A Paravirtualized Approach to Content-Based Page Sharing

Jacob Faber Kloster Jesper Kristensen Arne Mejlholm
Department of Computer Science, Aalborg University
Fredrik Bajers Vej 7E, 9220 Aalborg Ø, Denmark
{jk|cableman|mejlholm}@cs.aau.dk

June 4, 2007

Abstract

Recent advances in virtualization and hardware costs has made virtualization an appealing alternative to achieve server consolidation with performance isolation. Server consolidation however often results in much redundancy in main memory, because virtual machines tend to use the same kernels, libraries and applications.

In this paper we present a novel approach to reducing this redundancy by sharing memory between virtual machines in a virtualized system. The novel approach is that we use paravirtualized guest operating systems to participate in page sharing. In particular we depend on these to find virtual mappings that must be updated in order to share pages. This is less expensive than the alternative, using shadow page tables to alter virtual mappings in a fashion that is transparent to the guest operating system. Instead, we use the reverse mapping feature in Xen's paravirtualized Linux kernel to efficiently find virtual mappings for a given page. Furthermore with a paravirtualized operating system, it is possible to avoid sharing pages that are time critical.

We demonstrate that the approach is feasible by showing that it is able to share significant amounts memory and that the overheads of using this approach is smaller than using shadow page tables.

1 Introduction

As virtualization[8, 19, 20, 6] gains popularity, more and more exciting approaches to utilizing and/or optimizing the underlying virtualized platform, see the light of day.

While the hardware hosting the virtual system keeps steadily improving scalability, this is insufficient for some uses. Examples include running large amounts of dormant servers, workloads with high demands for available memory to cache used files or dynamically instantiating Virtual Machines (VMs) to respond to incoming traffic, as shown in Potemkin[23].

In order to ensure the scalability of virtualized systems, some bottlenecks must be overcome. Two typical approaches to this is to reduce the storage needed by VMs through memory sharing and Copy-on-Write (CoW) file systems.

Memory sharing is feasible because virtualized systems often run homogeneous VMs, often with the same basic services running. This implies a significant amount of redundancy in main memory, that can be reduced by use of simple CoW sharing techniques. To do so three tasks must be performed: 1) finding redundant pages, 2) sharing pages by updating virtual mappings for the pages containing redundancy and 3) breaking the page sharing for a given page, if the contents of it is about to be changed.

The first task is straightforward, one approach is to scan pages and identify duplicates by their content. This technique is referred to as content-based page sharing[24, 16]. The second task is more complicated, all page tables that are mapping a given page must be found and updated efficiently. The semantic gap[5, 12] between the Virtual Machine Monitor (VMM) and VMs however makes this non-trivial, because most VMMs record little information about the usage of pages within guest Operating systems (OSes). Quoting David Wheeler: *Any problem in computer science can be solved with another layer of indirection.* Not surprisingly, the traditional approach to overcome the semantic gap is to provide a layer of indirection between virtual addresses used in VMs and the machine pages they are mapping, by maintaining shadowed versions of guest OS page tables. In this paper we contradict this trend by actually avoiding a layer of indirection. Instead, we take a paravirtualized approach, by altering the guest OS to participate in finding the virtual mappings for a given page and then letting the VMM update the page tables. Our claim is that this is possible and can be made more efficiently compared to using shadowed structures. In the paper, the reverse mapping functionality in the Linux kernel is used to do this, but the approach should also be viable for other paravirtualizable OSes. For non-paravirtualizable OSes, the transparent approach earlier demonstrated[16], can be used as a fall-back solution. The third task, of breaking sharing, is trivial once task number two is achieved.

This paper focuses on the second and third task, so we only touch upon the first task. For more detailed information the reader is referred to [13, 14, 16, 15]. To sum it up, a page scanner is implemented to be invoked from the idle loop of the VMM, thus only machine cycles that would otherwise be wasted are used for the scanning. This uses compare-by-hash[9, 10] to efficiently find identical pages, by hashing the contents of pages and comparing the hash values by inserting them into a hash table. The process of scanning all intended pages is referred to as a page scanning round. Once a round is over, the hash table is flushed.

The structure of the paper is as follows: in Section 2 we discuss related work, in particular different approaches to memory sharing. In Section 3 we present the design of our paravirtual architecture and discuss which types of pages are feasible to share. In Section 4 we present an overview of the implementation, in particular the changes introduced to Xen. We conclude that section with an overview of the implementation status and explain further optimization. In Section 5 we show that this approach to memory sharing is feasible. Finally in Section 6 we conclude the paper and discuss further work.

2 Related Work

There exists several approaches to finding pages eligible for sharing. These can be divided into two different approaches: 1) passively using prior knowledge or 2) actively searching for identical contents.

Examples of the first approach include Disco[3] and XenFS[25], which rely on VMs using the same disk images to known which files are already present in memory. This means that a given file may be brought into memory once and mapped into several VMs simultaneously. Another example is Potemkin[23], which forks new VMs from an already running instance that is suspended, ensuring that forked VMs only utilize memory corresponding to the degree of how much they differ from the original VM.

Examples of the second approach are content-based page sharing in virtualized systems[24, 17, 16] and Mergemem[18]. The latter implemented transparent memory sharing for processes in the Linux kernel.

Content-based block sharing was explored in [22]. This allows VMs to use distinct disk images, without sacrificing memory sharing. Content-based buffer cache in [11], is similar to this, but instead of attempting to increase the effective size of main memory, this is focused on increasing the effective size of buffers.

Updating virtual mappings in order to share pages is, as mentioned, traditionally (in VMware, Disco, Potemkin etc.) achieved by changing mappings in an extra layer of indirection. Our approach is more similar to that applied by Mergemem, where we update the virtual mappings within the guest OS. This is similar to Memory hotplug project[21], which also needs to update page tables, amongst other things, in order to remove a range of machine memory addresses.

3 Architecture

The architecture, which we have dubbed the paravirtualized architecture or just para, is illustrated in Figure 1 on the following page. The purpose of this design is to make sharing of memory pages possible without the need for shadow page tables. Instead of relying on the shadow page tables to handle virtual mappings, the VMs will be aware of the fact that they are sharing pages with other VMs.

