Efficient Memory Sharing in the Xen Virtual Machine Monitor

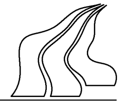
Jacob Faber Kloster
Jesper Kristensen
Arne Mejlholm

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

at

DEPARTMENT OF COMPUTER SCIENCE,
AALBORG UNIVERSITY
JANUARY 2006

Department of Computer Science
Aalborg University



Title:
Efficient Memory Sharing In the Xen Virtual Machine Monitor

Semester:
Dat5
2. September 2005 -
10. January, 2006

Group:
D515a, 2005-2006

Members:
Jacob Faber Kloster
Jesper Kristensen
Arne Mejlholm

Supervisor:
Gerd Behrmann

Copies: 5

Report – pages: 80

Appendix – pages: 6

Total pages: 86

Synopsis:

This report describes the design of an efficient mechanism for sharing memory between virtual machines. The design uses content-based page sharing, which uses a compare-by-hash technique to locate pages eligible for sharing. It uses a fast, low-collision, uniformly distributed hash function called SuperFastHash combined with common hash tables using open addressing. Reducing the number of pages by sharing provides an opportunity for doing overcommitment of memory through a technique called ballooning.

The mechanism will be implemented as a modification to the Xen Virtual Machine Monitor, which is a high performance paravirtualization framework built on top of the Linux kernel.

The background research for the report summarizes important work on virtualization.and memory sharing between virtual machines as well as explaining key concepts of Xen with focus on virtual memory.

Keywords: VM, VMM, hypervisor, Xen, virtualizability, paravirtualization, COW, memory sharing, shadow page table, ballooning, overcommitment, hash functions, SFH.

Preface

This first part of our thesis documents our background research on exploring memory sharing in a virtualization setting as well as the design of an extension to a known virtual machine monitor to enable it to do memory sharing.. The report has been developed as part of our DAT5 semester at Department of Computer Science, Aalborg University.

The second part will address the implementation and evaluation of the ideas presented in this part of the thesis.

We would like to thank Michael Vrable for the source code used in the Potemkin framework.

Jacob Faber Kloster

Jesper Kristensen

Arne Mejlholm

Contents

I Analysis 7

1 Virtualization 11
1.1 Motivation for Virtualization . . . 11
1.2 Virtualizability . . . 13
1.3 Other Approaches to Virtualization . . . 14
1.4 The Legacy . . . 16
1.5 Application of Virtualization in Research . . . 17

2 Xen 19
2.1 Xen Architecture . . . 20
2.2 Memory Management . . . 22
2.2.1 Memory in Xen . . . 22
2.2.2 Linux Page Tables . . . 23
2.2.3 Xen Page Tables . . . 25
2.2.4 Shadow Page Tables . . . 26
2.2.5 Page Faults . . . 28
2.2.6 Ballooning . . . 30

3 Motivation and Goals 31
3.1 Experiments . . . 31
3.1.1 Working Set Changes . . . 32
3.1.2 Sharing Potential . . . 33
3.2 Project Goal . . . 33
3.3 Limitations . . . 34

4 Related Work 35
4.1 Compare-By-Hash . . . 35
4.2 Copy-On-Write . . . 35
4.3 Shared Virtual Storage on VM/370 . . . 36
4.4 Transparent Page Sharing . . . 37
4.5 Content Based Page Sharing . . . 37
4.6 Flash Cloning . . . 38
4.7 Comparison of the Different Approaches . . . 38

5 Conclusion 41

II Design 43

6 Techniques 47
6.1 Hash Functions . . . 47
6.1.1 Fowler / Noll / Vo (FNV) . . . 47
6.1.2 Bob Jenkins (BJ) . . . 48
6.1.3 SuperFastHash (SFH) . . . 48
6.1.4 Secure Hash Functions MD5/SHA1 . . . 48
6.1.5 Evaluation . . . 49
6.2 Data Structures . . . 50
6.2.1 Hash Tables . . . 52
6.2.2 Hash Array Mapped Trie (HAMT) . . . 53
6.3 Comparison of Data Structures . . . 56

7 Architecture 59
7.1 Design Goals . . . 59
7.2 Components . . . 59
7.3 Feasibility of Perfect Sharing . . . 60
7.4 Flushing vs. Continuous Hashing . . . 61
7.5 Overall Designs . . . 62
7.5.1 Transparent Hypervisor Level . . . 63
7.5.2 Hypervisor Level . . . 65
7.5.3 Supervisor Level . . . 65
7.6 Advanced Details . . . 68
7.6.1 Algorithms in the Hypervisor Level Design . . . 68
7.6.2 Algorithms in the Transparent Hypervisor Level Design . . . 69
7.7 Evaluating the Designs . . . 70

8 Further Work 73

III Appendix 79

A Hash Collisions 81

Part I

Analysis

Part Introduction

This part of the report serves as an introduction to the problem area. Briefly stated: Our focus in the project is to address memory sharing between virtual machines in a virtualization environment. However in order to explain the problems in an adequate manner, we will not present a further description of the problem until we have provided the reader with a solid understanding of virtualization (Chapter 1) and the virtualization framework we intend to make use of (Chapter 2). Having established a common understanding we return to the problem in Chapter 3 and explicitly state our goals. Having stated the goals of the project, we compare approaches taken in related work in Chapter 4. Finally in Chapter 5 we will conclude this part of the report.

Chapter 1

Virtualization

Virtualization is built around two key concepts: A *Virtual Machine Monitor* (VMM) and a number of *Virtual Machines* (VMs). The VMM provides an abstraction that makes it possible to run one or more VMs on a single physical machine as pictured in Figure 1.1 on the following page. Traditionally the VMM provides the VM with a virtual interface[1] for each hardware component. The VMM validates accesses to the hardware through these virtual interfaces to ensure *isolation* between different VMs. Machine resources, such as network interface cards, are multiplexed, ensuring that each VM has the illusion that it is running on a single machine. We refer to an OS running in a VM as a *guest OS* and a system comprised of a VMM with a number of guest VMs as a *Virtual Machine System* (VMS).

The VMM described above is traditionally labeled a Type I VMM. The characteristics of a Type I VMMs architecture is that the VMM runs on the bare machine with a number of VMs. There is however another type, namely the Type II VMM, which is implemented inside a host operating system. [16, p. 22]

1.1 Motivation for Virtualization

There are plenty of motivating reasons to do virtualization. One of the main arguments usually is about doing server consolidation to save money on equipment and optimize the use of processor cycles. There are however many other reasons, we briefly sum up some of them:

- **Run several OSes on one machine:** One OS may be insufficient due to personal preferences or specific software needs.
- **Isolation:** Virtualization provides a powerful sandbox as it gives isolation in terms of quality of service, fault tolerance and data security.
- **Run legacy software:** Virtualization can be used to achieve binary compatibility for legacy software.
- **Testing purposes:** Because of the good isolation, virtualization provides a good environment for testing and debugging.

[1] Sometimes referred to as a shadow copy. This notion confuses concepts later in the thesis, so we will not adopt this terminology.

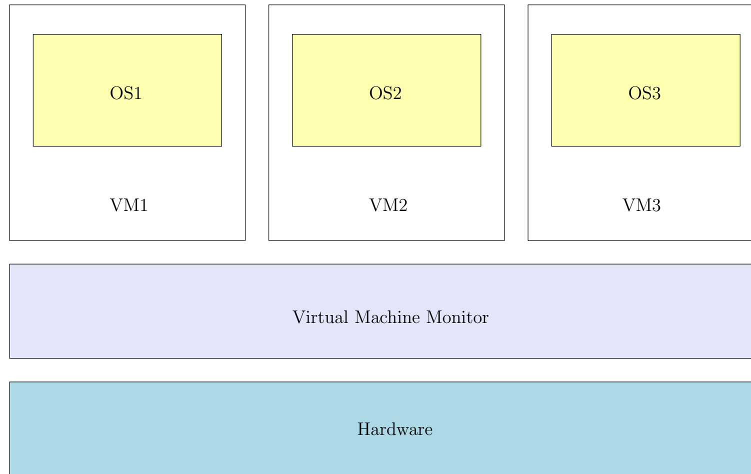


Figure 1.1: Traditional Virtual Machine Monitor Architecture

- **Migration of VMs:** Virtualization can be used to move VMs between different physical hosts.

All of these exciting features do, however, not come for free. The price payed for the isolation is performance.

Before going into the details about virtualization, we give a clear definition of the concepts. Though not taken directly out of "Survey of virtual machine research" [18], the following definitions of *simulation*, *emulation* and *virtualization* are inspired by that article[2]. The terms have since 1974 been used interchangedly and today there is, to our knowledge, no clear definition of the terms. We do however bring the definitions as they illustrate certain points and at least within this report the meaning of the terms should hereafter be well understood. It should be noted that by machine we mean the complete machine without main and secondary storage.

Definition 1 *(Simulation): A simulation is an imitation of machine V on a physical machine P.*

Definition 2 *(Emulation): An emulation is the simulation of machine V on machine P, where the machines are dissimilar ($P \neq V$).*

Software interpreters, such as the Java Virtual Machine, fall into the emulation category. They emulate the state of the machine V, while interpreting one instruction at a time.

Definition 3 *(Virtualization): Virtualization is the simulation of machine V on machine P, where the machines are identical ($P = V$).*

As it can be seen virtualization is a special case of simulation. This kind of simulation may at first seem undesirable, but it holds the key to a low performance overhead. We will discuss this to some length in the next section.

[2]It should be noted that the meaning of emulation has changed over time. Goldberg defines emulation, as being assisted by hardware or firmware.

1.2 Virtualizability

One of the major improvements on virtualization theory was the article [35] wherein Popek and Goldberg described, through an abstracted computational model, exactly what hardware features are necessary for a system to be virtualizable. The result was three theorems addressing first virtualizability, then recursive virtualizability and finally virtualizability for hybrid VMs. Theorems one and three are of interest to us, so we will briefly discuss them here. They do however depend on the notion of *privileged* and *sensitive* instructions, which we will discuss after we have introduced some terminology.

Modern processors have a number of different levels of privilege, numbered in rings from 0 to n, where 0 is the most privileged.

These rings are used to differ from levels of execution, such as between user space running in ring three and kernel space running in ring zero in the Linux kernel on the IA-32 architecture (x86), which has four rings. The most privileged mode, ring zero, in an OS is normally referred to as *supervisor* mode. Any call from user mode to supervisor mode is called a supervisor call or a system call.

In a VMS however, we need to ensure that the VMM is executing at the highest level of privilege to ensure isolation between VMs. Therefore we execute each VM in ring one and let the VMM run in ring zero. While the term supervisor mode refers to an OS running in kernel space, another term is needed to refer to the level of privilege of a VMM. This was named *hypervisor* mode and calls from supervisor mode to hypervisor mode is referred to as hyper calls. This understanding allows us to sum up the definitions from [35] of privileged and sensitive instructions.

A privileged instruction is defined as one which, if not executed with sufficient level of privilege, causes exception which again causes a *trap*. A trap is a jump from one level of privilege to a higher, because an exception has occurred.

Full virtualization relies on moving an OS that normally runs in ring zero to a less privileged ring. Thus any of the privileged instructions do not have sufficient privileges when executed and a trap to the VMM will occur. The VMM can then interpret which instruction was executed and perform any necessary code to create the side effects of the privileged instruction.

Sensitive instructions are divided into two groups: *Control sensitive* and *behavior sensitive* instructions. An instruction is control sensitive if it attempts to change the allocation of resources or affect the processor mode without a trap. A behavior sensitive instruction is one where the effect of the instruction is dependent on its location in memory or the level of privilege. We will however, as Popek and Goldberg, not differ between the two groups and just refer to them as sensitive instructions. Now returning to the central results of the article. Theorem one stated as follows:

> For any conventional third generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.

The authors note that very few machines at that time satisfy this property. This result is also true today on such common architectures such as the IA-32 architecture. Therefore the authors gave a relaxed version of the theorem entailing a *Hybrid VMS*

(HVMS). This is almost identical to a VMS, but it uses more processor cycles to interpret instructions instead of executing them directly. Theorem three stated:

> A hybrid virtual machine monitor may be constructed for any conventional third generation machine in which the set of user sensitive instructions are a subset of the set of privileged instructions.

This result has been used in numerous articles (e.g. [21], [40] and [29]) to identify the instructions that need to be interpreted in order to build a VMM for their respective architectures.

To sum up, the result leaves us with three interesting groups of instructions:

1. Non-sensitive, unprivileged instructions.

2. Sensitive, privileged instructions.

3. Sensitive, unprivileged instructions.

The first class requires no VMM intervention and can be executed directly on the processor. These are then instructions that ensure that we can get good performance with virtualization. The second class will trap when executed. The VMM must then interpret the instruction and provide the expected behavior. Finally the third class is the class of instructions that poses real problems. Use of these instructions must somehow be detected and control must be transferred to the VMM. [40]

Finally the current trend for hardware manufacturers is to support virtualization at the chip level[3]. In particular they have eliminated sensitive instructions that make the IA-32 architecture non-natively virtualizable [29, p. 12]. Thus these chips will still rely on trapping privileged instructions using full virtualization.

While this section has covered virtualizability with focus on the processor, there are also other concerns. These include ensuring valid memory management, updating time as well as delivering interrupts to corresponding VMs.

Finally the fact that we are dealing with multiple running OSes, where the usual terminology is usually intended for a single OS, indicates that we need to separate common OS terminology from virtualization terminology.

Starting at the highest level of privilege, we will refer to a switch to the VMM from a VM as a *hyper switch*, a switch to a VM from another VM is a *world switch* and finally as normal a *context switch* within an OS is a switch between processes or a switch to supervisor mode from user mode. When addressing both hyper and world switches, we will just refer to them as switches. Communication between VMs and the VMM typically happens through hyper calls, while communication between VMs (interdomain communication) typically happens through means introduced by the VMM.

1.3 Other Approaches to Virtualization

Virtualization has been used in numerous systems the last 40 years. Some of the most notable systems are the IBMs VM/370 systems[10], VMware ESX server[48]

[3] Intel has announced its Vanderpool Technology and AMD has announced its Pacifica chip.

(Type I) and VMware Workstation[43] (Type II), which provide *transparent* virtualization. By transparent we mean that OSes run unmodified on VMs. VMware ESX and Workstation both uses an on-the-fly binary translation to rewrite problematic instructions in the guest OS on runtime [48],[41]. VM/370 was one of the few architectures to actually support full virtualization.

Paravirtualization is another approach to virtualization, where the VM presented to the guest OS is not completely identical to the underlying machine. By changing the guest OS and the architecture of the VM, the overhead of identifying privileged instructions by trapping them can be avoided. Paravirtualization instead relies on manually identifying the use of sensitive instructions within the OS and porting the OS to explicitly call the VMM[29, p. 2].

Though first named in [50], as we shall see in Section 1.4 on the next page, it was by no means the first time it was applied.

Definition 4 *(Paravirtualization): Paravirtualization is the simulation of machine V on machine P, where the machines are* almost *identical ($P \approx V$).*

Modifying the OSes can make them virtualization friendly. E.g. Denali [50] and VMWare Workstation addressed performance on network interfaces by reducing the overhead of sending packets. VMware introduced a driver that reduced a total of twelve privileged operations in a guest OS to one privileged operation in the guest OS and twelve in the actual driver. Thus saving the overhead in unnecessarily switching between VMs.

DISCO [5] used paravirtualization to give VMs the abstraction of running on a single processor machine, while in fact running on a highly specialized NUMA multi processor machine. Thus they saved the cost of porting commodity OSes to the new architecture, which is more demanding than porting it to a paravirtualization system.

An often overlooked alternative, recently termed *paene-virtualization*[4] in [42][5], can be found in solutions such as VServer[44],[37]. The concept is to create the abstraction that every process is running as the only user space process on a dedicated machine. This should improve security between processes, which is useful to run e.g. web-servers isolated from other processes, while avoiding the performance penalty of virtualization.

The main drawback is that it does not provide as much isolation as virtualization does. As there is only one kernel running the whole system may crash in the case of a kernel failure.

Finally another approach called *pre-virtualization*[29] was proposed as an alternative to paravirtualization. Their work is based on the observation that the high performance of paravirtualization comes at an enormous engineering cost[6]. Therefore they aim to achieve high performance virtualization at a low engineering cost.

