On the Feasibility of Memory Sharing in Virtualized Systems

Content-Based Page Sharing in the Xen Virtual Machine Monitor

Jacob Faber Kloster Jesper Kristensen Arne Mejlholm

Department of Computer Science, Aalborg University
{jk,cableman,mejlholm}@cs.aau.dk

Abstract

The main memory of homogeneous virtualized systems contains significant amounts of redundancy. Removing the redundancy is beneficial, as the reclaimed memory increases the effective size of main memory and can be used to improve the performance of the entire virtualized system.

One approach to remove this redundancy is content-based page sharing. We design and implement this approach as an extension to Xen patched with Potemkin. The implementation is used to evaluate the potential for sharing pages and to validate the viability of the content-based page sharing method. Finally we perform an in-depth analysis of what the reclaimable pages are typically used for by guest operating systems. We find that the unified disk caches of modern operating systems are often the most significant cause of redundancy in virtualized systems.

Categories and Subject Descriptors D.4.2 [*Operating Systems*]: Storage Management – Virtual Memory; D.4.2 [*Operating Systems*]: Storage Management – Main Memory

General Terms Design, Experimentation

Keywords Virtual Machines, Virtual Machine Monitor, Memory Sharing, Copy-on-Write, Compare-by-hash, Shadow Page Tables

1. Introduction

Many servers today have excess resources and processing power compared to their purpose and are mostly idle, as they typically service requests that come in peaks. Virtualization[1, 2, 3] provides an abstraction over server resources in a way that allows *server consolidation*, i.e. the execution of several isolated Operating Systems (OSes) on the same machine. This abstraction is referred to as a *Virtual Machine* (VM) or *domain* as it represents a partitioning of the server resources. This partitioning may range from a static allocation to a dynamic policy where the resources are redistributed during the course of execution depending on the load of the system. If the allocations for the VMs exceed the amount of physical resources, the system is said to be *overcommitted*. A Virtual Machine Monitor (VMM) arbitrates the access to the resources in order to ensure correct partitioning and isolation.

Several VMMs are available, but only two are of interest to us in this paper. The first is Xen[4, 5, 6, 7, 8], an open-source VMM that uses paravirtualization[9]. The second is VMware[10, 11], which uses dynamic binary translation[12]. VMware is able to overcommit memory resources and dynamically adjust the allocations based on a number of policies. Xen on the other hand has not evolved such policies.

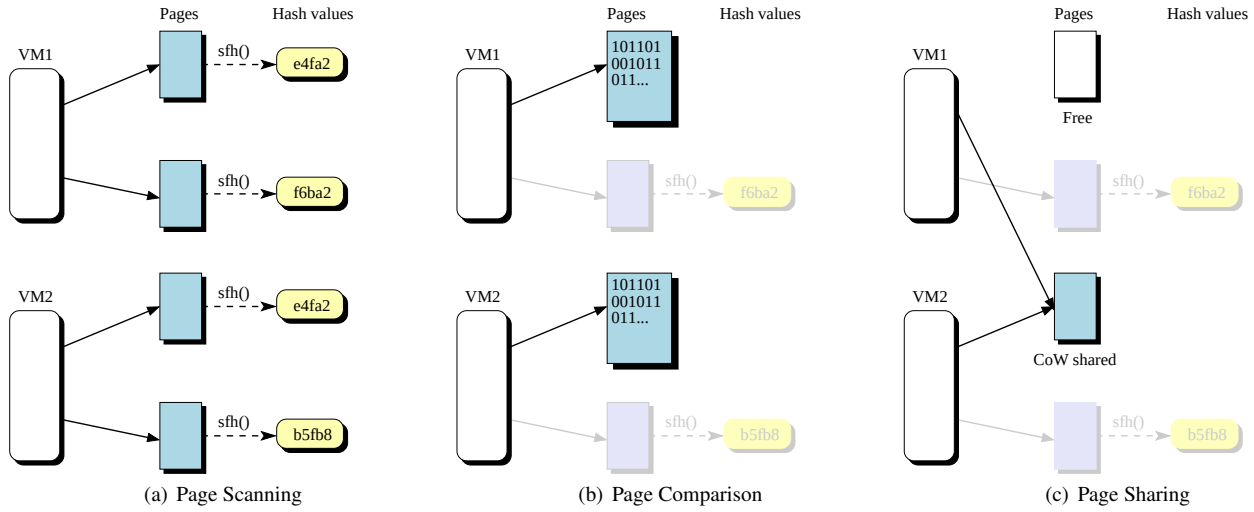
Each VM is fully isolated from the other VMs, so every VM executes an OS with a certain kernel and a number of applications. Many organizations use the same OS on most servers in order to save the inherent costs associated with mastering several OSes. If a consolidated server is running more than one instance of a given OS, it is probable that there is redundant information stored several times in main memory. If the VMs are running the same applications, the binaries of these are present in multiple places in main memory. Even if the VMs are not running any of the same applications, then it is likely that the VMs are using the same standard C libraries and other dynamically linked libraries. While most OSes are able to reduce memory consumption by accessing secondary storage through a common cache, they can still end up with the same data several places in their memory. Consider what happens when a file is copied: the new file will get a new disk cache entry and is written to disk. The old file however also has a disk cache entry and thus memory ends up with duplicate contents once again, so main memory can easily end up with redundant data, even within a single OS.

Put in general terms, the larger the degree of homogeneity in setups on consolidated servers, the larger is also the degree of redundancy. This means that a setup with VMs running different OSes or perhaps just different versions of kernels significantly lowers the probability of redundancy. By eliminating redundancy there will be freed an arbitrary amount of memory. Thus increasing the effective size of main memory and allowing the virtualized system to scale better where the size of main memory is a constraining factor. The freed memory may be used to improve resource consumption in various ways. One way is a static memory allocation policy, where freed memory is used for starting new VMs. Another more dynamic approach is where the freed memory can be used to increase the memory allocation of existing VMs to allow them to better handle peak demands.

There are two distinct approaches to finding redundant pages: compare all pages to the others based on content or use prior knowledge to know before-hand that the pages are identical. Obviously the second is the more efficient, but is often not sufficient or even possible.