We have chosen to split our design into three components to make a design that is dynamic and easy to implement. The three components are:

Page Hashing (PH) The page hashing component is responsible for hashing memory pages. The results produced by this component are sets of pages that are candidates for being shared. For more information about how the hashing process is conducted and the reasoning behind it, the reader is referred to our earlier work in [13, cha. 6], [14, p. 65-67] and [16, p. 3].

Copy-on-Write Sharing (CS) This component is placed in each of the VMs that participate in page sharing. Therefore it is also referred to as the paravirtualized driver. It receives shareable page addresses from a buffer that is shared by the VM and the VMM. It is gathering information about page table mappings for a given page in the VMs when setting up and tearing down shared pages and returning this to the VMM. At a first glance, it may seem as this is to rely too much on the VMs. However all new functionality is restricted to ordinary Xen auditing, so a malicious VM may at most harm itself.

Reference Manager (RM) The reference manager component is located in the VMM and is the main component of the design. It ensures that the right components are called in different situations e.g. setting shares up, hashing pages etc.

It is important to register the original state of pages before they are shared and track updates to these. This entails saving the number of references to a given shared page and whether it was writable before sharing it, so it can be restored to its original state when it is torn down. The reference counts are used by the VMM to track the usage of memory pages. If it drops to zero, then the VMM will consider it free, so it is critical to keep this bookkeeping correct.

3.1 Meta-data Dependent Sharing

The paravirtualized design was first designed in [13], and differs from our transparent design[14] in that it introduces changes to the guest OSes. Since it partly operates from within the guest OS, it has good knowledge about which pages it is

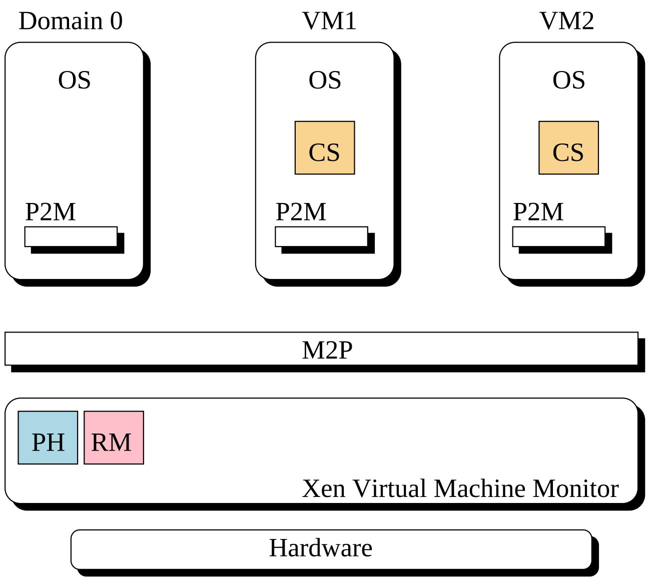


Figure 1: The paravirtualized design, where most of the functionality is placed in the Xen VMM. The machine-to-physical (M2P) and physical-to-machine (P2M) tables are also illustrated.

about to share. This knowledge can be used to make informed choices about which pages to share. In [15] we examined, what shareable pages was used for inside an VM for a set of different workloads. It is natural to make use of the results from this and only share the pages which we deem gives the overall system a performance gain. Generally put, it is a classic balance between processor cycles and memory usage, e.g. it might not feasible to setup a group of memory pages that are unshared momentarily, so only processor cycles are wasted.

The following is a discussion of whether to share different groups of pages. The grouping of the pages, into a set of categories, largely follows how the Linux OS views them.

Mapped pages: These are pages which are related to a block device, (e.g. the hard disk). They are mapped into the virtual address space of one or more processes. In this category of pages we typically find system libraries etc. This is in most situations a very attractive part to share and a significant percentile of the shareable pages on typical workloads. As they are backed on secondary storage and identical to other pages in the virtualized system, it seems probable that it is a shareable page that will not be broken down momentarily. These types of pages should therefore be shared.

Anonymous pages: These pages, as the name implies, are anonymous in that they are not backed on storage. We only know that they are mapped into the virtual address spaces of a set of processes. The number of shareable pages in this category is quite limited on typical workloads. The pages that are found shareable, should be long term shares because it is unlikely to have founded duplicates of this kind of pages. They must thus have rather static contents and should therefore be shared.

Cache only pages: This category of pages are only used by the page cache and at the moment not in use by any process. It is a very large group of pages on most workloads and has a high likelihood of being shareable. It is therefore monumental to share this kind of pages.

Kernel only pages: This category covers the pages only used by the kernel. These should not be shared for a set of reasons. An overall reason is that the Linux kernel does not expect page faults to these kinds of pages and the kernel address space is not designed to be remappable. In [15] we found that this category primary consist of the following two groups of pages, identified by a flag on the page descriptor: slab and reserved. Both will be discussed separately below.

- **Reserved pages:** About these pages we only know that the kernel uses them or they are corrupted memory frames[1]. It seems rather problematic to shares these pages so they are skipped.
- **Slab pages:** These pages belongs to the slab object caches[2]. Since these are likely to be used and are critical to the performance of the OS. For workloads where the redundancy caused by the slab object caches for some reason must be reduced, a better solution is to shrink the amount of pages used for the slab.

Free pages: This category consists of the free pages in the OS. For most workloads the amount of shareable page in this category is low. We decide not to share these pages, as we see more clever ways of returning these pages to the VMM than sharing the few that can be shared. Also these are often depended on to be quickly allocable, so sharing them are likely to be a performance overhead. One way is to just balloon the free pages out if the memory pressure within a VM is sufficiently low.

4 Implementation

Having provided an overview by discussing the architecture of the system, we now move on to the interesting details in the implementation. First necessary modification to the M2P in Xen are explained, followed by an explanation of how shared pages are set up and torn down. Subsequently issues with shared pages and grant tables in Xen are discussed. The section is ended with an overview of the implementation status and a discussion about further optimization.

The globally available Machine-to-Physical (M2P) table and the per VM Physical-to-Machine (P2M) table, as shown on the figure, gives us an address space called pseudo-physical [1]. This gives each VM the impression that it has a contiguous range of memory, as many OSes are not well suited for handling fragmented address spaces. We use pseudo-Physical Frame Numbers (PFNs) as unique identifiers for pages in the VMs during sharing as they remain static during the lifetime of a VM.

