Virtual Doppelgänger: On the Performance, Isolation, and Scalability of Para- and Paene- Virtualized Systems

Stephen Soltesz, Marc E. Fiuczynski*, Larry Peterson*, Michael McCabe+, Jeanna Matthews+*
*Department of Computer Science, Princeton University
+Department of Computer Science, Clarkson University

Abstract

Paravirtualization, popularized by the Xen hypervisor, is quickly expanding into commodity markets, with many in the IT sector considering it for a variety of purposes. It is appropriate for many usage scenarios, yet for those requiring strong isolation *and good performance and high scalability* there is at least one often overlooked alternative, which we call *paene*virtualization [1] Paenevirtualized systems are general-purpose, time-shared OSs retrofitted with abstractions to provide both namespace isolation and resource isolation. Examples of such systems include HP UX 11i Secure Resource Partitions, and Solaris 10 Zones, Virtuozzo for Linux, and Linux Vservers.

Both approaches to virtualization provide better isolation than traditional time-shared systems, but to a regular user, there is little tangible difference between the two; at a superficial level, one appears as a *virtual doppelgänger* of the other. Of course there are differences, and this paper digs below the surface to report on their strengths and weaknesses in terms of performance, scalability, and isolation.

1 Introduction

Operating systems face a fundamental tension between providing isolation and sharing among applications—they simultaneously support the illusion that each application has the physical machine to itself, yet let applications share objects (e.g., files, pipes) with each other. Today's operating systems designed for personal computers, adapted from earlier time-sharing systems, typically provide a relatively weak form of isolation (the process abstraction) with generous facilities for sharing (e.g., a global file system and global process ids). In contrast, hypervisors strive to provide strong isolation between virtual machines (VMs), providing no more support for sharing between VMs than the network provides between physical machines.

The point on the design spectrum supported by any given system depends on the workload it is designed to support. Workstation OSs generally run multiple applications on behalf of a single user, making it natural to favor sharing over isolation. Hypervisors are often designed to let a single machine host multiple unrelated applications, possibly running on behalf of independent organizations, as might be the case in a hosting or utility data center. In such a scenario, the applications have no need to share information, and in fact, it is important they have no impact on each other. Hence, hypervisors heavily favor isolation over sharing. However, when each virtual machine is running the same operating system and the same user level applications, the cost of this isolation can be unneccessarily high.

This paper investigates this tension between isolation and sharing in systems designed for an emerging scenario requiring scalability in the number of concurrently running VMs, especially where these VMs have substantial similarities in their software requirements. Examples of such scenarios include web/db/game hosting centers, utility data centers, and distributed hosting services such as Akamai and PlanetLab. Our study considers two general approaches to virtualization: *paravirtualization* as ex-

[1] Paene is Latin for "nearly" and pronounced just like the pasta.

emplified by Xen [1] and *paenevirtualization* as typified by Linux Vservers [10].

We have considerable experience with Vservers—i.e., a general-purpose Linux OS retrofitted with abstractions providing both namespace and resource isolation—as we use it to support a large number of researchers on PlanetLab [13, 2]. However, with the steep maturation of Xen from research prototype to production quality solution, there has been considerable momentum to use Xen for all scenarios involving VMs. For this reason, we take a closer look at Xen as an alternative to Vserver. In terms of performance, we find that Xen is a strong competitor to Vservers. However, especially in temrs of network I/O, we find that Vserver has the advantage due to the high CPU overhead required in Xen to manage the network. In terms of scalability, Vservers scale further than Xen for simple tests and have better response time when using a modest number of concurrently running VMs.

The paper is organized as follows. Section 2 teases apart the different requirements that users might place on virtual machine technology and characterizes the nature of the isolation properties of virtual machines, independently of how the underlying technology actually implements it. Section 3 compares reproduced performance results from Xen 1.2 with subsequent generations of Xen, and contrasts its performance with that attainable with Linux Vservers. Section 4 then evaluates the scalability of the two systems in the number of concurrently running VMs. Finally, Section 6 offers some concluding remarks.

2 Requirements

The purpose of virtual machine technology is to provide isolation between individual VMs. This section teases apart the different requirements that users might place on virtual machines. The discussion considers the full breadth of system-level support for virtual machines, from traditional OSs to low-level hypervisors. For clarity, we settle on a single set of terminology, drawn from the recent virtual machine literature: we refer to the underlying system as a virtual machine monitor (VMM) rather than an OS, and the isolated containers running on top of it as virtual machines rather than processes or domains.

We first outline the usage scenarios of VMs to set the context within which we compare and constrast the different approaches to virtualization. Next, we characterize the nature of the isolation provided by virtual machines, considered independently of how the underlying VMM is actually implemented. Finally, we discuss two different classes of VM technology from the standpoint of meeting the requirements of the usage scenarios demanding good performance, high scalability, and strong isolation.

2.1 Usage Scenarios

Virtual machine technologies are poised to be present throughout most computer systems. There are a number of exciting ideas to use VMs to secure work environments on laptops [18], ease management and improve utilization of large compute clusters, analyze unknown virus/work attacks in realtime [17], and debug difficult to track down system failures using time-travel [19]. However, today VM technology is predominantely used by: (1) *programmers* on workstations for software development and testing purposes, (2) *IT centers* to consolidate dedicated servers onto more cost effective hardware, and (3) *hosting* organizations to cost effectively sell virtual private servers at scale.