Specifically they insert an extra layer between the guest OS and the VMM. This layer runs as a part of the guest OS and makes sure that the low level API required by the guest OS interacts with the VMM. Basically pre-virtualization relies

[4] Paene is latin for nearly

[5] Early version, the final version is to be published in mid January

[6] They have been maintaining a port of the Linux kernel since 1997, so they understand this cost.

on the same approach as full virtualization: Trapping privileged instructions, which comes at a high cost. They therefore do an analysis and try to automatically identify sensitive and privileged instructions to achieve high performance. This analysis consists of many things, starting from full OS source code and gcc's generated assembly comments to a profile-feedback loop, where they gather details about the OSes behavior: First they compile the OS, then run a special application from the unmodified OS inside a fully virtualized VM. This application is specially designed to ensure a coverage of all privileged and sensitive instructions. Having examined these instructions, they can then use this knowledge and recompile the OS with support for virtualization. If a certain sensitive instruction is not found doing the analysis, then the approach falls back on full virtualization with the overhead of trapping the instruction. This description is rather brief and only serves as an overview. For more details, and there are many more, we refer the reader to the original article [29].

1.4 The Legacy

The last few years the use of virtualization have become increasingly popular. The concept however is by no means new. In fact it was explored as a means of doing timesharing in the 1960es. To give a quick overview we now give a short summary of the work done in that period. For the interested reader we recommend [34] and the definitive survey by the field specialist Robert P. Goldberg [18].

Conceptually not much has changed since then. While in those days a kernel was called a nucleus, the typical virtualization architecture used a VMM to monitor the interaction between the guest VMs and the hardware. One difference was however the focus, at least by Goldberg, on *recursive virtual machines*. The concept was to start another VMM from within a given VM. This could theoretically be done until the machine ran out of resources [17]. To our knowledge there was little practical use of the approach other than to allow a user to launch his own VM with an OS different from that provided by the administrator. Furthermore running a VMM from within a VM environment was considered a good test of the VMM [16, p. 103].

[31] argued that VMs were more "secure"[7] than the conventional multiprogramming approach to time-sharing back then. Today more than 40 years later we would perhaps rather say that VMs provide better isolation, nevertheless the result still holds [50].

The approach used in most systems running VMs then were to run a users jobs within VMs. Examples of this was the TENEX system running on the PDP-10[4] and the VM/370 system[10].

IBMs VM/370 system, as described by one of its creators in [10], was the first VMS able to run multiple distinct OSes on one machine. It provided abstractions of the IBM System/370 machine and offered this abstraction of a whole machine to every user. The reason why they chose to create such an VMS, was primarily that they intended to use it for their own testing and as a learning experience.

[7]The authors divide the notion of security into two concerns 1) reliability failure, meaning user space programs ability to interfere with correct operation of the machine and 2) security failure, meaning a program running in user space ability to take over control of other user space programs or the whole system

The beauty of the system was that it provided compatibility to virtually every OS running on the System/370. The system was extended to become *family-virtualizable* as opposed to *self-virtualizable*, allowing it to provide VMs abstracting any models of IBM System/360 (System versions 30 through 65)[34, p. 108]. Although this had not been named paravirtualization at that time, it still is one of the first, if not the first, examples of systems using paravirtualization (because $V \approx P$).

1.5 Application of Virtualization in Research

Virtualization has provided the platform for a number of interesting research projects. This section sums up the articles that we have encountered during the project. We present these to the reader to show the adversity gained by virtualization.

Virtualization was used in Terra[15] to establish a line of trust spanning from the hardware level up to the individual software components, thus achieving the goal of trusted computing.

Potemkin[46] used virtualization to build a framework to observe the behavior of Internet mal-ware (such as worms and viruses). By using virtualization they were able to service an entire network address space of 64K addresses on a handful of servers.

Friendly Virtual Machines[51] examined optimization and resource balancing through self-adjustment policies in the VMs.

VMs were used in vNUMA [6] to give the abstraction that an entire cluster of machines was a single NUMA machine.

Chapter 2

Xen

In this chapter we give an introduction of Xen. The introduction will start at a high level of abstraction by introducing the Xen approach to virtualization and its architecture. Then, as our focus is on memory, we will explain the details of Xens memory management.

Xen, introduced in [3], is a Type I VMM[1] as discussed in the previous chapter. The VMM runs in ring 0, while the guest VMs[2] are loaded into ring one and the user spaces of the guest VMs is run in ring three.

It uses paravirtualization to achieve high performance, typically reducing the overhead of virtualization to somewhere between 3% to 8%. This overhead was independently verified in [7]. The approach taken by Xen is to apply a *lazy safety checking* strategy, ensuring that as many microinstructions as possible are able to be run directly on hardware, thus achieving higher performance.

Xen originally made three changes to the virtual processor architecture and the guest OSes, which were described in [3, p.3] as:

- **Memory management:** There is no guarantee that a domain is allocated contiguous machine pages. Furthermore direct read access to hardware page tables is granted to guest OSes. Writes to the hardware pages are validated by the VMM.

- **Processor:** Guest OSes run at a lower privilege level than the VMM. Interrupts are replaced by an event system and exception handlers are extended. System calls are made directly to the guest OS, thus not involving the VMM. Finally time within a guest OS is extended with different notions of time.

The last mechanism originally changed by Xen was device I/O. This was however redesigned with focus on isolation in Xen version 2.0 as described in [13]. Furthermore with version 2.0 of Xen, live migration of VMs was also investigated [8] and implemented. As a result VMs can be moved from one physical machine to another[3].

Xen benefits from an active community. Therefore there are a lot of additions being constructed. We will shortly summarize some of the more significant of them.

[1] Also often referred to as a *hypervisor* in the Xen community.

[2] VMs are referred to as domains in the Xen community.

[3] Provided that all necessary storage is network provided and the network topology is local.

Partly to address the problems of managing a large number of hosts (provide network boot images and access transparency to them) and partly to provide a means for doing efficient checkpointing of VMs, a distributed file system, Parallax [49], is being developed.

Xen provides an excellent framework for doing testing and therefore many applications in testing and profiling has been proposed. PDB[24] is a framework built on top of Xen and GDB[4], which enables debugging of distributed applications (such as GRID applications). To diagnose problematic behavior in Xen and thereby facilitate profiling of VMs Xenoprof was introduced in [32], where they also identified several performance overheads in using network devices. A similar utility XenMon [20] was subsequently introduced.

Finally the current state as of this writing is that Xen has released its official 3.0 version. This includes support for the x86/64 and IA-64 architectures as well as support for the new processors with support for native virtualization. [36]

Knowing the high level capabilities of Xen, we now move on to something more concrete: Xens architecture.

2.1 Xen Architecture

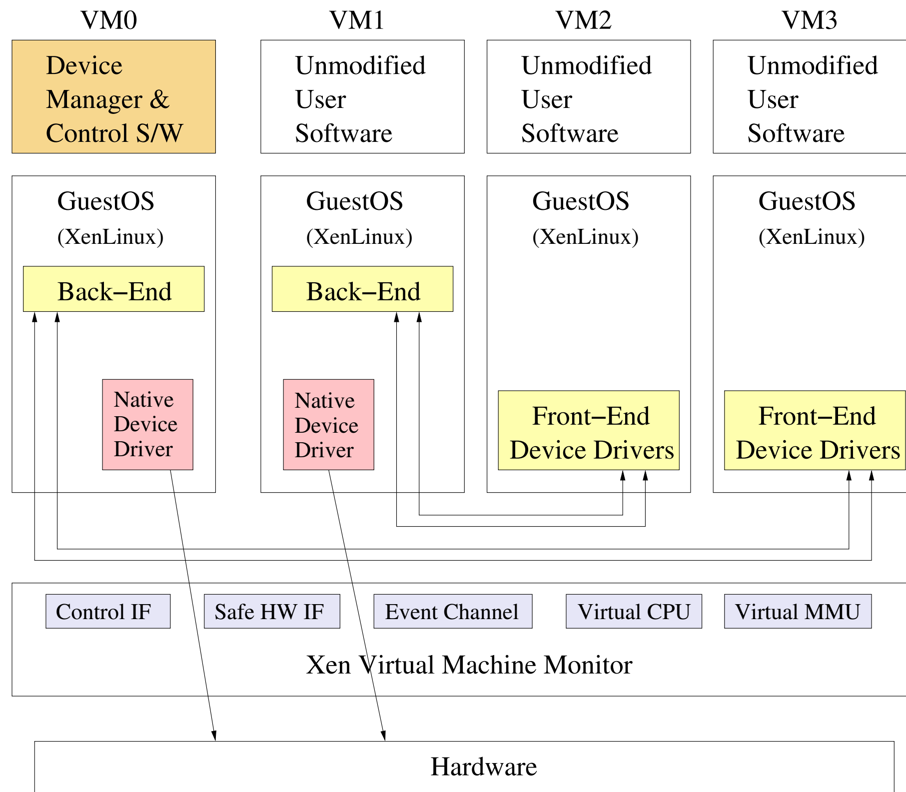


Figure 2.1: Xen architecture

[4] http://www.gnu.org/software/gdb/gdb.html

The architecture of Xen is as pictured in Figure 2.1 on the facing page[5]. At the bottom of the figure we have the hardware level, then the Xen VMM and finally the different VMs, with Domain-0 leftmost. We explain the architecture and key concepts of Xen using the figure. We will do this rather rigorously starting with the VMM.

Control IF: The Control InterFace exposes an interface to manage how the resources, like memory and processor time, are shared between the running VMs. The access to the control interface is restricted to a specially-privileged VM, known as *Domain-0*, which in the figure is called VM0. Domain-0 is required to be present at a machine running Xen, as it runs the application software that manages the control aspects of the Xen platform. This software then uses the exposed control interface in the VMM, when it has to change resource distribution amongst the VMs or create/destroy a VM. [3] [45]

Safe HW IF: The Safe HardWare InterFace exposes the hardware in a safe manner.

An *I/O space* is a restricted environment and is isolated much like a VM is. I/O spaces are the central part in the safe hardware interface. These I/O spaces are arranged such that each device performs its work isolated from the rest of the system. This is done to restrict possible harm done by device faults.

To achieve this the access privileges to the device I/O registers and interrupt lines are restricted. Furthermore where it is possible they protect against device to host misbehavior[6]. Also how the devices view the system is altered so it only sees the resources is has access to. [13]

Event Channel: The event channel, is used to transfer asynchronous events to a VM. In Xen events are used to replace hardware interrupts. Pending event are placed by the VMM in a per VM bitmap[7]. Pending events are delivered by setting a bit in the location corresponding to a given event in this bitmap for a given VM. The guest then receives notifications of pending events with a upcall from Xen. A VMs Guest OS can then check the bitmap to see which events they have pending. [3] [45]

Virtual Processor: The virtual processor in the VMM, illustrates that the VMs access a virtual processor, in which most of the instructions are directly given to the real processor. But a set of privileged instructions are paravirtualized, by requiring them to be validated and executed within the VMM as explained earlier in this chapter.

Virtual MMU: The virtual Memory Management Units responsibility is the validation of each page table creation and modification. This validations primary task is to make sure the pages that are pointed to by a VM is actually owned by the VM. Details about memory management will be discussed in depth in Section 2.2 on the next page.

[5]Based on `http://www.cl.cam.ac.uk/netos/papers/2005-xen-ols.ppt`

[6]Ensuring this completely requires hardware support e.g. from an IOMMU

[7]Referred to by The Xen community as bitmasks.

Having explained the VMM components, we now move on to Domain-0 (VM0):

Device Manager: The Device manager in Domain-0 is responsible for bootstrapping the device drivers, announcing device availability to the OSs, and exporting configuration and control interfaces. [13]

Control S/W: The Control SoftWare interacts with the control interface in the VMM, by use of hypercalls. This might be the creation of a new VM or the termination of one.

Native Device Driver: A native device driver is just a normal device driver. In Xen it has to be run in a privileged VM, that has access to the real device hardware. By supporting the native device driver there is no need to make special device drives for supporting Xen. They are restricted by the I/O spaces, used in the safe hardware interface, so the harm a faulting device driver can do is limited.

Front and Back-End Drivers: A frontend driver is provided access to a device through a back-end driver. The backend driver is responsible for receiving I/O requests from the frontend driver. These I/O requests are verified to ensure that they are safe. If so they are issued to the real device hardware. Since the backend driver has to be in a VM that has access to the real device hardware, it is often run in Domain-0, but can be run in other VMs that has been given a special DF_PHYSDEV flag.

To the kernel, the backend driver appears as a normal usage of in-kernel I/O functionality. When the kernel completes the I/O, the backend notifies the frontend, by use of an event channel, that data is pending. The data is transferred by the use of shared memory. [45]

Having established key concepts and the architecture of Xen, we now go into details about memory management as this is our primary focus.

2.2 Memory Management

In this section we address the changes Xen makes to memory management to make virtualization possible. To do this we sum up memory management in the Linux kernel and address the changes. In particular we will cover page tables, shadow page tables, page faults, ballooning and grant tables.

2.2.1 Memory in Xen

Xen allocates a small portion of the physical memory for its own use and it also reserves a fixed portion in the upper virtual address space of each guest VM on the system e.g. 64MB on the IA-32 architecture [3]. This is to prevent that the TLB is flushed every time a hyper switch is performed. All memory allocations are performed at a page level granularity and the VMM tracks the ownership and use of each page to enforce isolation.

In Linux the memory is normally allocated in contiguous blocks of machine memory, but because the VMM allocates at page level it can not be guaranteed that it will be a contiguous block. This can be a problem because most operating systems does not have good support for fragmented memory. To overcome this problem Xen introduces a pseudo-physical memory, often referred to as physical memory. We will adapt this terminology. Physical memory is a per guest VM abstraction of machine addresses. When using physical memory the address space will look like it is one contiguous range of memory from the guest VMs point of view. However it 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 page frames to physical ones. Furthermore each guest VM has a physical-to-machine table with the reverse mapping. [45]

Throughout the rest of this report we will use the terminology in Table 2.1. The page table (PT) mapping, which is placed in the guest OS, performs a translation from a virtual-to-machine address. The usage of PTs in both Linux and Xen will be explained in the next sections. 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. The latter is however is an optimization, which we will return to in Subsection 2.2.4 on page 26.

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

Table 2.1: Memory Translation Mappings

2.2.2 Linux Page Tables

In the following section the Linux PTs will be explained and the usage of them. This section is based on [19, cha. 3] and [30, cha. 14].

The PT structure in Linux consists of an architecture independent three level structure, even on systems where the architecture does not support it e.g. the IA-32 architecture. On most architectures the *Memory Management Unit* (MMU) handles the PTs.

Each process has a `mm_struct` that points into the *Page Global Directory* (PGD) of the process, which is a physical page in memory. On the IA-32 architecture the PGD is a single 4KB page and each entry is a 32 bit word, which means that it can contain 1024 entries. Each active entry in the PGD points to an entry in the *Page Middle Directory* (PMD), which again points to a *Page Table Entry* (PTE) entry. Finally the PTE points to the page where the actual user data is saved. If a page should be swapped out to backing storage the PTE will contain a swap entry for that page. It should be noted that when the PT of a process is loaded on the IA-32 architecture the *Translation Lookaside Buffer* (TLB) is flushed. The TLB is a cache

where address translations from virtual to physical is saved to speed up memory lookups.