1.1 Content-Based Page Sharing

The approach taken by VMware is content-based page sharing, a Compare-by-hash[13, 14] technique that effectively compares all pages in memory with each other. Sharing of identical pages is set up using standard Copy-on-Write (CoW) techniques.



(a) Page Scanning (b) Page Comparison (c) Page Sharing

Figure 1. A conceptual overview of the page sharing process. First the content of a number of pages is hashed (1(a)). If identical hash values are found, the pages are compared bitwise to ensure that the contents of the pages are identical (1(b)). Finally if the pages are identical then one page is freed and one is set up as shared (1(c)).

The main concepts of content-based page sharing are illustrated in Figure 1. Pages with identical contents are identified by the fact that hashing their content yield the same hash values. A naive implementation that compares each page with all the other pages would have a complexity of $\mathcal{O}(n^2)$. Thus hashing is used to speeds up the process. To effectively compare the produced hash values with each other, these are stored as tuples in a hash table along with the address of the page. If a collision occurs when inserting a tuple in the hash table, then possibly there is a set of identical pages.

One page is selected to be shared and one to be reclaimed in order to eliminate the redundancy. Any virtual mappings pointing to the identical pages are updated to point, with read-only permissions, to the shared page. This way a page fault is generated when a VM attempts to write to a shared page. This fault is referred to as a CoW fault. A new non-shared page is then allocated and the content of the shared page is copied to the new page. The new page is given to the VM, which then may modify it. Each shared page must have a reference count to record how many VMs are currently referencing it. This is incremented when setting shares up and decremented when a CoW fault occurs. If this count reaches zero, the VM writing to the page was the only user and the page may be converted back to a non-shared page.

This paper describes the design and implementation of a prototype extension to Xen that is able to do content-based page sharing. This is used to evaluate the viability of the content-based page sharing method and its ability to share pages as well as investigate what shared pages are typically used for. The content-based page sharing method has, at least to our knowledge, not been validated on a platform different from VMware.

The article is organized as follows: Section 2 outlines the architecture and its design goals. Section 3 provides elaborating details on the most interesting parts of the implementation and Section 4 presents experimental memory sharing results from various workloads. Finally Section 5 compares with related work and Section 6 concludes the paper.

2. Architecture

The architecture is lead by a few general design choices:

- The architecture should require no modification to guest OSes. Thus a larger amount of OSes can be supported and the implementation effort is reduced.
- Correctness of the system and isolation between VMs must be preserved.
- Keep the size and number of data structures minimal. There is little use of memory sharing if most of the reclaimed memory is spent on realizing page sharing.
- Defer work to when the system is idle, when possible, to avoid affecting system performance. Setting up shared pages that are torn down momentarily should be avoided.

The memory sharing system performs three distinct tasks: 1) scanning memory for identical pages, 2) scheduling of these tasks and page comparison and 3) setting up page sharing and removing it when a page is written to. These tasks are split into the following components:

Page Hashing (PH): hashes pages and produces a number of tuples consisting of *candidate machine addresses*, i.e. pairs of pages that are eligible for sharing.

CoW Sharing (CS): sets up and tears down shared pages by altering any virtual mapping that uses the page. Shared pages are mapped using CoW mappings, hence the name.

Reference Manager (RM): when the system is idle the component schedules page hashing. When candidate pages are found, the component examines these through a bitwise page comparison. If the pages are identical, the CoW sharing component is activated. Finally it handles reference counting for all shared pages to determine when the pages are no longer used by any VMs. Thus, this is the main logic of the system.

As the content-based page sharing patch is based on Xen, we follow the design of Xen and add our components as illustrated in Figure 2 on the next page.

The components and their designs are described in the following subsections.

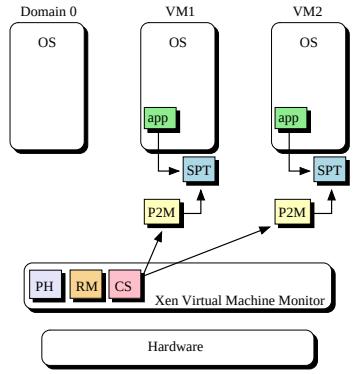


Figure 2. The simplified Xen architecture extended with our components. Our CoW Sharing (CS) component sets up page sharing in the Physical to Machine (P2M) mappings. These changes are then propagated to the corresponding Shadow Page Tables (SPTs).

2.1 Page Hashing

The design of the page hashing component relies on two entities: a hash function and a hash table using open addressing to resolve bucket collisions. Each entry in the hash table consists of page address and its hash value.

The quality of the hash function refers to three properties, namely it must be fast, ensure a uniform distribution of the hash values produced and finally have a low collision rate. If the hash function does not satisfy this, the performance of the hash table suffers. Hash collisions, i.e. when the contents of two non-identical pages produce the same hash value, are resolved by replacing the old hash table entry, thus in effect overwriting it. The probability of this happening is sufficiently low and the worst case is that a few sharing opportunities are missed. As these pages will be scanned again later, it is probable that they will be found if they actually are shareable.

The hash table is used as a temporary data structure during each scanning round, i.e. it contains no state that cannot be discarded at any given point. One page scanning round is when all the intended pages have been scanned. The hash table is reset after each scanning round. The motivation for keeping the hash table temporary is essentially a memory usage versus processor cycles trade-off, where we choose to spend cycles to keep the memory usage low. Consider the alternative where replacing hash table entries, given a machine address, is needed to keep the hash table up-to-date across several page scanning rounds. As the hash table is indexed by a given hash value, this is not possible. This is needed when a page changes content and thus produces a new hash value. Thus replacing entries given a machine address requires a back pointer that maps pages to their hash table entries. Therefore by discarding the information in the hash table between scanning rounds we avoid having to remove entries or keep them up-to-date.

2.2 CoW Sharing

The CoW Sharing component has two tasks: setting up page sharing and tearing this down. All virtual mappings that point to a set of identical machine pages must be found and replaced with mappings that point to a shared page to accomplish this.