These real-world scenarios have similar requirements in terms of isolation between VMs—it needs to be strong. However, they differ substantially in their need for scalability, performance and the ability to support multiple operating system environments. For example, programmers or testers are unlikely to run a number of heavily loaded VMs concurrently, and while they would prefer good performance, their tolerance for poor performance is likely to be high. On the other hand, they are likely to depend heavily on the ability to support mulitple operating system environments. In fact, the ability to run multiple software development and testing environments on the same hardware is the primary

advantage of VM technlogy in this scenario.

As another example, IT centers tend to consolidate on the order of a dozen physical servers into VMs running on a more cost effective system. They rely on the ability to run multiple VMs concurrently (one per service), but in many environments, the load on each service may be relatively light. For example, a small company may maintain their web server, mail server, DNS server, and ftp server all on one physical machine. By placing each service in its own VM, they can limit the impact of a software failure to one service and also simplify the administration of each service. In this scenario, there would be little benefit to sharing due to the user-level server software, but it is likely that multiple services would run on the same operating sytem.

In a third scenario, organizations selling virtual private servers at scale (e.g., web hosting companies, Akamai, PlanetLab, and so on.) likely have many copies of the same server software and underlying operating systems represented in their mix of VMs. They seek to benefit from an economy of scale and need to reduce the marginal cost per customer VM. Thus they are likely to be extremely sensitive to issues of scalability and performance as they try to carefully oversubscribing their physical infrastructure with as many VMs as possible without reducing overall quality of service.

Of these three scenarios, we focus primarily on the third which stresses the underlying virtual machine technology to its limits along the performance and scalability axes. While it is non-trivial to ensure isolation between VMs, the real challenge is to provide isolation *and high performance and scalability*.

2.2 Isolation Landscape

Isolation between VMs involves two largely independent dimensions: *resource isolation* and *namespace isolation*. Note that we do not use the term *fault isolation* in our characterization of VMMs, instead taking the view that fault isolation is implied by resource isolation: a VMM that provides resoruce isolation protects all correctly running VMs from both greedy and faulty VMs. After all, a fault that crashes the machine robs all VMs of the resources they expect.

Resource isolation corresponds to the VMM's ability to isolate the resource consumption of one VM from that of another VM; undesired interactions between VMs is sometimes called cross-talk [9]. Providing resource isolation generally involves careful scheduling and allocation of physical machine resources (e.g., cycles, memory, link bandwidth, disk space), but can also be influenced by VMs sharing logical resources, such as file descriptors and memory buffers. A VMM that supports strong resource isolation might guarantee that a VM will receive 100 million cycles per second (Mcps) and 1.5Mbps of link bandwidth, independent of any other applications running on the machine. On the other hand, a VMM that supports weak resource isolation might let VMs compete with each other for cycles and bandwidth on a demand-driven (best-effort) basis. Many hybrid approaches are also possible. For instance, a VMM may maintain strong performance isolation between classes of VMs while enforcing weaker isolation within each class (e.g., to support a low priority class of applications that are allowed to consume excess cycles).

In contrast, namespace isolation refers to the extent to which the VMM limits access to (and information about) logical objects, such as files, memory addresses, port numbers, user ids, process ids, and so on. A VMM that supports strong namespace isolation does not reveal the names of files or process ids belonging to another VM, let alone let one VM access or manipulate such objects. In contrast, a VMM that supports weak namespace isolation might support a shared namespace (e.g., a global file system), augmented with an access control mechanism that limits the ability of one VM to manipulate the objects owned by another VM.

The level of namespace isolation supported by a VMM affects at least two aspects of application programs. The first is *configuration independence*, that is, whether names (e.g., of files) selected by one VM possibly conflict with names selected by another VM. The second aspect is security; if one VM is not able to see or modify data and code belong-

ing to another VM, then this increases the likelihood that a compromise to one VM does not affect others on the same machine.

Taken together, these two dimensions of isolation provide a simple map of the design space, with different VMMs (OSs) corresponding to each possibility:

- Traditional timesharing systems offer both weak performance isolation and weak namespace isolation.
- Single-user multimedia systems offer strong resource isolation but weak namespace isolation.
- A traditional timesharing system augmented with support for security contexts, as exemplified by BSD jails [8], offer weak resource isolation and strong namespace isolation.
- A VMM designed for application hosting centers typically offer both strong resource isolation and strong namespace isolation.

It is important to keep in mind that this section puts forward just one possible view of the isolation landscape. There are certainly other criteria by which VMM strategies can be evaluated. For example, some VMMs offer the advantage of supporting more than one OS environment. As another example, a VMM can be characterized by its ability to migrate VMs between physical machines to avoid scheduled downtimes (e.g., for hardware maintenance) or to load-balance VMs. Finally, VMMs can be characterized according to the size of its trusted code base, the premise being that the less code there is to assure, the more secure the system is likely to be. While a VM that compromises the VMM is able to gain access to another VM's state—and hence, compromise the level of security supported by the VMM—there are many other security threats (e.g., physical access to the machines) that must first be addressed before this particular security issue becomes critical. Instead, we purely focus on isolation as it relates to performance and scalability, and return to these other criteria in Section 6.

2.3 Design Choices

Based on the current state of VMM research, it appears that two technologies are best positioned to support good performance, high scalability, and strong isolation: paravirtualizing systems such as Xen [1] and Denali [20], and paenevirtualizing systems—i.e., a general-purpose OS retrofitted with abstractions providing namespace and resource isolation—such as Linux Vservers [10], Solaris 10 (Zones + PRM), and Virtuozzo for Linux [16].