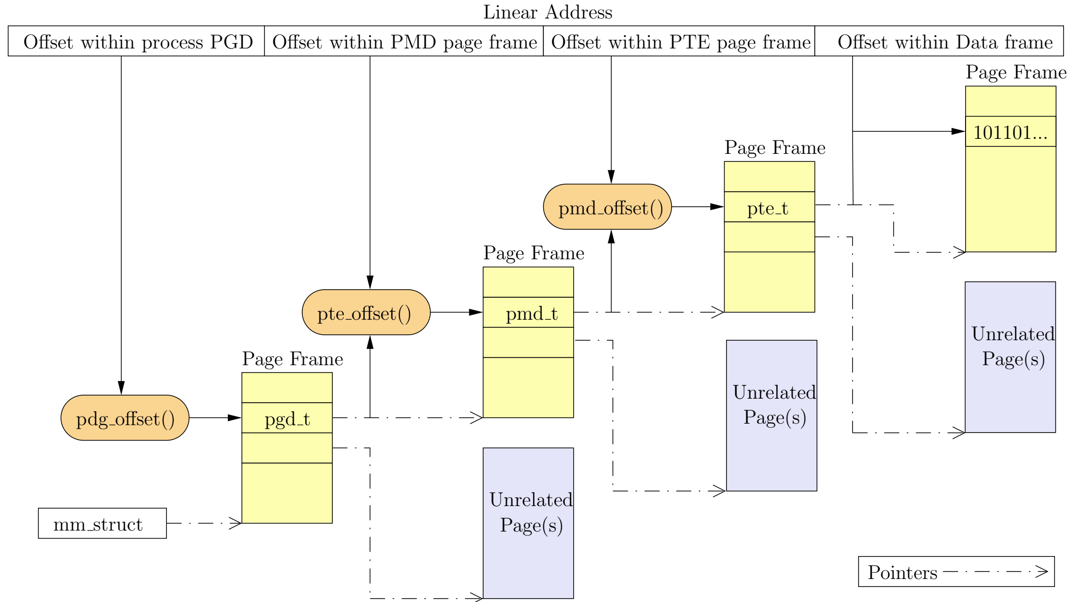


Figure 2.2: Lookup in the Linux page tables

Figure 2.2 illustrates how a lookup for a given page is performed. The linear address is split into four parts, where each part is used as an offset. The first three parts are used on the PTs and the last one on the actual page. First we find the right index into the PGD table by using the first part of the address as an offset. We follow the pointer located at that offset into the PMD page frame. Then we use the second offset from the address to find the right place in the PMD table and follows that pointer into the PTE. We use the third part of the address as an offset into the PTE and finds the pointer to the actual page containing the data. The last part of the address is used as an offset into the actual page.

As mentioned this three leveled PT structure is not supported on the IA-32 architecture. Actually the IA-32 only supports a two level structure, which is handled by directly "folding back" the PMD onto the PGD and it is optimized out at compile time [19, p. 33]. It should be noted that this is only true when not using the *Page Address Extension* (PAE), which adds four bits more to give 36 bit addressing. This will break the 4GB memory limit and provide a total of 64GB memory.

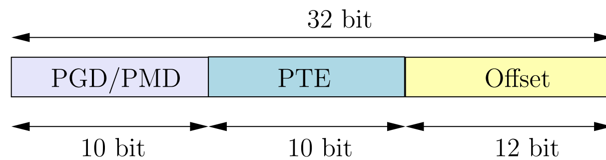


Figure 2.3: Virtual address on the IA-32 architecture

On the IA-32, without PAE, a virtual address is 32 bit, a single word. As it can be seen in Figure 2.3 on the preceding page the virtual address is split into three parts. The first 10 bits of the address is used as an index into the PGD, which means that the table has 1024 entries (2^{10}) and since each entry is 4 bytes wide it requires exactly one page (4KB) in memory. The next 10 bits are used as index into the PTE in the same manner. These lookups in the page tables gives the location of the actual page in memory and the last 12 bits are used as an offset into that page.

2.2.2.1 The Page Table Entries

Each entry in the page table structure consists of the structs `pgd_t`, `pmd_t` and `pte_t`, which again are architecture dependent.

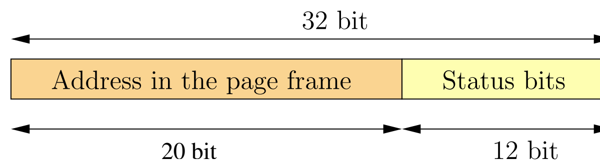


Figure 2.4: A single 32 bit page table entry on the IA-32 architecture

In Figure 2.4 a single page table entry is illustrated as it is implemented on the IA-32 architecture. The first 20 bits are used as the offset into the page frame as seen in Figure 2.2 on the preceding page. The last 12 bits are status and protection bits and are listed in Table 2.2, but it should be noted that the meaning of these bits varies between architectures.

Bit	Function
_PAGE_PRESENT	Page is resident in memory and not swapped out.
_PAGE_PROTNONE	Page is resident, but not accessible.
_PAGE_RW	Set if the page may be written to.
_PAGE_USER	Set if the page is accessible from userspace.
_PAGE_DIRTY	Set if the page is written to.
_PAGE_ACCESSED	Set if the page is accessed.

Table 2.2: Page Table Entry Protection and Status Bits [19].

2.2.3 Xen Page Tables

The most significant changes in the memory management system in Xen is the management of PTs and the possibility to create shadow page tables.

When using Page tables the guest OS has direct read access to the Page tables, while updates of the page tables must be validated by the VMM [3]. A single page can have five different types and they are mutually-exclusive. The two types PD (page directory) and PT (page table) are used to indicate that a page is part of a page table structure. The local descriptor table (LDT) and global descriptor table (GDT) is used by the guest OS if it does not support paging but segmentation.

The last type is used to indicated that a page is writable (RW). That the types are mutually-exclusive is very useful when validating a write to a page frame. This protects the page tables in the guest OS from accidentally or purposefully violating other VMs address space.

Furthermore each page in the memory has a reference counter, which keeps track of the number of references to it. A page can not be reallocated as long as it has a type and the reference counter has not reached zero. When a guest VM creates a new process it is expected to allocate and initialize its own PT from inside its address space and register it with the VMM. When it has registered with the VMM, there are two possible ways to make updates to the PTs. [45]

Instant validation: When a process in the guest OS wants to update one of its PTs it makes a hypercall (`mmu_update`), which transfers control to the hypervisor. The hypervisor then checks that the update does not violate the isolation of the guest VM. If it violates the memory constrains, the guest OS is denied write access to the page table. If it does not violate any constrains it is allowed to complete the write operation and update the PT. [45]

Just in time validation This method gives the guest OS the illusion that their page tables are directly writable. The VMM traps all writes to memory pages of the type PT. If a write occurs, then the VMM will allow writes to that page, but at the same time disconnects it from the currently active page table. In this way the guest OS can safely make updates to the page, because the updated entries cannot be used by the MMU until the VMM validates the page and re-connects it to the page table [45]. The VMM re-connects the page when: the TLB is flushed, a page in the unconnected page-tables page is accessed or the guest OS modifies another PTE entry in a separate PTE.

The PTs are only handled this way when the guest VM request it through a hypercall vm_assist. 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. [45]

2.2.4 Shadow Page Tables

As mentioned the normal PTs are address translations from virtual-to-machine, where each VM has direct read access to its own PT. The only restriction is updates, which has to be validated by the VMM.

When in SPT mode, the PT in the VM is not used directly by the hardware. In fact the guest OS does not have access to the SPT, which is used by the hardware [46] and the OS PT is not performing a mapping from virtual-to-machine, but a virtual-to-physical mapping.

The SPT gives the VMM the opportunity to track all writes performed to the pages in the PT. This have been used in e.g. live migration [8] to detect dirty pages while performing live migration. SPTs have a performance penalty in the form of keeping two PT structures up-to-date and the extra memory usage needed to save the SPT. The SPT is dynamically generated from the two mappings (PT, P2M), which can be seen in Figure 2.5 on the next page and can be discarded at any time.

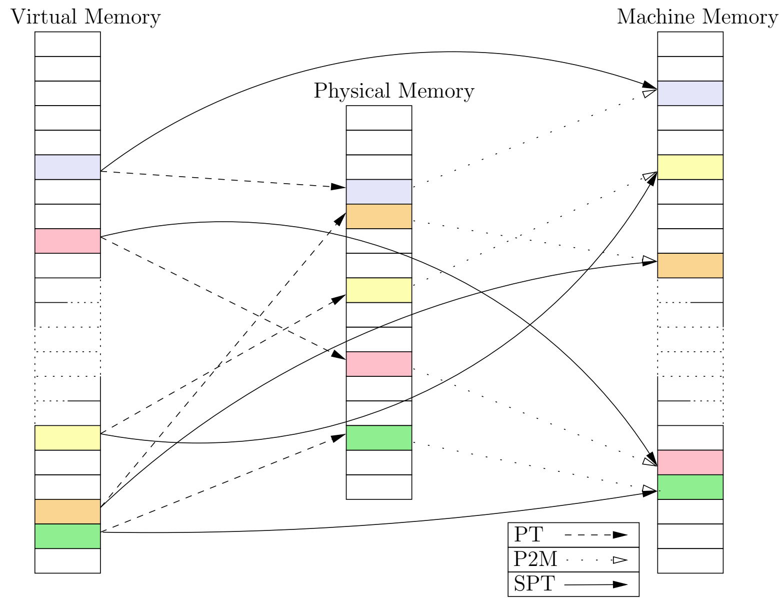


Figure 2.5: Mappings of Page Tables, Physical-to-Machine and Shadow Page Tables.

In fact each time a context-, world- or hyper switch occurs the SPT is discharged. This can however be optimized by having a SPT cache, which will take up extra memory to make SPT persistent between switches.

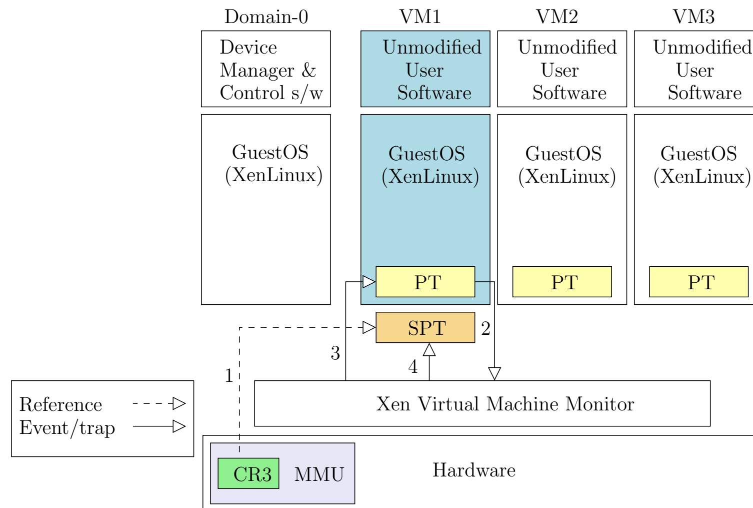


Figure 2.6: Update of the Guest OS Page Tables with Shadow Page Tables on the IA-32 Architecture.

Figure 2.6 illustrate three VMs, where VM1 is currently active and uses a SPT. The following enumeration explains how a change to the guest VMs PT is handle in shadow mode.

1. When Xen is not in SPT mode the active VMs page table is accessed via the

CR3 register in the processor. This means that all translations are handle by the hardware MMU. But as it can be seen in the figure, the CR3 register points to the SPT, which the VM does not have direct access to.

2. When the guest OS tries to perform a write to a PT it gets trapped by the VMM, which as always validates the write to ensure isolation.

3. The write is granted or denied and the guest OS is notified with an event.

4. If the write was granted then the write is also propagated to the SPT.

We have now explained the functionality behind page tables and shadow page tables in Xen. We will now explain what happens when the memory management unit raises an exception, also known as a page fault.

2.2.5 Page Faults

This section is about page faults in the Linux kernel and how Xen handles these. The subsection is mainly based on [19, Cha. 4] and [3].

The pages of a process are not necessarily in memory in Linux. Often they are swapped out to disk and if accessed a *page fault* is generated. A page fault can also occur when some page is marked as read-only and a write operation is performed on that page. Linux uses a *Demand Fetch* policy for dealing with pages that is not in memory, which means that a page is not fetched into memory before the hardware raises a page fault. What the OS does is that it traps the page fault and allocates a page frame to bring the needed page back into memory. Page faults are divided into two groups depending on how expensive they are: major and minor faults. A major fault occurs when an expensive operation is required to handle the fault e.g. a disk read is needed to fetch a page from swap. A minor, or soft, page fault is when it is fairly simple to correct the fault e.g. a write to a page, which has been marked as read-only.

When a page fault occurs on the IA-32 architecture the fault is trapped by the kernel and the exception handler is called. This reads the faulting address from the processor register CR2. The first thing the exception handler checks which context the page fault came from, kernel or user space. If the page fault happened in kernel space and the faulting address is in the kernels address space, the fault is handled. If the kernel tried to access memory outside its address space or if the page fault occurred in interrupt context the kernel generates an oops.

A segmentation fault (SIGV) will be generated and the process is terminated if the page fault occurred in user space and one of the following scenarios are true: The user space process has tried to access 1) a NULL pointer, 2) kernel space, 3) invalid virtual address or 4) tried to write to a read-only section in virtual memory. If none of these are true, then the exception table is accessed to find the address of the "fixup" code and the EIP register (program pointer) is updated to point to that code, which is then executed. When this has been done the control is handed back to the process that invoked the page fault.

In Xen page faults are handled as in unmodified Linux with the exception that a guest VM can not read the CR2 register, because it requires privileged instructions.

So the VMM has to copy the faulting address from the CR2 register into the stack frame of the guest OS, which has been extend with one word to contain the address. The faulted guest OS can then read the address and handle the fault, which may lead to updates of the page table. It should be noted that the handling of page faults is architecture dependent and the register mentioned in this section is only true for the IA-32 architecture.

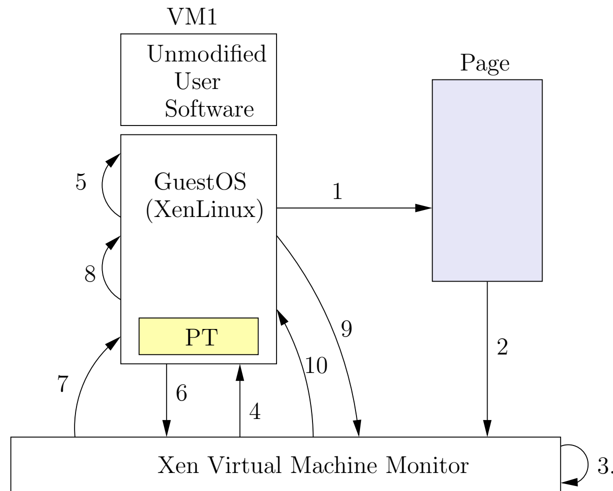


Figure 2.7: Handling of Page Faults in Xen

The following enumeration explains how Xen handles a page fault and which parts of the virtualized environment handles the different tasks. The page fault process is illustrated in Figure 2.7.

1. A process in the guest OS tries to read or write to a page, which triggers a page fault.
2. The VMM traps the page fault, which requires a hyper switch to the VMM.
3. The VMM copies the faulting address from the register in the processor into the extended stack frame.
4. The VMM notifies the guest OS by sending an event.
5. The guest OS reads the faulting address from the extended stack frame and finds the address of the "fixup" code in the exception stack.
6. We have to make a hyper switch back to the VMM to update the program counter such that the "fixup" code is executed. This is because the program counter can only be updated by executing privileged instructions.
7. A hyper switch back to the guest OS is required. This is because it is in the guest OS address space that the "fixup" code is located.
8. The "fixup" code is executed.

9. The execution of the “fixup” code requires that the PT in the guest OS is updated. This update has to be validated by the VMM and yet another hyper switch is required.

10. The process that provoked the page fault can continue its execution.

With the explanation of page faults, we have covered the usage of page tables. Dynamically changing the memory footprint of a VM is difficult. In the next section we explain a technique called ballooning, which is used to change the memory footprint of a VM.

2.2.6 Ballooning

If the VMM needs to reclaim memory pages from the guest OS it is put in an awkward position, because it has no knowledge about the usage of pages in the guest OS. Therefore it will have to make uninformed choices when selecting pages to swap to disc. The guest OS on the other hand has better knowledge and thus the best way to reclaim pages would be to make the guest OS give the VMM the amount of pages it requests. A technique for doing this, called ballooning, was introduced in [48].

Ballooning is a driver, which is loaded into the guest OS, from where it can be inflated, thus increasing memory usage load possibly forcing the OS to swap inactive pages to disc. When the memory load increases in the guest OS, it automatically starts its memory management algorithms and make its *working set* smaller. The working set is the set of virtual memory pages actively used by the current process. The pages allocated by the balloon can be used by the VMM and when the balloon is deflated the pages are returned to the VM.

