# Performance of VMware® VMI

VMware ESX Server 3.5

## Introduction

VMware® ESX Server 3.5 introduces support for guest operating systems that use VMware's paravirtualization standard, Virtual Machine Interface (VMI). This paper describes VMI and its performance benefits.

## x86 Virtualization Techniques

The first two subsections below discuss binary translation and hardware virtualization, both of which are techniques that can be used with unmodified guest operating system kernels. The third section introduces paravirtualization, a technique that involves modifications to the guest operating system or to other components within the guest virtual machine and which can be used in conjunction with either of the previous two techniques, or on its own as a complete virtualization strategy.

### Binary Translation

VMware pioneered x86 virtualization using a technique called binary translation. Binary translation modifies sensitive instructions on the fly to virtualizable instructions. Remaining instructions (such as those running in user mode) run un-modified in direct-execution mode at native speed. Because binary translation is performed on the binary code that gets executed on the processor it does not require changes to the guest operating system kernel. The first product to use binary translation to virtualize x86 operating systems was VMware Workstation 1.0, released in 1999. For almost a decade VMware has been perfecting this technique, making it a robust and highly reliable approach for x86 virtualization. Binary translation provides excellent compatibility with existing operating systems without significant performance trade-offs for most real-life workloads.

VMware uses several transparent techniques to gain control from the guest operating system whenever virtualization-sensitive events happen. These techniques involve some amount of virtualization overhead.

### Hardware-Assisted Virtualization

Both Intel and AMD have added hardware support for virtualization to their latest processors. These new processors allow trapping of sensitive events. This eliminates the need for binary translation and simplifies the hypervisor. This technique, however, still lags in performance compared to the more mature binary translation technique (see Reference 1), though future hardware improvements are expected to bridge this performance gap. Like binary translation, hardware assisted virtualization is a "full virtualization" solution, meaning the guest operating system is virtualized without the need to modify it.

### Paravirtualization

Paravirtualization is a technique in which a modified guest operating system kernel communicates to the hypervisor its intent to perform privileged CPU and memory operations. This technique reduces the work

required of the hypervisor, thus making it simpler than a binary translation hypervisor. Although paravirtualization does not eliminate virtualization overhead, it can improve guest operating system performance.

## Introduction to Paravirtualization

The idea of guest-host interaction is not a new concept; it has long been a part of VMware products in the form of VMware Tools. For example:

- The VMware SVGA driver shares data structures with the hypervisor to allow faster screen updates.
- VMware's high-performance virtual Ethernet driver, vmxnet, shares data structures with the hypervisor to reduce CPU overhead.
- VMware's "balloon driver" is used by the hypervisor to control the guest's memory usage.
- The VMware Tools service enables time synchronization between host and guest.

While such guest-host communication provides improved performance, and can be classified as paravirtualization, none of these examples involve changes to the underlying guest operating system kernel. However in order to paravirtualize the CPU and the memory management unit, changes to the guest operating system kernel are required. Open-source operating systems, such as Linux, allow us to make such changes.

## Issues with Paravirtualizing the Linux Kernel

The guest operating system modifications mentioned above are required due to the fact that the source code was initially written to run directly on native hardware. Early paravirtualization guest-host kernel interfaces allowed the guest to run only in a hypervisor, leading to the need for separate kernels, one for native hardware and one for use with a hypervisor. This meant that operating system vendors had to develop, test, and debug twice as many kernels. Applications also had to be certified to run on twice as many kernels.

Also, the lack of a standard guest-host interface in early implementations of paravirtualization led to frequent interface changes, which in turn caused version dependencies between the guest operating system kernel and the hypervisor.

These issues with early implementations of x86 paravirtualization hindered its adoption.

## Virtual Machine Interface (VMI)

In order to address the issues described above VMware proposed a new guest-host interface, called Virtual Machine Interface (VMI), which defines a set of hypercalls an operating system can use to communicate with the hypervisor (see Reference 2 for the VMI API).

The standardized interface provided by VMI allows the guest operating system kernel and the hypervisor to evolve independently. The VMI specification also makes it possible for other vendors to enable their hypervisors to support guest operating systems that use VMI.

VMI was designed to abstract native hardware. This feature, called transparent paravirtualization, allows a VMI-enabled kernel to run both on native hardware and on hypervisors that support the interface with no additional modification to the operating system kernel.

VMI code is included in Linux mainline kernels 2.6.22 and above (to enable the code, see Reference 3). The Ubuntu Linux distribution includes VMI support in version 7.04 ("Feisty Fawn") and later (see Reference 4). Novell and VMware are working together to include VMI in a future service pack for SLES 10 (see Reference 5).

## VMI Performance Benefits

VMware's VMI implementation offers a number of performance and resource-utilization benefits:

- The syscall entry and exit path is faster. This speeds up syscall-dominated workloads.
- VMI-enabled Linux kernels by default use an alternate timer interrupt mechanism that results in reduced physical CPU consumption, especially when the virtual machine is idle, and in more accurate timekeeping, even when running many virtual machines.
- Because the guest kernel communicates to the hypervisor its intent to perform memory management unit (MMU)-related operations, MMU virtualization overhead is reduced. Depending on the workload, this can have varying performance benefits.
- SMP virtual machines running VMI-enabled operating systems use shared shadow page tables (for more information on shadow page tables see Reference 6). As a result they have less memory space overhead than those running non-VMI-enabled operating systems.