Figure 1 illustrates the architecture of these virtualization approaches at a Birdseye level. Shown on the left, a hypervisor-based VMM runs separate copies of a conventional operating system kernel inside each VM; paravirtualizing hypervisors achieve greater scalability by modifying the OS running in each VM to make it aware that the hypervisor is beneath it. Shown on the right, a paenevirtualized system supports multiple UNIX VMs using a single OS kernel, with each VM appearing to the user as a separate UNIX system. The homogeneous nature of paenevirtualized VMs simplifies the underlying kernel's task of managing a large number of such VMs.

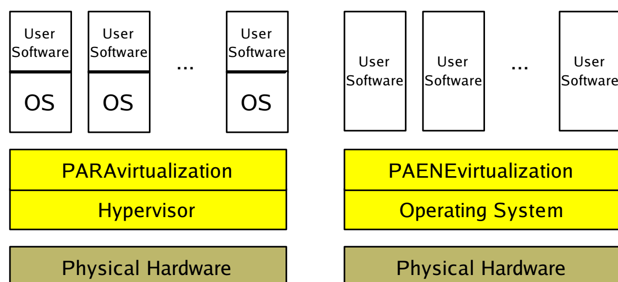

Figure 1: **Doppelgänger nature of para- and paenevirtualized systems.** At a birdseye level the main difference between these two systems is the trusted computing base. Paravirtualized systems let users run their own *untrusted* operating system within a VM, paenevirtualized systems do not.

The difference between paenevirtualized UNIX and traditional UNIX systems is the means by which the kernel verifies whether a process is authorized to perform a given action. The traditional multiuser UNIX model of an omnipotent `root` and

a number of less privileged user accounts is insufficient for paenevirtualized UNIX; every VM requires some superuser-like privileges, such as the ability to install packages, but must be denied others, such as the ability to load modules into the kernel. Paenevirtualized UNIX systems typically split the privileges traditionally granted to `root` into a number of finer-grained capabilities, and provide only a subset of those capabilities to the "root" inside a user-owned VM. By careful consideration of the capabilities needed to perform different roles associated with UID zero, it is possible to provide `root` within a VM with those capabilities that are necessary for maintaining the VM (e.g., manipulation of file ownership and permission), while removing those that could potentially affect other VMs on the same machine (e.g., loading a kernel module).

When ignoring performance, isolation, and scalability, a difference between paravirtualized and paenevirtualized systems is the *trusted computing base* (TCB). For paravirtualized systems, the TCB is the small underlying VMM. The OS running in each VM can be untrusted—i.e., in scenarios where customer may use their own custom OS. For paenevirtualized systems, only conventional user-level applications are untrusted, everything else is part of the TCB. It is important to note this difference, as prior evaluations of paravirtualized systems primarily demonstrate isolation between VMs using untrusted applications rather than untrusted OSs. Section refx shows that untrusted OSs running in a paravirtualized VM *can* successfully launch layer-below attacks, undercutting a previously established reason for using paravirtualized systems.

Finally, to understand many of the results reported in subsequent sections, it is helpful to recognized that paravirtualization is often achieved by *duplicating* resources (e.g., each VM has its own copy of the OS), whereas paenevirtualization is often achieved by *hiding* or *masking* resources (e.g., each VM is able to see only a subset of the file system). In other words, paenevirtualization achieves namespace isolation by layering a set of filters on top of the underlying logical resources, and it achieves resource isolation by layering a set of limits on top the underlying physical resource schedulers and allocators. In effect, this paper reports the differences between duplicating and hiding various components of the OS.

3 Performance

This section compares the performance of a paravirtualized system (Xen + XenoLinux) and a paenevirtualized Linux Kernel (PLK). As a reference point, we run many of the benchmarks used to evaluate those systems also on a stock Linux kernel. We repeat a number of the single VM benchmarks as described in Barham et. al. [1] and Clark et. al [3], which illustrate the base performance of these two systems. These experiments run within a single VM and exercise the whole system to characterize the relative performance of the two approaches for a range of server-type workloads.

Our Xen configuration consists of Xen 2.0.7 with XenoLinux based on a patched version of the vanilla 2.6.12 kernel from kernel.org in both the host (dom0) and guest domain (domU) for which results have not yet been reported elsewhere. Unless otherwise stated, the results reported for Xen are for a single guest domain. Our PLK configuration consists of a Linux 2.6.12 kernel patched with changes from FC4-1398, Vserver 2.0.1 and a new CPU scheduler developed at Princeton. To account for the difference between different variations of 2.6.12 used by XenoLinux and PLK, we also present the base performance numbers of stock FC4-1398 2.6.12 Linux kernels. The Linux distribution populating the filesystem is the latest release of Fedora Core 2, thereby letting us focus on comparing the architectural difference between para- and paene- virtualization. Where possible, we use the exact same filesystem to avoid variances due to using different portions of the disk.

All experiments are evaluated on an Intel SE7505VB2 motherboard with a 3.06Ghz Xeon uniprocessor, 4GB RAM, Intel E1000 GigE Ethernet, and a single Seagate Barracuda ST3200822A 200GB 7.2k RPM ATA-100 IDE disk. The Xeon processor has a 512KB L2 and 1MB L3 cache and

Config	Linux FC4	PLK	Xeno Linux
select TCP	5.91	5.92	**4.60**
mmap latency (32MB)	425.00	428.00	**1300.00**
mmap latency (64MB)	846.00	844.00	**2575.00**
fork process	127.57	**130.90**	**378.08**
exec process	462.58	**488.23**	**697.42**
sh process	1565.57	**1711.90**	**1741.60**
CS (2p/ 0K)	1.76	1.87	**3.12**
CS (2p/16K)	1.85	1.96	**3.12**
CS (2p/64K)	2.07	2.09	**3.29**
CS (8p/16K)	2.37	2.52	**3.92**
CS (8p/64K)	11.24	11.87	**14.07**
CS (16p/16K)	3.31	3.41	**4.88**
CS (16p/64K)	23.65	**26.35**	**28.10**
prot fault	0.68	0.69	1.07
page fault	2.31	2.35	**4.00**

Table 1: lmbench latency OS benchmarks - times in μs

hyperthreading disabled. To compare performance between Xen 1.0 and Xen 2.0, we also repeat the base performance evaluation on the identical hardware used by Clark et. al. to reproduce the results reported by Barham et. al.