Chapter 3

Motivation and Goals

Having introduced the problem domain, we now go on to address the goal of the project.

Main memory has always been a scarce resource in any environment. However in a virtualized environment it tends to become a bottleneck somewhat faster. Processors in desktop machines today are sufficiently large to support ten medium large (256 MB RAM) servers with little load. As most processor cycles today are wasted, this seems like a likely scenario. However supporting ten VMs with 256 MB RAM each puts us in a pickle, as common desktop computers today typically have 1-2 GB RAM. This forces us to lower the RAM allocation for each VM, thus potentially affecting performance by thrashing due to swapping. If we need to scale even further in terms of the number of VMs, the need for memory really becomes a bottleneck.

Dynamically reassigning the memory allocation from VMs that may not require all of their memory through ballooning (as described in Subsection 2.2.6 on the facing page) can improve the situation, but only to some extent. You can only remove a given amount memory from a VMs working set without seriously affecting performance. Furthermore determining how much this amount will have to be on runtime basis, as a VM may require more memory from one run to another.

Our approach, as others before us, is to limit the size of a VMs memory. If several VMs are running the same OS, then there is a probability that they share identical data, program code and software managed caches. Identifying these potential duplicates will allow us to reduce the memory usage.

The remainder of this chapter is divided as follows. In Section 3.1 we present the results of some preliminary experiments we conducted to investigate if there is potential for sharing memory between VMs. Having presented these, we state the goal of the report and finally describe limitations.

3.1 Experiments

The results in this section are not conclusive, meaning that there are too much potential causes of errors to conclude anything definite. Furthermore they are not representative as we have not conducted all of our intended experiments. We present them solely because they do indicate that there is a non-negligible potential for sharing memory, which serves as a motivation for us.

3.1.1 Working Set Changes

Our first concern was that the contents of memory changes so fast that sharing is not feasible.

We took snapshots of a single non-virtualized hosts memory by hashing[1] every page in memory. The hash values of one snapshot was then compared with the hash values of another snapshot taken later. The result of the analysis is shown in Figure 3.1.1, which pictures how the content of memory changes over time on an idle desktop machine.

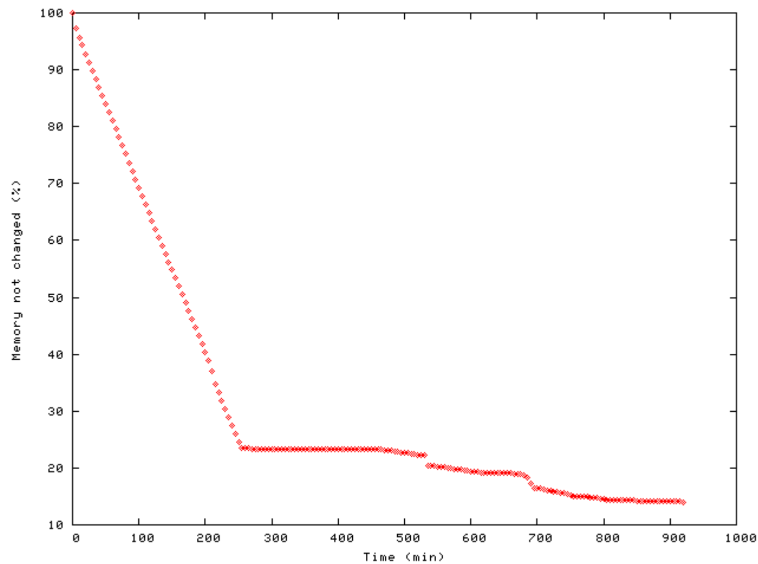


Figure 3.1: Memory content change on idle desktop machine

In particular the figure shows the first memory snapshot compared to all other snapshots. In the figure we notice that memory changes steadily over time. What is surprising is that it does not change as fast as expected, after 165 minutes 50% of the memory is still unchanged. We conducted this experiment three times total, all with similar results.

It should be stressed that these results are from an idle host. For the final report we plan to carry out similar experiments for both an idle and busy web server.

Working set changes were examined both in [8] and [46]. In [8] the changes in memory were crucial for the migration of VMs from host to host. They examined the working set for a set of different loads. Low loads changed roughly 20 MB, medium loads changed 80 MB and high loads changed roughly 200 MB of a total of 512 MB. They conclude that normal workloads will lie between these extremes. Vrable et al. [46] showed how much memory changes (within minutes) in VMs running only small services. None of the services changed more than 4 MB of memory within two minutes.

[1] The use of a hash function in this case was MD5, without handling conflicting hash values.

3.1.2 Sharing Potential

We examined a number of machines from common desktops to in use web servers and saw that typically $6-10\%$ of the pages in memory could be eliminated by using sharing[2].

Finally we experimented to see whether any two VMs would have anything to share. The experiment from which the results is illustrated in Figure 3.1.2, compared two VMs running separate instances of the Apache web server (named A1 and A2).

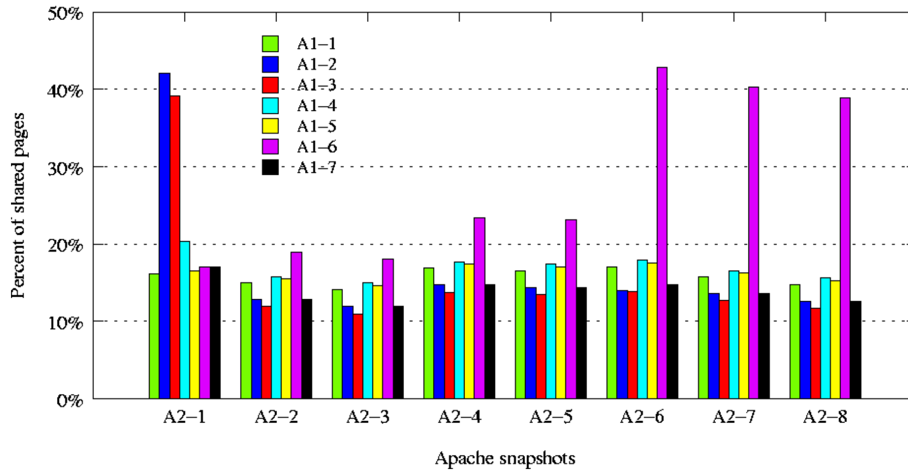


Figure 3.2: Comparison of shared pages between Apache web-servers, running in Xen domains

The first column in the figure presents a snapshot of A1 compared with three snapshots of A2. The next column shows the next snapshot compared with the same three snapshots as before and the third column continues the pattern.

We notice that some snapshots present the opportunity to share around 40%, but in average we should expect to share only around $10-20\%$[3].

Furthermore several real life examples on sharing between VMs exist in other related work, we will return to this work in the next chapter.

3.2 Project Goal

As the pervious section indicated, there is a potential for sharing memory between VMs. To fully explore this scenario, we will make an implementation and conduct a number of experiments. The implementation should perform well, so there is a chance that it will be included in the Xen framework. Formalized the goal of the project is to:

> Study existing techniques for doing memory sharing between Virtual Machines. This should be used to design and implement a solution that has

[2] This typically included $1-2\%$ of zero pages

[3] The servers were receiving a moderate load to the same non-dynamic pages, which was presumably cached. This could explain the high percentages.

little performance overhead. Finally the solution should be evaluated and compared to existing solutions.

3.3 Limitations

The basis for the implementation will be Xen version 3.0 and we limit it to modifying the XenLinux 2.6 kernel. Furthermore we will of course try to keep as much of the project architecture independent, but should we face the situation where architecture dependent code is needed, then we will favor the IA-32 architecture without PAE support[4]. We do this primarily because this is what Xen was primarily developed for[5] and because this is the only type of architecture we have access to.

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.

[4] This also entails that we will not be able to support the new hardware virtualization tecnology.
[5] Although not the only architecture anymore

Chapter 4

Related Work

This chapter summarizes previous work done on sharing memory between VMs. 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 only two approaches to sharing memory between VMs: 1) Use prior knowledge about certain blocks of data being identical or 2) actively compare the blocks to find identical blocks of data.

4.1 Compare-By-Hash

Compare-by-hash, as described in [22] and [23], is a technique to do fast comparisons of blocks of data. The reason why it performs well compared to a naive bitwise comparison of two blocks, is that it computes a hash value from two blocks. If the hash values collide, then there is a good probability that the two blocks are identical. 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.

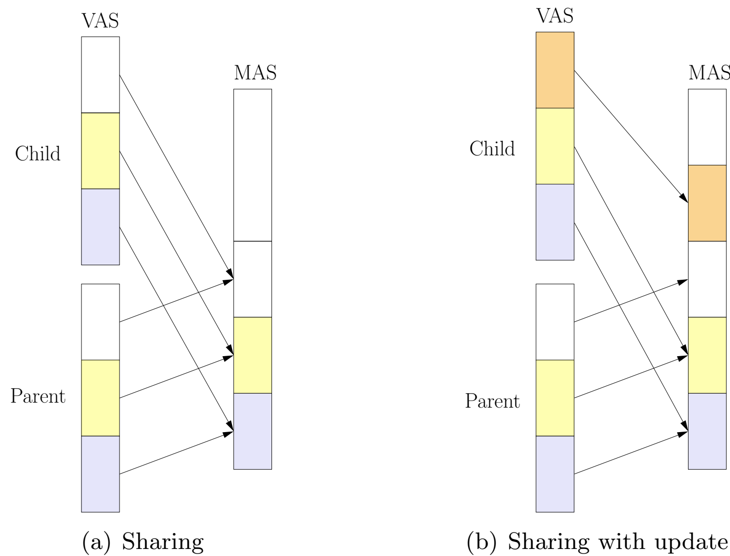
In [23] they list a lot of considerations that could indicate that the use of compare-by-hash is not correctly applied. One of the most important advises are that the technique should not be used as the only means to ensure correctness of data. Furthermore as hash functions are discovered to be unsecure at a large scale, the hash values should only be used temporarily i.e. thrown away after use.

4.2 Copy-On-Write

Copy-on-Write (COW), first introduced for memory sharing in the TENEX system in [4], is a *lazy optimization* technique generally applicable within computer science. As our focus is on memory, so we will explain it from this point of view.

The concept of COW is to make duplicates of data objects by not creating an identical copy, but instead giving a pointer to the original data object. Updates to data objects shared in this manner must be intercepted to make sure that no

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



(a) Sharing (b) Sharing with update

Figure 4.1: On the left picture both the parent and child have identical virtual address spaces. On the right picture an update has been performed by the child and a new page has been created.

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 sharing can be removed when it is unnecessary.

The technique is for instance used in the Linux kernel when forking processes as pictured in Figure 4.1. Each process has a PT that translates from Virtual Address Space (VAS) into the Machine Address Space (MAS). 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, which can be updated[2].

Having discussed sharing of pages within an 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.

4.3 Shared Virtual Storage on VM/370

Parmelee et al. reported briefly in [34, p. 117] that sharing of memory pages between VMs was possible on the CP-67 (one of the components of the VM/370). It was however tedious as the parts of memory had to be marked read-only and the task of identifying parts to share was left to the programmer.

Bagley et al. [1] proposed a set of changes to the VM/370 to allow sharing both main memory segments and auxiliary storage between VMs. As VMs were used to facilitate timesharing and a users jobs ran in a single VM, obtaining means of

[2]Further details can be found in [19, p. 87] and [30, p. 31]

doing interprocess communication was costly. They did this by altering the VMM to update the page tables entries of the guest VMs to point to the same machine addresses. Thus the user could designate a certain area of memory for communication. Whether they intended to use the scheme for actually reducing the memory footprint is doubtful.

Wahi investigated the feasibility of dynamically sharing pages on CP-67 in [47]. He proposed a policy to determine whether is was feasible to share pages at a given time depending on the system load.

4.4 Transparent Page Sharing

The Disco system [5] 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 uses prior knowledge to share identical blocks of data, which includes sharing caches and user space data based on their location on disc.

4.5 Content Based Page Sharing

VMware introduced[3] *Content-Based Page Sharing* (CBPS) in [48]. The concept is based on the Compare-by-hash technique as described in Section 4.1 on page 35.

A service 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 in different VMs, then there is a good probability that the current page is identical with the page that gave the same hash value[4]. VMware ESX server uses a 64 bit hash function [26] 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.

Returning to the concerns raised in [23], as the authors also notice, content-based page sharing is one of the good applications of compare-by-hash. If hash collisions occur, then the pages are just not considered for page sharing, thus removing the concern for collisions[5]. 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 and the correctness of the system is not coupled to one certain hash function.

VMware was able to identify as much as 42.9% of all pages as sharable and reclaim 32.9% of the pages from ten instances of Windows NT, doing real-world workload. Nine VMs running Redhat Linux were able to find 29.2% sharable pages and reclaim 18.7%. Reduced to five VMs the numbers were 10.0% and 7.2% respectively.

The CBPS principle was also implemented as a patch for the Linux kernel in mergemem[38] demonstrating that there is duplicate pages to eliminate within a single OS.

[3] Although the author does not directly take credit for the technique in the article, another article by VMware researchers [41] claim that it was VMware that introduced content-based page sharing.

[4] A naive approach, comparing each page with all other pages one at a time, has a complexity of $O(n^2)$

[5] We will return to this concern in Section 7.3 on page 60.

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 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. [33]

The same problem applies to sharing memory between different VMs. Depending on the implementation of the sharing scheme[6], 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. In an OS the attacker is able to tell which processes are running and then direct the attack. In the 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.

4.6 Flash Cloning

Another approach was taken by Potemkin[46], a Xen based framework, which is able to launch a large number of VMs by introducing new functionality termed *Flash Cloning* (FC) and *Delta Virtualization* (DV).

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[7], 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 it 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 referential image of size 128 MB and 116 clones of this image. All of the clones in use consumed a total of 98 MB, implying that each VM only changed roughly about one MB of its data.

4.7 Comparison of the Different Approaches

Now we discuss advantages and disadvantages of the CBPS and the FC approaches.

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

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

[7]Not unlike what we saw in the Linux kernel in Section 4.2 on page 35

as well as actually writing the new page. Therefore as updating pages is more expensive than the plain solution, 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 CBPS approach is that it introduces a new overhead: Finding candidate pages for sharing. Some scheme for scheduling the task must be devised. CBPS can identify all potentially sharable pages by contents, thus it can achieve as high or higher share percentage than FC[8].

While the FC approach has demonstrated great memory sharing numbers, it should be noted that Potemkin sets up their solution in a manner where the terms are almost perfect for sharing memory. While their setup are 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 FC.

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

Chapter 5

Conclusion

This part of the thesis first provided an overview of virtualization in Chapter 1. In particular we discussed how computers are able to host a virtual machine monitor, even if they are not fully virtualizable. Furthermore we summarized much of the important work done in the field of virtualization.

Then in Chapter 2 we used this basic understanding of virtualization to explain how the Xen framework is built. Our focus was mainly on memory management as this is the focus of this thesis, but we also gave an overview of the key concepts used in Xen. The most important results in this chapter were shadow page tables and ballooning. These constructs enable Xen to dynamically change the memory footprint of a virtual machine.

In Chapter 3 we presented the motivation behind the thesis and stated the goals of the thesis. In particular we explained that we have chosen to build an implementation of a scheme for sharing memory between virtual machines.

With our intentions to modify Xen to be able to share memory between virtual machines, we explored schemes for doing this in Chapter 4. We found that in general are two ways to do this, either compare pages or use prior knowledge to identify identical pages. As Vrable et al. [46] has already presented an a priori system for doing memory sharing in Xen, we therefore choose to base our implementation on content-based page sharing. This gives us an unique chance to verify the results provided by [48].

Part II

Design

Part Introduction

This part of the report is a mixture of two things. Foremost we present several possible designs in Chapter 7. We analyse the designs according to our design goals and finally choose which design to stick with.

The designs do, however, require some considerations to data structures. Therefore we use Chapter 6 to discuss data structures relevant to the design. The main focus of the discussion will be their space and performance overheads.

Finally in Chapter 8 we outline the work that lies ahead for the second semester of this master thesis project.

Chapter 6

Techniques

In this chapter we discuss techniques needed for the implementation of content-based page sharing. In particular we need to find a hash function that performs well to compare memory pages with. Therefore we first explain what we consider to be a good hash function (through three requirements) and then evaluate several hash functions based on these requirements. Having found a suitable hash function we investigate suited data structures to save the hash values of pages.