To realize this design six hypercalls are introduced, as shown in Table 1 on the next page. The `do_get_pfn` hypercall falls outside the category of the five others and it is introduced because we make some fundamental design changes to the M2P table. The normal M2P table in Xen maps a given Machine Frame Number (MFN) to one PFN. A mapping from a MFN to all the PFNs that are pointing to it is needed when pages are shared. This is explained in the following subsection.

4.1 The Machine-to-Physical Table

The M2P table contains mappings from MFNs to PFNs and it consists of an array with one-to-one relationships between MFN and PFN. It is, as mentioned, necessary to find all PFNs given a MFN in order to share pages between VMs. This is however

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

`do_setup_shares`	Is used to setup shared pages. It performs a sequence of validation checks to ensure that isolation is not broken between the VMs. Afterwards the page table mappings found by the CoW Sharing component are updated.
`do_cow_break`	Is used to tear down shares as a result of a VM trying to write to a write protected shared page. So a private non-shared copy is created for that VM and all page table mappings referring to the shared page are updated (CoW break).
`do_is_shared_page`	Is used to identify a page as shared.
`do_remove_gnttab_bit`	Used to clear a bit in a bitmap introduced to handle sharing and CoW breaking of pages shared by grant tables.
`do_set_gnttab_bit`	Used to set a bit in the bitmap mentioned in the previous hypercall.
`do_get_pfn`	Is used to query the M2P table.

Table 1: Hypercalls introduced in order to implement the paravirtualized design.

not possible with the M2P table as provided by Xen. The M2P table entries are therefore extended with single linked lists as exemplified in Figure 2 on the following page. This simple extension causes some issues as the M2P table is globally available and the VMs read its contents for different purposes. They are however not able to read the linked list extension, as they have not mapped the extension into their address spaces.

To be able to read the fully extended M2P table, the VMs would have to map the linked list elements in and out of their address space, once for each element in the list they what to access. Instead a hypercall (`do_get_pfn`) is introduced to query the extended M2P table and altered the guest OS to use it. This is acceptable as hypercalls are relatively inexpensive compared to mapping the linked lists in and out, as too much re-mapping may cause Transition Lookaside Buffer (TLB) flushes.

The elements in the linked list consists of a pointer to the next element (32 bit) in the list and a 32 bit word carrying meta-data. The word is encoded with a VMs unique 12 bit id number (`domain_id`) in the least significant bits and the PFN in the 20 most significant bits. Normally the unique id number is 16 bits, so this encoding will limit Xen to a maximum of 4096 VMs (2^{12}) at the same time, which is far more than Xen supports at the moment[23, p. 11]. This limitation is necessary as the maximum number of PFNs possible under our implementation is 1.048.576 (2^{20}), which corresponds to 4 GB of memory[2]. Memory is allocated to hold the list elements from the Xen heap, which is limited to 10 MB on a 32 bit system. This heap is normally used to contain meta-data about the VMs running on the system. The Xen allocator is not suited for small frequently allocated and deallocated memory chunks. We therefore build our own 8 byte mini allocator (much in the spirit of a normal slab object cache).

The effect of this simple M2P change is that the system cannot handle sharing of several identical pages within a single VM (also referred to as internal sharing). If pages are shared internally in the VMs, then one MFN would map to two PFNs or more and it is not possible to distinguish the two pages when updates to the page are applied, e.g. when the reference count increases. Internal sharing is therefore avoided in the VMs, which seems reasonable as the amount of internal sharing is

[2]We only support 32 bit systems without the Physical Address Extension (PAE), which restricts the system to a maximum of 4 GB of memory.

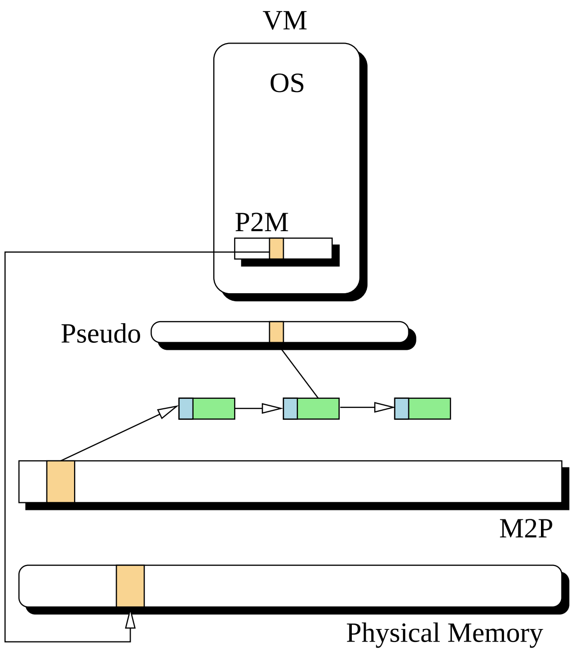


Figure 2: The M2P and P2M layout with the single linked list extension to the M2P table. Here a linked list added to an entry in the M2P table for a shared page. The page is shared between three VMs (the length of the list) even though only one VM is shown here.

quite limited.

4.2 Setting Up Shared Pages

When setting up shared pages in the paravirtualized design, there are a number of steps that have to be taken. The three main steps are: 1) finding and preparing the shareable pages, 2) using the Linux kernel's reverse mapping[15, p. 23-28] to efficiently retrieve mapping information and 3) updating the mappings from the VMM.

To reduce the trusted code base, it is ensured by the VMM, that updates are only allowed if the mappings being updated, map a page that is owned by a special pseudo VM called the share domain[3] and that they map a page belonging to the VM requesting the update.

Furthermore to ensure correctness in the virtualized system in step 3, it must be ensured that the contents of the page has not changed since the page was last compared. This is due to a race condition between the usage of the page that is to be shared and actually sharing the page: since an arbitrary amount of time may have elapsed since the page was placed into the buffer by the reference manager component and until the VM finds the mappings that needs to be updated, so the content of a page may have been changed at any point in between these two steps.