When a workload runs in user mode, the VMware virtual machine in which it is running will be in direct-execution mode. Because directly-executed code already runs at native speeds in both binary translation and VMI-style paravirtualization, workloads that spend the majority of their time in user mode gain only modest performance improvements from paravirtualization.

# Performance Data

Two different 32-bit Linux distributions were used in the benchmark tests presented in this paper:

- Ubuntu 7.04 (Feisty Fawn, kernel 2.6.20), which has VMI enabled by default.
- SUSE Linux Enterprise Server (SLES) 10, SP1 distribution, using an experimental 2.6.16 kernel with VMI patches. (See Reference 7 to download the latest SLES 10 SP2 kernel which includes the VMI patches)

**NOTE** You should see performance improvements similar to those documented in this paper with Linux kernel 2.6.22 and above when VMI is enabled.

The descriptions of the experiments in this paper use the following shorthand:

"VMI virtual machine" refers to a virtual machine running a VMI-enabled kernel and for which the VMI option is selected in ESX Server 3.5.

"Binary translation virtual machine" refers to a virtual machine for which the VMI option is deselected in ESX Server 3.5.

ESX Server 3.5 by default uses binary translation for 32-bit virtual machines. The tests in this paper use the performance of binary translation virtual machines as a baseline and compare that with the performance of VMI virtual machines.

The benchmarks in this paper can be broadly classified into two main categories:

- Non-workload experiments:

  Boot time and idle virtual machine CPU usage.

- Workload experiments:

  Kernel microbenchmarks, dbench & tbench, Apache compile, Linux kernel compile, MySQL-SysBench, Oracle-SwingBench, and virtual machine overhead memory tests.

## Benchmarking Methodology

When the physical system is CPU saturated, the reduction in CPU utilization provided by a VMI virtual machine will result in improved performance for that workload. For this reason, the workload-based experiments presented in this paper have been configured to saturate the CPU. The experiments were also conducted with only one virtual machine running at any time. This was to ensure that the virtual machine received all the CPU resources it requested.

**NOTE** Most real life workloads will not be CPU saturated. In such cases you might not see noticeable performance differences between binary translation and VMI virtual machines, even though the latter may consume fewer CPU resources.

Similarly, most real-life scenarios will run multiple virtual machines simultaneously. In such cases, the resources of underlying hardware are shared amongst the virtual machines. We ran single-virtual machine experiments for this paper because it allowed us to directly measure the performance impact of the virtualization techniques.

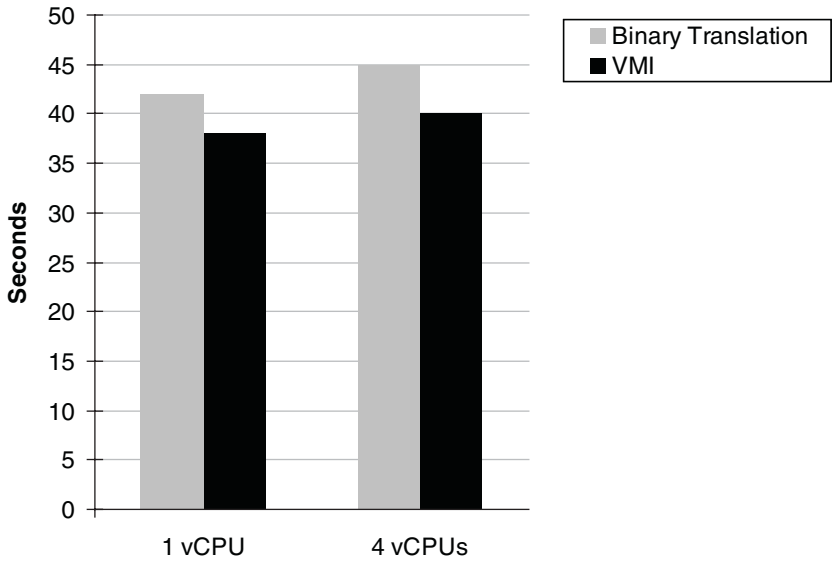
In order to avoid guest timekeeping inaccuracies, the VMware Tools service was used to log events into the virtual machine log file. The timestamps in this log file were then used to determine the time taken to complete the tasks. Previous experiments have shown this method to provide results that fairly closely match "wall clock" time. For workloads that involved separate client machines timing was performed on those client machines.

## Non-Workload Experiments

### Boot Time

This test measures the time taken for a Linux guest to boot from power on to completion of the `rc.local` script. Kernel mode operations performed during the boot process run more efficiently in VMI virtual machines than in binary translation virtual machines. For this reason VMI virtual machines boot faster, as can be seen in Figure 1.