6.1 Hash Functions

Before examining the different hash functions, we have to outline some requirements that the functions must comply with.

- Low collision rate to keep the lookup speed as close to constant time as possible, while producing as short a hash value as possible.
- The function is intended to be implemented in the kernel or VMM and thus it should preferably be architecture independent.
- The hash function should perform as good as possible, both in terms of memory usage and processor cycles.

We have chosen to examine five different hash functions and conduct a performance analysis on them. To our knowledge these hash functions are the ones that fulfill our requirements the best. This performance experiment will aid us in selecting the best suited hash function for our implementation. The hash functions that we have examined at will be explained in the following subsections.

6.1.1 Fowler / Noll / Vo (FNV)

FNV is a hash function used in everything from games to anti-spam filters. It is designed to be fast and at the same time have a reasonable low collision rate [12].

The FNV hash function comes in several different versions, but we have chosen the 16, 32 and 64 bit versions in two different implementations. The only difference between the two implementations is in the order of which it applies the *xor* and *multiply* operations in the core algorithm. From the core algorithm of FNV (Listing

6.1), it can be seen that it is only possible to construct hash values on data which can be divided with an octet (8 bits).

```
1      hash = offset_basis
2      for each octet_of_data to be hashed
3           hash = hash * FNV_prime
4           hash = hash xor octet_of_data
5      return hash
```

Listing 6.1: FNV Core Algorithm

It should be noted that the performance of this hash function depends on the selection of the prime number *FNV_prime* [12].

6.1.2 Bob Jenkins (BJ)

Bob Jenkins has constructed this hash function, which he claims is thorough and faster than other hash functions [26]. It is designed to be fast and perform well when used in hash tables. This means that the hash algorithm has been designed with considerations to uniform distribution of the produced hash values. The BJ hash function has also been used in the VMware ESX server [48]. It uses $3n + 35$ instructions to hash n bytes [26]. We have tested two different version of this hash function. The first function named 32a in Table 6.1 and 6.2 uses 8 bit fragments to perform calculations on, while 32b uses 32 bit words.

6.1.3 SuperFastHash (SFH)

This hash function is developed by Paul Hsieh to challenge the performance of Bob Jenkins hash function [25]. It uses 16 bit fragments internally instead of 8 or 32 bits as BJ does. The reason for using 16 bit is that IA-32 have hardware support for 16 bit unaligned words, which have a positive impact on performance on the IA-32 architecture. It should be noted that it still outperforms Bob Jenkins on other architectures.[25]

6.1.4 Secure Hash Functions MD5/SHA1

These two hash functions are secure hash functions, which means that they have a high probability of creating different hash values for each data object they parse. Secure hash functions are mostly used in cryptography, e.g. where it is important to verify the correctness of a message sent over a network.

MD5 generates a 128 bits message-digest of a message of arbitrary length, which should be unique for any given message [39]. It should be said that MD5 is no longer considered secure. It have been showed that it is possible to construct two useful messages, which have the same message-digest [27].

Secure Hash Algorithm 1 (SHA1) produces a 160 bits message-digest on any message with length $< 2^{64}$ bits as input. SHA1 is considered secure because it is computationally infeasible to find two messages which gives the same message-digest [11]. Internally SHA1 uses data blocks of 512 bits, hence SHA1 uses padding to make sure that the input data is dividable by 512 bit.

These two hash functions guarantee, with high probability, that the digest for a message (memory page) is unique, which is an important property for our usage. The downside is that the hash value is 128-160 bits in length, which requires more space when the hash values are stored for later use. Furthermore when comparing the hash values later on, it will take more time to compare values containing many bits.

6.1.5 Evaluation

We have made three experiments to find the best suited hash function for our usage. The experiment is executed on pseudo random data, where an input page (4096 bytes) is hashed with the given hash function. The experiment has been performed on 256MB (65536 pages), 512MB (131072 pages), 1GB (262144 pages), 2GB (524288 pages) and 4GB (1048576 pages) input files.

The first experiment tests how many collisions occurs on the input data sets. The results from our first (of three) data set can be seen in Table 6.1. Appendix A on page 81 contains the rest of results. The input data was pulled from the entropy pool of the OS and it was checked that there were no identical pages in the input file. From the table we can see that the 16 bit FNV hash is too short, as it has too many collisions[1]. The 32 bit keys have a few collisions but nothing that will have impact on the performance of the function. The 64 bit keys do not collide at all and as expected the two secure hash functions does not collide as well. So the only function that we can exclude on the basic of the collision table is the 16 bit FNV hash.

Hash function	256MB	512MB	1GB	2GB	4GB
FNV 16	63.3%	86.5%	98.2%	100%	100%
FNV 16a	63.3%	86.5%	98.2%	100%	100%
FNV 32	2	2	14	60	268
FNV 32a	2	2	14	60	268
FNV 64	0	0	0	0	0
FNV 64a	0	0	0	0	0
Bob Jenkins 32a	0	6	12	40	230
Bob Jenkins 32b	0	4	14	48	276
Bob Jenkins 64a	0	0	0	0	0
Bob Jenkins 64b	0	0	0	0	0
SuperFastHash	0	10	12	70	260
MD5	0	0	0	0	0
SHA1	0	0	0	0	0

Table 6.1: Collisions in the different hash functions on five different input sizes.

The second experiment is a timing test. It measures the processor time used when calculating hash values. To avoid the overhead of disc access, we load the entire input into memory. We have only performed this experiment on input files

[1] It should be note that the numbers for the 16 bit hash values is in percentage, because of the high collision rate.

with a size up to 1GB. This is because we do not have any machine, which can load the files into memory of larger sizes.

Hash function	256MB	512MB	1GB
FNV 32	1.22	2.44	4.87
FNV 32a	1.28	2.58	5.17
FNV 64	3.17	6.33	12.73
FNV 64a	3.06	6.10	12.29
Bob Jenkins 32a	1.05	2.10	4.21
Bob Jenkins 32b	0.55	1.12	2.22
Bob Jenkins 64a	1.85	3.71	7.36
Bob Jenkins 64b	1.13	2.27	4.45
SuperFastHash	0.38	0.75	1.50
MD5	1.09	2.17	4.38
SHA1	3.30	6.71	13.54

Table 6.2: Processor time (in seconds) used to hash data loaded into main memory.

The result of this experiment can be seen in Table 6.2. In our experiment it can be seen that the SFH function outperforms the other hash function in processor time spent on the function. From the experiment we can conclude that the SFH function is the fastest function, that still keeps an acceptable collision rate.

Our third experiment tests the distribution of hash values inside the range of values. Figure 6.1 on the facing page illustrates the distribution on a 32MB input file[2] with Bob Jenkins and SFH 32 bit versions. The x-axis is the page numbers[3]. The y-axis is the range of hash values that the given function hashed to. As it can be seen on the figure the distribution of hash values does not cluster together, which means that the distribution is acceptable in both functions. We can conclude that SFH does not suffer non-uniform distribution compared to BJ. Furthermore as it yields the best results in our tests, it is well suited for our implementation.

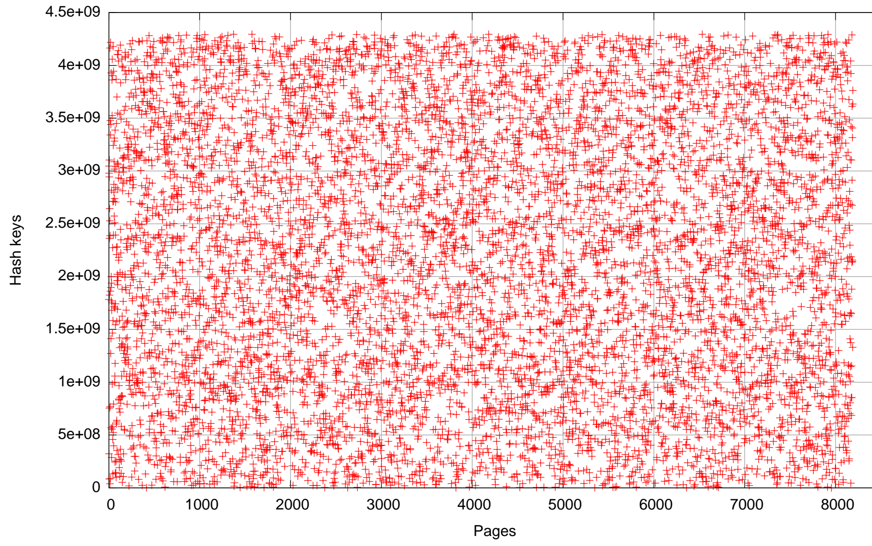
6.2 Data Structures

In this section we take a closer look at different data structures suited for our use. The focus will mainly be on hash tables as a general result is, that they are superior compared to tree structures in regards to space and speed consumption [28, p. 513]. The following section on hash tables is base on [9, cha. 11]. After that section we take a closer look at the Hash Array Mapped Trie data structure.

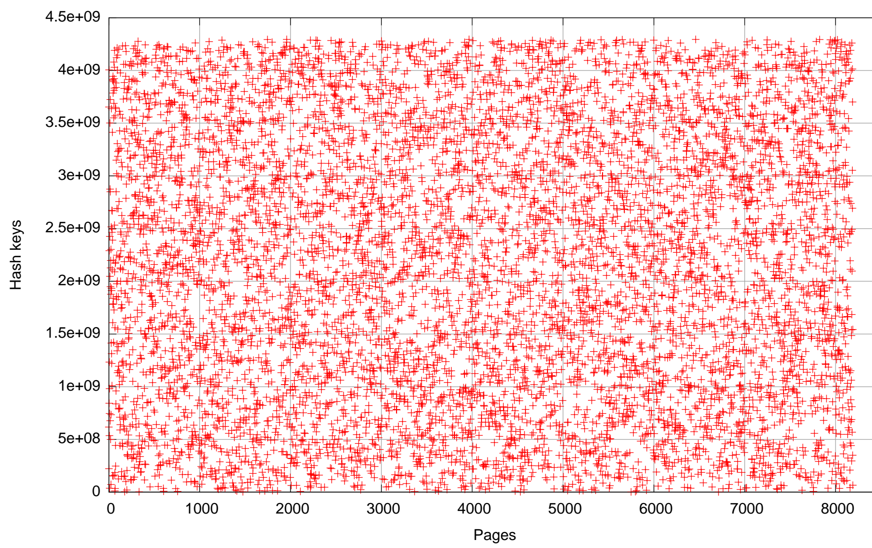
Before describing any data structures we give our requirements for choosing a data structure:

- The data structure, as with the hash functions, is intended to be implemented in the kernel and thus it should preferably be architecture independent.
- The data structure should also perform as good as possible, both in terms of memory usage and processor cycles.

[2]Inputs larger than 32MB produce graphs not well presented on paper
[3]A total of 8192 pages with the 32MB input



(a) 32 bit Bob Jenkins hash



(b) Super Fast Hash

Figure 6.1: Distribution of hash values in the two functions.

6.2.1 Hash Tables

Hash tables are used in many different areas in computer science because of their fast lookup speed in constant time on average. Hash tables work by using a hash function[4] to convert a given value into a hash value, which is used as a key in the hash table.

An ideal property of hash functions is that they should, on a range of values, calculate the hash keys so that they are uniformly distributed into the hash table. This distribution is important to avoid *collision* between keys in the hash table, because then the lookup speed can in worst case become linear *O(n)*.

The two most used techniques to handle collisions are chaining and open addressing.

Chaining: Chaining is typically implemented by using a linked list in the entries of the hash table, so when collisions occur the values are appended to the linked list. When using linked lists to handle collisions, the performance of the hash table will degrade to linear as the buckets gets filled.

The linked list should be double linked to ensure that delete operations on the list have a complexity of *O(1)*. If it was a single linked list, it would be necessary to traverse the list once more to find the prior element in the list.

When making an analysis of a hash table T, which uses chaining to resolve collisions, we have to look at the *load factor* α. This is n/m, where n is the set of stored elements in T and m is the number of buckets in T. The worst case performance is when all keys hash to the same bucket and we get one long linked list.

The average performance is based on the probability of keys hashing to the same bucket in the table. So if we make the assumption that a given key is equally likely to hash to a given bucket, then the length of a list in a given bucket $T[j]$ is n_j, where $j = 0, 1, \ldots, m-1$. This means that n will be $n = n_0 + n_1 + \cdots + n_{m-1}$.We assume that the time it takes to hash a value is $O(1)$ and to search for a key is linear depending on the length of the list. With these assumptions the average time on both successful and unsuccessful searches is $\Theta(1+\alpha)$. [9, p. 226-227]

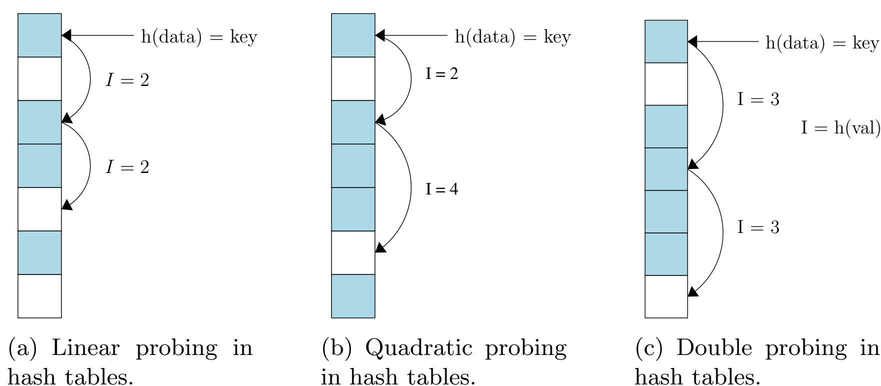
Furthermore if the number of buckets m in the hash table is at least proportional to the number of keys n in the table, then $n = O(m)$ and $\alpha = n/m = O(m)/m = O(1)$ with the assumption of uniform distribution.

Open addressing: Open addressing handles collisions in hash tables by probing or searching the hash table until an empty slot is found. There are three well known probing techniques:

- **Linear probing:** Linear probing is performed by having the same interval I between probes, which is pictured in Figure 6.2(a). It is simple to implement, but it has problems with the values clustering together.

[4] We point out that the hash function found in the previous section is not suited to index values into a hash table. A simpler function is needed.

- **Quadratic probing:** Quadratic probing uses an quadratic function $f(x) = ax^2 + bx + c$. This form of probing is illustrated in Figure 6.2(b).
- **Double hashing:** Double hashing uses another hash function to calculate the interval I to use when probing the hash table. I is used throughout a search and is first recalculated when inserting a new key (See Figure 6.2(c)).



(a) Linear probing in hash tables.

(b) Quadratic probing in hash tables.

(c) Double probing in hash tables.

Figure 6.2: Open addressing in hash tables (h() is a hash function)

As with chaining $\alpha = n/m$ is the load factor of the hash table, but with open addressing there is only one element in each bucket. This gives us that $n \leq m$. If we assume that the hash values are uniformly distributed and that any possible probe sequence is equally likely to occur. Then it can be shown that the number of probes in a unsuccessful search is at most $1/(1-\alpha)$ [9, p. 241-242].

When inserting new values into a hash table with open addressing it will also require at most $1/(1-\alpha)$ probes on average. When probing the hash table in a successful search, with a load factor $\alpha < 1$ then the expected number of probes is at most:

$$\frac{1}{\alpha} \ln \left(\frac{1}{1-\alpha} \right) \tag{6.1}$$

This is again under the assumption of uniform distribution and assuming that each value in the hash table is equally likely to be the value for which the search is conducted [9, p. 243].

6.2.2 Hash Array Mapped Trie (HAMT)

To avoid collisions all together a HAMT data structure can be used. The HAMT data structure was introduced in [2] and the rest of this subsection is based upon it.

It uses a combination of a trie[14] tree structure, arrays and (re-)hashing. A HAMT gives hash table properties, this being O(1) on lookup and insert. Furthermore it also provides a no collision guarantee and the data structure is dynamically

resized to the current data set size. This incurs some overhead, but it has been shown to perform at near hash table performance.