One option is to search through all address spaces of all processes in the VMs that own the pages, to replace the mappings. This requires knowledge about data structures in the VMs, which is not directly accessible to the VMM. This is also known as the semantic gap between the VMM and the VM[15, 16]. Also this is best performed from within the guest OSes, which violates our design goal about not modifying the OSes.

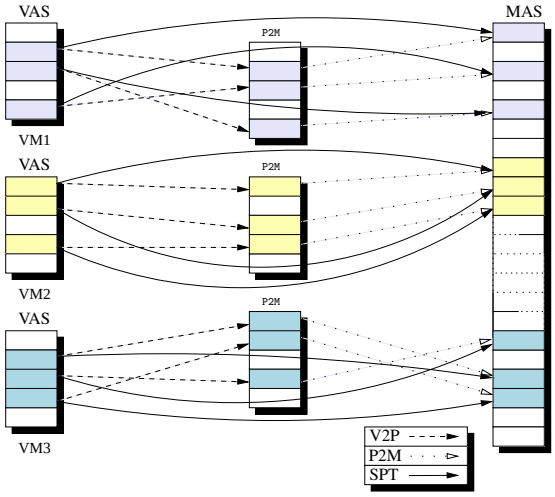


Figure 3. The address spaces of three VMs. The arrows represent the following mappings: Virtual to Physical (V2P), Physical to Machine (P2M), and Shadow Page Tables (SPT).

Another option is to use Xen to provide a level of abstraction between the normal virtual to machine mappings and use *shadow page tables*. This is illustrated in Figure 3. This scheme creates a new address space, which is referred to as the "physical" address space, as this represents what the VM believes is the machine memory resources. The mapping from physical to machine is denoted as the P2M.

A shadow page table is a temporary cache that is generated by collecting virtual to machine mappings from the guest OSes page tables[1] and exchanging the machine address corresponding to the mapping in the P2M table[17, 12]. This implies that there is a shadow page table for every process in every VM. Any changes in either the page tables of the guest OSes or the P2M table must be propagated to the shadow page tables. Attempts made by the guest OS to change the current page table (by changing the value of the Control Register 3 (CR3)) are tracked. When it happens, the Memory Management Unit (MMU) is set to point to the corresponding shadow page table by the VMM, instead of the page table that it intended on using. Thus it becomes transparent to the VMs that they are using shadow page tables.

By using this scheme there are three sets of mappings to maintain, so the use of shadow page tables comes at a cost, both in terms of the memory used for the P2M and the shadow page tables, but also in terms of performance spent on maintaining the different mappings. The use of shadow page tables thus comes at a cost, but with them, it is possible to set up sharing by modifying the P2M and shadow page table mappings for a given page, thus achieving our goal of not modifying the guest OS. It is expected that hardware support for shadow page tables in Hardware assisted Virtual Machine (HVM) processors[18, 19] will mitigate the performance costs[12].

2.3 Reference Manager

A number of policies are handled by the reference manager. The first is choosing which pages to scan and which to disregard. Ideally only the pages that are eligible for sharing are scanned. We leave this choice up to the user, but generally find that scanning only pages in use by VMs is the most effective, as this avoids unused pages. Other options include scanning continuously through the

[1] The guest OS regards these as machine addresses, but they are actually addresses in the physical address space.

entire machine address space or a pseudo-randomized permutation of this.

Another policy is when to establish the sharing between candidate pages. The page hashing component passes candidate pages to the reference manager component via a page sized buffer. This implies that further exploration of the candidate pages may be deferred for an arbitrarily long period of time. This way we can make bulk updates and avoid setting up shared pages just to tear them down momentarily. This also means that the contents of a page may very well have changed since it was first hashed, so a page comparison is needed. As two identical hash values alone cannot determine that two pages are identical, page comparison is needed anyway to ensure correctness. It is important that the atomicity of the page comparison, and the operations following it, are ensured. The contents of the pages must not change and it must be ensured that the page is either shared or not.

Finally it keeps a reference count for all shared pages in order to prevent memory leaks. When this reaches zero the page needs no longer to be shared.

2.4 Security and Isolation

The content-based page sharing method leaves at least one known security concern; a pendant to a similar concern in the mergemem project[20].

Whether a page is shared can be roughly determined by how long a write operation to the page takes, because a write fault to a shared page entails the process of copying the contents of the page to a newly allocated page. This is an overhead that is not present on a write operation to a non-shared page; hence there is a significant time difference.

A malicious VM may use this to construct arbitrary pages and detect whether they become shared. Thus the VM may infer that the information in the page is present elsewhere in the system and use this to brute-force guess at privileged information.

3. Implementation

The system discussed has been implemented as an extension to Xen 3.0.2 patched with Potemkin[17] and is available from our homepage[2]. We use Potemkin, as it provides some of the functionality needed to use CoW mappings of shared pages.

The current implementation is designed for the Intel 32-bit Architecture without Physical Address Extension (PAE) and we focus only on 4KB pages, so super pages are disregarded.

This paper presents a shortened overview of the implementation. For further details and more experimental results, we refer the reader to [21, 22, 23]. In the remainder of this section we discuss details of the system.

3.1 Scheduling and Reference Counts

The VMM only schedules page scanning when at least one processor is idle, to keep performance overhead minimal. Once the page scanning process commences it is allowed to continue until it either completes a round of scanning or a softirq is set as pending. Softirqs in Xen are signals that the system has more important work to do, so in this case the scanning process is terminated and Xen is allowed to respond to the softirqs. This ensures that the system remains responsive to events such as timer and hardware interrupts. Consequently only cycles that would otherwise be wasted are used. Should the system be mostly idle, the VMM is instructed to wait for a predefined period after a scanning has been completed, before scanning again.

Shared pages are owned by a pseudo domain called the clone domain, which enables the reference counting mechanism in Xen to be reused as reference count for the shared pages. Furthermore the page list of the clone domain provides a convenient logical container for the shared pages.

