A Comparison of Hardware Virtual Machines Versus Native Performance in Xen

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

The recent advances in hardware supported virtualization has recently gained popularity, partly due to the ability to run unmodified operating systems. This paper evaluates one of the new hardware virtualization supporting processors by comparing its performance during various workloads to the performance when running in native mode, i.e. without virtualization.

We find that hardware supported virtualization is capable of achieving near to native performance on some of the examined workloads, but also find some overheads on the remaining workloads. These overheads are inherently caused by the underlying Xen architecture, especially the emulated I/O devices and the use of shadow page tables.

When pinpointing the weaknesses of the hardware virtual machine implementation we found overheads of up-to 90% loss in hardware virtual machines compared to native performance.

1 Introduction

The recent attention to virtualization[9, 23, 24, 6] on the x86 server market, has brought a revival of well-understood virtualization techniques as well as innovating new approaches.

Full virtualization, i.e. virtualizing a system using a full trap-and-emulate scheme, is a well known technique that dates back 40 years[9]. Hardware Virtual Machines (HVMs[1]) is a recent incentive from AMD[3] and Intel[26] that is returning to the roots of virtualization, i.e. usage of a modified trap-and-emulate scheme. In order to fully virtualize a processor, it must have an instruction set where all privileged instructions cause a trap when executed by a Virtual Machine (VM)[20]. Instructions that does not adhere to this are called sensitive instructions, another way to put it is that the set of sensitive instructions must be a subset of the privileged instructions. Another technique to workaround this is to scan the binary code of guest Operating Systems (OSes) at runtime to patch the sensitive instructions. This is known as binary translation[2]. The Intel Architecture (IA-32) does not satisfy the previous demand[22], so the producers of the HVM chips had to modify the instruction set to respect this. Furthermore they extended the processor with two modes, root and non-root. The Virtual Machine Monitor (VMM) runs in root mode, i.e. it is allowed to run all privileged instructions. A VM is executed in non-root mode, which causes all privileged instructions to trap[26].

[1] With HVM we primarily mean Intel's VT-x. AMD's SVM has functionality that is identical to or similar.

Another popular approach is paravirtualization[29]. Paravirtualization, as implemented in Xen[4, 8], avoids the overhead of a full trap-and-emulate scheme, by altering the semantics of the underlying processor architecture and the OS to make explicit calls to the VMM in order to avoid the expensive trap-and-emulate or binary-translation schemes. Paravirtualization however comes at a cost, namely in the development and maintenance of modified OS kernels[14, 15]. Any approach to virtualization that allows the use of unmodified OSes should be preferable to many usage scenarios. This favors the HVM technology compared to paravirtualization, as paravirtualization typically only supports a limited number of OSes. It has however been pointed out that the first generation of HVM has some performance overheads, specifically the lacking hardware support for virtualization of the Memory Management Unit (MMU)[2]. This result was produced on a prototype VMM produced by VMware specifically for the purpose of evaluating HVM.

We deem that it is worthwhile to investigate the previous claim asserted by an independent source on another VMM, namely the Xen VMM. This has been chosen due to two factors: it supports HVM[21] and secondly because it is open-source, so micro benchmarking can be done by inserting code into the VMM.

This is not the only motivation for this paper. We have previously worked on implementing content-based page sharing[28] in Xen [10, 11, 12, 13]. This was however limited to the IA-32 architecture with paravirtualized kernels. A benefit from the study is thus that we have the chance to evaluate the required effort of porting content-based page sharing to a HVM VMM. Furthermore we previously found that parts of the shadow page table implementation in Xen was expensive to use[11]. This has been deprecated by a complete redesigned and rewritten version, which we would like to assess the efficiency of.

The remainder of this paper is structured as follows: in Section 2 we sum up important issues in the HVM implementation in Xen. In Section 3 we carry out the evaluation and present the results. In Section 4 we discuss related work and finally in Section 5 we conclude the paper.

2 Hardware Virtual Machines

We start by providing an overview of the HVM implementation in Xen. This is divided into four parts: Basic Input Output System (BIOS) emulation, processor virtualization, MMU virtualization and I/O virtualization.

Like on physical machines, OSes on HVMs also need a BIOS to provide boot and run-time services. In Xen, HVMs use the functionality from Bochs, an open-source IA-32 architecture emulator, to emulate a BIOS [7].

The processor is, as mentioned, virtualized by the means of hardware extensions. Most significantly the processor is extended with two modes called root and non-root. The VMM is executed in root mode, where all operations are allowed. VMs however are run in non-root mode, where all privileged instructions cause a certain type of traps called VM exits. Privileged instructions include instructions such as MOV to/from Control Register 3 (CR3), HLT and INVLPG. Furthermore page faults and external interrupts, not related to the currently running VM, also cause VM exits. On a VM exit, the VMM determines the cause of the trap and handles it. Afterwards the VM may be resumed by an VM entry operation. [21, 7]

The cost of VM exits are monumental to the performance of trap-and-emulate type VMMs, so these should be kept to a minimum [2]. Because this technique is well understood and used, most VMMs generally perform well with processor intensive workloads. System calls generally is a concern in this context, because most should be executed without VMM intervention.

The MMU is currently virtualized by the use of software managed shadow copies

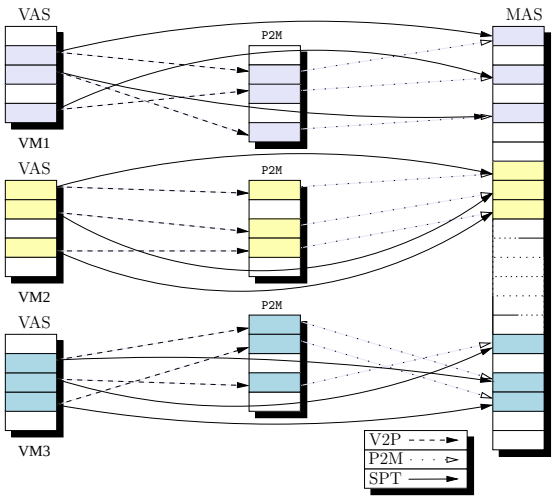


Figure 1: The address spaces of three VMs. The arrows represent the following mappings: Virtual to Physical (V2P), Physical to Machine (P2M), and Shadow Page Tables (SPT).

of memory structures. These are necessary to ensure isolation between VMs, that may otherwise create mappings to arbitrary memory pages, if not confined to the pages allocated to the VMs. In particular, an abstraction, called the physical address space, over all virtual address spaces in VMs is created, as shown in Figure 1. Addresses in this space have a one-to-one mapping to machine pages, while the addresses in guest page tables may have a many to one mapping to physical addresses.

The new address space abstraction is however not usable by the MMU, so transitive translations from guest page tables (virtual to machine) must be handled in software by the use of a temporary shadow page table. Xen has approached this translation from several different angles, the first was to save the old shadow page tables, track updates to the guest page tables and then resynchronize them when needed[7]. A more recent approach was to track all updates to guest page tables and update the shadow page tables immediately. Currently it is believed that a hybrid approach should yield the best performance. The flexibility of the shadow page table abstraction can be used to optimize the virtualized system for certain usage scenarios, e.g. memory sharing, as demonstrated in [27, 28, 11, 13]

Virtual devices are emulated through a QEMU device manager running in the privileged VM. Figure 2 on the next page shows the information flow when I/O is performed in HVM. When a VM executes an I/O operation, it causes a VM exit. The VMM then blocks the VM and extrapolates that the VM exit was caused by an I/O operation. The request is then relayed to the device manager in the privileged VM by the VMM. The device manager then carries out the I/O operation. Once the operation is done, the device manager responds to the VMM, which forwards the result to the VM, unblocks it and resumes execution of the VM.