3.1 Base Operating Systems Benchmarks

We ran the *development* benchmark subset of McVoy's *lmbench* [11] version 3.0-a3 tool, as those experiements target particular OS subsystems. For 17 of the 30 lmbench DEVELOPMENT microbenchmarks, the performance of PLK, XenoLinux, and stock Linux are nearly identical; i.e., within the margin of error. Table 1 shows the results of the latency microbenchmarks for which there is a performance discrepancy between stock Linux, PLK, and XenoLinux.

The first row in Table 1 shows that latency to handle TCP select is a bit faster for XenoLinux when compared to stock Linux and PLK. This is the only case where XenoLinux has lower latency, which happens to be due to minor code path difference between the vanilla 2.6.12 and FC4-1398 2.6.12 kernels.

The next two rows show mmap latencies for 32MB and 64MB files. As can be seen the latencies scale wrt to the size of the file, rather than there being a constant overhead for XenoLinux. While this makes perfect sense, we were initially surprised because previous discussions with respect to this benchmark made it seem like it would be a constant overhead of a few tens of microseconds.

The following three rows in Table 1 show the performance of *fork process*, *exec process*, and *sh process* across the systems. XenoLinux is slower when compared to stock Linux. Barham et al. [1] describe why this is the expected behavior for XenoLinux: hypercalls to the VMM update a large number of page tables. PLK is nearly identical for the fork sh benchmark, a bit slower for exec process, and only a bit faster than XenoLinux for sh process. Technically, PLK should more closely match stock Linux for these latter two benchmarks and we are still investigating what causes the additional delay.

The next seven rows in Table 1 show context switch overhead between different numbers of processes with different working set sizes. As explained by Barham et al. [1], the $1\mu s$ to $4\mu s$ overhead for these microbenchmarks are due to hypercalls from XenoLinux into the VMM to change the page table base. In contrast, there is little overhead between the stock and paenevirtuailzed versions of Linux.

Finally, for protection fault and page fault handling, XenoLinux is more than 1.4x slower than stock Linux, which is also due to Xen VMM hypercalls. For PLK the cost for these operations are essentially identically to stock Linux.

Table 2 shows the results of the bandwidth based microbenchmarks for which there is a performance discrepancy between stock Linux, PLK, and XenoLinux. The first three rows show an odd theme: XenoLinux drastically outperforms both stock Linux and PLK for strictly user-level benchmarks. The differences between these should be much smaller. We have run these benchmarks with sufficient *warm up* iterations to factor out cache effects and page fault effects and continue to get the same results. Not shown in the table are the

Config	Linux FC4	PLK	Xeno Linux
bcopy (libc)	597.70	583.08	729.12
bcopy (hand)	585.72	571.82	744.62
mem write	834.60	795.22	1038.00
mem read	2237.14	1910.02	1933.02
pipe	1993.80	1906.32	2294.30
AF UNIX	2838.82	2768.52	2668.70
file reread	1739.16	1656.12	1809.22
mmap reread	2241.82	1922.36	1942.62

Table 2: lmbench bandwidth OS benchmarks - in Mbytes/sec

bandwidth numbers for the host domain (dom0) under Xen. For the bcopy (libc) benchmarks, it falls right in between PLK and XenoLinux—i.e., 660Mbytes/sec. The cause for this is not well understood. One difference between these systems is that XenoLinux uses a 100HZ clock, whereas PLK and stock Linux use a 1000HZ clock[2]. Clock interrupts in conventional Linux would need to be on the order of 8μs to account for the difference in performance. However, that does not account for the difference between the results for host versus guest domain results for Xen. Further investigation into these discrepancies is required.

The bottom four rows listed in Table 2 do involve the operating system and in XenoLinux's case the hypervisor. Here the results are mixed. There are two takeaways from these results: (1) overall the performance of XenoLinux is remarkably good compared with stock Linux and PLK, (2) yet microbenchmarks seldom are indicators of overall system behavior for real workloads [4].

3.2 Network Performance

We examine TCP performance over a private Gigabit Ethernet LAN that is unloaded. We use a HP DL320 server with a 3.4GHz Xeon uniprocessor system that generally is faster than our test system. This lets us measure receive and transmit performance independently to a machine that can both source and sync data faster than our test system. The *iperf* benchmark was used to perform these measurements. Both sender and receiper applications were configured to use different TCP window sizes, and iperf selects the appropriate socket buffer size that is larger than the max bandwidth delay product. The results presented are a median of four tests running for 10 seconds, transferring between 400MB-1GB.

[2] We were negligent not to eliminate this difference, but are out of time to correct for the submission.

Config	Linux FC4		PLK		Xeno Linux	
	rcv	snd	rcv	snd	rcv	snd
TCP Win 16K	524	593	524	593	348	425
TCP Win 32K	888	938	888	938	678	815
TCP Win 64K	941	941	941	941	787	867
TCP Win 128K	941	941	941	941	888	915
TCP Win 256K	941	941	941	941	902	929

Table 3: iperf performance with TCP windows sizes ranging from 16K to 256Kbytes.