**Figure 1.** Boot Time (Lower is Better)

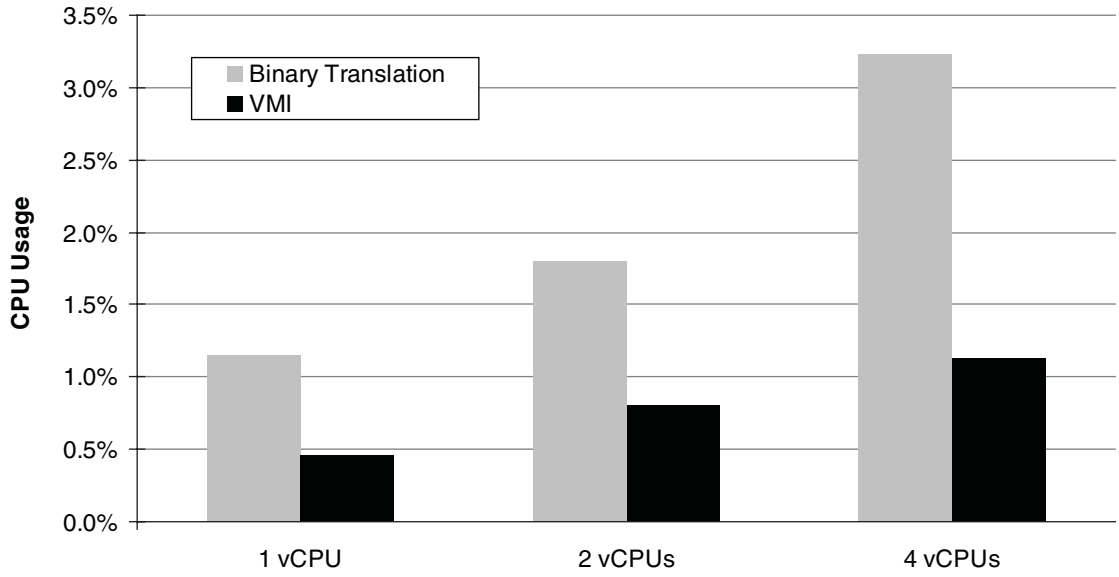



The test was run in Ubuntu 7.04 on two virtual machines: the first with one vCPU and 1GB of RAM; the second with four vCPUs and 1GB of RAM. Setup details are included in "Boot Time Experiment Details" on page 15.

### Idle Virtual Machine CPU Usage

Virtual machines consume some CPU cycles for timer interrupts and handling network broadcast packets, even when those virtual machines are completely idle. Because the timer devices deliver timer interrupts to each of the virtual CPUs, the CPU usage of an idle virtual machine increases with more virtual CPUs. VMI virtual machines consume fewer timer interrupts than binary translation virtual machines, and thus consume fewer CPU cycles when idle. As shown in Figure 2, this difference in CPU usage is more pronounced with more virtual CPUs.

**Figure 2.** Idle Virtual Machine CPU Usage (Lower is Better)




This test was run in Ubuntu 7.04 on virtual machines with one, two, and four vCPUs, all of which had 1GB of RAM. Setup details are included in "Idle Virtual Machine CPU Usage Experiment Details" on page 16.

# Workload Experiments

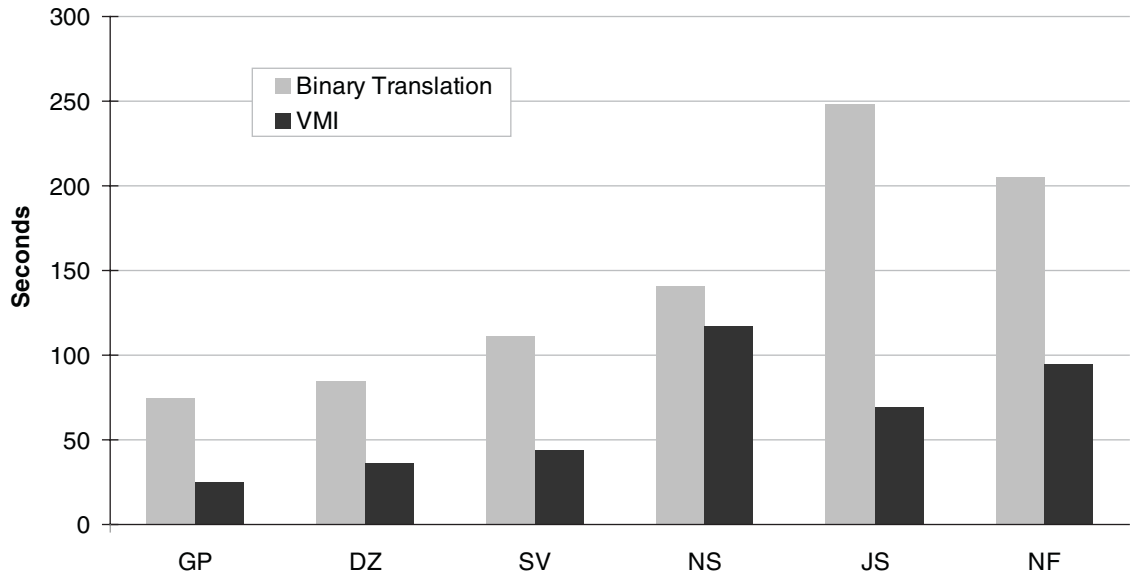
## Kernel Microbenchmarks

Real-life workloads spend the majority of their time executing user code and less time executing kernel code. Both binary translation and VMI-style paravirtualization incur CPU overhead for kernel code, but not for user code. In order to measure this CPU overhead we used kernel microbenchmarks that fully utilize the CPU and cause the virtual machine to spend almost all its time in kernel code. The following kernel microbenchmarks were used:

- getppid (GP): A C program that measures the system call overhead by repeatedly issuing getppid system calls. This stresses the syscall entry and exit path.
- divzero (DZ): A C program that measures the processor fault handling overhead by repeatedly generating and handling divide-by-zero faults. This stresses the fault-handling code.
- segv (SV): A C program that measures the processor fault handling overhead by repeatedly generating and handling segmentation faults. This stresses the fault-handling code.
- nativethread switch (NS): A C program that measures context switch overhead by creating threads and repeatedly switching between them. This stresses MMU virtualization.
- javaswitch (JS): A Java program that measures context switch overhead by repeatedly switching between multiple Java threads. This stresses MMU virtualization.
- nforkwait (NF): A C program that measures process creation overhead by forking processes and waiting for them to complete. This stresses MMU virtualization.

Figure 3 shows that kernel microbenchmarks run significantly faster in a VMI virtual machine, as expected. This is because VMI virtual machines are more efficient at handling syscalls than binary translation virtual machines. While these results show a significant performance gap between VMI virtual machines and binary translation virtual machines, the performance gap is highly dependent on the workload.

**Figure 3.** Kernel Microbenchmarks (Lower is Better)




These benchmarks were run in SLES 10 SP1 with a VMI-patched 2.6.16 kernel on a virtual machine with one vCPU and 1GB of RAM. Setup details are included in "Kernel Microbenchmarks Experiment Details" on page 16.

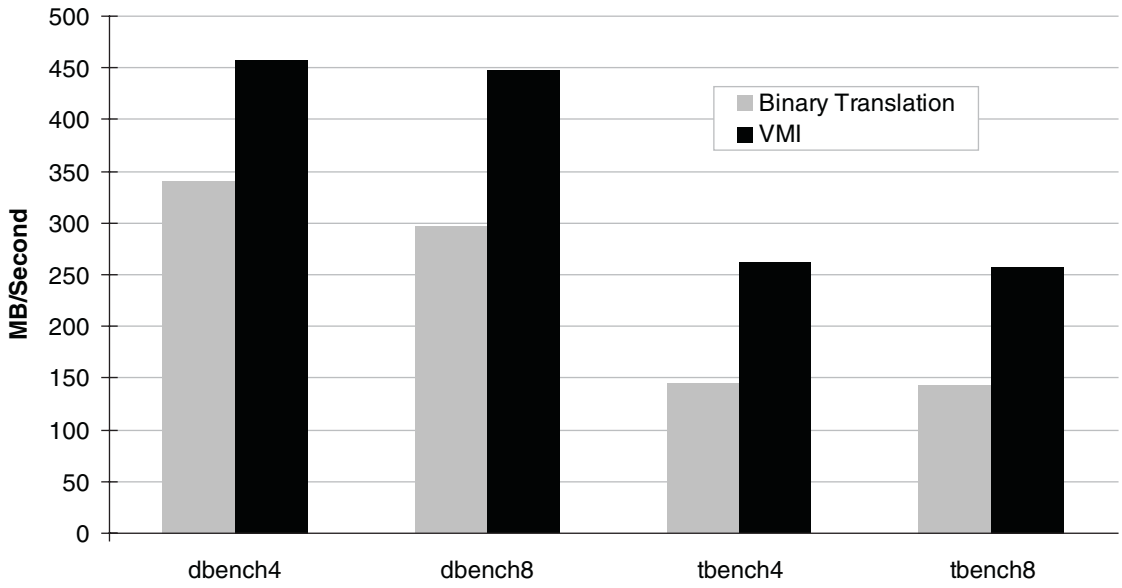
### Dbench and Tbench

Dbench and tbench (see Reference 9) simulate the disk and network I/O component, respectively, of SMB I/O traffic. Dbench uses asynchronous I/O, which does very little disk access due to filesystem buffering. Tbench has client and server components, both of which were run on the same virtual machine for this experiment. For this reason the tbench results were not affected by network latency.

With no disk access and no network latency, both of these benchmarks behaved like kernel microbenchmarks, exercising I/O syscalls until the CPU was fully utilized. The results, shown in Figure 4, were thus limited by CPU resources, with the higher performance of the VMI virtual machines coming from their ability to handle syscalls with increased CPU efficiency.

**NOTE** High I/O latencies can make the CPU overhead insignificant. Although not shown in Figure 4, the performance gap between VMI and binary translation virtual machines can decrease as I/O latencies increase.

**Figure 4.** Dbench and Tbench Benchmarks (Higher is Better)




These benchmarks were run in SLES 10 SP1 with a VMI-patched 2.6.16 kernel on a virtual machine with one vCPU and 1GB of RAM. We ran the benchmarks with four threads ("dbench4" and "tbench4" in Figure 4) and eight threads ("dbench8" and "tbench8" in Figure 4). Setup details are included in "Dbench and Tbench Experiment Details" on page 16.