The data structure consist of two kinds of tables, sub-hash tables and a root hash table. We will refer to both kinds of tables as a table. A table consists of a number of entries, which can be a key/value pair or a map/base pair. A base is a pointer to a child sub-hash table, in which each key will share a common prefix. A map is a bitmap of n bits[5] that holds the information about which of the entries in the sub-hash table are in use. A tables first entry is indexed as zero, likewise the first bit in a bitmap is indexed as zero.

The sub-hash tables are not of a fixed size, as they are resized according to the number of entries. The number of entries is also reflected by the bits in the parents bitmap. Specifically the number of bits set is equal to the number entries in the sub-hash table.

We will now present a search example step by step, with a given lookup key. In the example the hash values computed are 8 bit in size. The root hash table has eight entries and each bitmap has 8 bits. This means that we use 3 bits from the key at a time, since 3 bits can be arranged in 2^3 different ways, which is the size of the bitmap. Below is an enumerated list of the steps in the example, the numbers corresponds to the circled numbers in Figure 6.3.

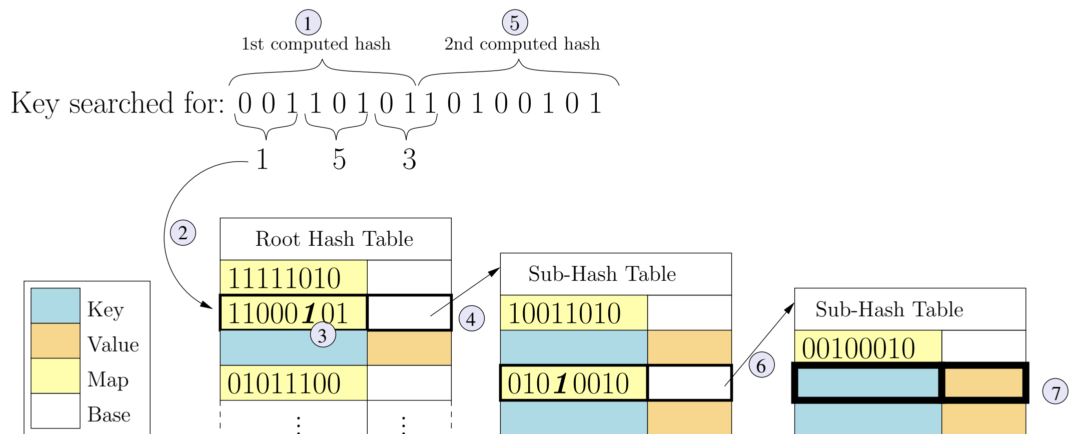


Figure 6.3: HAMT with bitmap exemplified

1. An 8 bit hash is computed from the lookup key and used as key.

2. The first 3 bits of the computed key, which in base ten is one, is used to index into the root hash table, which is the second entry in the table, because it starts at index zero as mentioned above. The entry contains a map/base pair.

   If there had been a key/value pair and the key had not matched the searched key, then the HAMT had not contained the key searched for. The same applies if the entry had been empty.

[5] This is typically 32 bit, but it depends on the processor architecture.

3. The next 3 bits, which in base ten is five, are used to index into the bitmap. The bit we get is the sixth bit with value one (emphasized in the figure) again because we count from zero. This indicates that there is an entry in the sub-hash table, which can either be a key/value or a map/base pair.

   However if this bit had been zero, then there was no entry in the sub-hash table with a longer prefix of the searched key, which had implied that the HAMT had not contained the key searched for.

4. The ones before the sixth bit are counted and gives a result of two. This is then the index into the sub-hash table, pointed to by base. This gives us the third entry, which contains a map/base pair.

5. There is now only two bits left in the key and we need three to index into the bitmap, this is solved by rehashing. This adds 8 bits more to the key.

6. The next 3 bits, which in base ten is three, can now be used to index into the map. We read the fourth bit, containing a one bit. We count the ones before the fourth bit, which gives us a result of one. Now we follow the base pointer and go to index one, the second entry, which contains a key/value pair.

7. We do a comparison between the key stored and the one we are currently searching for. If they are identical, the key/value pair is found, else it is not in the HAMT since there exist no longer prefix of the searched key.

Normally the root hash table is set to 32 entries initially and resized dynamically as the HAMT grows in size. If there is prior knowledge of the number of keys the HAMT is going to contain, then it can be optimized with a preset size of the root hash table.

We now know how searching is performed in a HAMT, another function is the insert operation, which is more complex than the search operation. As we have prior knowledge of the data set size, we can create the root table at the needed size and avoid resizing. Therefor we omit the more complex details of insert and search with root resizing and refer the reader to [2].

To insert data into a HAMT, a search on the key to be inserted is first performed. This can have two outcomes:

- An empty bucket is discovered. If this is in the root hash table, we simply insert the key/value pair into the empty bucket. However if the empty bucket is found in a sub-hash table, a one bit is placed in the appropriated place in the map bitmap. The key/value pair is inserted into the sub-hash table.

- The search discovers a key/value pair. A new sub-hash table has to be created and the existing key/value pair replaced with a bitmap and base pointer to the created sub-hash table. The existing key/value pair and the inserted pair is then inserted into the new sub-hash table if they can be uniquely separated, else yet another sub-hash table is created an so forth.

When removing or adding an entry from a sub-hash table, the entries in the original sub-hash table are actually copied to a new one.

Some optimizations are used to improve performance: A free lists of sub-hash tables of different sizes that are not currently in use are kept and de-fragmented. The de-fragmentation means memory is freed to a free memory pool, so we for example do not get too many sub-hash tables with two entries, in the free list. Another optimization is the use of lazy root hash table re-sizing, which is not applicable to our use, since we can estimate the size of the hash root table from the start, which is a further optimization.

The space used by a HAMT can be estimated, e.g. with an average of four colliding keys in the same entry in the root hash table. The used can be estimated as : [2]

$$N + \frac{1}{4}N(1 + 0.188 - 0.073) \approx 1.279N \tag{6.2}$$

where N is the number of key/value pairs. The constants are based on probability theory about how the keys will be distributed in the tables. Let r indicate the average number of keys that collide in the same root hash table bucket. With $r = 4$ the space used is estimated to $1.28N$ and with $r = 1$ the space used is estimated to $1.65N$. The lower r is, the better performance can be expected. This is because a lower number of sub-hash tables has to be traversed, however more space is used. The same applies the other way, if $r = 8$ only $1.14N$ is used, but the performance is lower, since more sub-hash tables has to be traversed. [2]

6.3 Comparison of Data Structures

We will now evaluate the data structures described in the previous section. The key observation to choosing a data structure is that we have prior knowledge about the size of our data set (number of page frames). Furthermore we remind the reader that doing dynamic allocation within the kernel is expensive in terms of operations on data structures and the memory they use.

The performance of the hash tables is dependent on the load factor α, which reflects the number of buckets and elements in the table.

If we are using chaining to resolve collisions in hash tables, then we will have to use double linked lists. These linked lists will incur the overhead of two additional pointers, which each takes up one word in memory, per element in the list. Furthermore the linked list will require dynamic memory allocation.

If we use open addressing to handle collisions we do not have the memory overhead of pointers. But to use open addressing optimally we have to find a "fair" load factor relative to the size of the table.

To find a fair estimate of this load factor, we now make some estimations on the maximum memory size of 4GB (1048576 pages). The formulas introduced in Section 6.2.1 on page 52 will now be used for doing calculations on the hash table data structure. In the following example we have chosen the number of buckets to be the number of pages plus 10%.

$$\alpha = \frac{1048576}{1048576 + \lfloor 0.10 \times 1048576 \rfloor} \approx \frac{1048576}{1153433} \approx 0.90 \quad (6.3)$$

$$\frac{1}{0.90} \ln\left(\frac{1}{1 - 0.90}\right) \approx 2.63 \quad (6.4)$$

If we have a load factor of roughly 0.90 as in Equation 6.3, it yields approximately 2.63 probes in a successful search. An insert on a table with the same load factor will require $1/(1 - 0.90) \approx 11.0$ probes. Furthermore we present additional results in Table 6.3 to show that an overhead of 10% additional buckets seems reasonable. This is however something that can be experimented with later and easy to fine tune after the implementation is complete.

Overhead	Load	Search	Insert	Buckets
2%	0.98	4.01	51.0	1069547
6%	0.94	3.04	17.6	1111490
8%	0.92	2.81	13.5	1132462
9%	0.91	2.71	12.1	1142947
10%	0.90	2.63	11.0	1153433
11%	0.90	2.56	10.0	1163919
12%	0.89	2.50	9.33	1174405
16%	0.86	2.29	7.24	1216348
20%	0.83	2.15	5.99	1258291

Table 6.3: Memory overheads with different load factors in open addressing.

The HAMT data structure can not be directly compared with the hash table performance, since the approach is quite different. However the HAMT does not fulfill the requirements for the data structures as well as the hash tables does.

The three main reasons are as follows:

1. We can not predict the memory usage perfectly, an implementation in the kernel space or in VMM might require a number of allocations. Which as mention above is expensive.

2. The complexity, meaning how easy it is to validate that there are no errors in the implementation, are higher than with a hash table. This is of high relevance to us since the data structure might be used in the VMM.

3. The last reason is that the search can be expected to perform quite well, possibly better than a hash table. This is also what they showed in [2]. But the insert can be expected to have a much higher overhead than in an open addressing solution. Because on an insert, if we want a memory overhead of around $10\% - 15\%$, there will be an average of eight keys per entry in the root hash table. Hence a sub-hash table is in use, which means that on each insert, the entire sub-hash table has to be copied, which again leads to a performance overhead. And this performance overhead on insert is larger than that of hash table. This concurs with the performance results in [2].

These performance overheads can however be reduced, but will require a root hash table with as many entries or more, as there are keys to be inserted. This is not coherent with the idea behind the HAMT data structure, as the quality of the tree structure is not used to its fullest.

There is a significant memory overhead in having a root hash table with a number of entries equal to the number of keys. With an average of one key per root hash entry, it will incur a calculated memory usage of $1.65N$ (65% overhead) and by empirical testing $1.63N$ (63% overhead) both results can be found in [2], where N is the number of inserted keys.

We conclude that the most space efficient data structure is the hash table using open addressing under the assumption of uniform distribution. Furthermore the HAMT data structure is more complex to implement than a hash table. This means that the best suited data structure for our requirements is hash tables with open addressing and to use the SFH to hash the memory page into 32 bit values.

Chapter 7

Architecture

In this chapter we introduce different architectures for the implementation. We do this in a top down manner, where we start by introducing our design goals and the architectures at the component level. After that we move the focus to the algorithm level, where we describe how the different steps are performed. Finally we evaluate the designs in accordance to our design goals.

7.1 Design Goals

Before getting into the actual designs, we present the design goals ordered top-down by importance.

Scalability: Our aim is to improve Xens memory usage, so that we are able to scale to a higher number of VMs than what can normally be achieved with the same amount of machine memory.

Performance: The implementation should have as low a performance overhead as possible.

Security: The isolation provided by the VMS scheme should not be compromised.

Reusability: The Potemkin framework should provide us with some of the mechanisms needed to realize our implementation.

Simplicity: Care should be taken not to unnecessarily make the VMM more complex than necessary.

With these goals in mind, we now present the components which are shared throughout the different design architectures.

7.2 Components

To make the design of the architecture presentable we split it into three components:

Page Hashing (PH): This component creates a hash value of each memory page in the address space of the component.

COW Sharing (CS): This component handles write faults and shares pages using COW. This component depends on having access to the page tables.

Reference Manager (RM): This is the most complex component and therefore it has two subcomponents. The RM keeps track of the reference count for a shared page and handles tearing down shared pages based on this reference count.

The two subcomponents of the RM are:

- **Hash Indexing (HI):** Maintains a hash table with open addressing 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.
- **Page Comparison (PC):** Compares two memory pages to see if they are bitwise identical.

Basically the PH component produces hash values of pages and delivers them to the RM component. The RM component then uses the subcomponents to determine whether there are any duplicate pages. If this is the case then the RM component tells the CS components to update the page tables of the VMs sharing the pages. To ensure the VMM will allow multiple VMs to access the same machine page frame, we need to introduce a new *shared read-only page* type to the types described in Subsection 2.2.3 on page 25. When a write is attempted to a shared page, the CS component handles the fault.

Alternatively the RM component could be split into three full components giving us a total of five components. These components rely on communication with a low overhead because they have a high degree of coupling. Therefore placing them in separate levels (e.g. one in the VMM and one in a VM) may incur a significant performance overhead, because this will require additional switches. Thus we will in the following regard them as one component. Before getting into the concrete designs, there are some overall design considerations that should be discussed. We will address these in the following sections and return to the actual designs in Section 7.5 on page 62.

7.3 Feasibility of Perfect Sharing

As explained before, when hashing memory pages there is a probability that two pages, which are not identical, will produce the same hash value (a collision). To deal with this situation we propose two different approaches: 1) deal with the collision using open addressing or 2) disregard the old value and only use the new value[1]. The latter option simply has the effect of not considering the pages for sharing. We will refer to the first option as "perfect sharing". This section will argue about whether it is feasible to do "perfect sharing" or not.

[1] It should be noted that we only address collisions caused by the hash functions that hash pages, not the hash function used when inserting into the hash table.

To be able to actually do "perfect sharing" we need to do additional modifications to the open addressing scheme, which we have chosen to use in our implementation. Either we can use a linked list to handle these special collisions (which costs an additional factor two for every colliding entry and the data structure will become more complex) or we allow the hash table to fill the colliding entry into the next available bucket (this will however affect the performance of a search, as it becomes as expensive as an insert). So this additional feature will come at a high cost.

As we shall see shortly the probability of these collisions occurring is so low that the best choice is just to ignore the collisions. It should be noted that this is just the probability for collisions between any pages. The probability that the collisions will happen with pages that are actually candidates for sharing should be even lower. We will use the maximum number of pages as the number of input pages i.e. $2^{20} = 1048574$ pages for 4GB.

To calculate the probability of collisions occurring, we perform a number of different calculations. One way is to use the birthday paradox as used in [48],[22]. We can calculate, using a 32 bit hash function, that we only need an input size of 65536 pages ($\sqrt{2^{32}} = 2^{16}$) to have a 50% probability of a collision.

Another way is, as Henson described in [23], to calculate the probability for encountering one or more collisions as:

$$1 - (1 - 2^{-b})^n = 1 - (1 - 2^{-32})^{1048576} \tag{7.1}$$

$$\approx 0.0244\% \tag{7.2}$$

where b is the number of bits in the produced hash value and n is the number of input pages. Equation 7.1 shows that the probability of this is 0.0244%. Ít should be noted that this is the theoretical probability. If we return to the hash functions evaluated in Section 6.1.5 on page 49, we can calculate the frequency of collisions on the SFH function as

$$\frac{collisions}{pages} = \frac{278}{1048576} \tag{7.3}$$

$$\approx 0.0265\% \tag{7.4}$$

and as it was random data used in the experiments, so we are positive that the collisions occurring in the tests were actually caused by collisions, not identical input data. The fact that the actual frequency of collisions is almost equal to the theoretical probability, confirms the conclusion from the other section that the hash function have a low collision rate. Keeping in mind that we have a low collision hash function and because we know that the probability of hash collision is small, we can conclude that there is no need to do "perfect hashing" as the overhead is too high and not much is gained by doing it.

7.4 Flushing vs. Continuous Hashing

In this section we consider two different approaches to page hashing.

Flushing	Continuous
Coarse grained hashing control	Fine grained hashing control
	Possibly unnecessary page comparisons
	Extra reverse lookup mapping

Table 7.1: Summary of the advantages and disadvantages of the two approaches

The first approach is the more simple of the two: When the VMM detects that the system is idle it schedules page hashing[2]. The hashing is done over a large region (possibly all of it) of the machines address space. All pages that can be shared are identified and sharing is established. When this has been completed the hash values are discarded and the next time page hashing is done it starts from fresh. We shall refer to this approach as “Flushing”.

The second approach, which we will refer to as “Continuous Hashing”, keeps the content index up-to-date from one round of page hashing to another[3]. To be able to keep the content index up-to-date we need to be able to delete entries from it. To do this we need a reverse lookup mechanism, which implies a data structure of at least one word (a pointer to a given content index entry for every machine page frame).