An important note is that the HVM Xen team has changed the IDE DMA emulator from a synchronous mode where a VM waits for the disk I/O request to an asynchronous mode, where disk I/O is handled in a new thread[7].

Another approach to virtualizing I/O, that probably scales better than emulating devices, is to modify the guest OS to use front- and back-end drivers[25]. To achieve this, Xen provides both Virtual Block Devices (VBD) and Virtual Network InterFaces (VNIF) [7].

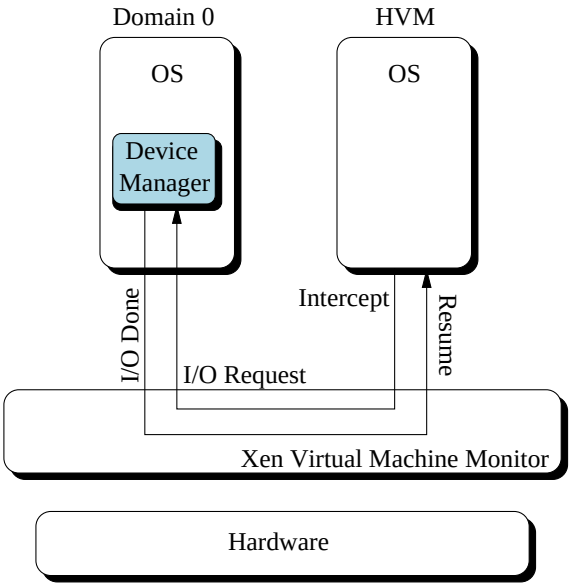


Figure 2: I/O requests from a guest OS on Xen using HVM. A Device Manager (DM) runs inside the privileged VM, which handles requests from guest OSes running on HVM.

3 HVM Evaluation

The first part of this section describes the benchmarking setup and the last part presents and discusses the results. A top-down approach is used, i.e. first a general application benchmarking is applied to test all OS subsystems together. Then benchmarks that tests specific subsystems are used in order to locate any specific overheads there might be. Finally, micro benchmarks are applied to the shadow implementation, in order to investigate whether this might be some of the cause for any bad performance in the memory subsystem.

3.1 Setup and Methodology

We use the latest stable version (Xen 3.0.4 compiled from source) of Xen for the benchmarks. The evaluation was carried out on a VT-x supporting Intel Core 2 Duo E6300 1.86 GHz processor with 2 GB of memory and a a 100 Mbit network interface.

Each VM was allocated one virtual processor, 1 GB of memory and 10 GB of disk space. The native (non-virtualized) OSes were limited using OS configurations to have the same resources. The privileged VM was allocated two processors and the remaining memory.

Table 1 on the following page shows a list of the OSes we tried to use on HVM. Surprisingly, for a virtualization approach that should support any OS, only a few of the tested OSes worked under HVM.

Debian Linux 2.4 worked fine in HVM, but unfortunately the 2.4 Linux kernel did not support the hardware in native mode. Furthermore in HVM Windows Vista, there was a known bug in the network driver (Xen uses QEMU to emulate a rtl8139 network interface), which caused it to drop all incoming packages.

It should be noted that we kept configuration to a minimum. This means that HVM benchmark scores may very well be improved through customizations that favor the HVM architecture. For a given OS, the native and the HVM OS were

Operating System	Install	Preinstalled
Debian 3.1 (Linux 2.4)	**successful**	**successful**
Debian 4.0 (Linux 2.6)	**successful**	**successful**
FreeBSD 5.3	unsuccessful	not tested
FreeBSD 5.5	unsuccessful	not tested
FreeBSD 6.2	unsuccessful	unsuccessful
NetBSD 3.1	unsuccessful	not tested
OpenBSD 4.0	unsuccessful	not tested
Minix 3.1.2a	unsuccessful	not tested
Solaris 10 11/06 x86	unsuccessful	unsuccessful
Windows 2003 Server ENT	unsuccessful	not tested
Windows XP SP2	**successful**	**successful**
Windows Vista	**successful**	**successful**

Table 1: List of tested OSes. The Install column refers to whether it worked during the installation procedure and the Preinstalled column means that it was run after being installed by other means (usually through QEMU).

configured identically, except for the different virtual devices imposed by the HVM implementation in Xen.

The software used during the experiments was as follows:

OSDB: A database benchmark. Version 0.21 with Mysql 5.0.32 and 5.0.41 (for Debian 4.0 and Windows XP respectively) was used.

SPECweb99: A web server benchmark, where Apache 2.0.59 was used.

Scimark: Version 2.0 with Sun JDK 1.6.0_01. The benchmark runs a number of computations from the Java Virtual Machine (JVM).

Bashmark: A mixed variety of system tests. Version 0.6.2 was used.

Nbench: A variety of processor intensive computations. Version 2.2.2 was used.

NetIO: A TCP and UDP benchmark, that examines throughput with packages of different sizes. Version 1.2.6 was used.

Aim9: A large UNIX benchmark suite. Version 9.1.10 was used.

Kernel compile: The compilation of a 2.6.21.1 Linux kernel.

MemoryMotion: An application written by us, custom made to exercise page tables. This executes a large number of processes sequentially, which allocate large amounts of memory, writes to it and then frees it.

The results presented in the actual benchmarking are the median of a number of experiments, usually three or seven, depending on the time spent on executing the benchmarks.

The Scimark results presented here is a combined score, because the scores differ so little. Scimark is composed of six different metrics. The Scimark benchmark is run seven times and the results are used to calculate the median of each of the metrics. We then calculate a relative measure (the percentage) from the median on HVM to the median on native for each of the metrics. Finally we take the mean of all of the percentages of the different metrics, to end up with a single score. This results in a standard deviation between 0.18 and 0.39 (of the percentages), so this seems reasonable.

3.2 Comparing HVM with Native Performance

The benchmarking is started by a set of experiments that show the performance of HVM under various conditions. To favor the HVM we start with a workload, Scimark, that is primarily processor intensive. Then benchmarks that should exhibit some degree of everyday server use, are applied. Additionally on the Debian systems also run a kernel compilation. Finally the MemoryMotion stress test is run. This should expose the worst sides of the shadow implementation in HVM. The experiments are run on Debian 4.0 and Windows XP systems.

The results of these experiments are shown in Figure 3 on the next page, where the HVM benchmark scores are normalized relative to the score on native (which is 100%). The first experiment, the JVM benchmark Scimark, reveals no surprises. The benchmark is processor intensive and does not require much I/O, thus no VM exits are required. Therefore it is able to achieve very close to native performance.

The next experiment, SPECweb99, measures the throughput from four remote web clients requesting a mixture of static and dynamic web contents from an Apache web server. The experiment is run with three different workloads, low (16 connections), medium (50 connections) and high (84 connections). These values were chosen due to the load on Debian native, i.e. 84 connections was the point where Debian native stopped having conforming connections. The light workload reveal some overhead on Windows XP (6.33%), but nothing significant compared to the other workloads. On the medium and heavy workloads, the throughput drops drastically. Furthermore it was observed that the HVMs stopped having conforming connections at the medium workload, while both of the native machines could handle this load. As for the response time, this also increases drastically in HVM. With 84 connections on Windows XP HVM, the response time is 1283 micro seconds (596 native) and on Debian, this goes from 321 micro seconds native to 688 micro seconds in HVM. If this is compared to the loss on the figure, then it can be seen that the factor between response time and throughput is roughly the same.

