
41	dgram_pipe	740003.33	DataGram Pipe Messages
42	pipe_cpy	993485.00	Pipe Messages
43	ram_copy	4161174195	Memory to Memory Copy

Table B.8: Results from the AIM benchmark running Potemkin.

Appendix C

Unabridged Performance Evaluation Results

The tables in the appendix shows the complete results from the performance evaluation.

Linux Build Time (sec)			OSDB IR (tup/sec)		
No.	Sun Fire	700 MHz AMD	No.	Sun Fire	700 MHz AMD
1	1045.785	6225.540	1	2225.59	149.05
2	1048.451	6217.711	2	2200.75	164.96
3	1048.070	6233.248	3	2204.90	197.18

OSDB OLTP (tup/sec)			dbench (MB/sec)		
No.	Sun Fire	700 MHz AMD	No.	Sun Fire	700 MHz AMD
1	142.31	257.56	1	619.110	37.7613
2	153.37	305.44	2	613.467	38.8351
3	153.52	327.84	3	604.343	37.2971

SPECweb99 (KB/sec)		
No.	Sun Fire	700 MHz AMD
1	133.3	384.9
2	133.2	385.4
3	133.1	385.5

Table C.1: Benchmark results for Ubuntu and Debian.

Linux Build Time (sec)			OSDB IR (tup/sec)		
No.	Sun Fire	700 MHz AMD	No.	Sun Fire	700 MHz AMD
1	330.983	1493.652	1	803.14	84.18
2	331.923	1507.197	2	887.09	116.46
3	332.840	1499.235	3	840.55	113.40

OSDB OLTP (tup/sec)			dbench (MB/sec)		
No.	Sun Fire	700 MHz AMD	No.	Sun Fire	700 MHz AMD
1	48.02	129.21	1	169.227	9.66996
2	42.88	164.52	2	180.794	9.57138
3	45.26	169.92	3	181.147	9.61325

SPECweb99 (KB/sec)		
No.	Sun Fire	700 MHz AMD
1	134.0	382.8
2	135.0	383.1
3	134.1	383.5

Table C.2: Benchmark results for Xen without shadow page tables.

Linux Build Time (sec)			OSDB IR (tup/sec)		
No.	Sun Fire	700 MHz AMD	No.	Sun Fire	700 MHz AMD
1	374.880	1546.802	1	906.17	84.14
2	378.612	1507.197	2	814.31	106.13
3	383.407	1488.584	3	840.99	110.96

OSDB OLTP (tup/sec)			dbench (MB/sec)		
No.	Sun Fire	700 MHz AMD	No.	Sun Fire	700 MHz AMD
1	49.62	132.33	1	183.425	9.95482
2	42.01	158.27	2	167.056	9.35365
3	48.15	166.00	3	155.398	9.14352

SPECweb99 (KB/sec)		
No.	Sun Fire	700 MHz AMD
1	133.7	363.8
2	133.9	362.7
3	131.7	362.2

Table C.3: Benchmark results for Xen with shadow page tables.

Linux Build Time (sec)			OSDB IR (tup/sec)		
No.	Sun Fire	700 MHz AMD	No.	Sun Fire	700 MHz AMD
1	349.611	1560.033	1	755.42	76.78
2	343.987	1560.449	2	745.54	89.81
3	348.022	1561.003	3	777.77	91.62

OSDB OLTP (tup/sec)			dbench (MB/sec)		
No.	Sun Fire	700 MHz AMD	No.	Sun Fire	700 MHz AMD
1	41.53	127.51	1	156.707	9.1172
2	44.30	133.15	2	151.537	9.0078
3	42.30	132.96	3	196.764	9.2946

SPECweb99 (KB/sec)		
No.	Sun Fire	700 MHz AMD
1	90.0	361.4
2	81.6	358.5
3	80.6	358.4

Table C.4: Benchmark results for Potemkin.

Linux Build Time (sec)		
No.	Sun Fire	700 MHz AMD
1	346.040	1562.059
2	338.363	1555.163
3	338.603	1554.630

OSDB IR (tup/sec)		
No.	Sun Fire	700 MHz AMD
1	794.57	80.68
2	771.21	88.83
3	782.68	87.42

OSDB OLTP (tup/sec)		
No.	Sun Fire	700 MHz AMD
1	42.36	128.32
2	40.90	137.02
3	39.28	132.06

dbench (MB/sec)		
No.	Sun Fire	700 MHz AMD
1	181.003	9.80688
2	167.460	9.33071
3	185.098	9.37882

SPECweb99 (KB/sec)		
No.	Sun Fire	700 MHz AMD
1	88.2	361.4
2	88.7	359.3
3	81.6	359.3

Table C.5: Benchmark results for our implementation (no sharing, only scanning).

Linux Build Time (sec)		
No.	Sun Fire	700 MHz AMD
1	340.544	1570.807
2	338.827	1582.591
3	339.838	1582.806

OSDB IR (tup/sec)		
No.	Sun Fire	700 MHz AMD
1	799.89	94.02
2	747.44	93.46
3	760.34	98.67

OSDB OLTP (tup/sec)		
No.	Sun Fire	700 MHz AMD
1	40.49	125.11
2	41.35	125.62
3	39.97	134.63

dbench (MB/sec)		
No.	Sun Fire	700 MHz AMD
1	152.171	9.67219
2	166.034	9.75403
3	163.975	9.72496

SPECweb99 (KB/sec)		
No.	Sun Fire	700 MHz AMD
1		359.4
2		361.2
3		357.3

Table C.6: Benchmark results for our implementation (sharing and scanning).