So this approach is more costly in terms of updating the content index and the memory required for the data structure. Furthermore there is a higher probability that keys are outdated, which implies that there is a higher probability that we are going to compare pages that are not identical.

Table 7.1 summarizes the pros and cons of the two approaches. As the two approaches both have desirable features and there is no compromise, we deem that the matter of which approach is preferable is best determined in a series of experiments when the implementation has been constructed.

Having presented the overall decisions we now move on to presenting the architectures of the different designs.

7.5 Overall Designs

The following subsections will outline our different design ideas for the architecture of the implementation. 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 7.6 on page 68. This means that any optimization are left out in this description.

We propose three different designs. The first design, like VMwares design, is completely transparent to the guest OS. The second requires few modifications to the OS and the third attempts to move functionality away from the VMM and into the privileged VM Domain-0.

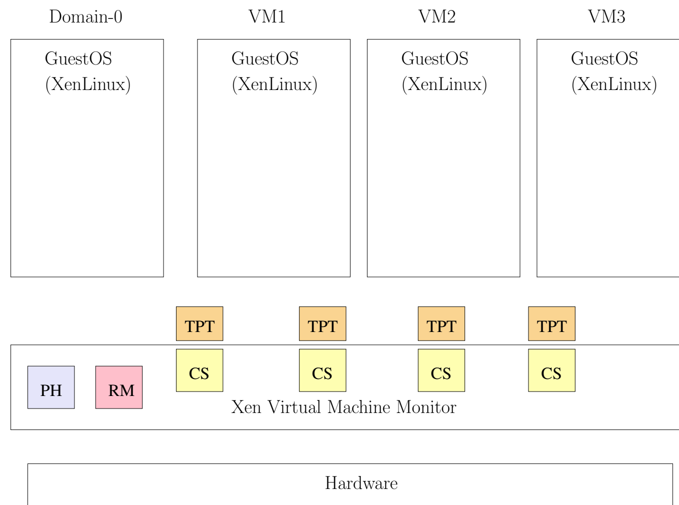


Figure 7.1: Transparent Hypervisor Level design

7.5.1 Transparent Hypervisor Level

In this design all the components are, as pictured in Figure 7.1, placed in the VMM. The design is. To achieve this transparent solution, we introduce a level of abstraction between the pages that the VM is allocated and actual machine memory. Now the Virtual Address Space (VAS) of a process points into what we will refer to as Physical Address Space (PAS). Then we introduce a *Translation Page Table* (TPT)[4] that translates from PAS into the Machine Address Space (MAS) as illustrated in Figure 7.2 on the following page.

Now we have introduced the level of abstraction that we wanted. However we need to alter the memory management to take advantage of the changes. There are two options 1) change the OS to use a two level lookup scheme (tedious and expensive) or 2) use SPTs (as explained in Section 2.2.4 on page 26). The first option is not usable in this design, as it requires modifications to the guest OS.

The TPT can be used to give the VM the illusion that it has a contiguous address space. This TPT can also be used to share memory pages between VMs without them knowing about it. This is done by changing the translation in the TPT, so one or more TPT entries point to the same machine address. The CS component in the VMM, can now handle the sharing of pages, by updating the TPTs. Changes to the TPTs must be propagated 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

[2]Further work will be either to describe a policy on when a system is idle or find an appropriate time to schedule hashing of pages.

[3]This does not imply that all keys are updated on writes to their corresponding pages. It simply implies that there is at most one key for one page at all times.

[4]The Xen P2M mapping introduced in subsection 2.2.1 on page 22, does already provide this abstraction. Unfortunately it is placed inside the guest VM, so it can not be used in this design as it requires modification to the OS.

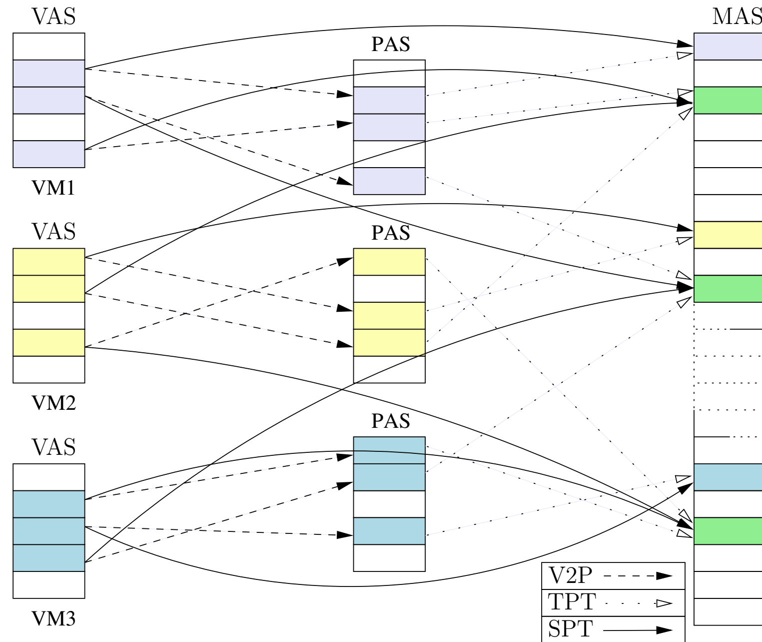


Figure 7.2:

VMs, which are only able to access their own address space. 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 to achieve and typically comes at the expense of performance [15]. Furthermore placing the PH component in the guest VM could possibly introduce a new Denial-of-Service (DoS) 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. This approach 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, it is expensive to do this. The latter argument also applies to placing the RM component in Domain-0.

The advantages of this design are that it can be done completely transparent to the VMs, hence no porting 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. Also the decision of when to schedule the hashing of pages, may introduce some complexity to ensure fairness between VMs. Additionally the TPTs and SPTs introduce both memory usage overhead and performance overhead in keeping the tables updated. Plus a separate policy scheme is needed if a given VM does not wish to participate in the sharing.

Furthermore this design should be directly compatible with Potemkin framework, which should save us some implementation effort as that framework already has a number of the features we require.

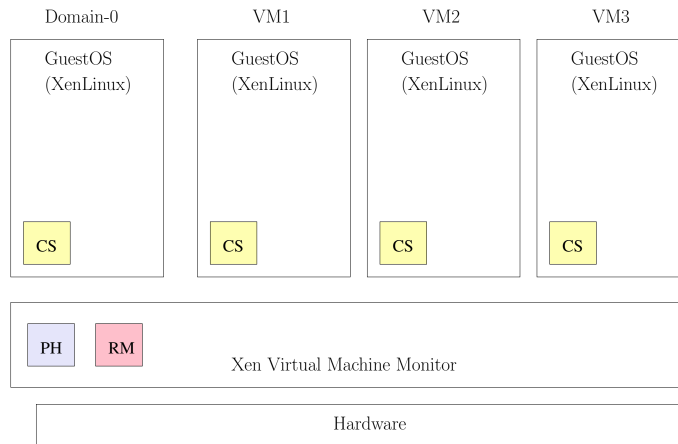


Figure 7.3: Hypervisor Level design

7.5.2 Hypervisor Level

This design is a modification of the transparent hypervisor level design, where all the components, except for the CS component, are placed in the VMM. Placing the CS component in the guest VMs will allow us to avoid the overheads of using TPTs and SPTs. The design is as illustrated in Figure 7.3.

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 updates to the page tables are validated by the VMM, thus making us able to check that the CS component actually updates the page table to a valid address.

In order to let the guest VM handle write faults to shared pages, the VMM needs to trap the write fault, detect that it was a write operation to a shared page and notify the guest VM of this. This kind of communication is normally done through events, therefore we need to introduce a new event in this design.

We also need to introduce a new type of hyper call to make sure that before the sharing is made final, there is a page comparison to ensure that the pages have not changed since the initial comparison. This is necessary because a VM may make changes to its memory while processing the event mentioned above.

The advantage of this design is that there is minimal modification to the guest OSs, only the CS component. The disadvantages to this design, are the same as in the transparent hypervisor level design except for the TPT and SPT overheads.

7.5.3 Supervisor Level

In this design the components are divided between the privileged VM Domain-0 and the guest VMs. This design is as illustrated in Figure 7.4 on the next page. The VMs contain the CS components and the placement of this has the same reasons as in the hypervisor level design. Each VM also contains a PH component, which means that it has the responsibility of hashing its address space. The RM component is placed in the trusted and privileged Domain-0. It receives hash values and machine addresses of the hashed pages generated by the PH component in the different VMs.

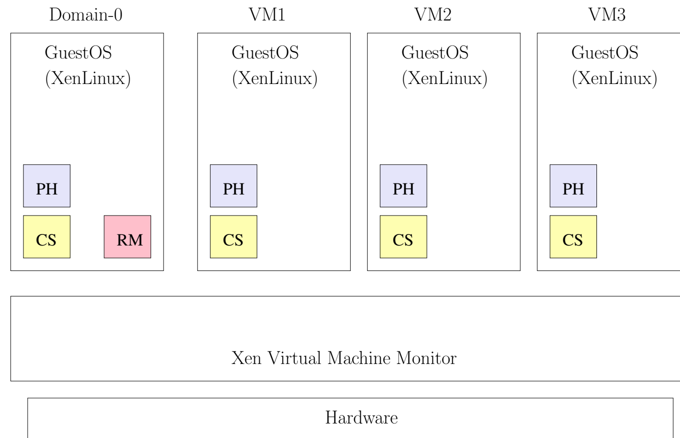


Figure 7.4: Supervisor Level design

The transferring of the values is performed by the use of shared memory.

The advantages of this design is that only a minimal complexity is added to the VMM. There are some problems with this design, which we will address in the following subsection.

7.5.3.1 Supervisor Level Design Problematic

The approach of this section is as follows: First we will go through an algorithm and then get into the problems with this design. Many of the problems are serious and hard to avoid.

We will now describe step by step how the actual sharing is created, which is shown in Figure 7.5. The numbers on the arrows correspond to numbers in the enumerated list below.

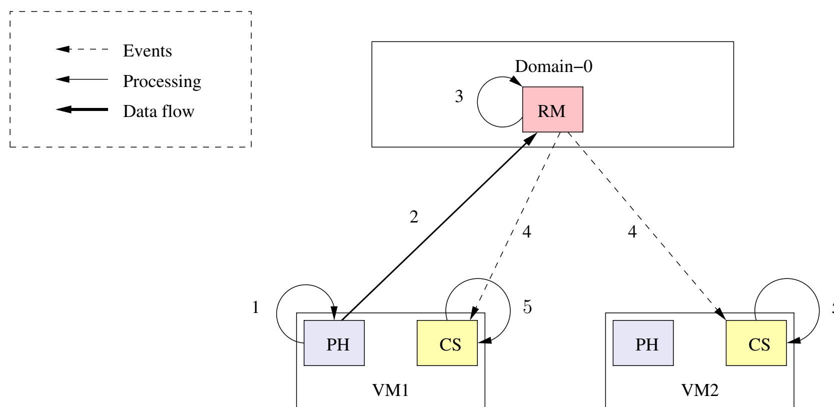


Figure 7.5: Supervisor Level Sharing Algorithm

1. **Hashing:** VM1's PH component hashes a page.

2. **Notify RM:** VM1 delivers the hash key and the machine address of the hashed page to the RM component. This delivery can be done asynchronous.
3. **Content Index Lookup:** The RM checks the content index for identical keys, indicating whether the page has been seen before. The lookup can have four different outcomes:
   (a) **Key not seen before:** Add key to content index and terminate.
   (b) **New key for old address:** Delete old entry, add new key and terminate.
   (c) **Key seen before, same address:** Terminate.
   (d) **Key seen before, different address:** Possibly an opportunity to share. Request page comparison.
4. **Page Comparison:** If the pages are not identical, then we have a hash collision. However if the pages are actually identical, then proceed and update the reference count for the page. Also the VMs are sent an asynchronous event that they have a page, which is to be shared. There is however a demand that the VMs participating in sharing a given page, must not be run until they receive that event.
5. **Page Table Update:** The VMs sharing the pages update their page tables to point to the same page and mark this as read-only. This has to be done as the first thing a guest OS does before any pages might be changed.

7.5.3.2 Problems

The main problems with the design are: Two problems in the sharing algorithm and one problem in handling write faults. These will now be explained:

- Step two in the sharing algorithm has a fundamental problem, that we can not trust the hashes and machine addresses that the guest VM sends to the RM, as described earlier. The machine addresses can be validated to be inside the sending VM address space, but there is a overhead in doing so.
- The problem with step five in the sharing algorithm is that we can not be sure that the guest OS does not modify any pages, before the page tables are updated to contain the shared pages. An example of this: A page is to be shared, but the guest OS kernel uses this for some internal kernel structures, so it might be modified before we get to update the page table to point to the new shared page. The changes the kernel made to the old page are now lost as the page table points to the shared page.

  Another problem in this step is that Domain-0 can not check that the VM actually did update its page table.

  These kind of flaws are hard, if not impossible, to avoid in this design.
- The problem in handling write faults is that the VM can tell the RM that it has removed access to a shared page, while actually keeping the access. The RM cannot verify this.

As can be seen this design requires placing trust in the VMs, which may compromise the isolation of the system. It should be noted that while we have not made the greatest effort to solve these problems, we firmly believe that they are unavoidable, at least when performance is a criterion. Thus this design is discarded.

7.6 Advanced Details

Having introduced the architectures, we will now describe the details of the designs. To realize our designs we need at least two algorithms, one to share pages and one to remove the sharing of pages. We remind the reader that events and data-flows are asynchronous and hyper calls are synchronous. The asynchronous steps can be batched, meaning e.g. that a set of hashed values can be sent together. We start by presenting the hypervisor level design as this contains the most interesting details.

7.6.1 Algorithms in the Hypervisor Level Design

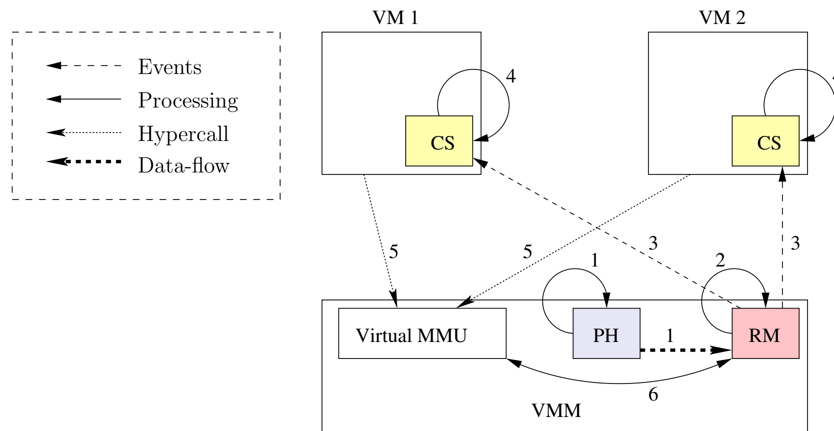


Figure 7.6: Hypervisor Level Sharing Algorithm

We will now give a description of how a sharing is obtained, step by step. To do this we illustrate the algorithm in Figure 7.6. The numbers in the figure corresponds to the steps of the algorithm.

1. **Hashing:** PH component hashes a page and afterwards the hash value and machine address is delivered to the RM component.

2. **Content Index Lookup:** The RM checks the received values for identical values in the content index, indicating whether the page has been seen before. This lookup can have four different outcomes:

   (a) **Key not seen before:** Add key to content index and terminate.

   (b) **New key for old address:** Delete old entry, add new key and terminate (Only if using continuous hashing as described in Subsection 7.4).

   (c) **Key seen before, same address:** Terminate.

(d) **Key seen before, different address:** Possibly an opportunity to share. Request page comparison.

3. **Page Comparison:** If the pages are actually identical, then send an event to the VM that they have to share the page. This can be done asynchronous as long as the involved VMs are not run. If they are not identical then we have a hash collision.

4. **Page Table update:** The VMs sharing the pages update their page tables to point to the same page and mark it as read-only.

5. **Commitment:** The Updates to the page tables are validated as they always are by the VM. As the pages may have changed since step 3, we need to ensure that the pages are still identical. Therefore we do a page comparison.