3.2 Reducing Memory Usage

One of the design goals is to minimize the memory used by the implementation itself. Any memory used for the implementation is allocated from the Xen heap, which is also used to store meta data about VMs[17]. This is limited, so any memory used for memory sharing impacts the ability to spawn VMs. This subsection explains how memory usage is minimized.

The implementation uses the 32-bit SuperFastHash[24] function to hash the contents of pages[3]. While architectures with more than 4 GB of memory (2^{20} = 1048576 pages) need a 64-bit hash function, we have found that a 32-bit function has a collision rate that is sufficiently low on systems with less than 4 GB of memory.

For a 32 bit hash function we can calculate the probability of a collision when hashing a page as:

$$\begin{aligned} 1-(1-2^{-b})^{n} &= 1-(1-2^{-32})^{1048576} \\ &\approx 0.0244\% \end{aligned}$$

where b is the number of bits in the produced hash value and n is the number of input pages[13]. We have in earlier experiments found that the actual collision rate was 0.0265% on random data[21], so the collision rate of the hash function seems to correspond with the probability. Hence on a system with 4 GB of memory, the page scanning process should result in roughly 278 collisions and the probability that these pages are shareable is low. So using chaining or other methods to ensure that all shareable pages are found, is not feasible.

As for the size of the hash table, each entry in the hash table consumes 8 bytes (one 32-bit hash value and one 32-bit machine address). Also, the hash table needs extra entries (we choose 10%) to ensure good performance[25, p. 528]. This is unreasonable as this consumes roughly 8.8 MB on a 4 GB machine and the Xen heap is about 10 MB in size.

To remedy this, the hash value space is partitioned into fixed size intervals and these are iterated over once every scanning round. Only the hash values that belong to the current interval are inserted into the hash table. This reduces the size and memory usage of the hash table to a fixed constant. Thus the uniformness of the hash function becomes more important. This is because the hash table has a possibility of overflowing, in case the hash values are unevenly distributed over the current interval of the hash value space.

We have, through empirical means, found 1.7 MB (209.715 entries $\times$ 10% $\times$ 8 bytes) to be a reasonable size for the hash table. If the number of pages in the system is larger than 209.715 (840 MB), then we split the hash value space into n intervals:

$$\begin{aligned} \left\lceil \frac{total\_pages}{(entries \times 10\%)} \right\rceil &= n \\ \left\lceil \frac{2^{20}}{(209.715 \times 10\%)} \right\rceil &= 5 \end{aligned}$$

As exemplified above, 4 GB of memory (2^{20} pages) results in 5 intervals. If the number of pages in the system is less that 209.715, then no intervals are needed and we only use the space needed for the actual number of entries. Using this scheme we may spend processor cycles on hashing the pages several times to reduce the

[2] `http://www.cs.aau.dk/~htj/students/xen-cbps/`

[3] We have assessed that this is in fact a high quality hash function through empirical studies. See [21] for more details.

memory usage. As we only hash pages when the system is idle, this seems reasonable. Note that the reason that this is possible is that the content of the hash table is kept temporary. This is unlike VMware's ESX server that uses a permanent hash table, which is kept up-to-date[11]. Thus they use a percentage of the total amount of pages in the system, where we use a fixed constant.

3.3 Changing Page Mappings

To share and reclaim pages it is, as explained in the previous section, necessary to update virtual mappings of the pages involved. This entails updating the P2M tables of the VMs as well as the shadow page tables pointing to the pages. The global machine to physical (M2P) table, that is used to control which VMs are allowed to access a given page, must also be updated. The modified system has changed the safety checking mechanisms in Xen to allow any given VM to point to its own pages or a shared page, instead of just its own pages.

The algorithm for setting up shared pages follows:

1. **Pause Domains:** the domains are marked as paused to ensure atomicity of the algorithm.
2. **Page Comparison:** the pages are compared to ensure that the pages are bitwise identical.
3. **Allocate Page and Copy Contents:** a new page is allocated and the contents of one of the identical pages are copied to it[4]. Finally the owner of this page is set as the clone domain.
4. **Update P2M Tables:** we add entries for the new page to the P2M tables and remove access to the old pages by invalidating them.
5. **Update M2P Table:** the M2P table must also be updated to reflect the changes, just like the P2M tables.
6. **Invalidate Shadow Page Table Entries:** iterate through all shadow page tables used by the VMs and invalidate entries that point to the old pages.
7. **Reference Count Update:** the reference counts of the reclaimed pages are decremented and as a result the pages are freed to the domain heap. The reference count of the shared page is also incremented.
8. **Unpause Domains:** finally the domains are allowed to resume execution.

The shadow page tables are updated on demand when invalid entries are encountered; therefore it is sufficient to invalidate shadow page table entries instead of resynchronizing all the shadow page tables.

A CoW fault is generated when a VM attempts to write to a shared page. Atomicity is also needed in this case, but this is ensured by the fact that the fault is handled in interrupt context. The fault is handled as follows:

1. **Allocate Page and Copy Contents:** a new private page is created by copying the contents of the shared page to a newly allocated page that is owned by the domain that caused the fault.
2. **Update M2P Table:** the M2P table must associate the new private page with the correct domain.
3. **Update P2M Table:** add the new private page instead of the old page to the P2M table of the faulting domain.
4. **Invalidate Shadow Page Table Entries:** the shadow page table entries that point to the old page also needs to be invalidated on CoW faults.
5. **Reference Count Update:** increment the reference count of the new page and decrement it for the old page.

Optionally if the reference count for a given shared page is one when the fault occurs, then the shared page can be converted back to a non-shared page instead of allocating a new page and copying the content from the old page to the new.

3.4 Implementation Status

The current implementation is intended to be a prototype used for evaluating memory sharing between VMs and is not intended for production use. The efforts needed for it to be of production value is to add proper support for shutting domains down, cloning domains and live migration. While the implementation is able to do overcommitment, it is not able to handle the situation where the system is committed beyond its capacity and more pages are needed. There are a number of possible solutions to this: 1) the least important domain may be shut down, thus freeing its pages, 2) suspend it to disk, 3) swap a number of pages to disk and bring them back into memory when needed or 4) reclaim some pages from each of the domains. Besides these issues, the implementation is fully capable of reclaiming redundant pages from memory.