Figure 3 on the next page shows the process of setting up a shared page. The steps are described in the following:

1. **Finding and preparing**: This step consists of finding pages that are candidates for being shared. The main work is performed by the page hashing

[3]This domain is also referred to as the clone domain and is used as a container for all the shared pages.

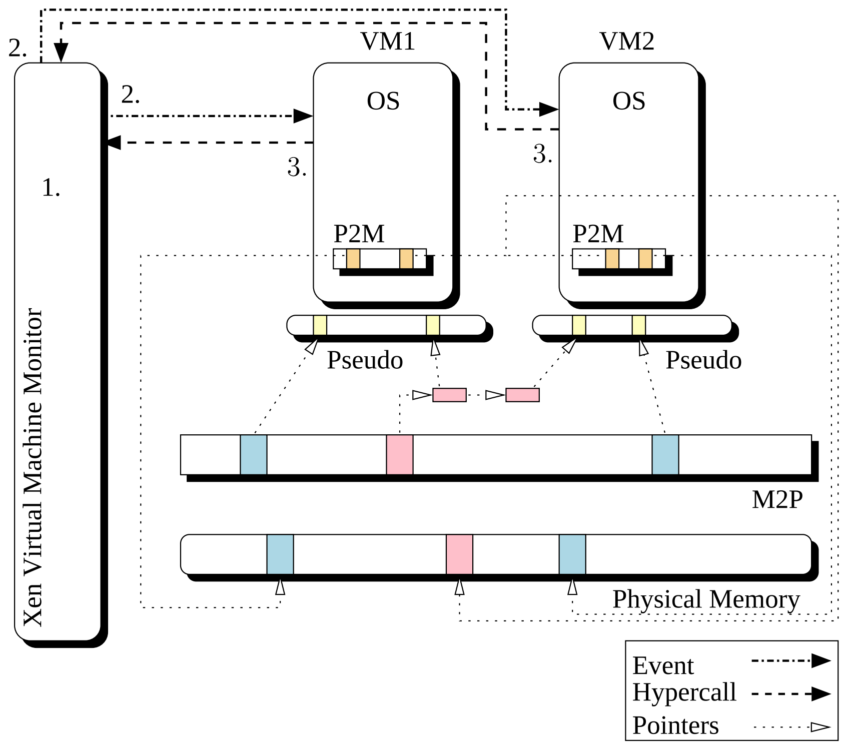


Figure 3: The process of setting up a shared page. The pink page (middle one in physical memory) is shared between the two VMs. The pointers show how M2P and P2M tables are related to each other.

component. When two shareable pages have been found, they are passed along to the reference manager component, which allocates a new page (the page to be the shared page) and copies the content of one of the old pages to the new page. After the copy, the reference manager component sends an event, with the addresses of the old and new page, notifying the VM that a page is ready for sharing and it should find all mappings that have the old page mapped. In fact the mappings is not sent immediately with an event but is batched in a buffer until a certain threshold is reached or a scan round is completed. We have chosen to batch the updates to not overload the VMs with virtual interrupts. Furthermore consider the case where an event is sent each time a shareable page is found. This would require a switch between the VMM and each of the VMs one at the time, which would introduce a considerable overhead as the current page table has to be changed for each VM participating in the sharing of a given page. This also forces a flush of the TLB.

Pages that changes their contents relatively fast will be invalid for sharing while in the buffer. If they had been shared, they would have triggered a CoW break immediately after setup, which would have been undesirable and the sharing would have been wasted.

2. **Getting mapping information**: Events are handled as virtual interrupts by VMs and are therefore handled by an interrupt handler (top half). The interrupt handler schedules the pending work to a work queue for later processing, when not in interrupt context (bottom half). When the work from the queue is being processed, the reverse mapping functionality in the Linux

kernel is used to find the mappings that need to be updated. These mappings are then sent synchronously back to Xen via the `do_setup_share` hypercall for updating.

3. **Update the mappings**: When `do_setup_shares` is called, it first goes through a set of safety checks to verify that the updating requests are legal, which includes a page comparison. Afterwards the page table mappings are updated and all new mappings are set to read-only. So if a VM tries to write to a share page, a page fault will be triggered.

   Next the number of references to the old page is saved, so the page can be restored with the same number of references if a CoW break should occur. The number of references from the old page is then transfered to the new shared page and the old page is freed to the Xen domain heap.

   Then the M2P table is updated by either adding a new element to an existing linked list or a new linked list is created. In Figure 3 on the preceding page it is shown that the middle page (the pink one) is a shared page between the two VMs and that the M2P table is extended with a linked list with one element for each VM. Furthermore the figure shows how the P2M tables are mapping PFNs back to MFNs.

   Finally the last step is to start tracking every reference taken and given to the shared page at VM level. This means that the amount of references taken by each single VM must be tracked, such that the number of references can be set correctly for a given page in the VMs in the case of a CoW break. Then control is passed back to the VM, which updates the P2M mapping and the page is now shared. Fortunately Xen handles this easily with a few modifications.

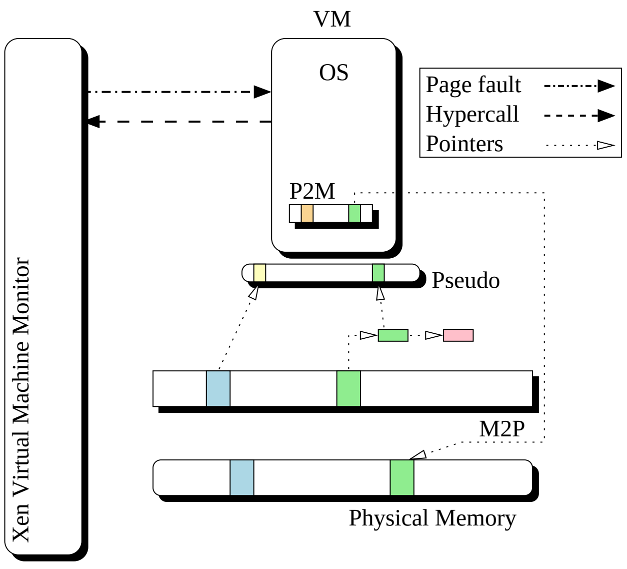
4.3 Tearing Down Shared Pages

When pages have been shared between VMs, it is likely that some of them will be written to at some point in time. When this happens, a CoW-break is triggered and a private non-shared copy must be created, i.e. all original state about the page must be restored. The process of doing a CoW break consists of four steps in our design: 1) detecting that the page fault is in fact to a shared page, 2) using reverse mapping to find all the page table entries that needs to be updated, 3) update the mappings in the page table from the VMM, 4) continue along the normal page fault path in the VM.