Table 3 presents the results using the default MTU of 1500 bytes. Both stock FC4 Linux and PLK achieve line GigE rate with a TCP window size of 64Kbytes. In contrast, XenoLinux approaches GigE rates only with large window sizes.

While this might seem unremarkable, achieving near GigE rates is a significant achievement for XenoLinux on Xen 2.0. This is because packets flow from the GigE NIC to the host domain that then forwards them to the XenoLinux guest domain. In this way the Xen 2.0 hypervisor no longer needs to replicate iptables-like forwarding support, as was done in Xen 1.0.

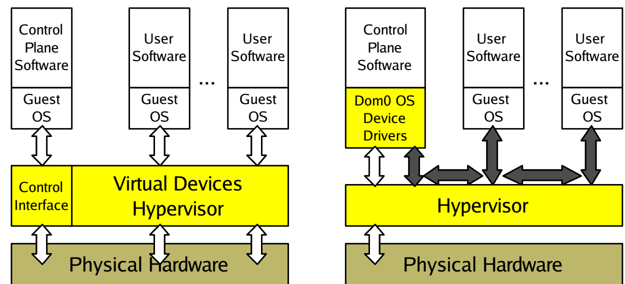

Figure 2: **Xen 1.0 vs. Xen 2.0 architecture.**

Figure 2 depicts the high-level architectural design of Xen 1.0 and Xen 2.0, and illustrates the path packets take through the system. The salient difference between these two generations of Xen is the move of device drivers out of the hypervisor. For Xen 1.0 the Guest OSs (such as XenoLinux) use Xen-aware device drivers to access phyical devices. In contrast, for Xen 2.0 privileged Guest OSs (typically dom0) can use their native device drivers via a safe hardware interface exported by the hypervisor; however, unprivileged Guest OSs use special *front-end* device drivers that communicate with dom0 to access the disk or receive network packets. Fraser et al. [5] describe the new I/O architecture for Xen 2.0. A key contribution of their architecture is the combination of high performance with resilience to device driver faults, the need for which is motivated by Swift et. al. [15, 14].

The Xen 2.0 I/O architecture resembles a conventional microkernel model with device drivers running in separate VMs (address spaces), as exemplified by QNX [7]. Hand et al. [6] argue that VMMs are microkernels done right. Certainly it seems that Xen's *cut* across the software stack is at the right level and achieves remarkable performance.

Nonetheless, the approach imposes a non-trivial amount of overhead. Menon et al. [12] profile Xen 2.0 to characterize the overhead of their new model under high-bandwidth scenarios. They report that on the receive path the TLB miss rate—defined as the ratio of the number of misses to the number of instructions executed—is an order of magnitude higher (14x - 25x) when compared to stock Linux. Their results also show that there are no specific hotspots, but rather the TLB misses in the driver domain are spread across the entire TCP receive code path. The bottom line is that for a GigE bandwidth benchmark Xen fully utilizes the 2.4GHz Xeon test system and cannot achieve GigE line rate to XenoLinu, while stock Linux has cycles to spare under the same load.

In contrast, we observe that Xen on a 3.06GHz Xeon CPU can nearly operate at link rate. In the future, PlanetLab nodes will be operating as multihomed routes with GigE NICs. For this reason, we are very interested what kind of forwarding rates can be achieved using Xen and Vserver with two GigE NICs. We developed a very simply TCP forward proxy that reads packets from one socket and writes (forwards) them as quickly as possible to another socket. Running this application on XenoLinux with two GigE NICs we observe a forwarding rate of 417Mbits/sec with a 128K TCP window. In contrast Linux and PLK can forward packets at GigE link rate (941Mbits/sec) with a 64K TCP window.

3.3 Relative Benchmark Performance

Overall the single guest performance of XenoLinux is remarkably good with it matching or even besting base Linux in most cases. Data demonstrating this for Xen1 versus Linux 2.4 was presented in [1] and independently verified in [3]. In Figure 3, we update this data for Linux 2.6 and Xen 2 including measurements of osdb, dbench, a Linux kernel compile, FourInARow,a CPU intensive component of Freebench and the full geometric mean of all the Freebench tests. This data was taken on the same hardware platfrom as [3] specifically a Dell 2650 dual processor 2.4 GHz Xeon server with 2 GB RAM and a Maxtor 36 GB 15000 RPM SCSI drive. Xen guests are allocated 128 MB of memory and a 2 GB LVM backed virtual block device. Each score reports the average of 10 trials and the standard deviation of the trials is shown with error bars. For the most part, the data confirms that newer versions of Xen retain their excellent performance relative to base Linux. However, there are some interesting details.

First, for some benchmarks, especially dbench, performance is significantly worse in domain0 than in a regular guest VM. This reflects the fact in domain0, one processor is running both the benchmark code and handling the disk I/O. While for a regular guest OS, the benchmark code and the disk I/O handling use different processors. To illustrate this, we include measurements on Xen2 with SMP disabled. This forces the 1 guest case to run both the benchmarking code and disk I/O as in the domain 0 case. (Note: we disabled SMP by using "xm pincpu" to