Finally it was designed for the Intel 32 bit architecture, which means that it presumably is not compatible with any other architecture, and not able to take advantage of the new HVM architectures.

4. Results

In this section we evaluate the content-based page sharing method in regards to its applicability in sharing memory. We also investigate which types of pages are typically found shareable.

The first two experiments illustrate the implementations ability to share large amounts of pages. These are referred to as "best case" workloads, as they exhibit a degree of homogeneity that is unrealistic in normal production grade server environments.

The next set of experiments evaluates the potential for sharing pages on more realistic workloads. The best workloads for this would undoubtedly be real production workloads. Unfortunately we had no access to such, so synthetic workloads will have to suffice. Therefore a number of benchmarks are used to create a significant load on the virtualized system.

Finally the last set of experiments provides an in-depth investigation of what shareable pages are typically used for.

4.1 Experiment Setup and Metrics

The machine used to carry out the experiments was a 2.6 GHz Intel Pentium 4 Northwood with hyperthreading enabled and 2 GB of memory. Each VM had an allocation of 128 MB of memory and used shadow page tables. Domain-0 had a memory allocation of 1 GB and the remaining memory was left free. The VMs were running Debian on a Xen modified 2.6.16 Linux kernel, with a minimum of services executing, e.g. cron and sshd.

The benchmarks used throughout the experiments were Dbench version 3.04 emulating eight clients, OSDB 0.21 with MySQL 5.0.21 and 40 MB datasets, SETI 5.12 and the compilation of a 2.6.16 Linux kernel.

The metrics used in the experiments are:

Shared: is the amount of pages that are set up as shared pages at the moment, i.e. the level of redundancy in main memory.

Reclaimed: is the amount of pages that have been reclaimed from the VMs, i.e. the effective increase in main memory.

[4] This simplifies a number of concerns in the implementation. Alternatively one page could be set up as shared and the other page reclaimed. The allocation and copy operations are not nearly as expensive as synchronizing the shadow page tables, so the gain is minimal.

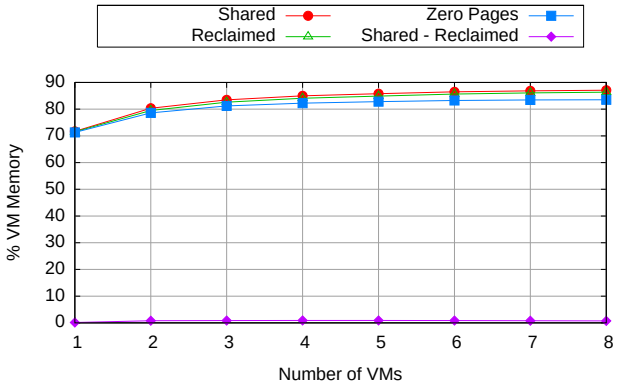


Figure 4. Sharing metrics for idle VMs. The graph shows the metrics for the aggregated measurements from eight incrementally started VMs. The shared pages consist mainly of zero pages as the VMs are idle.

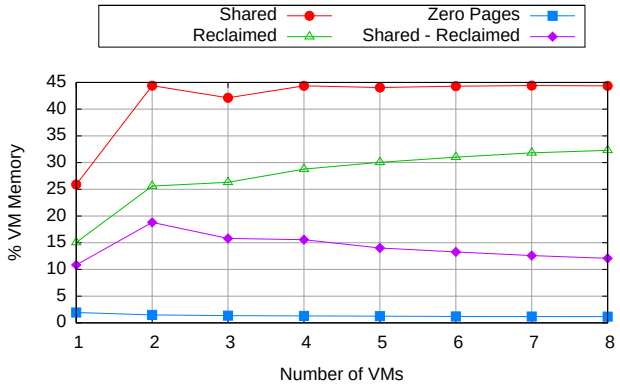


Figure 5. Sharing metrics for VMs compiling kernels. The VMs are started incrementally and the metrics show the aggregated results. The sharing percentage is high and there is a low amount of zero pages.

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

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

4.2 Best Case Workloads

The first experiment investigates the amount of shareable pages on VMs that are completely idle. A total of eight VMs are started incrementally with two and a half minutes delay. The amount of shared pages is measured right before each VM is started.

The results of the experiment are presented in Figure 4. While the amount of shared pages is high, most of the shared pages are zero pages. Furthermore the "Shared - Reclaimed" metric is close to zero, indicating almost no diversity in the shared pages. In the experiment 884 MB of memory was reclaimed.

The purpose of the second experiment is twofold. It illustrates what happens to zero pages when VMs are subjected to a non-idle workload. Secondly it shows the amount of redundant pages on a workload with homogeneous VMs.

VMs are started incrementally, as in the first experiment, but the delay is now 30 minutes and the VMs are executing kernel compiles. This ensures that the VMs are left in roughly the same state when they are done compiling, so the amount of shareable pages should be high.

The results from this experiment are presented in Figure 5. Once VM number two is started, the percentage of shared pages is roughly linear, so we can validate that the amount of shared pages in each VM is roughly the same. Additionally nearly all zero pages have been used, so there is no use of sharing zero pages unless the system consists of idle VMs. When VMs are started they typically consist of only zero pages[5], so the VMM will typically share all the zero pages only to break down most of them as soon as the VM is subjected to a workload. We deem that this is an unnecessary overhead and have therefore disabled sharing of zero pages per default. Notice that the "Shared - Reclaimed" metric has grown significantly higher, indicating that the amount of shared pages is more diverse than in the first experiment. At the end of the experiment a total of 331 MB of memory was reclaimed from the VMs.

The metrics show that there is some sharing after the kernel compilation on the first VM. As there are no other VMs running at this time, this sharing must be caused by redundancy within the guest OS. We refer to this as *internal sharing* and this is further investigated in Section 4.5.

[5] Because the VMM fills the pages with zeros in order to avoid security breaches. This process is known as "page scrubbing".