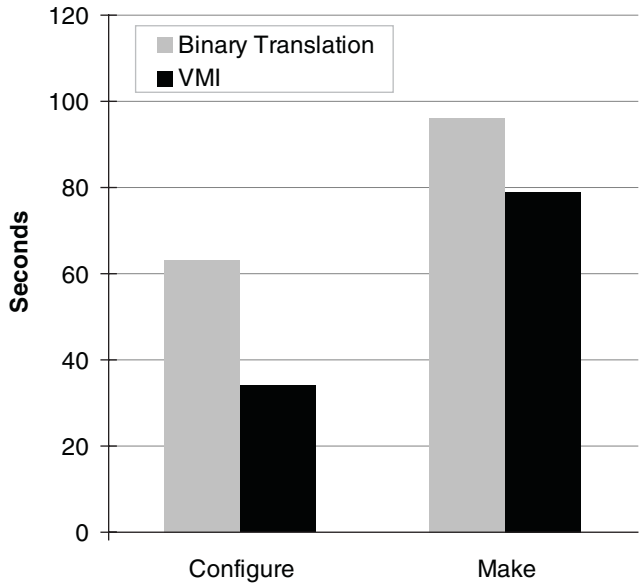
### Apache Compile

This experiment represents the typical job of compiling open-source software — in this case Apache, a popular open-source web server (see Reference 10). This involves two steps: configure and make. Both steps create many processes, and both, though especially the configure step, resemble the process creation kernel microbenchmark, nforkwait, included in "Kernel Microbenchmarks" on page 7.

Because there are many process creation tasks, all of which stress MMU virtualization, VMI virtual machines perform the job faster than do binary translation virtual machines, as is shown in Figure 5.

While both configure and make create many processes, those created during the make step live longer and spend most of their time in user mode. This explains why VMI virtual machines offer a larger performance gain for configure than for make.

**Figure 5.** Apache Compile (Lower is Better)



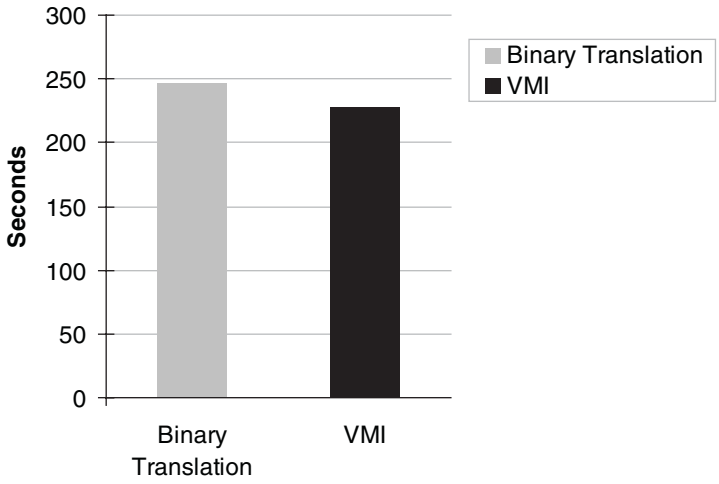

These tests were run in SLES 10 SP1 with a VMI-patched 2.6.16 kernel on a virtual machine with one vCPU and 1GB of RAM. Setup details are included in "Apache Compile Experiment Details" on page 16.

### Linux Kernel Compile

The Linux kernel compile job, like the Apache compile job described previously, creates many processes. But, like the make step in that job, these tend to be long-lasting processes that run mostly in user mode.

While the VMI virtual machine is much faster at process creation than the binary translation virtual machine, there is no benefit experienced in user mode. The VMI virtual machine therefore provides only a modest performance increase for this workload, as shown in Figure 6.

**Figure 6.** Kernel Compile (Lower is Better)




These tests were run in SLES 10 SP1 with a VMI-patched 2.6.16 kernel on a virtual machine with one vCPU and 1GB of RAM. Setup details are included in "Linux Kernel Compile Experiment Details" on page 16.

### MySQL-SysBench

MySQL is a popular open-source database (see Reference 11) and SysBench is a benchmarking tool designed to evaluate its performance (see Reference 12). SysBench can run two different type of queries:

- Simple query: This query issues select system calls with no updates to the database.
- Complex query: This query performs transactions and rollbacks with reads and writes in between.

Neither of the query types alter the database table size. SysBench, which was run on a separate client machine, was configured to send multiple simultaneous queries to the MySQL database with zero think time. We used a simple database that fit entirely in memory. As a result, this workload both saturated the virtual CPU and generated significant network activity, with very little disk I/O.

VMI virtual machines performed better than binary translation virtual machines in this experiment for both simple queries (as shown in Figure 7) and complex queries (as shown in Figure 8). The higher performance of the VMI virtual machines comes from their ability to handle syscalls with increased CPU efficiency.