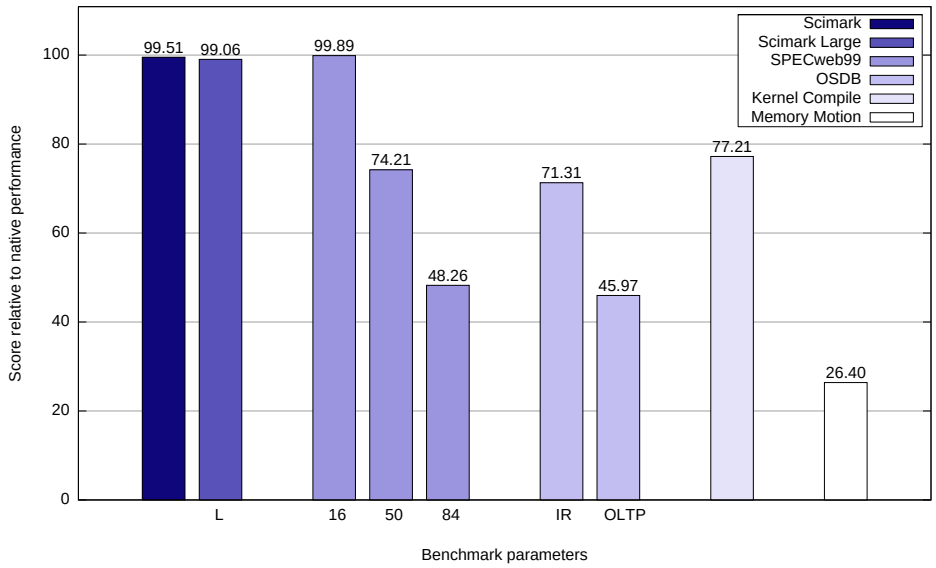
The third experiment, OSDB, consists of two parts: Information Retrieval (IR), mostly retrieving existing data, and On-Line Transaction Processing (OLTP), which also updates the database with new content. The experiment was run by a remote connection on a closed local area network. The overheads found resemble those of the SPECweb99 benchmark, except from the OLTP on the Windows XP HVM. It has a remarkable speed-up of 316.05% compared to native Windows XP. This will be further investigated later in the section.

Next the compilation of a vanilla Linux kernel on Debian was timed. The overhead again closely resembles that of IR on OSDB, with 22.79% drop in performance. Common for these is that they both are I/O intensive.

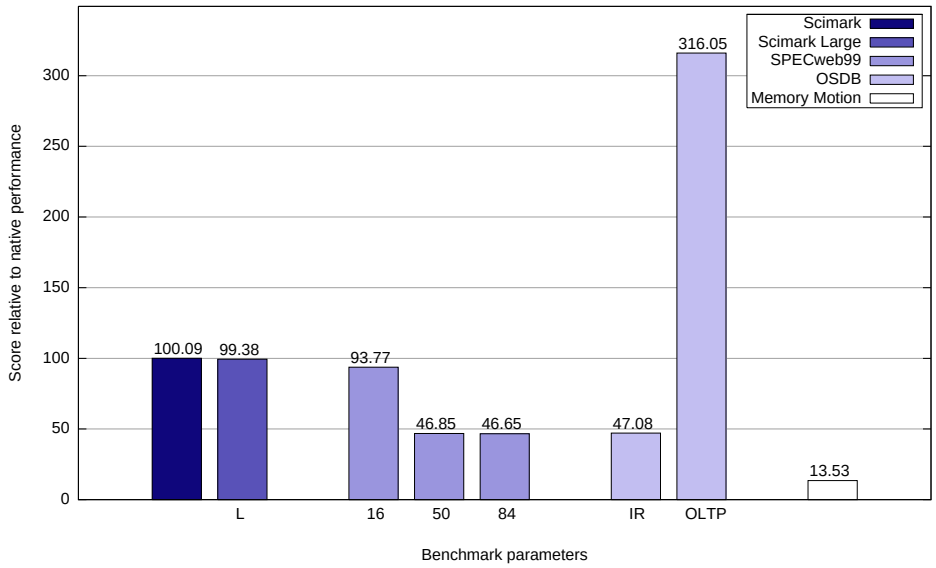
The final overall benchmark is our own MemoryMotion, which stresses the system with an extreme amount of process creation and memory allocation. On native Debian the running times were roughly 33 seconds, while the HVM took 125 seconds, thus almost giving a factor of 1:4 in difference. The Windows XP version ran for 46 seconds natively, while the HVM took 340 seconds (1:7). This exemplifies the overhead of using shadow page tables to virtualize the MMU.

To summarize, most of the benchmarks revealed overheads of some sorts. To localize these, we carry out a number of macro benchmarks, where each one is designed to test a specific subsystem of the OS. We start with a further evaluation of the virtualized processor, just to rule out that Scimark may not be representative. Afterwards we investigate I/O performance and end the evaluation with an investigation of the memory subsystem.

Besides the subsystem specific benchmarks, we also run Aim9, a UNIX benchmark that is good at evaluating the costs of system calls. As some of these are privileged instructions and thus cause VM exits, this should also give a good all-



(a) Debian



(b) Windows XP

Figure 3: Overall benchmark results. The results from the different benchmarks on Debian 4.0 (3(a)) and Windows XP (3(b)).

Test	Native	HVM	Loss
Numeric Sort	814	812	0.26%
String Sort	112	111	0.53%
Bitfield	387860000	386170000	0.44%
Fp Emulation	99	98	0.52%
Fourier	21492	21404	0.41%
Assignment	28	28	0.47%
Idea	4063	4050	0.31%
Huffman	1754	1744	0.56%
Neural Net	32	32	1.05%
Lu Decomposition	1249	1242	0.55%

Table 2: Results from **Nbench** on **Debian**. Very near to native speeds are achieved.

Test	Native	HVM	Loss
add_short	2360006	2344818	0.64%
add_int	3194935	3180409	0.45%
add_long	3194935	3180409	0.45%
add_float	1242392	1236381	0.48%
add_double	1242185	1235888	0.51%
mul_short	779845	774552	0.68%
mul_int	891619	886808	0.54%
mul_long	892263	887512	0.53%
mul_float	745475	741952	0.47%
mul_double	745351	741705	0.49%
div_short	116151	115596	0.48%
div_int	93194	92728	0.50%
div_long	93178	92728	0.48%
div_float	319443	317891	0.49%
div_double	319496	317841	0.52%

Table 3: Results from the **arithmetic** benchmarks in **Aim9**. There are almost no VM exits, so very close to native speed is achieved.

round overview of the HVM costs.

3.3 Processor Usage

To assert that HVM performs well on processor intensive workloads, we apply the Nbench, Bashmark and Aim9 benchmarks.

Table 2 presents the results from the Nbench benchmark. As with the Scimark benchmark, the loss is very little. For the record, Bashmark shows the same tendencies, but the results are omitted here.

Table 3 and 4 on the next page show additional results from Aim9. The pure arithmetic operations cause no apparent overheads. The benchmarks in Table 4 execute as many invocations of a predefined function as possible. A function call pushes the return address and function parameters to the process stack. The stack is an in-memory page, so if there were any decreases in memory access (read and write), then this benchmark should probably show so. There is however no noticeable overheads. Such workloads can therefore favorably be virtualized using HVM. On the other hand, such workloads would probably not cause any significant overheads in any other virtualization technology, because most instructions may be executed without interference from the VMM.