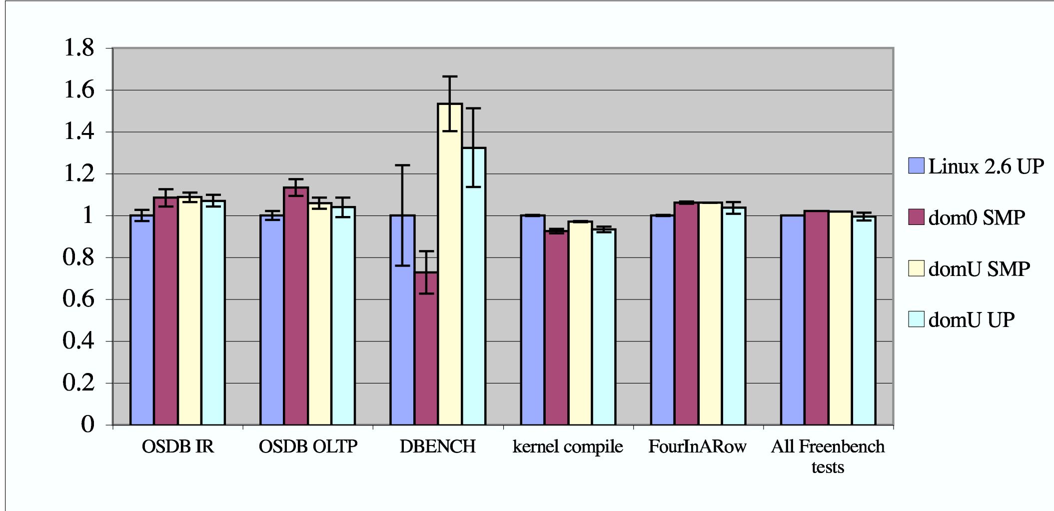

Figure 3: **Relative performance of Linux and XenoLinux.** Four sets of measurements for each benchmark are shown for each benchmark - Linux 2.6 UP has SMP disabled, Xen2 dom0 with SMP enabled, Xen2 guest domU with SMP enabled and Xen2 guest domU with SMP disabled. All scores are the average of 10 runs of the benchmark with standdard deviation shown with error bars.

ping both domain0 and the 1 guest to the same CPU.

Second, Xen is actually slightly better then base Linux for most of the tests run. It certainly seems counterintuitive that introducing a virtualization could improve performance. However, we believe that there are several things are work here. For Xen, the LVM-backed VBD is optimized for throughput rather than latency while Linux is optimized more for latency. For these benchmarks, we expect that optimization for throughput is more effective. Also, on Xen, all I/O is going through two elevator sorting algorithm and the lower level deeper queue allows for more pipelining of parallel requests. A recent paper by Fraser et al. [5] also describes an oscillatory behavior of Linux 2.4 memory system when doing bulk writes. The Xen implementation avoids this problem. It is somewhat surprising that Xen would show an advantage on a CPU intensive application like FourInARow. However, we saw the same slight advantage for all benchmarks in the Freebench suite. We summarize these by including the geometric mean of all the Freebench suite in the figure as well.

4 Scalability

This section evaluates the scalability of Xen and Vservers. We first evaluate how many active VMs can be instantiated on each system. Then we repeat a number of the scalability benchmarks reported by Barham et. al. [1] and compare the performance of the two systems. All experiments are evaluated on the same hardware as described in Section 3.

4.1 Phone Booth Packing

Phone booth packing is a contest of cramming as many people as possible into a booth designed for one person. We have a similar benchmark to determine the upper limit of packing *concurrently active* VMs onto a single physical host. It essentially is a footprint benchmark measured in terms of available disk and memory space. We evaluate the virtualization approaches in two dimensions: guaranteed reservation vs. overbooking. For guaranteed reservation the underlying VMM admits only the number of VMs that it can guarantee the availability of a reserved amount of resources. For overbooking, the VMM admits more VMs and services them on

a best effort basis. The assumption being that in the common case VMs will utilize far less resources than their maximim limits permit.

With guaranteed reservations (i.e., 100MB memory and 1GB disk space) both systems scale similarly, as the limiting factor is available memory and disk space—you can only cut a pie so often. The real challenge is in reducing the footprint of individual VMs when overbooking the physical resources.

In general, for hypervisor-based systems there is an inherent overcommittment of virtual memory pages and disk blocks to each VM. To reduce the memory footprint, such systems need to reduce duplicate state–introduced by each VM running a Guest OS and its applications–*after the fact*. While there are several proposed techniques to reduce duplicate memory pages [18, 17], each comes with its own drawbacks. For example, content-based page sharing techniques [18] compute hash values by scanning pages to identify duplicate pages and then uses CoW virtual memory mechanisms to reduce the memory footprint. Similarly, the delta-virtualization techniques described by Vrable et. al. [17] are very specialized for short-running applications (operating at the level of seconds or minutes)—not the usage scenarios we are investigating.

These memory footprint optimization techniques could also be applied to paenevirtualized systems to reduce memory footprint. However, their primary benefit is to reduce the text segments of common applications. When properly configured, this is something that paenevirtualized systems already do inherently. For example, on PlanetLab and other Linux Vserver-based systems (such as Lycos Europe), the filesystem is shared using a file-level CoW scheme. Many of the common executables and shared libraries are thereby shared on disk between VMs (vservers). More importantly, their inodes are the same and file paths appear the same across VMs. This setup lets the dynamic loader map them into virtual memory such that they are shared read-only across processes, even those that run in separate VMs.

Squeezing as many VM's filesystem onto a disk as possible, though, is more an exercise in frugality and ease of use rather than in reducing duplicate state. The ideal case for Xen might be to use file-backed virtual block devices (fVBD). A fVBD can be configured as a sparse file, which lets one nicely overbook the amount of disk space. However, the current implementation of the backing store for fVBDs caches I/O in the host domain, i.e., it behaves more like a RAM-disk that eventually persists the data to the backing store. This in fact provides remarkable performance, yet it does not meet the kind *durability* requirements that databases, email systems, etc. require—people typically are not happy when a *disk* looses their data. For this reason, one needs to use LVM-backed VBDs on Xen as the storage devices for VMs. One could start with a small LVM extent and then dynamically increase it when the VM requires more disk space. For example, a XenoLinux guest OS using ext3 could dynamically resize its filesystem in response to the LVM being resized. Yet this requires explicit cooperation by the guest OS.