4.3 Synthetic Workloads

Having examined sharing under good conditions, we turn to more realistic workloads. We use benchmarks as these are easy to deploy and show how content-based page sharing behaves when subjected to substantial processor loads. These workloads are not representative of real-world workloads, but they still illustrate how the virtual memory mechanisms of the kernel work.

In the remaining experiments zero pages are disregarded. The shared pages metrics is the most interesting metric (it shows the redundancy in main memory), so the others are disregarded.

In this experiment four VMs are started concurrently. After the VMs are started, they are left idle for two minutes to ensure that all system services have been started. Hereafter benchmarks are started and allowed to run until they terminate.

We run four different experiments, where the VMs are running the same benchmark. The first is the kernel compilation benchmark, which is characterized by a large number of small files being brought into memory. OSDB is a database stress test, which executes a large number of queries on a MySQL database. The Dbench benchmark emulates the load on a file server generated by eight network clients. The SETI benchmark is a processor intensive workload. A fifth experiment consists of four VMs, where each VM is executing one of the four different benchmarks. This workload is referred to as the Mixed workload. Note that the operations performed by the Dbench benchmark are derived from the same dataset and the kernel compilation uses the same version of the kernel source code. Except for this, the benchmarks are using distinct datasets. The kernel compilation takes roughly 40 minutes, while the other benchmarks run for 70-80 minutes. The amount of shared pages is measured once every minute.

The results of the experiments are presented in Figure 6 on the next page. The figure shows that the number of shared pages range from as low as 2% to 30%. 5% (roughly 25MB of 512MB total memory) appears to be the average amount of shared memory. Note that after a while, the number of shared pages on the Mixed workload rises to match the 5%. This is interesting as this workload is the most heterogeneous workload examined here and should exhibit the lowest degree of redundancy. We remind the reader that the VMs used during these experiments were only running

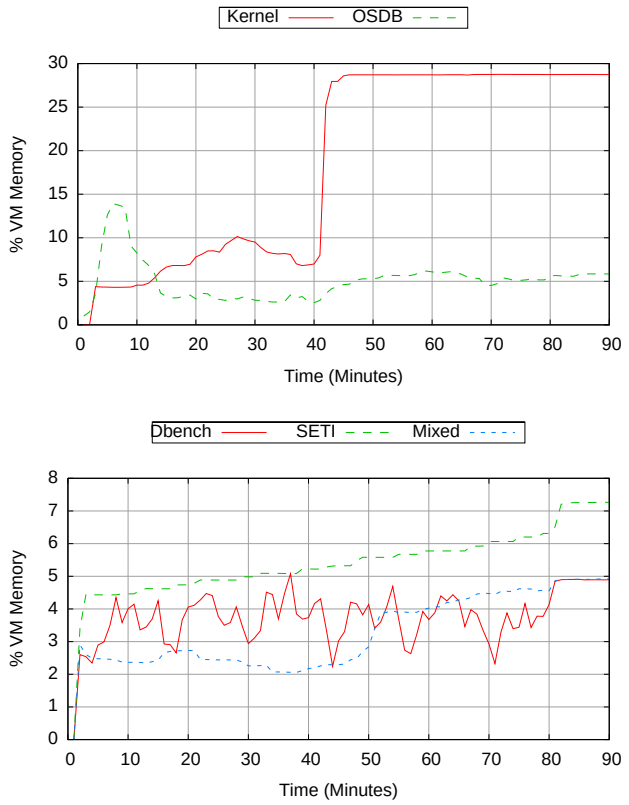


Figure 6. Shared pages as a percentage of the total number of pages for four VMs on different workloads. The results are separated into two graphs to increase clarity.

minimal system services, so real-world workloads would probably yield larger page sharing rates.

4.4 Usage of the Shareable Pages

The last experiments investigate what the shareable pages found during the previous experiment were used for by the guest OSes.

The VMM does not record enough information about the usage of pages by guest OSes to sufficiently determine what shareable pages are used for. So *VM introspection*[26] is applied, i.e. the VM abstraction and kernel is used to bridge the semantic gap by returning the contents of data structures related to the shareable pages.

The machine addresses of the candidate pages are sent by the VMM to the VMs owning the pages, instead of setting the pages up as shared. The VMM does this by filling a buffer that is shared between one VM and the VMM. A kernel module then categorizes each page within one of the following five mutually exclusive categories:

Inode or mapped pages: a page containing data from disk, which is mapped into the virtual address space of one or more processes.

Anonymous pages: mapped into the virtual address space of one or more processes, but the contents of the pages are not backed on storage.

Cache only pages: the pages that are only used by the cache, i.e. not currently used by any processes.

Kernel pages: only used by the kernel.

Cache Only Pages		
File	Count	Pages
.../setiathome.i686-pc-linux-gnu	28742	2180
/usr/lib/i686/cmov/libcrypto.so.0.9.8	9771	680
/bin/bash	8963	612
/usr/lib/i686/cmov/libcrypto.so.0.9.7	7055	826
/usr/lib/libstdc++.so.6.0.8	6649	468
/usr/sbin/sshd	3031	208

Inode Pages		
File	Count	Pages
/lib/libc-2.3.6.so	14358	302
/lib/tls/libc-2.3.6.so	12512	299
/usr/lib/i686/cmov/libcrypto.so.0.9.8	6030	388
/usr/lib/libstdc++.so.6.0.8	5846	380
/usr/bin/boinc_client	4302	301
/lib/ld-2.3.6.so	4099	80

Table 1. The corresponding file names of the pages most frequently found shareable during the SETI workload. The SETI application is frequently found along with standard binaries and libraries. The SETI executable is responsible for 8.5 MB (2180 × 4KB page size) of shareable pages between the four VMs.

Free pages: these pages are marked as free by the guest OS, i.e. they have a page count of zero.

As shareable pages are not set up as shared during this experiment, it is likely that the same pages are encountered multiple times. This is a subsequent experiment, so there are no guarantees that the pages found during this experiment are the same as those in the last experiment, though it is probable. The results in this section are thus only indications of what the pages found in the previous section were used for.