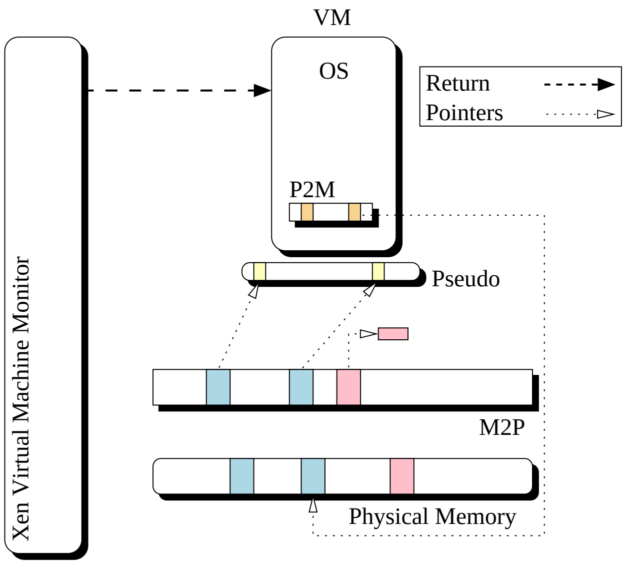
Figure 4(a) on the next page shows a single VM with a shared page and a non-shared page. As it can be seen on the figure the shared page is in a M2P extended linked list. Figure 4(b) shows the same setup just after a CoW break has been performed on the shared page.

1. **Detecting the page fault**: The first step is to detect that a given page fault is a write fault to a shared read-only page. When the fault has been identified as belonging to a shared page, the virtual address is read from Control Register 2 (CR2) and converted into a MFN by walking the current page table. Subsequently, the MFN is stored in the meta-data of the virtual processor. Each VM on the system has a virtual processor that stores the state of the physical processor when the VM is not running on the processor. The page fault is then propagated to the VM that caused the fault.

2. **Finding the mappings**: When the fault is propagated to the VM, it activates the standard Linux `do_page_fault` routine. This is modified to check if a MFN has been set in the virtual processors meta-data. If so, then the reverse



(a) State before a CoW break on a shared page.



(b) State after a CoW break on a shared page.

Figure 4: The process of doing a CoW break illustrated as a before (4(a)) and after (4(b)) state.

mapping functionality is used and all the mappings, that need to be updated, are found. They are then passed back to the VMM with the `do_cow_break` hypercall.

3. **Updating the mappings**: As with the setup of shared pages, a range of safety checks are performed to ensure isolation and correctness. A new page is allocated and the contents of the old shared page is copied into the new page. Next all mappings are updated with the same read-write permissions as before the page was shared and the number of references to the page are restored.

   Next M2P mapping needs to be updated and because it is during a CoW break it is known that the page is in a linked list. This means that the element belonging to the page is simply removed from the list or the list is completely removed. The MFN of the new page is then used to create a new M2P mapping with the PFN of the original page.

4. **Page fault**: The control is then returned to the VM, which updates the P2M mapping and continue down the normal page fault path of the Linux kernel. This is necessary as the CoW break could potentially be a side effect of a real page fault i.e. a write fault to a page that was read-only before it was shared. This is in fact highly probably as the Linux OS shares pages during forking of new processes and when allocating memory with the `malloc` function the pages allocated are pointed to the zero page until written to. Therefore the page fault could easily be a side effect of a VMs internal CoW breaks, so the page fault should still be handled by the OS after the original state of the page is restored.

4.4 Grant Table Issues

Grant tables[4, 7] is a general mechanism in Xen, which is used to share or transfer machine frames between VMs. They are used during such operations as disk access and network send and receive, where the privileged VM (domain 0) uses grant tables to swap I/O with non-privileged VMs. Grant tables can also be used without the involvement of a privileged VM.

Grant tables has two modes in which they can operate: 1) sharing mode where one VM share a frame with another VM and 2) transfer mode where one VM transfer ownership of a frame to another VM. The first mode is used by the I/O back-end driver in Xen to transfer data read from or written to disk by the non-privileged VMs. This is done by sharing frames between the privileged VM (`P`) and a non-privileged VM (`A`) in the following way:

1. VM `A` creates a grant reference and sends it to VM `P` by the use of an virtual interrupt. This grant reference may grant both read and write access.

2. VM `P` then uses the grant reference to map the frame into its address space.

3. VM `P` performs the memory access it needs.

4. VM `P` unmaps the frame.

5. VM `A` removes the grant reference and thereby ends the sharing of the frame with VM `P`.

The latter mode in which a frames ownership is transfer to another VM is used by the network front-end and back-end drivers to send and receive packets. The transfer is performed as shown below:

Related to	**Location**	**Modified code**	**New code**	**Total**
Setup of shares	Linux	6	901	907
Setup of shares	VMM	2	683	685
CoW break	Linux	25	443	486
CoW break	VMM	11	235	246
Page tracking	VMM	78	125	203
Page scanning	VMM	0	1250	1250
Ref. manager	VMM	0	820	820
Grant tables	Linux	24	17	41
Total number of lines				4638

Table 2: The amount of code either modified or added to Xen to make content-based page sharing possible without shadow page tables.

1. VM `A` creates an accept grant reference and sends it to VM `P`.
2. VM `P` uses the reference to hand over one of its frames to VM `A`.
3. VM `A` then accepts the transfer and thereby changes the ownership of the frame and clears the reference used.

An unprivileged VM may grant write permissions on a CoW shared page to another VM using grant tables. Thus an unprivileged VM may alter the contents of a shared page without triggering a CoW-fault. A simple solution to this problem is to detect when a shared page is part of a granted reference and force a CoW break on the shared page before it is granted.