Test	Native	HVM	Loss
fun_cal	309068488	308070109	0.32%
fun_cal1	329007565	328053573	0.29%
fun_cal2	207154807	201738064	2.61%
fun_cal15	70001199	69796467	0.29%

Table 4: Results from the **function call** benchmarks in **Aim9** with different numbers of parameters. Again, these cause few VM exits.

Test	Native	HVM	Loss
disk_rr	264189	271946	-2.94%
disk_rw	233184	239792	-2.83%
disk_rd	1442916	373720	74.10%
disk_wrt	363167	379521	-4.50%
disk_cp	276558	163102	41.02%
sync_disk_rw	1001	1967	-96.50%
sync_disk_wrt	1384	623	54.99%
sync_disk_cp	1340	581	56.64%
disk_src	108155	101673	5.99%
link_test	110416	103541	6.23%
creat-clo	365811	237370	35.11%
dir_rtns_1	1541152	1524325	1.09%

Table 5: Results from the **disk and file system** benchmarks in **Aim9**.

3.4 I/O Performance

This part of the evaluation focuses on the I/O performance in HVM. In particular the results of the I/O Aim9 specific tests are discussed as well the results from Dbench and NetIO.

Table 5 shows the disk I/O related tests in Aim9. The results clearly demonstrate the nature of the underlying I/O devices, namely those emulated by QEMU, as described in Section 2. The speed up in `sync_disk_rw` is likely due to the change from synchronous to asynchronous in the disk device manager. On the native machine, the application is forced to wait for the write request to complete, where on HVM, the request is buffered and the VM allowed to resume while the privileged VM is writing the changes to disk. This is also most likely the explanation for the speed up we previously saw in the OSDB OLTP benchmark on Windows XP.

Figure 4 on the next page, which shows the results from running Dbench with a varying amount of clients, helps to determine whether the overheads in the previous table may be amortized. At least for the disk I/O operations that are typical for the Dbench workload, there seems to be a constant overhead of $20 - 30\%$. This seems to correspond nicely to the loss of performance in the kernel compilation, which also is rather I/O intensive.

As for network I/O, Table 6 on the following page reveals some minor overheads, but nothing really significant. Furthermore it should be noted that both `tcp_test` and `udp_test` operate on the local host, so no network traffic is actually transmitted.

Figure 5 on the next page on the other hand shows a different result. This is the result of the NetIO benchmark, that uses an actual client and server setup. Notice the constant loss in TCP receive throughput and the immense loss in UDP receive throughput.

A possible explanation for the decreased performance on I/O, could be the world switch introduced by the switch to the privileged VM on all I/O. This causes the Translation Lookaside Buffer (TLB) to be flushed, which is an inevitable conse-

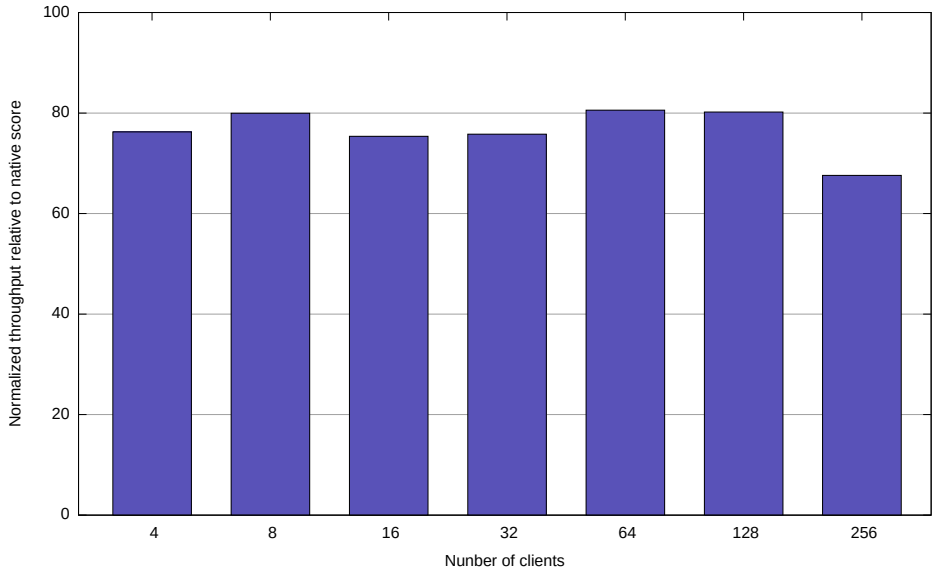


Figure 4: Normalized throughput from running the **Dbench** benchmark with a varying number of clients.

Test	**Native**	**HVM**	**Loss**
tcp_test	231832	224085	3.34%
udp_test	415804	402562	3.18%
fifo_test	851189	740826	12.97%
stream_pipe	762617	733387	3.83%
dgram_pipe	705469	683063	3.18%
pipe_cpy	986238	955978	3.07%
shared_memory	628552	552635	12.08%

Table 6: Results from the **IPC** benchmarks in **Aim9**.

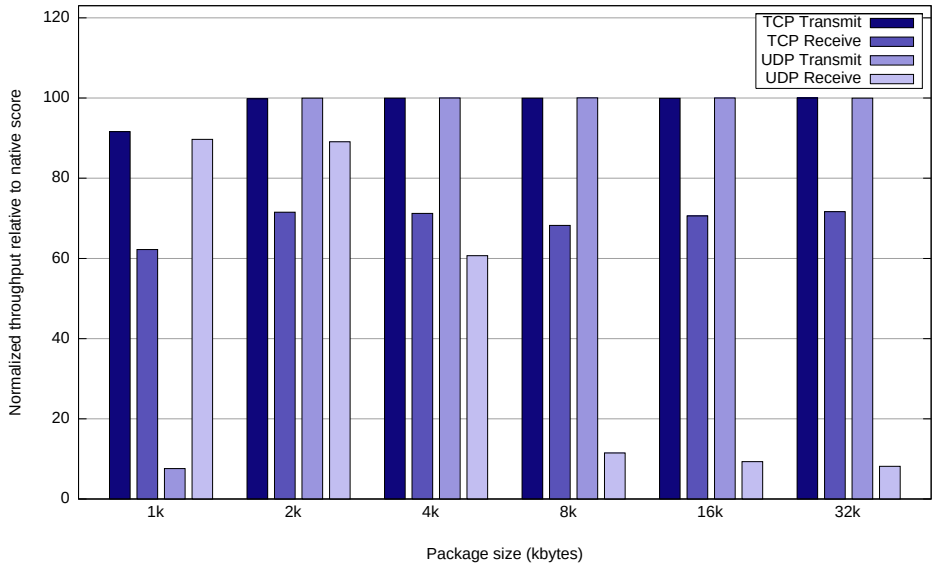


Figure 5: Normalized throughput from running the **NetIO** benchmark. Notice the severe reductions in the throughput.

Test	Native	HVM	Loss
jmp_test	20341543	20235671	0.52%
ram_copy	1685692168	1676916422	0.52%
num_rtns_1	191926	189058	1.49%
trig_rtns	883019	876915	0.69%
mem_rtns_1	3352382	3338330	0.42%
string_rtns	2647	2628	0.72%
matrix_rtns	5754215	5723657	0.53%
array_rtns	1667	1659	0.48%
sort_rtns_1	1083	1057	2.40%
signal_test	454740	453115	0.36%
misc_rtns_1	26743	11323	57.66%

Table 7: Results from the **library** benchmarks in **Aim9**. Notice the loss on `misc_rtns_1`.

quence of the I/O architecture in Xen (and not necessary in native mode). Another explanation is the penalty taken when memory mapping files, because this requires page table updates, that must also be propagated to the shadow page tables used by HVM. Most likely both are part of the cause.