6. **Reference Count Update:** In order to increment the reference count for the shared page two things must be validated. The updated page table entry points to: 1) VMs own address space or 2) a shared page. This must be done for each PT update performed by the VMs.

The algorithm does not introduce any hyper calls and only an asynchronous event is introduced in step 3. So there are no new performance overheads in the form of hyper switches.

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

1. The VMM receives a write fault from the MMU, it translate this into an event and sends it to the corresponding VM.

2. The VM receives the event of a shared read-only write fault and it makes a copy of the shared page by updating its current page table.

3. The page table 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 introduced a new event and the VMM needs to check for this new type when it validates page table updates. Now that we have presented the hypervisor level design we will now continue with the next design.

7.6.2 Algorithms in the Transparent Hypervisor Level Design

This design differs from the hypervisor level design, since the CS component is placed in the VMM and there is an abstraction provided by TPTs and SPTs. This gives the advantage that the CS component just updates the TPT with the new mappings. Thus the physical addresses in the different TPTs maps to the same machine address. This simplifies the algorithms, because th VMs are never involved. As with the other design we now present the algorithm for sharing pages:

The two first step of this algorithm are the same as in the previous section.

3. **Page Comparisons:** If the pages are actually identical, then proceed to step 4. If they are not identical then we have a hash collision.

4. **Atomic Phase begins:** The following operations are critical. Therefore we need to ensure that there are no switches from the VMM to a VM during the following steps, as this may result in incorrect behavior.

5. **Mappings are updated:** The VMM changes the mappings in the TPTs.

6. **Shadow Page Table flushed:** The SPTs are invalid as the mappings they contain have been changed. Therefore we need to flush the SPT cache.

7. **Reference count update:** The VMM changes the reference count for the shared pages.

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

The SPTs will be regenerated when normal operation resumes, therefore in this design it would be preferable to batch the operations, as regenerating the SPTs is expensive if performed each time a single page is shared. Furthermore doing page hashing by flushing in this design (as described in 7.4 on page 61) should incur the lowest performance overhead.

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.

2. We copy the content of the page to a free page and update the TPT entry to point to this new page. Finally the current SPT is flushed to ensure that the changes are propagated[5].

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

7.7 Evaluating the Designs

Having presented the different designs and pointed out their details, we now return to the design goals. As the design goals reflect our wishes for the implementation, the design that adheres best to the design goals should be the best design for us. Lets first make it clear that the supervisor level design is discarded due to its flaws and to work around them will create an unacceptable overhead.

Scalability: The design goal of scalability in terms of memory usage, can be achieved in both remaining designs. But the transparent hypervisor level design, has significantly higher memory overhead. This is due to the use of a complete

[5]Further work will investigate if we can avoid flushing the SPT.

one-to-one mapping from each VM physical-to-machine address. It might be possible to compress this mapping to contain only the pages that are shared. However there is still a memory overhead in just using shadow pages, which can not be avoided unless a really small amount of shadow page table cache is kept. This will then lead to a high performance overhead in generating new shadow page tables on each context switch.

Performance: As for the design goal of low performance overhead, we can give an estimate of how expensive an architecture will be, by counting the different world and hyper switches when establishing the sharing of pages. For the transparent hypervisor level design it is easy to count since there are none, because it is all done in the VMM. The hypervisor level design adds one hyper switch. The hyper switch happens when the guest OS updates its page table to contain a new machine address of a shared page. The amount of switches in handling a write fault (including the write faults) are in the hypervisor level design two and in the transparent hypervisor level design one.

Based on the hyper switch count, the goal of low performance overhead can be achieved by both designs. But the transparent hypervisor level design suffers from a performance overhead since it has to keep the shadow page table updated.

Security: The design goal about security can be archived by both designs. But easiest with the transparent hypervisor level design. Since an extra page comparison has to be made in the hypervisor level design to make sure of complete isolation.

Reusability: The design goal of reusability in terms of what the Potemkin framework can provide will best be achieved in the transparent hypervisor level design. This is because the Potemkin framework has already implemented many of the features (such as SPTs, TPTs and COW sharing etc.) we need. Thus we will be able to save some implementation efforts.

Simplicity: Lastly the design goal of simplicity, is hard to predict at this point. However the PH component in the two designs can be made quite simple, with small amount of code added to the VMM. It can also be made so it is cleanly separated from the rest of the VMM, in terms of not messing with the existent code. The same applies to the RM component.

The CS component and the job of decrementing the reference count can also be done cleanly in the transparent hypervisor level design, due to the use of TPT and shadow page table. However in the hypervisor level design the CS component, will have to intervene with the guest OS.

Another issue with the hypervisor level design is its need to intervene with the validation of a page table updates. If not done with care, it will dirty the VMM.

To sum up, the simplicity design goal can be accomplished by both designs, but most cleanly by use of the transparent hypervisor level design.

Having gone through the design goals, we can now conclude that the design which fulfills the scalability and performance design goals best is the hypervisor level design. The security, reusability and simplicity design goals are best or easiest fulfilled by the transparent hypervisor level design. This leads us to conclude that if the hypervisor level design can be implemented in a reasonable fashion, then this is the top choice. However if this is not possible the transparent hypervisor level design is the choice.

Chapter 8

Further Work

This part of the report first addressed the techniques needed to implement page sharing between VMs. In particular we found a fast hash function with low collision rate and a uniform distribution called SuperFastHash. Furthermore we found that the most efficient data structure for storing produced hash values is a hash table using open addressing.

In Chapter 7 we introduced our designs. First we divided the functionality needed to implement sharing of pages into components. Then we explained the architecture of the designs using these components and explained how they interact. Finally we analyzed the designs according to the design goals introduced in that chapter. We point out that the designs are not yet completed, as the low level details are still missing from the report.

To conclude this part of the report, we now explain what we expect to accomplish in the last part of the project.

As we noticed in Chapter 3, the initial experiments on memory sharing have not been completed to our satisfaction and these will be completed.

The design part is, as mentioned above, not complete and the low level details should be specified as we become more comfortable with the Xen and Potemkin source code.

A major part of the upcoming work is to actually build the implementation. Our approach will presumably be to build a prototype that uses as little policies as possible. Then when the framework is in place, we can move on to fine-tune the details on e.g. when to schedule hashing of pages and if flushing the content index is optimal.

Another important issue is to find an appropriate time to schedule the hashing of pages. Either we need to detect when the system is idle or determine if we can come up with a specific event in the system, that indicates that it is a good time to hash pages.

Finally when the implementation is done we can start our experiments. We expect to do the following experiments:

Fine-tuning: The first set of experiments will be made to fine-tune the implementation to determine the factors that may be variable.

Stress Testing: We will perform a series of application level tests to determine how

well the framework performs.

Design Testing: If possible we would like to implement both designs, mainly because of two things: 1) We expect the hypervisor design to have the lowest overhead in terms of memory usage. 2) It is quite possible that the only kernel, we will implement this solution for, is the Linux kernel. Therefore we regard the transparent solution as a general solution to fall back on. If used for this, it would be great to be able to compare the performance of the two designs.

Phone booth Testing: We will also test how many VMs we can fit into one physical host.

Coexistence Test: If our framework is compatible (and we hope so) with the Potemkin code, then we can investigate whether there is anything to gain by combining flash cloning with content-based page sharing.

Sharing Test: Finally the perhaps most important part of the experiments is to determine how high a percentage of memory can be shared. Furthermore we would like to investigate how sharing behaves on specific workloads and determine if the sharing rate decreases as a function of time.

With this work list we conclude the report.

Bibliography

[1] J. D. Bagley, E. R. Floto, S. C. Hsieh, and V. Watson. Sharing data and services in a virtual machine system. In *SOSP '75: Proceedings of the fifth ACM symposium on Operating systems principles*, pages 82–88. ACM Press, 1975.

[2] Phil Bagwell. Ideal hash trees. Technical report, Institute of Core Computing Science, Swiss Institute of Technology Lausanne, Es Grands Champs, 1195-Dully, Switzerland, 2001. http://lampwww.epfl.ch/papers/idealhashtrees.pdf.

[3] 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.

[4] 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.

[5] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, 1997.

[6] Matthew Chapman and Gernot Heiser. Implementing Transparent Shared Memory on Clusters Using Virtual Machines. In *Proceedings of the 10th USENIX Workshop on Hot Topics in Operating Systems (HotOS-X), Santa Fe, NM*, 2005.

[7] 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.

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

[9] 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.

[10] Robert J. Creasy. The origin of the vm/370 time-sharing system. *IBM Journal of Research and Development*, 25(5):483–490, 1981.

[11] Donald E. Eastlake and Paul E. Jones. Us secure hash algorithm 1 (sha1). http://www.ietf.org/rfc/rfc3174.txt?number=3174.

[12] Glenn Fowler, Phong Vo, and Landon Curt Noll. Fvn hash. http://www.isthe.com/chongo/tech/comp/fnv/.

[13] 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.

[14] Edward Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, 1960. ISSN 0001-0782.

[15] 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.

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

[17] Robert Philip Goldberg. Architecture of virtual machines. In *Proceedings of the workshop on virtual computer systems*, pages 74–112, 1973.

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

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

[20] Diwaker Gupta, Rob Gardner, and Ludmilla Cherkasova. Xenmon: Qos monitoring and performance profiling tool. Technical report, Hewlett-Packard Laboratories, 2005.

[21] 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.

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

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

[24] Alex Ho, Steven Hand, and Timothy L. Harris. Pdb: Pervasive debugging with xen. In *GRID*, pages 260–265, 2004.

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

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

[27] Vlastimil Klima. Finding md5 collisions - a toy for a notebook. Cryptology ePrint Archive, Report 2005/075, 2005. http://eprint.iacr.org/2005/075.

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

[29] 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.

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

[31] 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.

[32] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Willy Zwaenepoel. Diagnosing performance overheads in the xen virtual machine environment. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 13–23. ACM Press, New York, NY, USA, 2005. ISBN 1-59593-047-7.

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

[34] R. P. Parmelee, T. I. Peterson, C. C. Tillman, and D. J. Hatfield. Virtual storage and virtual machine concepts. *IBM Systems Journal*, 11(2):99–130, 1972.

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

[36] 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 *In Proc. of the 2005 Ottawa Linux Symposium*, pages 65–77, July 2005.

[37] Herbert Pötzl. Linux-vserver technology. http://linux-vserver.org/Linux-VServer-Paper, 2004.

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

[39] Ronald L. Rivest. The md5 message-digest algorithm (rfc 1321). http://www.ietf.org/rfc/rfc1321.txt?number=1321.

[40] 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.

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

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

[43] 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.

[44] Linux-VServer Team. Linux-vserver homepage. http://linux-vserver.org/.

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

[46] 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.

[47] P. N. Wahi. On sharing of pages in cp-67. In *Proceedings of the workshop on virtual computer systems*, pages 127–149, 1973.

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

[49] Andrew Warfield, Russ Ross, Keir Fraser, Christian Limpach, and Steven Hand. Parallax: Managing storage for a million machines. In *Proceedings of the 10th USENIX Workshop on Hot Topics in Operating Systems (HotOS-X), Santa Fe, NM*, June 2005.

[50] 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.

[51] Yuting Zhang, Azer Bestavros, Mina Guirguis, Ibrahim Matta, and Richard West. Friendly virtual machines: leveraging a feedback-control model for application adaptation. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 2–12. ACM Press, 2005.

Part III

Appendix

Appendix A

Hash Collisions

The hash collision experiment has been performed on 256MB (65536 pages), 512MB (131072 pages), 1GB (262144 pages), 2GB (524288 pages), 4GB (1048576 pages), 8GB (2097152 pages), 16GB (4194304 pages), 32GB (8388608 pages) and 64GB (16777216 pages) input files. The tables in this section, shows the number of collisions throughout three runs of the hash function on the same input file size.

Hash function	**256MB**	**256MB**	**256MB**	**Frequency**
FNV 16	63.3%	63.2%	63.3%	63.2%
FNV 16a	63.3%	63.2%	63.4%	63.3%
FNV 32	2	0	0	0.6
FNV 32a	2	0	0	0.6
FNV 64	0	0	0	0
FNV 64a	0	0	0	0
Bob Jenkins 32a	2	0	0	0.6
Bob Jenkins 32b	0	2	2	1.3
Bob Jenkins 64a	0	0	0	0
Bob Jenkins 64b	0	0	0	0
SuperFastHash	0	0	2	0.6
MD5	0	0	0	0
SHA1	0	0	0	0

Table A.1: Collisions in the different functions on the three different data sets with a 256MB (65536 pages) input file.

Table A.6 on page 84 contains test with a single data set on random data, which is between 8 to 64GB in size.

Hash function	**512MB**	**512MB**	**512MB**	**Frequency**
FNV 16	86.5%	86.4%	86.6%	86.5
FNV 16a	86.5%	86.4%	86.6%	86.5%
FNV 32	2	4	8	4.6
FNV 32a	2	4	8	4.6
FNV 64	0	0	0	0
FNV 64a	0	0	0	0
Bob Jenkins 32a	6	8	6	6.6
Bob Jenkins 32b	4	8	0	4.0
Bob Jenkins 64a	0	0	0	0
Bob Jenkins 64b	0	0	0	0
SuperFastHash	10	4	2	5.3
MD5	0	0	0	0
SHA1	0	0	0	0

Table A.2: Collisions in the different functions on the three different data sets with a 512MB (131072 pages) input file.

Hash function	**1GB**	**1GB**	**1GB**	**Frequency**
FNV 16	98.2%	98.2%	98.1%	98.1%
FNV 16a	98.2%	98.2%	98.1%	98.1%
FNV 32	14	14	16	14.6
FNV 32a	14	14	16	14.6
FNV 64	0	0	0	0
FNV 64a	0	0	0	0
Bob Jenkins 32a	12	10	24	15.3
Bob Jenkins 32b	14	6	16	12
Bob Jenkins 64a	0	0	0	0
Bob Jenkins 64b	0	0	0	0
SuperFastHash	12	22	20	18
MD5	0	0	0	0
SHA1	0	0	0	0

Table A.3: Collisions in the different functions on the three different data sets with a 1GB 1GB (262144 pages) input file.

Hash function	**2GB**	**2GB**	**2GB**	**Frequency**
FNV 16	100%	100%	100%	100%
FNV 16a	100%	100%	100%	100%
FNV 32	60	58	58	58.6
FNV 32a	60	58	58	58.6
FNV 64	0	0	0	0
FNV 64a	0	0	0	0
Bob Jenkins 32a	40	88	84	70.6
Bob Jenkins 32b	48	58	68	58.0
Bob Jenkins 64a	0	0	0	0
Bob Jenkins 64b	0	0	0	0
SuperFastHash	70	66	60	65.3
MD5	0	0	0	0
SHA1	0	0	0	0

Table A.4: Collisions in the different functions on the three different data sets with a 2GB (524288 pages) input file.

Hash function	**4GB**	**4GB**	**4GB**	**Frequency**
FNV 16	100%	100%	100%	100%
FNV 16a	100%	100%	100%	100%
FNV 32	268	248	290	268.6
FNV 32a	268	248	290	268.6
FNV 64	0	0	0	0
FNV 64a	0	0	0	0
Bob Jenkins 32a	230	286	244	253.3
Bob Jenkins 32b	276	236	221	244.3
Bob Jenkins 64a	0	0	0	0
Bob Jenkins 64b	0	0	0	0
SuperFastHash	260	264	234	252.6
MD5	0	0	0	0
SHA1	0	0	0	0

Table A.5: Collisions in the different functions on the three different data sets with a 4GB (1048576 pages) input file.

Hash function	**8GB**	**16GB**	**32GB**	**64GB**
FNV 16	100%	100%	100%	100%
FNV 16a	100%	100%	100%	100%
FNV 32	950	3940	16245	65613
FNV 32a	950	3940	16245	65613
FNV 64	0	0	0	0
FNV 64a	0	0	0	0
Bob Jenkins 32a	992	4048	16347	65475
Bob Jenkins 32b	1068	4146	16152	65611
Bob Jenkins 64a	0	0	0	0
Bob Jenkins 64b	0	0	0	0
SuperFastHash	1118	4306	17350	69721
MD5	0	0	0	0
SHA1	0	0	0	0

Table A.6: Collisions in the different hash functions on four different input size running from 8 to 64GB.