# Proceedings of the Linux Symposium

## Volume Two

July 19th–22nd, 2006
Ottawa, Ontario
Canada

## Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

## Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

## Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin

# Virtual Scalability: Charting the Performance of Linux in a Virtual World

Exploring the scalability of the Xen hypervisor

Andrew Theurer
*IBM Linux Technology Center*
habanero@us.ibm.com

Karl Rister
*IBM Linux Technology Center*
kmr@us.ibm.com

Orran Krieger
*IBM T.J. Watson Research Center*
okrieg@us.ibm.com

Ryan Harper
*IBM Linux Technology Center*
ryanh@us.ibm.com

Steve Dobbelstein
*IBM Linux Technology Center*
steved@us.ibm.com

## Abstract

Many past topics at the Linux Symposium have covered Linux Scalability. While still quite valid, most of these topics have left out a hot feature in computing: Virtualization. Virtualization adds a layer of resource isolation and control that allows many virtual systems to coexist on the same physical machine. However, this layer also adds overhead which can be very light or very heavy. We will use the Xen hypervisor, Linux 2.6 kernels, and many freely available workloads to accurately quantify the scaling and overhead of the hypervisor. Areas covered will include:

1. SMP Scaling: use several workloads on a large SMP system to quantify performance with a hypervisor.
2. Performance Tools: discuss how resource monitoring, statistical profiling, and tracing tools work differently in a virtualized environment.
3. NUMA: discuss how Xen can best make use of large system which have Non-Uniform Memory Access.

## 1 Introduction

Although virtualization has recently received much press and attention, it is not a new concept in computing. It was first added to IBM mainframes in 1968 and has continued to evolve ever since [1]. Traditionally, virtualization has been a capability that only high-end systems possessed, but that has begun to change in recent years. Commodity x86 servers first gained virtualization capabilities through a technique that is known as full virtualization where each guest operating system was provided a completely emulated environment in

which to run. While functional, this approach suffers degradations in performance due to the fact that all interaction between the guest operating system and the physical hardware must be intercepted and emulated by the hypervisor.

A competing approach to full virtualization known as paravirtualization has emerged that attempts to address the deficiencies of full virtualization. Para-virtualization is a technique where guest operating system are modified to provide optimal interaction between the guest and the hypervisor layer upon which it is running. By modifying the guest operating system, some of the performance overheads that are associated with full virtualization are eliminated which leads to increased throughput capability and resource utilization. Para-virtualization is, however, not without its own challenges. The requirement that the guest operating system be modified is chief among these. Each and every guest that the end user desires to run in the virtualized environment must be modified in this manner. This is a high cost to pay in time and manpower. It also means that the support of closed source operating systems is entirely at the discretion of the controlling development organization.

This paper focuses on paravirtualization as implemented by the Xen [2][3] project. While the expectation that the overhead of paravirtualization with Xen is low, its original design and development occurred on relatively small systems such as those with one or two CPUs. It is only in the last year of development that support for SMP guests has been added along with support for greater than 4GB of memory through the addition of 32-bit PAE and native 64bit support. For these reasons, many of the design decisions were aimed at excising the utmost performance in these types of configurations, but these decisions may not lend themselves to scaling upward with similar performance expectations. As x86 hardware becomes increasingly competitive at the high end—systems with 128 logical CPUs and hundreds of gigabytes of memory are now possible—the desire to run virtualized environments on systems of this scale will increase. This paper will examine the performance characteristics of Xen in this type of environment and identify areas that development should focus on for scaling enablement.

## 2 What is scalability?

For this study, scalability applies to scaling up the number of processors within a single system. As such, we proceeded to measure and analyze various workloads running from one to as many as 16 processor cores. Without a hypervisor, using a single operating system, the process to measure scalability was fairly clear: start with one processor, run workload(s), record results and analysis data, and repeat, incrementing processors until a maximum is reached. However, introducing a hypervisor adds a twist to measuring scalability. We still want to measure the throughput increase as we scale up the processors, but how exactly do we do this? Not only can we scale up the number of processors, but we can also scale up the number of guest operating systems. So, we have take a two-pronged approach to this:

1. Start with one processor running a workload on one guest; guest is assigned that processor. Add a processor and a new guest; new guest is pinned to the new processor. Scale to N processors.