3.5 System Calls

Now the results of the different library benchmarks in Aim9 are shown. As can be seen from Table 7, at a first glance, the only interesting result is that of `misc_rtns_1`. This particular benchmark imposes an intensive workload by executing a variety of inexpensive system calls, e.g. `getuid()` or `getpid()`, and a variety of expensive system calls, e.g. `getpwuid()` or `getgrgid()`. Despite being system calls, the first set of system calls do not access privileged information, only kernel data, and thus do not trigger VM exits. The latter system calls access files (`/etc/passwd` and `/etc/group`), so the process must open and memory map the file to read it. I/O and memory mapping causes VM exits, so it is understandable why this particular benchmark reveals so much loss. We investigated this further by timing all of the non-I/O requiring system calls and found only 0.88% loss in performance when executing a large amount of calls. We then timed the `getpwuid()` system call and found no less than 68.39% loss.

3.6 Memory

The remaining set of interesting results from the Aim9 benchmark are shown in Table 8 on the next page. These are memory specific and should supplement the results from our MemoryMotion benchmark nicely. We notice a number of interesting issues, for instance the difference between `page_test` and `brk_test`. Both call the `brk()` system call intensively. This call sets the end of the data segment for the calling process, which in effect increases the size of the virtual address space of the process, but without allocating any memory. When the new virtual memory is written to, a standard CoW fault occurs and new pages are allocated. Actually this is what distinguishes the two tests: `brk_test` only manipulates the size, where `page_test` also modifies the contents of the page pointed to by the new virtual memory area. In effect `brk_test` does not trigger much page table updates, while `page_test` ends up allocating a lot of pages and mapping these into memory. These changes must also be reflected in a shadow page table, hence this overhead.

The remaining tests in Table 8 shows the same tendencies due to the internal use of massive forking. When a process is forked, the VMM must also create and

Test	Native	HVM	Loss
page_test	590226	93211	84.21%
brk_test	3133755	3106896	0.86%
exec_test	1144	212	81.47%
fork_test	9215	788	91.45%
shell_rtns_1	242	56	76.86%
shell_rtns_2	243	56	76.95%
shell_rtns_3	242	56	76.86%

Table 8: Results from the **memory** benchmarks in **Aim9**. The large penalties are taken due to the shadow page tables.

maintain another shadow page table. While these extreme amounts of forking are seldom usable, they do pinpoint the problem with a software virtualized MMU. The overhead is simply too large on workloads with many, short living processes, or processes that continuously modify their page tables, e.g. using `mmap` and `munmap` for files.

3.7 Micro Benchmarking

In this section we will present the results of micro benchmarking the code used in the new shadow page table implementation, which is used when running HVM guests. The reasoning behind doing these micro benchmarks are that we have experienced that certain parts of the shadow code is rather slow. We saw this in earlier micro benchmarks and in the actual use of it in [11]. Furthermore the performance of the new and old shadow code has been a topic of discussion on the Xen related mailing lists. A final reason for benchmarking this code is that we expect the shadow page tables, to be a rather important and performance critical part of the new HVM support in Xen.

3.7.1 Experiment Setup

The setup of the benchmarking largely follows what already has been presented in the former experiments. The SPECweb99 benchmark is run in a significantly shorter time period than before (7 minutes). Our own MemoryMotion benchmark was run as presented before.

Two measures are used for timing the individual functions: 1) the Time Stamp Counter (TSC), which gives us the number of clock cycles elapsed in a given code segment, 2) the time used in nano seconds on the same code segment.

It should be noted that the TSC is a 64 bit counter which is only reset on a processor reset. Overflow is not an issue since only small intervals are measured and it is a 64 bit counter [1, p. 403][16, p. 228]. Hardware features that might change the processor frequency have been disabled. The test was performed on a multiprocessor machine, so there still might be some uncertainties like the processors local TSC might not be synchronized with each other (the highest observed difference has been roughly 600 cycles). These kinds of uncertainties should however even out the differences in the measurements, so they can still be compared against each other, with these reservations. The measurements cannot be compared directly to measurements on other machines unless exactly the same processor, memory timings etc. are used.

The functionality provided by Xen is used to get the time used in nano seconds. This however also uses the TSC, but uses different techniques to improve the reliability of the TSC in general and in a virtualized context. Such things as keeping track of processor frequency changes etc. which is needed to convert the

Function:	Count	Mean	Std. dev.	Max.	Min.	Sample size
sh_page_fault						
in cycles	17344499	6545.3	3789.9	16275	714	2329
in nano sec.	17344499	3591.7	2029.9	8802	468	2329
sh_x86_emulate_write						
in cycles	9031366	3280.7	816.9	8708	2044	2328
in nano sec.	9031366	1843.2	438.3	4752	1178	2328
sh_page_fault exits:						
Fault fixed						
in cycles	10938993	3804.4	1292.1	54635	581	23292
in nano sec.	10938993	2140.7	693.8	29362	401	23292
mmio						
in cycles	2693365	7300.4	1278.8	60949	5670	5776
in nano sec.	2693365	4001.6	685.9	32742	3127	5776
Not a shadow fault						
in cycles	3986234	572.7	198.0	2429	301	8542
in nano sec.	3986234	412.8	111.2	1467	255	8542

Table 9: Results from running the **SPECweb99** benchmark in **Debian**. The topmost part of the table shows measurements of individual shadow page table related functions. The lower part is an in-depth measurement of the exit points from the `sh_page_fault` function. **Notice** that the memory mapped I/O (mmio) exit point is the expensive one and is triggered frequently in this I/O intensive workload.

time to nano seconds. Furthermore the timing itself takes time. This was measured to a minimum of 63 cycles (116 nano seconds), a maximum of 105 cycles (143 nano seconds) and finally a mean of 71.5 cycles (121.6 nano seconds). All in all it is necessary to have the nature of these timers in mind when interpreting the results.

The measurements are taken by pseudo randomized sampling, where only a certain factor of the possible measurements are used. An example is where only the n'th measurement is used, where n is a pseudo random number in the range 2500 – 5000. This range is adapted to the specific workload, so a fair amount of measurements are registered without slowing the system too must down. There is a counter for the number of times a code segment has been executed. This is not included in the pseudo randomized sampling, so it gives us the exact number of executions.

3.7.2 Results

The results of the micro benchmarks are presented in a set of tables on the following pages. They are arranged so each row is a separate measurement of a code segment. There are two sets of measurements in each table: 1) first a set of functions that are benchmarked, 2) a set of the different exit points from the `sh_page_fault` function. The two sets cannot be directly compared since they are from different runs.