**Figure 7.** MySQL-Sysbench Simple Query (Higher is Better)

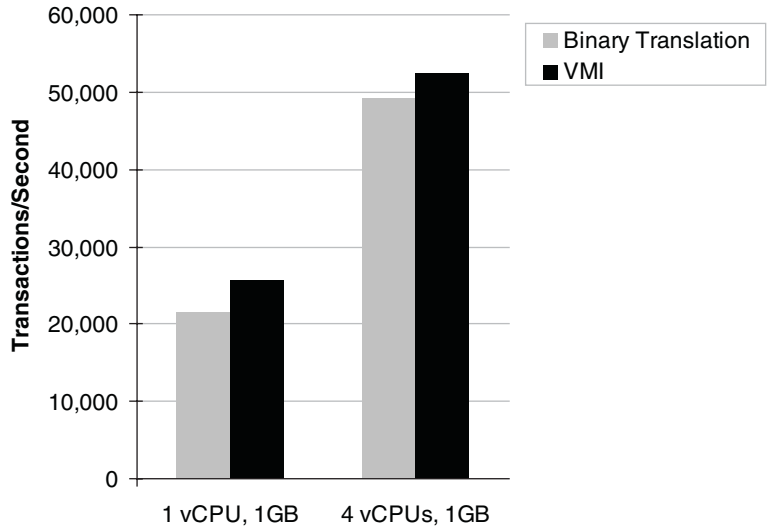
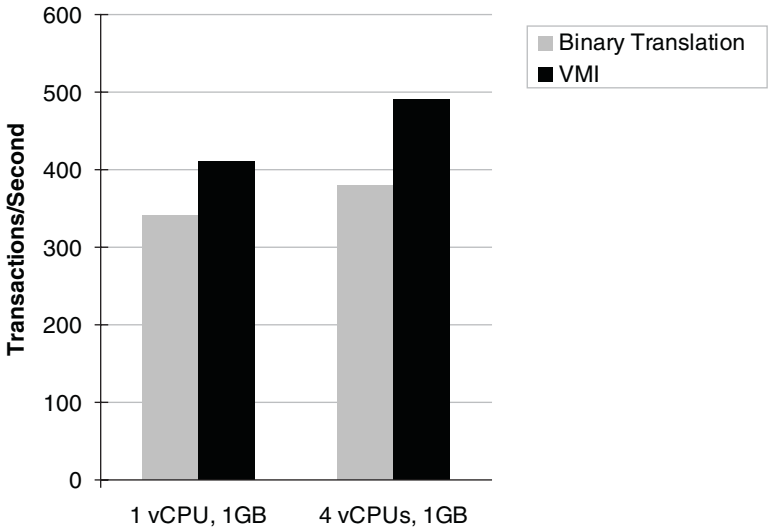



**Figure 8.** MySQL-Sysbench Complex Query (Higher is Better)



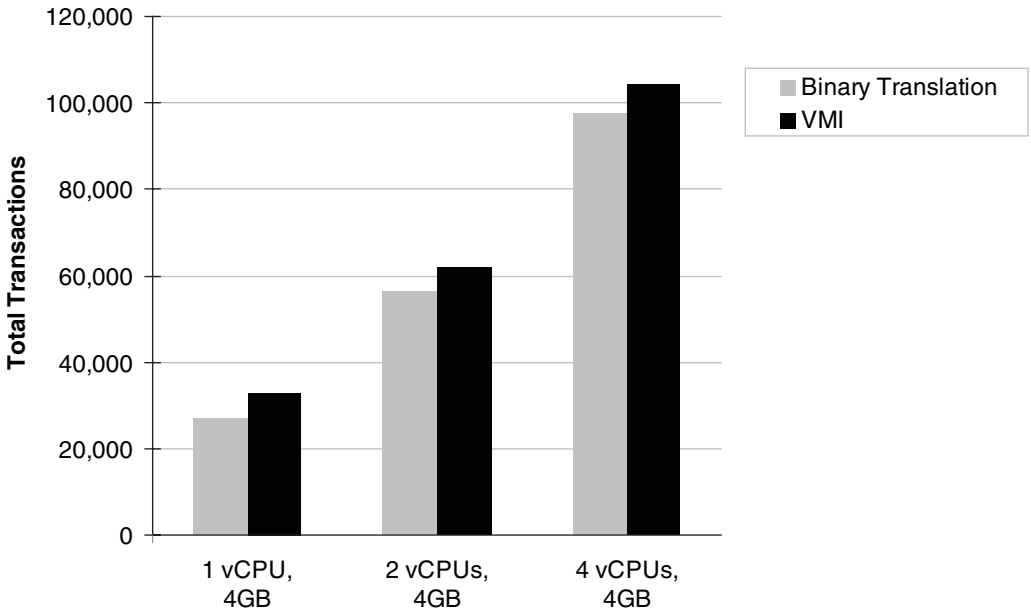

These tests were run in SLES 10 SP1 with a VMI-patched 2.6.16 kernel on a virtual machine with one vCPU and 1GB of RAM. Setup details are included in "MySQL-SysBench Experiment Details" on page 16.

### Oracle-SwingBench

SwingBench (see Reference 13) is a benchmarking tool used to drive load against an Oracle database. The SwingBench OrderEntry workload, used in this test, simulates transaction processing. For the purpose of comparing binary translation and VMI virtual machines, we configured the SwingBench client to drive the Oracle database to CPU saturation.