2. Start with one processor running a workload on one guest; guest is assigned that processor. Add a processor, but not a new guest; assign the processor to the existing guest. Scale to N processors.

To compare these results, we also have two scenarios which do not use a hypervisor:

1. Start with one processor running a workload on Linux. Add a processor and a new instance of the workload on the same Linux OS. Scale to N processors.

2. Start with one processor running a workload on one Linux OS; Add a processor and repeat. Scale to N processors.

## 3 Tools

Virtualization of resources such as processors, I/O, and time can create some unique problems for scalability analysis. Abstracting these resources can cause traditional performance and resource analysis tools not accurately reporting information. Many tools need to be modified to work correctly with the hypervisor layer. The following were used to conduct this study.

### 3.1 sysstat package

The sysstat[4] package provides the `sar` and `iostat` system performance utilities. `sar` takes a snapshot of the system at periodic intervals, gathering performance data such as CPU utilization, memory usage, interrupt rate, context switches, and process creation. `iostat` takes a snapshot of the system at periodic intervals, gathering information about the block devices for the disks such as reads per second, writes per second, average queue size, average queue depth, and average wait time. Both of these utilities will only gather the statistics for the domain in which they are running. To get a more complete view of the system performance, it sometimes helps to run the utilities simultaneously in multiple domains. For example, `iostat` running in a guest domain will only gather the disk I/O statistics as seen within the domain. To get a more complete picture of the disk I/O behavior one can run `iostat` in Domain 0 to also gather the statistics for the backend devices that are mapped to the guest domain.

| CPU | Total Pct | Virtual CPUs |
|---|---|---|
| 0 | [070.5] | d0-0[006.3] d2-1[064.2] |
| 1 | [065.0] | d0-1[000.0] d2-2[065.0] |
| 2 | [072.3] | d0-2[000.1] d2-3[072.1] |
| 3 | [087.3] | d0-3[000.2] d2-0[087.1] |

Figure 1: Sample vm-stat output

### 3.2 vm-stat

As mentioned above, the `sar` utility gathers statistics for CPU utilization. When running in a guest domain, however, the CPU utilization statistics are meaningless since the domain does not have full usage of the physical CPUs. What is needed is a complete view of total system CPU usage broken down by domain. The `xentop` utility provides CPU utilization statistics, but it displays them in real time using an ncurses interface. `xentop` is not useful when running automated performance tests where the performance data are collected for later analysis. Therefore, we wrote a new utility, `vm-stat`, which queries the Xen hypervisor at periodic intervals to get the CPU utilization statistics, broken down by domain. `vm-stat` writes its output to standard output, making it useful for automated performance tests.

Figure 1 shows sample output from `vm-stat`. The first two columns show the physical CPU usage. The subsequent columns show how each physical CPU was allocated to a virtual

CPU in the various domains. For example, `d2-3[072.1]` states that the CPU was allocated to domain ID 2, virtual CPU 3 for 72.1% of the time.

### 3.3 Xenoprof

Xenoprof[5][6] adds extensions to OProfile[7].

From the OProfile *About* page:

> OProfile is a system-wide profiler for Linux systems, capable of profiling all running code at low overhead. OProfile is released under the GNU GPL.
>
> It consists of a kernel driver and a daemon for collecting sample data, and several post-profiling tools for turning data into information.
>
> OProfile leverages the hardware performance counters of the CPU to enable profiling of a wide variety of interesting statistics, which can also be used for basic time-spent profiling. All code is profiled: hardware and software interrupt handlers, kernel modules, the kernel, shared libraries, and applications.

From the Xenoprof home page:

> Xenoprof allows profiling of concurrently executing virtual machines (which includes the operating system and applications running in each virtual machine) and the Xen VMM [virtual machine monitor] itself. Xenoprof provides profiling data at the fine granularity of individual processes and routines executing in either the virtual machine or in the Xen VMM.
>
> Xenoprof is modeled on the OProfile profiling tool available on Linux systems. Xenoprof consists of three components: extensions to the Xen virtual machine environment, an OProfile kernel module adapted to the Xen environment, and OProfile user-level tools adapted to the Xen environment.