4.5 Implementation Status

The implementation in its current state consists of changes in both the kernel running inside the VMs and in the VMM. Table 2 gives a rough overview on the number of lines of code that was needed to make paravirtualized content-based page sharing possible. The code in its current state is stable enough to perform experiments with proof of concept in mind, but it is not production ready and further stabilizing is needed. In the remainder of this subsection, a set of isolation breaching issues are described. All of these should be trivial to implement.

We have an implementation issue with the allocation of pages to the pseudo domain (share domain). As it is implemented at the moment, pages are allocated in the references manager and a reference is taken to the newly allocated page, so it cannot be freed and a bit is set in a bitmap indicating that the page is allocated as a shared page. This bitmap is used later on, after the current scan round has been completed, to release the reference taken during the allocation. This is done to ensure that the page is freed again if it was not setup as a shared page. There can be many reasons for why it was not setup as shared e.g. filtered out as part of a grant reference or the contents of the page had changed. This creates the following issue: If a large amount of pages are identified as shareable, but for whatever reasons are not set up as shared, then the domain heap may be depleted and cause the VMM to crash. Newly allocated pages are only freed as a consequences of pages being freed during setup or when the pages, that have failed to be setup by the guest OS, are freed once a scanning round is completed.

This problem could be solved by delaying the allocation of the new page until the `do_setup_shares` hypercall is called. A workaround for this memory problem is to only allocate pages for shares that are known to succeed in being set up. There is however a twist about allocating the page during the setup hypercall. Somehow the other VMs that participate in a given sharing has to be informed about the newly allocate pages address and insure that only one new page is allocated per shared page. This may be solved by using a bitmap and a data structure to indicate which pages that have be allocated to which shared page. Another solution could be to apply a policy that restricts the amount of uncompleted shares to be setup at any given time and thereby control the amount of allocated unshared pages. This restrictive limit could be adaptive to the number of free pages on the Xen domain heap.

Another issue is that any VM may read the M2P table and thereby find out which pages that are shared pages and map these into their address space, as the table is globally available. This is possible as the validation checks allow all updates to pages owned by the share domain. This could be corrected by inserting an extra check in the audit code that only gives references to pages by checking that the VM, trying to map in the page, is actual part of the sharing. This could be done by reading the extended M2P table that reflects which VMs and PFNs in these VMs are allowed to be used.

The implementation has one more issue, which has the consequence that a VM can re-map a shared page as writable and thereby change the content of the shared page without triggering a CoW fault. This leads to an isolation problem, which could be used to attack another VM by altering system critical information. This should be handled by extending the auditing code to the hypercalls used when changing mappings.

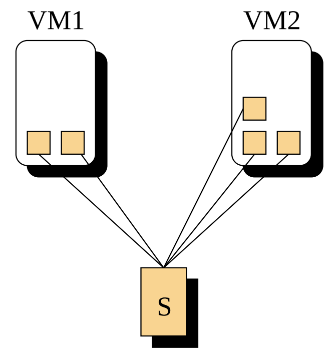
4.6 Further Optimization

In the following, when CoW is written, it refers to the kind introduced in content-based page sharing and when copy-on-write is written, it refers to the normal kind used in a the standard Linux OS.

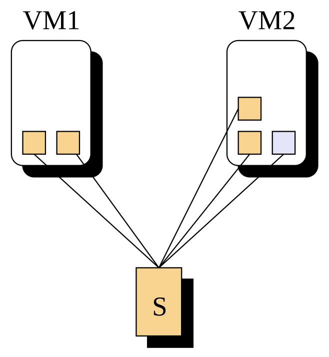
An optimization, that should improve the performance of the paravirtualized implementation, will now be explained. Consider the following example: A set of processes share a page in a normal OS copy-on-write fashion. The normal OS operation on such a page, if one of the processes tries to write to it, is that a page fault is triggered. The page fault then copies the content to a new page and the page table of the process is updated to point to the new private copy with write permission. After this, the process continues from where the page fault was triggered. Pages involved in this internal OS copy-on-write sharing, are likely to be shared with pages in other VMs, since they contain data which is shareable inside a single OS so it is also likely that they are shareable between VMs.

In the following it will be shown how a page fault is handled on a page, that is CoW shared (between VMs) and copy-on-write shared internally in a OS. We use Figure 5 on the next page to illustrate the example.

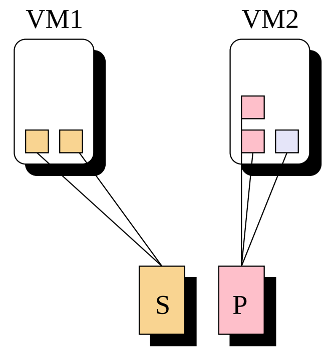
The following will occur in the current implementation: when a page fault on a CoW shared page is triggered inside a given VM, the sharing of the page is broken in all cases, meaning a private copy is created for that VM. In the example, this results in a break of the CoW shared page and the VM gets a private copy (step **(a)** to **(c)** in Figure 5). The private copy still has a set of processes using it in a copy-on-write fashion, which means that the page is mapped read-only by the processes. The result is that the faulting process, the one that triggered the page fault, gets its own private copy of the VM's private copy of the page, due to the normal copy-on-write procedure inside the OS (step **(d)**). The VM's private read-only copy of the page will then be found in the next scanning round and is set up to be replaced with the shared copy again (step **(e)**). This happens each time an internal copy-on-write page fault happens on such a copy-on-write CoW shared page. Hence there is a short term private copy of a CoW shared page.



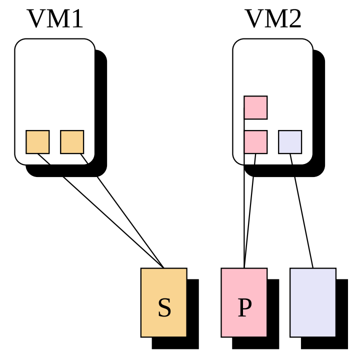
(a) A CoW shared page.



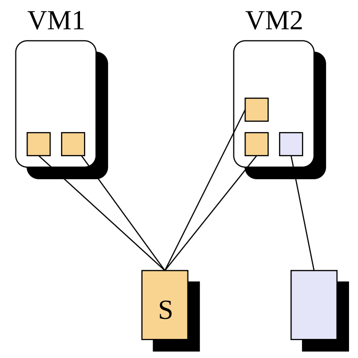
(b) A copy-on-write fault triggered on a CoW shared page.