All the tables are compiled from a much larger set of measurements (34 functions and 7 exits points) where only the ones with a significant time used and number of executions are selected for these concise tables. Furthermore all experiments have been run three times with no significant differences from the ones presented here. A set of unabridged results from one of the different runs can be found in Appendix

Function:	**Count**	**Mean**	**Std. dev.**	**Max.**	**Min.**	**Sample size**
sh_page_fault						
in cycles	32320864	5627.8	3594.4	57582	595	4328
in nano sec.	32320864	3102.7	1934.1	30930	405	4328
sh_x86_emulate_write						
in cycles	10824508	3368.8	1313.1	52801	2002	3153
in nano sec.	10824508	1890.1	703.4	28369	1158	3153
sh_page_fault exits:						
Fault fixed						
in cycles	21479937	3823.1	1602.8	56980	595	45812
in nano sec.	21479937	2141.3	893.2	30619	413	45812
mmio						
in cycles	106233	7351.4	3780.5	55657	5614	183
in nano sec.	106233	4027.5	2025.3	29906	3097	183
Not a shadow fault						
in cycles	10715854	368.0	734.9	50260	245	22872
in nano sec.	10715854	290.8	464.7	27012	217	22872

Table 10: Results from running the **MemoryMotion** benchmark in **Debian**. The topmost part of the table shows measurements of individual shadow page table related functions. The lower part is an in-depth measurement of the exit points from the `sh_page_fault` function. **Notice** that in this memory and process intensive workload there is a higher amount of shadow page faults compared to the SPECweb99 workload, which has a longer running time.

Function:	Count	Mean	Std. dev.	Max.	Min.	Sample size
sh_page_fault						
in cycles	26547825	10525.3	6078.7	35819	616	3658
in nano sec.	26547825	5723.9	3256.5	19275	417	3658
sh_remove_l1_shadow						
in cycles	626788	21978.7	6628.2	77042	6321	1441
in nano sec.	626788	11859.6	3550.8	41359	3469	1441
sh_x86_emulate_write						
in cycles	23214867	2925.5	631.3	8379	2135	3658
in nano sec.	23214867	1652.5	338.2	4571	1230	3658
sh_page_fault exits:						
Fault fixed						
in cycles	27599474	4593.4	9765.4	2216452	735	117442
in nano sec.	27599474	2557.7	5234.1	1187557	487	117442
Not a shadow fault						
in cycles	3422605	575.7	483.9	50519	287	14592
in nano sec.	3422605	409.1	261.2	27154	243	14592

Table 11: Results from running the **SPECweb99** benchmark in **Windows**. The topmost part of the table shows measurements of individual shadow page table related functions. The lower part is an in-depth measurement of the exit points from the `sh_page_fault` function. **Notice** that the `sh_remove_l1_shadow` is the most expensive function in the benchmark.

A.

Below is a short overview of the elements in the tables:

Count: Amount of times this code segment (often a function) has been run in the measured time period.

Mean, Std.dev., Max., and Min.: Statistics of the cycles or nano seconds a code segment is used.

Sample size: The number of pseudo randomized samplings in this measurement.

The results are presented in a set of tables one running Linux with SPECweb99 (see Table 9) and with MemoryMotion (see Table 10). The same benchmarks in Windows (see Table 11 and Table 12). We will not explicitly reference the tables each time they are used; the reader is encouraged to reference them when needed for better understanding or evidence in the following.

sh_page_fault: From table 9 it can be seen that the function executed most frequently is the `sh_page_fault` function. This is expected as this function is the entry point for most of the shadow related code, since Linux uses demand paging to populate page tables. This actually applies to all benchmarks in this section. Compared to the other functions, it is rather expensive on average, however there seems to be a high degree of diversity in the measurements as it can be seen on the standard deviation, maximum, and minimum values. By making a separate run where we measured on the different exits points from the function (as seen in the lower part of the table), it can clearly be seen why there is this diversity. There is a fast exit when it is detected not to be a shadow fault. Two exits which are the expensive

Function:	**Count**	**Mean**	**Std. dev.**	**Max.**	**Min.**	**Sample size**
sh_page_fault						
in cycles	86042670	9982.0	6240.7	56700	602	11508
in nano sec.	86042670	5432.5	3343.3	30457	409	11508
sh_remove_l1_shadow						
in cycles	4384	23911.9	8066.0	71169	7329	144
in nano sec.	4384	12895.5	4320.9	38212	4009	144
sh_x86_emulate_write						
in cycles	75278737	2996.9	562.1	13006	2149	11508
in nano sec.	75278737	1690.7	301.1	7050	1237	11508
sh_page_fault exits:						
Fault fixed						
in cycles	75555141	4436.9	1909.8	57092	987	160621
in nano sec.	75555141	2467.7	1035.9	30671	622	160621
Not a shadow fault						
in cycles	10780573	363.1	464.1	49623	245	23178
in nano sec.	10780573	284.5	302.8	26670	218	23178

Table 12: Results from running the **MemoryMotion** benchmark in **Windows**. The topmost part of the table shows measurements of individual shadow page table related functions. The lower part is an in-depth measurement of the exit points from the `sh_page_fault` function. **Notice** that in this process and memory intensive workload there is high amount of emulated writes and shadow page faults compared to the SPECweb99 workload in Windows.

ones, one in the case of a Memory Mapped I/O (MMIO) related page fault, lastly a generic shadow fault.

The MMIO fault is significantly more expensive than the generic fault, but it is triggered less often. Since this is an I/O intensive workload (SPECweb99) this explains why the `sh_page_page` is more expensive here, than seen in the MemoryMotion workload (a non I/O intensive workload) on Debian. This is simply because the expensive MMIO part of the code is triggered much more frequently. In the Windows benchmarks it is seen that the SPECweb99 workload has a lower percentage of its shadow page faults of the "not a shadow fault" kind, which explains the slightly higher mean in the number of cycles. If comparing the same kind of workload on the two platforms against each other, a tendency towards the shadow page faults triggered by Windows being the most expensive can be seen. Furthermore it is triggered much more frequently in Windows benchmarks than in the Linux counterparts.

sh_x86_emulate_write: The `sh_x86_emulate_write` function is triggered when a HVM guest tries to write to a page table, which is then emulated. The function is about half as expensive as the shadow page fault. The number of times it is called varies depending on the workload, from one third of the times to almost as many times as the shadow page fault on these workloads. It seems to have the same execution time on a different workload on the same OS.

Content-Based Page Sharing: The following paragraph is more or less only relevant for the readers that have read our master thesis on content-based page sharing[11].

In relation to content-based page sharing this investigation showed that the `sh_remove_l1_shadow` is quite expensive, actually the most expensive seen in this micro benchmark. This influences the content-based page sharing as this function is likely to be used often if we decide to port our current implementation to this new version of the Xen shadow page table implementation. It removes all mappings to a l1 (PTE) shadow from a given l2 (PGD) shadow. We will have to do this on each l2 table that might have a shadow that contains a mapping that must be changed. This can either be due to a copy-on-write page fault or a setup of a shared page.

3.7.3 Summary

All in all the new shadow page table implementation seems sane, but we will, if we port the content-based page sharing to it, still experience it as the main overhead. Relative to HVM guests it might be optimized further, but none of the real expensive function are triggered often in this context. The two main functions, `sh_page_fault` and `sh_x86_emulate_write`, are essential and it is not possible to avoid such expenses as long as there is no hardware support for shadow page tables or other solutions exists.

4 Related Work

VMware earlier demonstrated similar results[2]. They, however, compared HVM performance with the performance of their own VMM. They found that the most significant performance penalty on the HVM is due to the lack of MMU virtualization. Both producers of HVM capable processors, Intel and AMD, have declared their intentions for developing further hardware support for this with their "Extended Page Tables" and "Nested Paging" respectively[19, 3].

The performance of virtual networking in paravirtualized Xen was examined in [18] and [17]. Both found large overheads and proposed a number of different solutions to reduce these. The performance was significantly increased, but the authors pointed out that networking, especially receiving, remained a bottleneck. Furthermore VMware workstation experienced similar overheads when virtualizing networking[25], they found that using customized front- and back-end drivers greatly reduced the overhead and provided near to native performance.