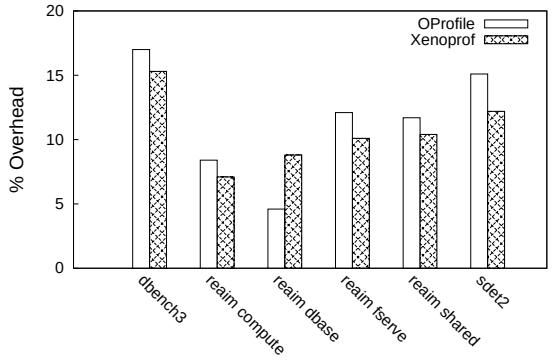


Figure 2: Overhead of OProfile and Xenoprof by benchmark

Others have contributed to Xenoprof since its release. Xiaowei Yang added enhancements to map IP samples of passive domains to the Xen/kernel symbol tables. Andrew Theurer added support for the x86_64 architecture.

One of our concerns was whether Xenoprof added sufficient overhead to change the characteristics of a workload being profiled. It would be difficult to diagnose performance issues of a certain workload if the profiler significantly changed the behavior of the system under the workload. We ran a series of benchmarks with and without profiling code to determine the profiling overhead for both OProfile and Xenoprof. The results are shown in Figure 2.

The data revealed that Xenoprof adds overhead that is comparable to that of OProfile. Since in

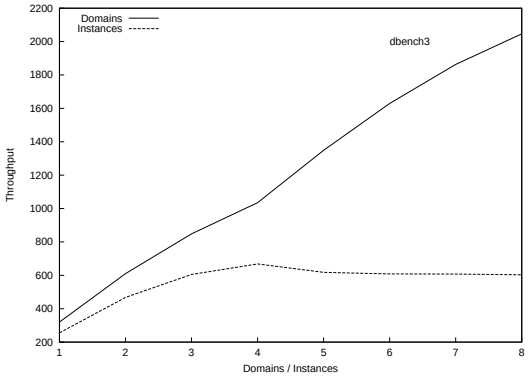


Figure 3: Dbench3 Aggregate Throughput

our experience we have not noticed the overhead of OProfile to significantly change the characteristics of a workload being profiled, we expect that Xenoprof will have a similar negligible effect on workloads that it is profiling.

# 4 Scaling up the number of domains

It is possible that in certain workload situations, running several small domains can yield performance that is preferable to that of running a single large Linux instance on the same hardware. One such instance is illustrated in Figure 3. In this particular case, we have the dbench workload running in two different configurations. In the first configuration, labelled *instances*, we are running one instance of dbench per CPU (the task is pinned on the CPU) on independently mounted tmpfs file systems. Each of these tasks represents a separate workload instance running side by side on the same system image. While a simple concept, this setup illustrates a fundamental scaling problem with the Linux kernel, the system wide dcache lock. Dbench stresses the dcache lock. The more work that is added, the more the system begins to strain under the load.

Xen offers an approach that can be used to overcome this deficiency and not only maintain performance but actually yield sizable performance advantages. By running separate Linux guest domains (each domain pinned on a single CPU, just like the dbench instances on baremetal Linux) we can effectively add more dcache locks which reduces contention and therefore increases performance. As the "Domains" line in the figure indicates, testing of this configuration on Xen yields a very high performance setup for this particular workload.

## 4.1 NUMA Support

Scaling up the number of domains on a large box can involve spanning multiple NUMA nodes. Without NUMA policies in the hypervisor, guests which have processors from one node may have memory allocated from another node, increasing memory latency and reducing throughput.

The core strategy for Xen on NUMA systems is to provide local resources whenever possible with exceptions for function. The two main resources are physical CPUs and memory. With minimal information extracted from ACPI tables, we are able to provide mechanisms for domains to acquire local resources leading to domains using CPUs and memory contained within a single NUMA node. Xen also allows for a domain's resources to span more than one NUMA node. Currently, we are not providing any optimizations for providing dynamic topology information to the guest, though that is an area of future work.