Figure 7 on the following page summarizes the usage of the shareable pages. Most shareable pages fall into the cache only category, indicating that the disk caches of the different VMs contain some of the same files. This is not surprising as the disk cache is allowed to take up the inactive parts of memory. The workloads also revealed that almost no free pages or anonymous pages are found eligible for sharing. The workloads with long running processes have some sharing due to in-use pages containing file-backed application data. On the OSDB workload this reached as much as 34.3%. In the remainder of this subsection, we examine the usage of the shareable pages on the SETI workload in more detail. We use two different counts: *Count* is the number of times that we observed a page in a given category is shareable. *Pages* reflects the amount of shareable pages with unique page addresses. The latter is practical as it shows the amount of pages that have been found shareable during the experiments and reflects how many pages would have been set up as shared. In effect it shows how many pages could be shared if no CoW breaks occur during the experiment.

Table 1 shows the contents of the disk cache that are most frequently found shareable during the SETI workload. The table reveals such libraries as `libc` and `ld`, which makes sense as they are used by the init process. As this process is present on all the VMs, this is guaranteed to be shareable. Generally such libraries are likely to be shareable on virtualized systems, given that the VMs are using the exact same versions of the libraries.

The processes responsible for the most shareable pages are listed in Table 2 on the following page. Note the long running system services, such as the sshd and cron daemons. Such services can be shared on almost any workload as long as the guest OSes are using the same libraries and binaries. On homogeneous VMs it goes that the more system services that are running on every VM,

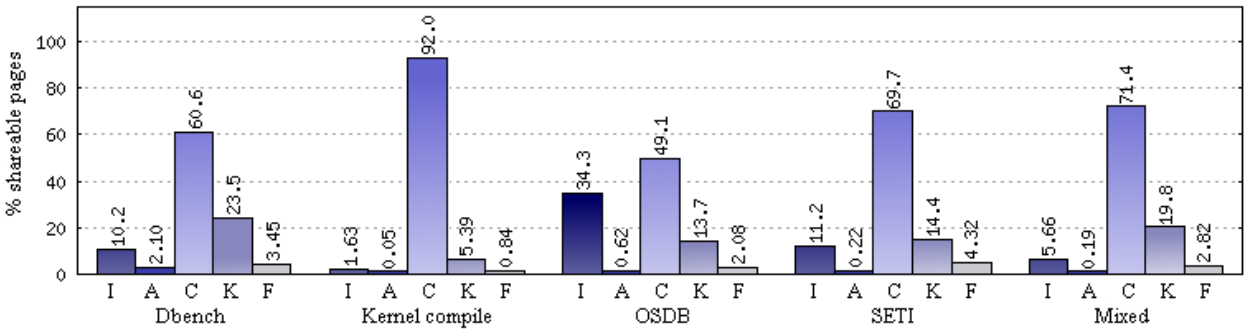


Figure 7. The percentages of Inode Pages (I), Anonymous pages (A), Cache only pages (C), Kernel pages (K) and Free pages (F) found during the different experiments. The pages used for disk cache dominates the shared pages found during the experiments.

Inode			Anonymous		
	Count	Pages		Count	Pages
boinc_client	26254	2153	dhclient	165	18
sshd	6303	678	sshd	135	18
syslogd	4939	558	cron	129	12
cron	4933	549	syslogd	95	11
init	4070	441	init	72	6

Table 2. Top five of processes using shareable pages found during the SETI workload. Besides the `boinc_client`, most of the processes are system services.

Kernel Pages		
Workload	**Count**	**Pages**
OSDB	99442	3810
SETI	45416	4198
Kernel compile	174977	4632
Dbench	164899	4633
Mixed	88182	3903

Table 3. Shareable kernel pages for the four VMs during the different workloads. Count is the number of times a page in the category has been found shareable. Pages is the amount of unique page addresses within the category. The number of unique page addresses is roughly constant on all the workloads, with 4 MB of shareable pages on each VM.

the greater the level of redundancy. Furthermore these processes even result in some anonymous shareable pages.

The shareable pages that are used by the kernel can be further divided into two categories depending on the flags set on the page descriptor: reserved pages and pages used for slab caches[27] (80% and 20% respectively). The reserved pages are pages that only have the reserved flag set on the page descriptor. This means that they are either unusable or only used by the kernel.

Table 3 shows the amount of shareable kernel pages on the different workloads. The amount of unique page addresses are roughly equal on all the workloads, so such an amount of pages can be expected to be shared on any workload where the VMs are using the same kernel version.

		Inode	Anon	Cache	Kernel	Free
Dbench	P	0.0%	12.0%	71.7%	4.40%	11.7%
	C	0	851	5067	311	832
	U	0	246	146	8	264
Kernel	P	0.0%	0.00%	97.4%	0.98%	1.57%
	C	0	3	490722	4973	7954
	U	0	3	5162	47	1512
OSDB	P	0.0%	0.43%	89.7%	1.74%	8.10%
	C	0	134	27928	543	2524
	U	0	6	6122	46	1098
SETI	P	0.0%	0.15%	96.4%	0.37%	3.06%
	C	0	29	17889	69	568
	U	0	3	11	6	42

Table 4. Distribution of shareable pages due to internal sharing. (P) is the percentage, (C) is the amount of times a page was observed as shareable, and (U) is the amount of unique page addresses. Notice the amount of shareable pages in the Cache only category in the OSDB and kernel compilation workloads.

4.5 Internal Sharing

Finally to conclude the evaluation, we investigate what the internal sharing observed in Section 4.2 is used for. Instead of four VMs, only one VM is used. The results of this experiment are shown in Table 4. Here the disk cache is also responsible for most of the shareable pages. The internal sharing on most of the workloads is negligible, except for the kernel compilation and OSDB workloads. A further investigation, presented in Table 5 on the next page, of what the disk cache contains on the kernel compilation workload revealed that the internal sharing was due to the results of the compilation that was copied during the process, so their content end up multiple times in the disk cache. The shareable pages during the OSDB workloads were due to the MySQL database file, which contained a non-zero pattern. These two workloads exemplify how easily identical contents may end in memory within a single VM.