A framework for determining why the cost of virtualizing I/O is so high was presented in [5]. Hopefully this will in the future lead to significant advances in virtual I/O, or at least lead to an understanding of the cost.

5 Conclusion

In this paper we have examined the performance of hardware supported virtual machines on Xen compared to native performance. We found that HVM has some overheads, I/O and memory virtualization, due to emulated devices and shadow page tables. Thereby we have, as an independent third-party, asserted the results from VMware, that claimed that the lacking support for MMU virtualization in the first generation of HVM processors was a problem. Both producers of the HVM capable processors have acknowledged this and have announced extensions to the architecture in the next processor generation.

Generally, processor intensive workloads yield near to native performance and the performance is sometimes indistinguishable. Other workloads however exhibit troubles, e.g. any workload that exercises the shadowed page related structures, for instance through excessive process creation and large memory allocations, shows significant overhead, caused by the extreme load of keeping the shadow copies updated. This was micro benchmarked and we found that the implementation seems effective with no apparent overheads, it is the sheer frequency of doing the updates that is causing the performance overhead.

Furthermore there seems to be a constant overhead in the emulation of I/O devices, due to how Xen has chosen to provide I/O for the VMs. Finally there seems to be some networking troubles, in particular a constantly lower TCP receiving throughput and a drastically reduced UDP receiving throughput. We can however not rule out that this latter result is not due to the lack of proper networking configuration, as default settings were applied.

References

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

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

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

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

[5] 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.

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

[7] Yaozu Dong, Shaofan Li, Asit Mallick, Jun Nakajima, Kun Tian, Xuefei Xu, Fred Yang, and Wilfred Yu. Extending xen with intel virtualization technology. *Intel Technology Journal*, 10(3):193–203, August 10, 2006.

[8] 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`.

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

[10] 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.

[11] 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`.

[12] 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.

[13] 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.

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

[15] Joshua LeVasseur, Volkmar Uhlig, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. Pre-virtualization: Soft layering for virtual machines. Technical Report 2006-15, Fakultät für Informatik, Universität Karlsruhe (TH), July 2006.

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

[17] Aravind Menon, Alan L. Cox, and Willy Zwaenepoel. Optimizing network virtualization in xen. In *USENIX Annual Technical Conference*, pages 15–28, May 2006.

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

[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 10, 2006.

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

[21] 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.

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

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

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

[25] 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.

[26] 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.

[27] 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.

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

[29] 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`.

A Micro Benchmarks Results

Function:	Count	Mean	Std. dev.	Max.	Min.	Sample size
sh_page_fault						
in cycles	32320864	5627.8	3594.4	57582	595	4328
in nano sec.	32320864	3102.7	1934.1	30930	405	4328
sh_invlpg						
in cycles	5758	142.8	31.5	329	105	206
in nano sec.	5758	164.2	16.7	267	142	206
sh_gva_to_gfn						
in cycles	32450648	457.0	11.7	819	448	4328
in nano sec.	32450648	330.1	6.2	525	326	4328
sh_update_cr3						
in cycles	15883	1017.7	386.6	9121	609	2489
in nano sec.	15883	632.4	211.9	4969	412	2489
sh_guess_wrmap						
in cycles	23232	1355.5	449.5	19208	623	2936
in nano sec.	23232	816.1	240.9	10380	420	2936
get_page_type						
in cycles	10865538	137.4	52.7	1071	105	4327
in nano sec.	10865538	160.0	30.9	746	142	4327
emulate_map_des						
in cycles	21487025	560.7	750.3	49798	525	4327
in nano sec.	21487025	383.1	402.0	26760	363	4327
sh_x86_emulate_write						
in cycles	10824508	3368.8	1313.1	52801	2002	3153
in nano sec.	10824508	1890.1	703.4	28369	1158	3153
sh_x86_emulate_cmpxchg						
in cycles	10662517	2840.1	104.5	4067	2415	1572
in nano sec.	10662517	1606.7	56.0	2261	1380	1572
sh_make_shadow						
in cycles	11305	5266.2	490.3	11186	2751	2936
in nano sec.	11305	2915.9	263.9	6086	1560	2936
l2e_propagate_from_guest						
in cycles	95933	243.0	895.5	49952	70	3125
in nano sec.	95933	216.4	479.8	26847	120	3125
l1e_propagate_from_guest						
in cycles	43469881	107.9	50.0	441	70	4328
in nano sec.	43469881	141.5	26.2	319	120	4328
guest_walk_tables						
in cycles	86258537	191.4	760.9	49385	126	4328
in nano sec.	86258537	186.1	407.6	26543	150	4328
sh_map_and_validate						
in cycles	21487025	1709.3	899.4	51478	777	4327
in nano sec.	21487025	1001.7	482.0	27664	502	4327
get_page_and_type						
in cycles	10795960	835.8	898.4	50134	798	3032
in nano sec.	10795960	532.9	481.3	26944	513	3032
get_page						

Continued on the next page

– **continued from last page**

Function:	**Count**	**Mean**	**Std. dev.**	**Max.**	**Min.**	**Sample size**
in cycles	10928545	167.6	753.0	49427	105	4327
in nano sec.	10928545	175.9	403.5	26565	138	4327

Table 13: Results from running the **MemoryMotion** benchmark in **Debian**. It shows measurements of individual shadow page table related functions.

Function:	**Count**	**Mean**	**Std. dev.**	**Max.**	**Min.**	**Sample size**
sh_page_fault						
in cycles	17344499	6545.3	3789.9	16275	714	2329
in nano sec.	17344499	3591.7	2029.9	8802	468	2329
get_page_type						
in cycles	5195063	142.9	62.2	1365	105	2327
in nano sec.	5195063	162.9	33.9	817	138	2327
sh_gva_to_gfn						
in cycles	21223706	477.3	46.3	917	448	2329
in nano sec.	21223706	341.2	25.0	582	322	2329
sh_update_cr3						
in cycles	114702	1176.5	345.1	7595	623	2328
in nano sec.	114702	717.2	185.8	4155	420	2328
sh_guess_wrmap						
in cycles	102779	655.2	57.2	1659	616	2319
in nano sec.	102779	436.6	30.8	975	412	2319
sh_remove_write_access						
in cycles	278	2909.7	1743.6	4900	1652	3
in nano sec.	278	1643.7	934.6	2711	972	3
sh_invlpg						
in cycles	612798	127.8	31.4	287	91	2328
in nano sec.	612798	155.3	18.6	300	135	2328
emulate_map_des						
in cycles	9292585	609.6	97.9	1834	525	2328
in nano sec.	9292585	410.4	55.8	1072	363	2328
sh_x86_emulate_write						
in cycles	9031366	3280.7	816.9	8708	2044	2328
in nano sec.	9031366	1843.2	438.3	4752	1178	2328
sh_x86_emulate_cmpxchg						
in cycles	261219	2596.3	210.5	4739	2114	2324
in nano sec.	261219	1476.9	113.5	2621	1215	2324
sh_make_shadow						
in cycles	35863	2844.0	254.6	5117	2247	2319
in nano sec.	35863	1611.4	136.9	2832	1286	2319

Continued on the next page

– **continued from last page**

Function:	**Count**	**Mean**	**Std. dev.**	**Max.**	**Min.**	**Sample size**
l2e_propagate_from_guest						
in cycles	118877	131.4	42.0	532	70	2327
in nano sec.	118877	156.8	25.4	592	120	2327
l1e_propagate_from_guest						
in cycles	27345619	116.0	58.8	469	70	2329
in nano sec.	27345619	147.0	32.0	337	120	2329
guest_walk_tables						
in cycles	47860790	207.7	187.8	3500	126	2329
in nano sec.	47860790	196.4	100.4	1962	150	2329
sh_map_and_validate						
in cycles	9292586	1723.7	777.3	6489	777	2328
in nano sec.	9292586	1011.0	418.2	3567	502	2328
get_page_and_type						
in cycles	1729909	889.7	93.8	2471	798	2328
in nano sec.	1729909	562.5	51.2	1410	510	2328
get_page						
in cycles	8454456	165.6	96.3	756	105	2322
in nano sec.	8454456	176.2	53.9	487	138	2322

Table 14: Results from running the **SPECweb99** benchmark in **Debian**. It shows measurements of individual shadow page table related functions.