By utilizing the Xen's existing infrastructure for mapping virtual CPUs to physical CPUs, we create domains and specify which physical CPUs will be utilized initially by each domain. Exposing the topology of which physical CPUs are within which nodes of a NUMA system is

all that is needed to ensure proper selection of physical CPUs for an in-node domain.

The NUMA-aware page allocator for Xen strives to provide memory local to the requester as much as possible. We do, however, make exceptions for DMA pages. Without an IOMMU, we currently must prefer a non-local, but DMA-able page if a guest specifically requests such pages. Without this bias one can imagine a scenario where the resulting page is local to the requester but ultimately the guest cannot make use of a page beyond 32-bit DMA limits.

In responding to a domain's request for memory, we can utilize the vcpu to CPU mapping as a method of equitable distribution of memory across the nodes that a domain might be within. This ensures that a domain, no matter which virtual CPU is running, will have some memory local to the node to which the physical CPU belongs.

#### 4.1.1 Topology Discovery

Xen's NUMA topology discovery is dependent on the presence of an ACPI System Resource Affinity Table (SRAT). This table provides mapping information for memory and CPU resources to their respective proximity domain (node). Parsing of the memory affinity tables contained within the SRAT yields an array of physical memory address ranges and the node to which they belong. The CPU to proximity domain mapping populates a CPU to node structure. This discovery is invoked prior to initializing Xen's heap and provides topology information required for initializing the per-node heap array properly.

#### 4.1.2 Page Allocator Implementation

Xen's heap of free pages is implemented as a buddy allocator. The heap is split into three zones: xen, domain, and dma. There are various methods for requesting memory pages from each zone. To aide in handing out pages local to the node of the requester, we further divide each zone in to a collection of pages per-node. When we initialize and add pages to the heap, we determine to which node the pages belong and insert accordingly. When handing out pages, we can provide pages from the required zone (required for functionality when requesting DMA-able pages) by exhausting the available memory for the target node before using pages from a non-local node.

In addition to subdiving the heap's zones, we also wanted to preserve the existing, non-NUMA aware API for requesting pages from the heap. This allows us to progressively modify areas to make them NUMA aware through performance tuning.

#### 4.1.3 Performance with NUMA policy

Memwrite is a simple C program designed to calculate memory access throughput. We use it to stress the importance of node-local memory allocation. The benchmark allocates a buffer of memory and proceeds to write across the entire buffer. We calculate the throughput by dividing the size of the buffer and the time it took to write a particular value to the entire buffer. Memwrite also can fork off any number of child processes which will duplicate the write of the buffer. Multiple parallel writes add additional stress to the NUMA memory interconnect which is bandwidth and latency constrained as compared to local processor to memory access.

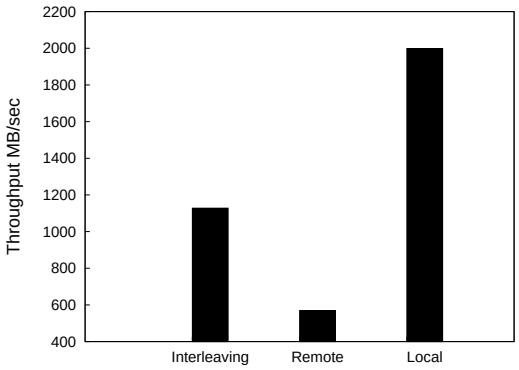


Figure 4: Memwrite with Various NUMA Policies

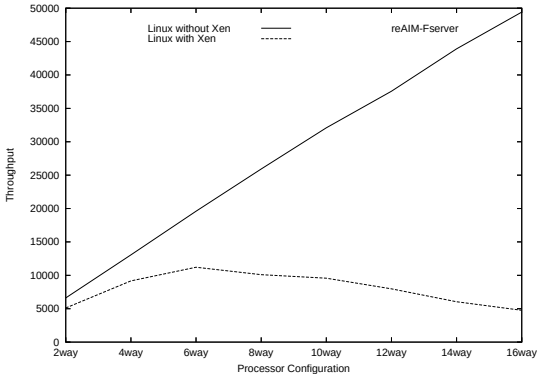


Figure 5: Guest scalability: baseline

Running baremetal NUMA-aware Linux, we utilized the `numactl` tool to force the memory and CPU selection to illustrate possible scenarios for guests running on NUMA hardware, but without NUMA support in Xen.