The Oracle database in this test spends the majority of its time running in user mode and so the binary translation virtual machine performed quite well and the VMI virtual machine offered only a modest performance gain, as shown in Figure 9.

**Figure 9.** Oracle-SwingBench Query (Higher is Better)

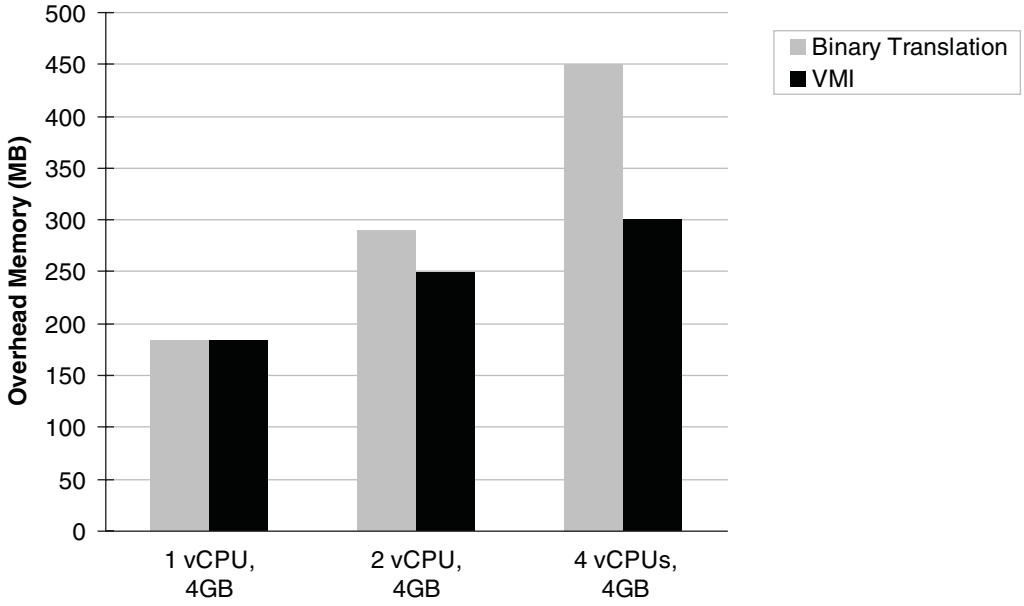



These tests were run in SLES 10 SP1 with a VMI-patched 2.6.16 kernel on a virtual machine with one, two, or four vCPUs and 4GB of RAM. Setup details are included in "Oracle-SwingBench Experiment Details" on page 16.

### Virtual Machine Overhead Memory

This experiment compares the overhead memory usage of VMI virtual machines and binary translation virtual machines. The overhead usage was collected after completion of the Oracle-SwingBench tests described in "Oracle-SwingBench" on page 12. As shown in Figure 10, the overhead memory usage of VMI virtual machines is less than that of binary translation virtual machines and the difference increases with more virtual CPUs. The reduction in overhead memory usage in VMI virtual machines comes from their ability to use shared shadow page tables for their virtual CPUs.

**Figure 10.** Memory Overhead (Lower is Better)




These tests were run in a SLES 10 SP1 with a VMI-patched 2.6.16 kernel on a virtual machine with one, two, or four vCPU and 4GB of RAM. Setup details are included in "Overhead Memory Experiment Details" on page 17.

# Conclusion

This paper shows that VMI-style paravirtualization offers performance improvements for a wide variety of workloads. However the performance gains depend heavily on the nature of those workloads. Workloads that spend the majority of their time in user mode provide only a modest gain when run in a VMI virtual machine. Workloads that spend a significant portion of their time in kernel mode, or that are memory management unit intensive, see the largest benefit from running in a VMI virtual machine.

Non-VMI-enabled guests continue to perform quite well with binary translation, and remain an appropriate choice for many situations. VMware is also working with hardware vendors to improve hardware virtualization. With support for binary translation, hardware virtualization, and VMI-style paravirtualization, ESX Server 3.5 allows the choice of the best virtualization technique for each virtual machine based on its workload and guest operating system.

# References


1 Agesen, Ole and Adams, Keith, *A Comparison of Software and Hardware Techniques for x86 Virtualization* http://www.vmware.com/resources/techresources/528

2 http://www.vmware.com/pdf/vmi_specs.pdf

3 http://kb.vmware.com/kb/1003644

4 http://www.ubuntu.com/getubuntu/download

5 http://www.novell.com/products/server/

6 *VMware ESX Server 2 Architecture and Performance Implications* http://www.vmware.com/pdf/esx2_performance_implications.pdf

7 SUSE Linux Enterprise Server (SLES) 10, SP2 kernels containing VMI patches can be downloaded from: http://ftp.suse.com/pub/projects/kernel/kotd/SLES10_SP2_BRANCH/i386/kernel-vmi.rpm http://ftp.suse.com/pub/projects/kernel/kotd/SLES10_SP2_BRANCH/i386/kernel-vmipae.rpm

8 http://www.vmware.com/interfaces/paravirtualization/performance.html

9 http://samba.org/tridge/dbench/README

10 http://www.apache.org/

11 http://www.mysql.com/

12 http://sysbench.sourceforge.net/

13 http://www.dominicgiles.com/swingbench.html

# Test Environment

This section details the hardware and software environment in which the tests described in this paper were run.