Function:	**Count**	**Mean**	**Std. dev.**	**Max.**	**Min.**	**Sample size**
sh_page_fault						
in cycles	86042670	9982.0	6240.7	56700	602	11508
in nano sec.	86042670	5432.5	3343.3	30457	409	11508
sh_invlpg						
in cycles	38277	126.9	29.7	483	98	3565
in nano sec.	38277	163.2	443.9	26640	138	3565
sh_gva_to_gfn						
in cycles	96431759	483.5	58.6	1162	448	11508
in nano sec.	96431759	344.8	31.5	709	326	11508
sh_gva_to_gpa						
in cycles	680	108.7	38.2	392	70	655
in nano sec.	680	147.0	20.7	300	123	655
sh_update_cr3						
in cycles	263354	579.5	274.0	7770	504	11485
in nano sec.	263354	395.7	146.9	4249	356	11485
sh_guess_wrmap						
in cycles	12989	792.5	772.5	50561	77	8392
in nano sec.	12989	511.0	413.8	27168	131	8392

Continued on the next page

– **continued from last page**

Function:	**Count**	**Mean**	**Std. dev.**	**Max.**	**Min.**	**Sample size**
sh_remove_write_access						
in cycles	34142	3234.8	852.1	4585	1183	69
in nano sec.	34142	1818.8	456.3	2543	720	69
get_page_type						
in cycles	21658393	158.6	31.4	560	105	11507
in nano sec.	21658393	171.5	18.7	698	142	11507
sh_remove_l1_shadow						
in cycles	4384	23911.9	8066.0	71169	7329	144
in nano sec.	4384	12895.5	4320.9	38212	4009	144
emulate_map_des						
in cycles	75278749	716.7	121.4	11354	658	11508
in nano sec.	75278749	466.5	65.0	6165	435	11508
sh_x86_emulate_write						
in cycles	75278737	2996.9	562.1	13006	2149	11508
in nano sec.	75278737	1690.7	301.1	7050	1237	11508
sh_x86_emulate_cmpxchg						
in cycles	12	3685.8	211.9	4207	3339	11
in nano sec.	12	2064.5	113.4	2343	1879	11
sh_make_shadow						
in cycles	13209	4729.4	1312.8	54369	2471	8392
in nano sec.	13209	2631.6	703.5	29220	1414	8392
l2e_propagate_from_guest						
in cycles	69505	114.6	42.6	693	70	11235
in nano sec.	69505	146.5	23.5	458	120	11235
l1e_propagate_from_guest						
in cycles	851416517	103.9	57.0	462	70	11508
in nano sec.	851416517	139.5	30.7	334	120	11508
guest_walk_tables						
in cycles	257753178	239.3	41.9	2436	154	11508
in nano sec.	257753178	212.8	22.2	1388	165	11508
sh_map_and_validate						
in cycles	75329616	1416.9	593.4	4172	770	11508
in nano sec.	75329616	844.9	317.9	2322	499	11508
get_page_and_type						
in cycles	21428850	852.6	40.5	2149	798	8948
in nano sec.	21428850	541.8	21.9	1237	513	8948
get_page						
in cycles	21904879	163.8	99.5	1260	105	11508
in nano sec.	21904879	173.6	54.2	758	138	11508

Table 15: Results from running the **MemoryMotion** benchmark in **Windows**. It shows measurements of individual shadow page table related functions.

Function:	Count	Mean	Std. dev.	Max.	Min.	Sample size
sh_page_fault						
in cycles	26547825	10525.3	6078.7	35819	616	3658
in nano sec.	26547825	5723.9	3256.5	19275	417	3658
sh_invlpg						
in cycles	348341	143.2	34.7	364	112	3563
in nano sec.	348341	163.8	19.9	281	142	3563
sh_gva_to_gfn						
in cycles	24509507	468.3	53.2	1127	441	3658
in nano sec.	24509507	337.0	40.3	2055	323	3658
sh_gva_to_gpa						
in cycles	406	105.7	18.2	252	84	383
in nano sec.	406	146.6	9.8	225	131	383
sh_update_cr3						
in cycles	983618	906.7	388.2	7126	511	3658
in nano sec.	983618	571.6	208.8	3903	360	3658
sh_guess_wrmap						
in cycles	54286	779.3	73.8	1526	126	3472
in nano sec.	54286	507.0	40.9	1065	157	3472
sh_remove_write_access						
in cycles	94571	3211.9	879.4	6692	1148	191
in nano sec.	94571	1807.0	470.8	3671	701	191
get_page_type						
in cycles	5922134	183.7	66.0	1428	105	3657
in nano sec.	5922134	184.7	35.8	852	142	3657
sh_remove_l1_shadow						
in cycles	626788	21978.7	6628.2	77042	6321	1441
in nano sec.	626788	11859.6	3550.8	41359	3469	1441
emulate_map_des						
in cycles	23220915	744.7	94.6	2835	658	3658
in nano sec.	23220915	481.7	51.1	1601	435	3658
sh_x86_emulate_write						
in cycles	23214867	2925.5	631.3	8379	2135	3658
in nano sec.	23214867	1652.5	338.2	4571	1230	3658
sh_x86_emulate_cmpxchg						
in cycles	6064	6013.0	3947.6	12593	2408	48
in nano sec.	6064	3308.6	2116.5	6836	1372	48
sh_make_shadow						
in cycles	58745	3108.4	333.7	6734	2359	3472
in nano sec.	58745	1756.0	180.8	3698	1350	3472
l2e_propagate_from_guest						
in cycles	650825	176.5	69.5	1155	70	3548
in nano sec.	650825	181.9	39.4	709	120	3548
l1e_propagate_from_guest						
in cycles	309776112	104.7	62.3	469	70	3658
in nano sec.	309776112	140.5	33.6	338	120	3658
guest_walk_tables						
in cycles	74278247	270.9	143.5	2730	126	3658
in nano sec.	74278247	230.3	77.0	1545	154	3658

Continued on the next page

– **continued from last page**

Function:	**Count**	**Mean**	**Std. dev.**	**Max.**	**Min.**	**Sample size**
sh_map_and_validate						
in cycles	23568635	1238.0	576.3	6867	770	3658
in nano sec.	23568635	749.5	309.0	3765	499	3658
get_page_and_type						
in cycles	2677425	966.2	135.3	3997	798	3658
in nano sec.	2677425	602.9	73.0	2227	513	3658
get_page						
in cycles	9433666	155.8	86.9	525	105	3658
in nano sec.	9433666	170.3	47.7	439	138	3658

Table 16: Results from running the **SPECweb99** benchmark in **Windows**. It shows measurements of individual shadow page table related functions.