We benchmarked two worst-case allocation scenarios against allocating the memory local to each node. The results are shown in Figure 4. The first case, labeled *Interleaving*, distributes the memory across a set of nodes, which creates significant traffic over the interconnect. The second case, labelled *Remote*, selects CPUs from one node and the memory on a separate node. Both of these cases can happen without NUMA support to help allocate memory and processors. With NUMA-aware Xen, the processor and memory selection can be controlled to provide local-only resources resulting in increased performance.

# 5 Scaling up the size of a single domain

In this section we attempt to scale a single Linux guest, running on top of the Xen hypervisor, to many processors. In doing so, we show that some parts of the hypervisor inhibit the guest's ability to scale to 16 processors, compared to Linux running without a hypervisor.

For these tests, we are using reAIM benchmark's file server workload. This workload emulates the characteristics of a Linux file server. Running with just Linux, no hypervisor, yields excellent scalability to 16 processors. Throughput from 2 processors to 16 has a scaling factor of 7.47 (out of 8.0). The same Linux kernel, patched to with work Xen, has a scaling factor of 0.932 at 16 processors. The results from these two runs are shown in Figure 5. Obviously, we have some serious challenges prohibiting similar scalability.

## 5.1 Scaling Issues

There are several scaling issues with this type of situation. Most of them center around Xen's implementation of guest memory management. When a guest makes changes to an application's memory, it must keep Xen in the loop. Operations like page fault handling, fork, etc., require Xen's participation. This is critical to keep a guest from accessing other guests' memory.

Xen allows the guest to write directly to a temporarily detached page table. The guest can

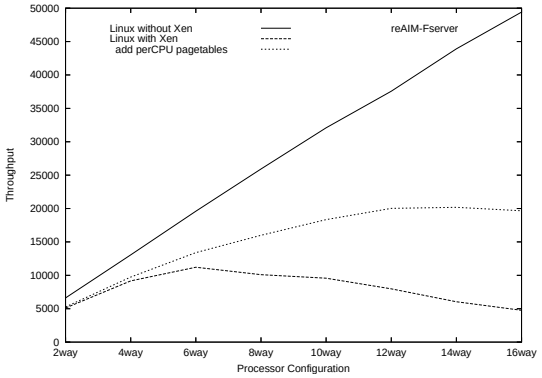


Figure 6: Guest scalability: added per CPU writable page tables

update this page table with machine addresses rather than pseudo-physical addresses. Since there is no shadow page table, Xen must verify the entries that are written by the guest. Virtually all of the operations to support this are protected by a single, domain-wide spinlock. The spinlock protects primarily the two (and only two) writable page tables per guest.

We have prototyped a change to support two writable page-tables per virtual processor and reduced the number of uses of the domain-wide lock. The results are shown in Figure 6. This alone can yield up to four times throughput gain on 16-way guests.

However, scalability still lacks on large processor guests. We turn to OProfile to find our next problem. As we increase processors in the guest, we see a disproportionate increase in functions related to TLB flushing. On closer inspection we see that most TLB flushes use a CPU bit-mask containing all processors currently running that guest. Many of these TLB flushes are for a particular application context, and thus only need to flush processors which are: (1) running the same guest and, (2) are running in the same context. As of this writing, we do not have a implementation for this, but we believe this could yield a substantial improvement.

There are also other issues with TLB flushing. Xen uses a system-wide spinlock when implementing an interprocessor interrupt (IPI) to flush many processors. Xen will flush the local processor's TLB, then acquire the lock, then send an IPI to other processors in the bit-mask to invalidate their TLB. With many processors trying to flush a set of processors at the same time, we have lock contention on the flush lock. Invariably many of the processors wanting to aquire this lock will wait, and in doing so, many of the processors it would like to flush will have been flushed while waiting for this lock. If we can determine which processors were flushed after we flushed our local processor, but before we acquired the lock, we can remove those processors from the IPI bit-mask.