(c) A new private copy (P) is created after a CoW break on S by VM2.



(d) Process gets a private copy after a copy-on-write fault inside the OS.



(e) The VM's private copy (P) is set up as shared again.

Figure 5: Typical sequence of actions when a VM writes to a CoW shared page. Steps (a) through (e) will happen without any optimization. With the optimization only steps (a) and (e) are needed.

The preceding situation can be optimized when it is known that it is an internal copy-on-write page fault, the page count of the CoW shared page is just decremented and it is left to the guest OS to handle the rest. This yields the same result as in the double fault described above. In Figure 5 this corresponds to going directly from step (a) to (e).

In the case where there is only one mapping for the internal copy-on-write shared page left inside a given VM, then the OS will try to convert it to a writable mapping. This can however not be allowed, since the page is CoW shared. To handled it, the CoW sharing is broken for the VM and it is given a writable mapping of the page, just like it is done without the optimization.

This optimization should remove unnecessary CoW breaks on pages, that are set up as CoW shared pages again momentarily after the break.

5 Results

We evaluate the effectiveness of the implementation by comparing it to our transparent implementation. This entails three experiments: 1) ensuring proof of concept, in that the implementation is able to share significant amounts of pages, 2) that it is actually faster than the transparent implementation and 3) finally we examine the overheads in the transparent designs use of shadow page tables and see how the paravirtualized version performs in the same conditions.

The machine used to conduct the experiments was a 2.6 GHz Intel Pentium 4 Northwood without hyperthreading and 2 GB of memory. Each VM had an allocation of 128 MB of memory. Domain-0 had a memory allocation of 1 GB and the remaining memory was left free. The VMs were running Debian Linux, with a minimum of services executing, e.g. cron and sshd.

The first set of experiments consists of compiling the 2.6.16 Linux kernel in the following manner. First 4 VMs were started and allowed to finish their boot sequence. Then the kernel source code was unpacked inside the VMs, thus filling their page caches. Then page sharing was activated and as much memory as possible was shared e.g. mostly the page cache. This ensures that there will be a significant amount of CoW breaks once the VMs start executing a workload. Finally the unpacked kernels were compiled in each of the VMs at the same time.

Figure 6 on the following page shows the amount of pages that are shared at different times during the experiment. The number of shared pages are pulled from the system every minute. The differences in the amount of shared pages between the two implementations are likely due to the fact that the paravirtualized design do not share the kernel only pages and the free pages categories, as explained in Section 3.1. After the kernels has been unpacked 2-4 minutes into the experiment, the sharing percentage raises to about 68% and 83%. It then steadily drops to 20% and 30% as the kernels are compiled and the memory pressure is increased in the VMs. The sharing percentage rises to 60% and 80% after the kernel compiles has completed, which is mostly due to the files used for the compilation and the resulting kernel binaries in the page cache, as well as the fact that the system is mostly idle, which yields more time for scanning for shareable pages. If the reader is interested in more details about what actually is shared, the reader is referred to our earlier work in [15]. The figure clearly demonstrates that the paravirtualized approach is feasible and that it is capable of sharing pages.

To see the actual performance differences between the two approaches, a set of performance benchmarks are run where the first one consists of timing the kernel compilation process. The setup is the same as under the sharing experiment above. The results of this experiment can be found in Table 3 on the next page. The table shows that there is no performance loss during page sharing in the paravirtualized

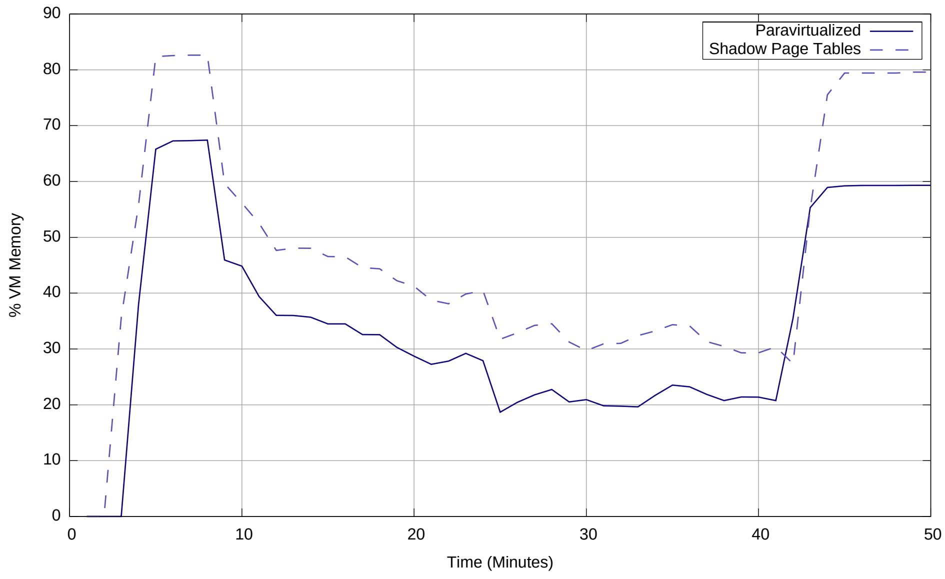


Figure 6: Sharing results from a kernel compilation (4 VMs).

	No Sharing	**Sharing**
Para	2010.81	2006.46
Shadow	2018.53	2050.65

Table 3: Timing of the kernel compilation process with and without sharing for both implementations of content-based pages sharing. Lower scores are better.

design compared to paravirtualization without sharing. To the contrary there are indications of a small performance enhancement with sharing. This may be due to improved memory locality, which may increase cache hit rates as observed in [24, p. 187] and [17]. This experiment shows that there is a significant performance different between using the paravirtualized and the shadow page table approach. The shadow page table approach has in fact a 2.20% overhead with sharing and a 0.99% overhead without sharing. This preliminary performance experiment shows good indications that the paravirtualized approach has better performance than the shadow approach.