## Test Systems

### Server Hardware (System A)

System: IBM System x3500
Processors: Two 3.0 GHz Intel Xeon dual-core processors
- L1 cache: 32KB
- L2 cache: 4MB (shared)

Memory: 5GB
Internal storage:
- LSI Logic 53c1030 PCI-X Fusion MPT dual Ultra320 SCSI adapter
- Maxtor SCSI 74GB 10K RPM hard disc drive (10K5 73WLS)

Network interface cards:
- Two Broadcom NetXtreme BCM5721 Gigabit Ethernet controllers

### Server Hardware (System B)

System: HP DL380 G5
Processors: Two 2.66 GHz Intel Xeon 5160 quad-core processors
- L1 cache: 128KB
- L2 cache: 8MB
- L3 cache: 8MB

Memory: 32GB
Internal storage:
- Compaq Smart Array P400 Controller
- Maxtor SCSI 74GB 10K RPM hard disc drive (10K5 73WLS)

Network interface cards:
- Two Broadcom NetXtreme BCM5708 Gigabit Ethernet controllers
- Two Intel 82571EB Gigabit Ethernet controllers

### Server Software

VMware software: ESX Server 3.5, experimental build

### Client Hardware

System: “white box”
Processors: Two dual-core 2.0 GHz AMD Opteron model 270 processors
- L1 cache: 64KB
- L2 cache: 1MB (shared)

Memory: 5GB
Internal storage: IDE controller, 130GB ATA hard disc drive
Network interface card (two): Broadcom NetXtreme BCM5704 Gigabit Ethernet controller

For client-based workloads, a cross-over Ethernet cable was used.

### Client Software

Operating system: Ubuntu 6.06 (“Dapper Drake”)

## Details of Individual Tests

### Boot Time Experiment Details

This test was performed on the system described in “Server Hardware (System A)” on page 15.

### Idle Virtual Machine CPU Usage Experiment Details

This test was performed on the system described in "Server Hardware (System A)" on page 15.

### Kernel Microbenchmarks Experiment Details

This test was performed on the system described in "Server Hardware (System A)" on page 15.

In this test we forked 16 instances of each kernel microbenchmark and waited for all the instances to finish. We then measured the total time taken, and reported the average of five runs, in seconds.

### Dbench and Tbench Experiment Details

This test was performed on the system described in "Server Hardware (System A)" on page 15.

We used Dbench 3.04 and reported the average throughput numbers in MB/second for a five minute run (following a two-minute warmup). We used the standard load-description file, `client.txt`, that came bundled with the benchmark.

### Apache Compile Experiment Details

This test was performed on the system described in "Server Hardware (System A)" on page 15.

We measured the time required to compile Apache source version 2.2.3. The test included `./configure` and `make` with 16 job threads (using `-j16`) and we reported the average of three runs, in seconds.

### Linux Kernel Compile Experiment Details

This test was performed on the system described in "Server Hardware (System A)" on page 15.

We measured the time required to compile the Ubuntu Linux kernel 2.6.20 source tree with 16 job threads (using `-j16`) and reported the average of five runs, in seconds.

### MySQL-SysBench Experiment Details

This test was performed on the system described in "Server Hardware (System A)" on page 15.

We used MySQL Distribution version 5.0.26 and SysBench version 0.4.8.

We added the following configurations to the MySQL configuration file:

key_buffer=16M
max_allowed_packet=16M
thread_stack=128K
thread_concurrency=8
query_cache_limit=1M
query_cache_size=16M

Binary logging was also disabled in MySQL.

Sysbench was run with 64 threads and 0 think time (oltp-connect-delay=0, oltp-user-delay-max=0, oltp-user-delay-min=0).

We measured the average transactions per second for a three minute run and reported the average of three runs.

### Oracle-SwingBench Experiment Details

This test was performed on the system described in "Server Hardware (System B)" on page 15.

We used Oracle 10g R2 and SwingBench 2.2. SwingBench was run on the client system with the following configuration:

SwingBench configuration: OrderEntry (PLSQL)
Number of users: 30
Pooled: 5

LogonDelay: 0
MinDelay: 0
MaxDelay: 0
MaxTransactions = -1
QueryTimeout = 60
DriverType: thin

We measured the total transactions completed in five minutes and reported the average of three consecutive runs.

The database files were hosted on a separate virtual disk which was set to nonpersistent mode to ensure repeatable results even after SwingBench modified the database files.

### Overhead Memory Experiment Details

This test was performed on the system described in "Server Hardware (System B)" on page 15.

We made the overhead memory measurements using `esxtop` and reported values from the **ovhdmax** column in the memory screen.

**VMware, Inc. 3401 Hillview Ave., Palo Alto, CA 94304 www.vmware.com**

Revision 20080216 Item: ESX-ENG-Q108-595