To do this, we make use of an existing feature in Xen, the `tlbflushclock`. The tlbflushclock is a global clock which is incremented by all processors' TLB flush. Xen also keeps a per-processor clock value for their own most recent flush. So, we use this to reduce the bit-mask before we send an IPI to TLB flush. When a processor wants to flush a set of processors, it first flushes its own TLB, then records the tlbflushclock value. It then spins for the flush-lock, after which we check to see if any of the CPUs in the bit-mask have their own tlbflush-clock value greater (more recent flush) than our local flush. If so, those processors have already TLB flushed and don't need a flush again, reducing the bit-mask for the IPI. The effect of this patch is shown in Figure 7.

We still have some scaling challenges, so we asked the question, "Are the writable page tables really efficient on an SMP guest?" We hypothesize that with many processors, we have a much greater chance that writable page tables may be flushed back early, with few entries updated. The overhead to replicate and detach an entire page, only to flush the page back in and

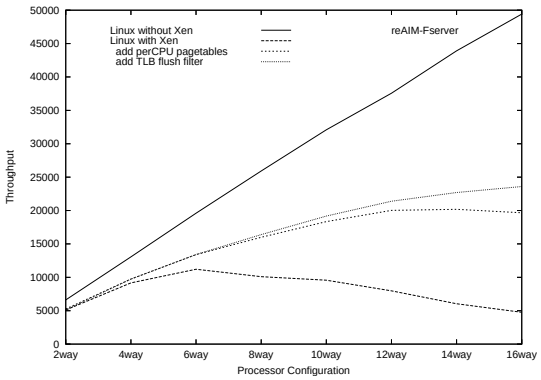


Figure 7: Guest scalability: added TLB filter

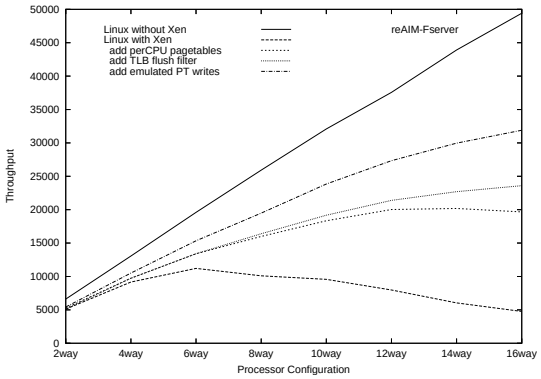


Figure 8: Guest scalability: forced page table write emulation

verify changes, may be too much if there are only a few changes to the page. Simply bypassing writable page tables and verifying each write as it occurs may have less overhead. So, we modified Xen to always handle each page table write fault individually. The results for SMP, shown in Figure 8, demonstrate that this is a more efficient method.

## 6 Conclusion

Although Xen's paravirtualization technique has previously shown to perform extremely well on uniprocessor guests on UP and 2-way SMP systems, this paper has exposed many challenges to achieve similar performance on large SMP systems. However, we believe that with more analysis and development, the Xen hypervisor will overcome these obstacles.

## 7 Trademarks and Disclaimer



## References

[1] `http://www-03.ibm.com/servers/eserver/zseries/virtualization/features.html`

[2] 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* SOSP'03, October 19–22, 2003, Bolton Landing, New York, USA.

[3] Ian Pratt, Keir Fraser, Steven Hand, Christain Limpach, Andrew Warfield, *Xen 3.0 and the Art of Virtualization,* Linux Symposium, Ottawa, 2005.

[4] `http://perso.wanadoo.fr/sebastien.godard`

[5] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, Willy Zwaenepoel, *Diagnosing Performance Overheads in the Xen Virtual Machine Environment,* First ACM/Usenix Conference on Virtual Execution Environments (VEE'05), Chicago, Illinois, June 11–12, 2005.5 `http://www.usenix.org/publications/library/proceedings/vee05/full_papers/p13-menon.pdf`

[6] `http://xenoprof.sourceforge.net`

[7] `http://OProfile.sourceforge.net/news`