In contrast, overbooking disk space is very simple with the (previously mentioned) file-level CoW scheme used by Linux Vservers. Each vserver filesystem is essentially a subtree of the overall filesystem, into which a vserver is confined using the `chroot` system call. The file-level CoW scheme reduces the otherwise 150Mbyte footprint required for a small Fedora Core 2 installation down to a 12Mbyte. Vservers provides a simple disk-space and inode-count limit placed on each vserver to prevent it from completely consuming all of the system's space or inodes. For example, on PlanetLab we routinely put 250-300 vserver VMs onto a single node and give each VM a 5GB disk quota, while the underlying disk itself is only 100-150GB large.

To end this discussion with some emperical data, we examined XenoLinux and Vservers' ability to instantiate a large number of VMs concurrently. Barham et al. [1] showed that it is possible to host roughly 128 VMs running SPEC INT2000 using XenoLinux. This is not a real-world workload corresponding to any of the scenarios we discussed in

Section 2. Instead, we configured a number VMs with Apache version 2.0.46 web servers in its default configuration. With Vserver we stopped when we reached 1024 VMs running Apache on our hardware configuration—we could have gone further. Note that each in its default configuration Apache creates up to 8 httpd processes. On the same hardware with Xen we only succeeded to install and run 80 VMs each limited to 48MB. We did not use the balloon driver with which we probably could have instantiated 1.5x-3x more, which nonetheless would have been 5x-7x lower compared to PLK.

4.2 VM Concurrency

This section compares the performance of running multiple VMs concurrently. Our focus is on running multiple instances of SPEC WEB99, similar to the configuration used by Barham et al.

Achieving good SPEC WEB99 scores equires both high throughput and bounded latency. When a client request gets stalled due toa badly delayed disk read, then the connection will be classed as *non-conforming* and will not contribute to the score.

Figure 4 shows the results of running 1, 4, 16 copies of the SPEC WEB99 benchmark in parrallel on our uniprocessor machine. For both PLK and XenoLinux each instance of SPEC WEB99 exercises a separate Apache server isolated into its own VM. Different TCP port numbers were used for each web server to enable the copies to be run in parallel.

Note that in Figure 4 the base case (V1) –running a single instance of Apache for SPEC WEB99 on Linux Vserver– is significantly better than the corresponding base case for stock Linux reported by Barham et al. Based on our experience, stock Linux is faster than PLK in this case. Given that our Xen (X1) base case roughly mirrors their prior results (550 conforming clients), we speculate that the bulk of this performance difference is due to horrible overhead imposed by the 2.4.x based LinuxSMP system in handling the multiple Apache processes.

The better overall performance of PLK is primarily due to the lower overhead imposed by the paenevirtualization approach. As a result, there simply is more CPU left to serve clients. In contrast, as revealed in Section 3.2, at GigE data rates there are few cycles left even for one XenoLinux to process client requests. The performance for Xen certainly would be better if we enabled SMP. However, in comparison to PLK, this would only have masked the CPU overhead imposed by paravirtualization and their microkernel-style I/O architecture for Guest OSs.

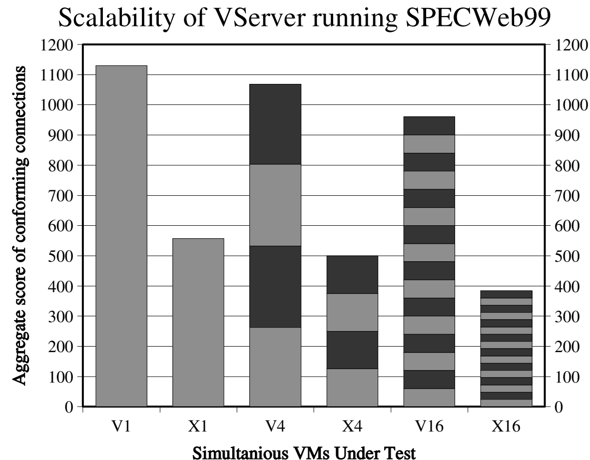

Figure 4: **Web server performance**

Note that PLK's CPU scheduler is identical to the Linux scheduler, augmented only to limit the amount of CPU a VM (vserver) can maximally consume. As we scale from 1 to 16 VMs, it ensures that each Apache server obtains a fair share of the CPU.

5 Isolation

Barham et. al. [1] wanted to do a backoff between Xen and other systems, but they lacked a freely available paenevirtualized Linux solution free available. In this section we discuss the isolation properties of PLK and compare it with Xen.

As one simple example, we ran a Quake III game server in one VM for both Xen and PLK and measured the latency to the server from another system on the local LAN. We then instantiated 9 antisocial VMs each running a CPU hog (infinite loop). Xen's borrowed virtual time scheduler does a good job at isolating the Q3 server from the CPU hogs,

with latencies varying between 1-15 milliseconds. As mentioned previously, PLK's CPU scheduler is identical to the Linux scheduler, augmented only to limit the amount of CPU a VM (vserver) can maximally consume. It can schedule the processes of each vserver at a fine grain and treats the Q3 server processes as an interactive process. The Linux scheduler is optimized for interactive processes, giving such higher priority over CPU bound processes. Consequently, even with CPU hogging processes running in 9 different vservers, the round trip packet latencies to the server remain below one millisecond *most of the time*[3].