5. Related Work

The concept of memory sharing on virtualized systems is not new. Several virtualization projects have approached it in significantly different ways.

VMware[11] use the content-based page sharing method, much like it is described in this paper. Our implementation differs in a number of ways. The systems we support are limited to 32-bit

File	Count	Pages
.../i386/boot/compressed/vmlinux.bin	100913	932
/usr/src/linux/vmlinux	97771	1170
/usr/src/linux/.tmp_vmlinux2	69715	1169
.../linux/arch/i386/boot/vmlinux.bin	32729	458
.../i386/boot/compressed/vmlinux	32729	458
/usr/src/linux/System.map	13677	189
/usr/src/linux/.tmp_System.map	13675	189
/usr/src/linux/.tmp_kallsyms2.o	2197	43
/usr/src/linux/drivers/net/s2io.o	684	9
/usr/src/linux/drivers/net/s2io.ko	684	9

Table 5. Internal sharing during kernel compilation. Top ten of the file names of pages backing the most commonly found pages due to internal sharing during the kernel compilation on a single VM. The same content is present multiple times in the disk cache.

architectures without PAE, as we use a 32-bit hash function to keep the memory usage low. VMware supports PAE and 64-bit, so they need a 64-bit hash value. We prefer to scan all pages belonging to a particular VM, where VMware uses random scanning. VMware uses chaining to handle colliding hash values, where we discard the colliding values. As explained earlier, these differences are what enable us to keep the memory usage due to data structures low. Finally, as concluded earlier, we avoid sharing zero pages per default.

Content-based page sharing has also been attempted without virtualization in the mergemem[28, 20] project for earlier versions of the Linux kernel.

Disco[29] uses file system meta data from VMs to keep track of which files are in memory. This ensures that a given file is only brought into main memory once in the virtualized system and it is possibly shared between VMs. Thus the memory usage is reduced and the average cost for bringing files into memory is lowered. The approach resembles the disk cache used by most modern OSes, the difference being that it is dealing with interdomain sharing of files, and is not limited to a single OS. The approach is also being implemented for Xen in XenFS[30], where it has been dubbed *interdomain shared cache*. This approach relies on the VMs using the same physical media as backing storage in order to share memory. If this is not the case, the approach of XenFS combined with content-based block-level sharing, as it was explored in [31], should prove effective. As the previous section showed, most of the shareable pages are due to pages in the disk cache, so this should be almost equivalent to content-based page sharing on most workloads. It will however miss the few anonymous pages, the free pages and the pages used by the kernel.

Content-based Buffer Cache[32] is an attempt to avoid the disk access penalty, by enhancing the effective disk buffer size. Like content-based page sharing, it uses a compare-by-hash approach to identify buffer entries and eliminate these, thus increasing the effective size of the buffer.

Another approach was taken by Potemkin[17], which uses the VM abstraction to quickly clone a new VM when needed. By using *delta virtualization*, i.e. a VM only occupies as much memory as its memory differs from the VM it was cloned from, they are able to keep the initial memory footprint low. The memory footprint is bound to grow larger as its memory contents changes. Therefore if the VMs are intended to run for a longer period of time, a combination of both delta virtualization and content-based page sharing could prove effective to keep the memory footprint as low as possible.

6. Conclusion

We examined content-based page sharing, a method first introduced by VMware, as a means of reducing the level of redundancy in main memory. When redundancy is eliminated, virtual machines consume less memory and the memory freed can be used to do overcommitment.

Main memory is scanned to find identical pages. Identical pages can be shared between virtual machines, thus freeing them for other uses. The pages are shared using Copy-on-Write, so attempts to write to the shared pages trigger faults, which cause the virtual machine monitor to create new private copies that may be modified. As long as the interval between creating a shared page and a write operation to that page is sufficiently long, the system may use the freed page for other purposes. If the interval is not long enough, then it may constitute an overhead. This is often the case with zero pages, which are rapidly used on most workloads.

We implemented content-based page sharing on top of Potemkin, a patched version of Xen, and used this to examine a number of different workloads. We found that the method is viable and in general there is almost always something to share. The amount of shareable pages is highly dependent on the workload and the degree of homogeneity in the virtual machines running the workload. We examined the usage of the shareable pages and found that the main cause of redundancy was the disk caches of the virtual machines. The pages used by system services are often shareable between homogeneous virtual machines, i.e. both the executables as well as the libraries used by the services. Thus we were able to conclude that an interdomain shared cache would be almost equivalent to using content-based page sharing on most workloads, because most modern operating systems have unified caches.

The implementation uses shadow page tables to keep the virtual machine unaware of the changes we apply. Shadow page tables are a second level of paging that are maintained by the virtual machine monitor, thus causing a performance overhead. The cost of doing content-based page sharing is twofold, one is the cost of scanning for shareable pages, and the second is breaking shared pages. As the implementation only scans for pages when the system is idle, only otherwise wasted processor cycles are used for this. Breaking sharing is done as a part of the write operation to a shared page and this is bounded by the shadow page table implementation. Hardware assisted virtualization technology that supports shadow page tables, may however mitigate this cost.

Acknowledgments

We would like to thank Gerd Behrmann and Henrik Thostrup Jensen for their valuable input on both the implementation and their persevering efforts and insights on our writings.

Also we would like to thank the Xen team and in particular Keir A. Fraser for answering questions about Xen, as well as for providing Xen as open-source, without which our work would not have been possible. A great thanks goes out to Michael Vrable for providing us with the Potemkin source code and answering questions about it. Finally we would like thank Jacob Gorm Hansen for suggesting improvements to early versions of this paper.

References

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

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

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

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

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

[6] 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`.

[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 *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, May 2005.

[9] 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`.

[10] 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.

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

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

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

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

[15] 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, Washington, DC, USA, 2001. IEEE Computer Society.

[16] 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.

[17] 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.

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

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

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

[21] 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.

[22] 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`.

[23] 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.

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

[25] Donald Ervin Knuth. *Sorting and Searching*, volume 3. Addison Wesley Longman, 1998.

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

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

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

[29] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. pages 143–156, 1997.

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

[31] 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`.

[32] 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.