Secondly the Aim9 benchmark suite is run to see the performance under circumstances that stress shadow page tables. Table 4 on the following page shows two of the memory related benchmarks. These simply run a loop that either forks processes or executes another script. The reader should be aware that the numbers presented, in this experiment, is the mean of three runs taking 60 seconds each per test. Overall the results shows that there is an overhead in using shadow page tables. The `exec` and `fork` tests, executes a large amount of new processes and thereby creates new shadow page tables, when used in the transparent design. These experiments also show that the effect of sharing pages do not have much, if any, impact on the performance of the operations shown here. This applies to both implementations.

The only time critical part of both designs is the CoW breaking of shared pages, since it primarily takes place in the page fault context. To benchmark this exact situation we created an experiment where that following steps are carried out: 1) first a set of 3 VMs are started, 2) a Linux kernel is unpacked in each VM, to fill the page cache and thereby share roughly 26900 pages (105 MB) with shadow page

Mode	Test	Mean	Std. Dev.
Para, no sharing			
	`exec_test`	408.66	0.94
	`fork_test`	1586.66	12.28
Para, with sharing			
	`exec_test`	407.00	0.81
	`fork_test`	1579.00	14.14
Shadow, no sharing			
	`exec_test`	351.00	2.82
	`fork_test`	1200.00	34.41
Shadow, with sharing			
	`exec_test`	347.66	1.69
	`fork_test`	1151.33	39.50

Table 4: Memory related performance counts from the large UNIX benchmark suite Aim9 version 9.1.10. The results are based on three runs. Higher scores are better.

Mode	Mean	Std. Dev.
Para, no sharing	1.56	1.34
Para, with sharing	1.56	1.07
Shadow, no sharing	1.11	0.99
Shadow, with sharing	5.78	0.42

Table 5: The measurements are the mean time it takes to allocate 24832 pages (97 MB) of memory and write to them under different conditions. Each measurement is based on 9 datasets. Again lower scores are better.

table and roughly 22300 (87 MB) in the paravirtualized implementation, 3) then 24832 pages (97 MB) are allocated and written to, which forces the VM to perform approximately 22400 CoW breaks on shared pages (87 MB). This last step is timed and is used as the result of the experiment.

The results of the experiment are shown in Table 5. It shows that there are no significant overheads in the allocation of pages and writing to these, with or without sharing enabled in the paravirtualized design. This means that there are no visible overheads in performing CoW breaks in that design. Contrary to this, there is a significant overhead of the CoW breaks in the transparent design. In fact it takes five times as long to allocated and write to the same amount of pages with sharing as without, as it can be seen in the table.

6 Conclusion

In this paper we presented a novel approach to memory sharing in virtualized systems. It distinguishes itself from earlier approaches in that it bridges the semantic gap between the virtual machine monitor and virtual machines, by relying on the virtual machines to use the reverse mapping feature in the Linux kernel in order to effectively find virtual mappings for a given page. This conveniently also allows us to avoid pages that might not be feasible to share, e.g. free pages and pages that are used as slab caches. This is a trade-off between sharing as much memory as possible and the processor cycles spent on breaking and setting up sharing. For most workloads it should be feasible to share only pages that have relatively static contents.

We showed that shadow page tables are expensive and that our approach with paravirtualized operating systems leverages a more efficient solution in terms of performance. For non-paravirtualizable operating systems, we also provide an approach that is transparent to guest operating systems.

The cost of using this approach is the time spent on altering a guest operating system to participating in the sharing. As we however have developed a reasonable interface for this, the cost of building the necessary functionality for other operating systems should be low.

6.1 Future Work

First of all, the paravirtualized driver needs further stabilizing work, which includes implementing support for live migration, shutting down virtual machines etc. It should be trivial to port the shadow specific parts of the code to support hardware virtual machines (HVMs) and Xen's new shadow implementation, as well as making the paravirtualizing driver work seamlessly with these. It could be worthwhile to examine the limits of our modified M2P table, to determine whether it is possible and feasible to share pages internally within a single virtual machines. Alternatively internal sharing could be handled by a Mergemem approach, where the operating system eliminates duplicate pages by itself. Finally it could be investigated whether it is possible to minimize the number of hypercalls needed to setup and tear down sharing.

In the broader sense, it would be worthwhile to investigate which other paravirtualizable operating systems are capable of efficiently finding all virtual mappings for a given page. Furthermore operating systems supporting hotplug memory could be used to increase the memory allocation for a given virtual machine, depending on how much it has already shared.

It could prove interesting to extend Xen to support a more adaptive memory allocation for virtual machines. If using paravirtualized drivers, these could easily be extended to use operating system functionality to swap out pages that are infrequently used or balloon pages out if the memory pressure within the operating system is low. Virtual machines could, before swapping out any pages, hash the contents of the pages to see if the page is shareable.

References

[1] 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.

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

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

[4] Vineet Chadha, Ramesh Illikkal, Ravi Iyer, Jaideep Moses, Donald Newell, and Renato Figueiredo. I/o processing in a virtualized platform: A simulation-driven approach. In *VEE '07: Proceedings of the 3rd International ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*. ACM Press, 2007.

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

[6] Simon Crosby and David Brown. The virtualization reality. *Queue*, 4(10): 34–41, 2007. ISSN 1542-7730.

[7] 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`.

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

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

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

[11] Charles B. Morrey III and Dirk Grunwald. Content-based block caching. In *23rd IEEE, 14th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST2006)*, May 2006.

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

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

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

[15] Jacob Faber Kloster, Jesper Kristensen, and Arne Mejlholm. Determining the use of interdomain shareable pages using kernel introspection. `http://www.cs.aau.dk/library/cgi-bin/detail.cgi?id=1168938436`, January 2007.

[16] Jacob Faber Kloster, Jesper Kristensen, and Arne Mejlholm. On the feasibility of memory sharing in virtualized systems: Content-based page sharing in the xen virtual machine monitor. `http://services.cs.aau.dk/public/tools/library/details.php?id=1180896308`, Febuary 2007.

[17] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, December 2004.

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

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

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

[21] Joel Schopp, Keir Fraser, and Martine J. Silbermann. Resizing memory with balloons and hotplug. In *Proceedings of the 2006 Ottawa Linux Symposium*, July 2006.

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

[23] 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.

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

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