PLK can also properly resource isolates VMs running fork-bombs and other antisocial processes. However, it currently lacks support to isolate processes that cause excessive disk I/O (e.g., a sustained dd). We are working on adding such a disk I/O resource controller that acts as a filter to ensure that all vservers (VMs) obtain a fair share of the I/O bandwidth.

Barham et al. point out that Linux cannot run multiple instances of Postgresql due to conflicts in the SysV IPC name-space. On PLK this problem no longer exists, as the SysV IPC name-space is virtualized for each Vserver security context. Consequently, we can run multiple instances of OSDB; *however, we ran out of time to complete those measurements in time for the submission*.

6 Conclusion

Paravirtualization, popularized by the Xen hypervisor, is quickly expanding into commodity markets, with many in the IT sector considering it for a variety of purposes. In this paper, we have compared the performance of Xen to Linux Vservers which represents a viable option to Xen for many usage scenarios that require both strong isolation *and good performance and high scalability*. Linux Vservers are one example of a class of systems that we call *paene*virtualization systems. Paenevirtualization systems are general-purpose, time-shared OSs retrofitted with abstractions to provide both namespace isolation and resource isolation. We find that the performance of Xen is quite competitive with Linux Vservers offering a substantial advantage in lower overhead for network intensive applications. While paenevirtualization systems do not offer some advantages of paravirtualization such as the ability to support multiple OS environments, we conclude that for many environments, especially those that are oversubscribing their physical infrastructure with as many VMs as possible without reducing overall quality of service, paenevirtualized systems are a more attractive alternative.

[3]Our scheduler implementation was finished two weeks ago and we suspect a minor bug in the token bucket handling that causes 90 millisecond hick ups every four seconds.

References

[1] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the Art of Virtualization. In *Proc. 19th SOSP* (Lake George, NY, Oct 2003).

[2] Bavier, A., Bowman, M., Culler, D., Chun, B., Karlin, S., Muir, S., , , and and, T. S. Operating System Support for Planetary-Scale Network Services. In *Proc. 1st NSDI* (San Francisco, CA, Mar. 2004).

[3] Clark, B., Deshane, T., Dow, E., Evanchik, S., Finlayson, M., Herne, J., and Matthews., J. Xen and the art of repeated research. In *Proc. USENIX '04, Freenix Track* (Jul 2004).

[4] Draves, R. P., Bershad, B. N., and Forin, A. F. Using Microbenchmarks to Evaluate System Performance. In *Proc. 3rd Workshop on Workstation Operating Systems* (Apr 1992), pp. 154–159.

[5] Fraser, K., Hand, S., Neugebauer, R., Pratt, I., Warfield, A., and Williamson, M. Safe Hardware access with the Xen Virtual Machine Monitor. In *Proc. First OASIS* (Oct 2004).

[6] Hand, S., Warfield, A., Fraser, K., Kotsovinos, E., and Magenheimer, D. Are Virtual Machine Monitors Microkernels Done Right? In *Proceedings of the Tenth Workshop on Hot Topics in Operating Systems (HotOS X)* (Santa Fe, New Mexico, June 2005).

[7] Hidlebrand, D. An architectural overview of QNX. In *Proc. UKERNEL* (Apr 1992), pp. 113–126.

[8] Kamp, P.-H., and Watson, R. N. M. Jails: Confining the Omnipotent Root. In *Proc. 2nd Int. SANE Conf.* (Maastricht, The Netherlands, May 2000).

[9] Leslie, I. M., McAuley, D., Black, R., Roscoe, T., Barham, P. T., Evers, D., Fairbairns, R., and Hyden, E. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE J. Sel. Areas Comm. 14*, 7 (1996), 1280–1297.

[10] LINUX VSERVERS PROJECT. http://linux-vserver.org/.

[11] MCVOY, L., AND STAELIN, C. lmbench: Portable Tools for Performance Analysis. In *Proc. USENIX '96* (Jan 1996), pp. 279–294.

[12] MENON, A., SANTOS, J., TURNER, Y., JANAKIRAMAN, G., AND ZWAENEPOEL, W. Diagnosing Performance Overheads in the Xen Virtual Machine Environment. In *Proc. 1st VEE* (Jun 2005).

[13] PETERSON, L., ANDERSON, T., CULLER, D., AND ROSCOE, T. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proc. HotNets–I* (Princeton, NJ, Oct 2002).

[14] SWIFT, M. M., ANNAMALAI, M., BERSHAD, B. N., AND LEVY, H. M. Recovering device drivers.

[15] SWIFT, M. M., BERSHAD, B. N., AND LEVY, H. M. Improving the reliability of commodity operating systems. pp. 207–222.

[16] SWSOFT. Virtuozzo Technology. http://www.sw-soft.com/en/products/virtuozzo/.

[17] VRABLE, M., MA, J., CHEN, J., MOORE, D., VANDEKIEFT, E., SNOEREN, A. C., VOELKER, G. M., AND SAVAGE, S. Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm. In *Proc. 20th SOSP* (Brighton, UK, Oct 2005).

[18] WALDSPURGER, C. A. Memory resource management in VMware ESX server. In *Proc. 5th OSDI* (Boston, MA, Dec 2002), pp. 181–194.

[19] WHITAKER, A., COX, R., AND GRIBBLE, S. Configuration Debugging as Search: Find the Needle in the Haystack. In *Proc. 6th OSDI* (San Francisco, CA, Dec 2004), pp. 181–194.

[20] WHITAKER, A., SHAW, M., AND GRIBBLE, S. D. Scale and Performance in the Denali Isolation Kernel. In *Proc. 5th OSDI* (Boston, MA, December 2002), pp. 195–209.