Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors

Stephen Soltesz
Dept. of Computer Science
Princeton University
Princeton, New Jersey 08540
soltesz@cs.princeton.edu

Herbert Pötzl
Linux-VServer Maintainer
Laaben, Austria
herbert@13thfloor.at

Marc E. Fiuczynski
Dept. of Computer Science
Princeton University
Princeton, New Jersey 08540
mef@cs.princeton.edu

Andy Bavier
Dept. of Computer Science
Princeton University
Princeton, New Jersey 08540
acb@cs.princeton.edu

Larry Peterson
Dept. of Computer Science
Princeton University
Princeton, New Jersey 08540
llp@cs.princeton.edu

ABSTRACT

Hypervisors, popularized by Xen and VMware, are quickly becoming commodity. They are appropriate for many usage scenarios, but there are scenarios that require system virtualization with high degrees of both *isolation* and *efficiency*. Examples include HPC clusters, the Grid, hosting centers, and PlanetLab. We present an alternative to hypervisors that is better suited to such scenarios. The approach is a synthesis of prior work on *resource containers* and *security containers* applied to general-purpose, time-shared operating systems. Examples of such container-based systems include Solaris 10, Virtuozzo for Linux, and Linux-VServer. As a representative instance of container-based systems, this paper describes the design and implementation of Linux-VServer. In addition, it contrasts the architecture of Linux-VServer with current generations of Xen, and shows how Linux-VServer provides comparable support for isolation and superior system efficiency.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures; D.4.8 [**Operating Systems**]: Performance—*Measurements, Operational analysis*

General Terms

Performance Measurement Design

Keywords

Linux-VServer Xen virtualization container hypervisor operating system alternative

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EuroSys'07, March 21–23, 2007, Lisboa, Portugal.
Copyright 2007 ACM 978-1-59593-636-3/07/0003 $5.00.

1. INTRODUCTION

Operating system designers face a fundamental tension between isolating applications and enabling sharing among them—to 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 commodity operating systems, designed for personal computers and 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 full isolation between virtual machines (VMs), providing no more support for sharing between VMs than the network provides between physical machines.

The workload requirements for a given system will direct users to the point in the design space that requires the least trade-off. For instance, workstation operating systems generally run multiple applications on behalf of a single user, making it natural to favor sharing over isolation. On the other hand, hypervisors are often deployed to let a single machine host multiple, unrelated applications, which may run on behalf of independent organizations, as is common when a data center consolidates multiple physical servers. The applications in such a scenario have no need to share information. Indeed, it is important they have no impact on each other. For this reason, hypervisors heavily favor full isolation over sharing. However, when each virtual machine is running the same kernel and similar operating system distributions, the degree of isolation offered by hypervisors comes at the cost of efficiency relative to running all applications on a single kernel.

A number of emerging usage scenarios—such as HPC clusters, Grid, web/db/game hosting organizations, distributed hosting (e.g., PlanetLab, Akamai, Amazon EC2)—benefit from virtualization techniques that isolate different groups of users and their applications from one another. What these usage scenarios share is the need for efficient use of system resources, either in terms of raw performance for a single or small number of VMs, or in terms of sheer scalability of

Exhibit 1036
Microsoft Corporation v. Edge Networking Systems, LLC

concurrently active VMs.

This paper describes a virtualization approach designed to enforce a high degree of isolation between VMs while maintaining efficient use of system resources. The approach synthesizes ideas from prior work on *resource containers* [2, 14] and *security containers* [7, 19, 12, 25] and applies it to general-purpose, time-shared operating systems. Indeed, variants of such container-based operating systems are in production use today—e.g., Solaris 10 [19], Virtuozzo [23], and Linux-VServer [11].

The paper makes two contributions. First, this is the first thorough description of the techniques used by Linux-VServer for an academic audience (henceforth referred to as just "VServer"). We choose VServer as the representative instance of the container-based system for several reasons: 1) it is open source, 2) it is in production use, and 3) because we have real data and experience from operating 700+ VServer-enabled machines on PlanetLab [17].

Second, we contrast the architecture of VServer with a recent generation of Xen, which has changed drastically since its original design was described by Barham et al. [3]. In terms of performance, the two solutions are equal for CPU bound workloads, whereas for I/O centric (server) workloads VServer makes more efficient use of system resources and thereby achieves better overall performance. In terms of scalability, VServer far surpasses Xen in usage scenarios where overbooking of system resources is required (e.g., PlanetLab, managed web hosting, etc), whereas for reservation based usage scenarios involving a small number of VMs VServer retains an advantage as it inherently avoids duplicating operating system state.

The next section presents a motivating case for container based systems. Section 3 presents container-based techniques in further detail, and describes the design and implementation of VServer. Section 4 reproduces benchmarks that have become familiar metrics for Xen and contrasts those with what can be achieved by VServer. Section 5 describes the kinds of interference observed between VMs. Finally, Section 6 offers some concluding remarks.

2. MOTIVATION

Virtual machine technologies are the product of diverse groups with different terminology. To ease the prose, we settle on referring to the isolated execution context running on top of the underlying system providing virtualization as a *virtual machine* (VM), rather than a *domain*, *container*, *applet*, *guest*, etc.. There are a variety of VM architectures ranging from the hardware (e.g., Intel's VT.) up the full software including hardware abstraction layer VMs (e.g., Xen, VMware ESX), system call layer VMs (e.g., Solaris, VServer), hosted VMs (e.g., VMware GSX), emulators (e.g, QEMU), high-level language VMs (e.g., Java), and application-level VMs (e.g., Apache virtual hosting). Within this wide range, we focus on comparing hypervisor technology that isolate VMs at the hardware abstraction layer with container-based operating system (COS) technology that isolate VMs at the system call/ABI layer.

The remainder of this section first outlines the usage scenarios of VMs to set the context within which we compare and contrast the different approaches to virtualization. We then make a case for container-based virtualization with these usage scenarios.

2.1 Usage Scenarios

There are many innovative ideas that exploit VMs to secure work environments on laptops, detect virus attacks in real-time, determine the cause of computer break-ins, and debug difficult to track down system failures. Today, VMs are predominantly used by programmers to ease software development and testing, by IT centers to consolidate dedicated servers onto more cost effective hardware, and by traditional hosting organizations to sell virtual private servers. Other emerging, real-world scenarios for which people are either considering, evaluating, or actively using VM technologies include HPC clusters, the Grid, and distributed hosting organizations like PlanetLab and Amazon EC2. This paper focuses on these three emerging scenarios, for which efficiency is paramount.

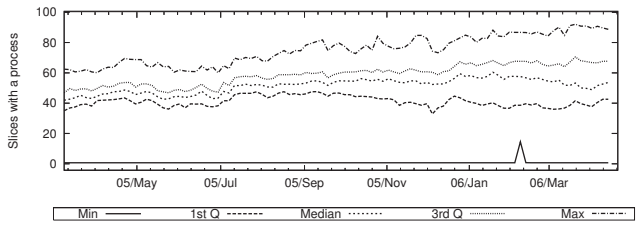
Compute farms, as idealized by the Grid vision and typically realized by HPC clusters, try to support many different users (and their application's specific software configurations) in a batch-scheduled manner. While compute farms do not need to run many concurrent VMs (often just one per physical machine at a time), they are nonetheless very sensitive to raw performance issues as they try to maximize the number of jobs they can push through the overall system per day. As well, experience shows that most software configuration problems encountered on compute farms are due to incompatibilities of the system software provided by a specific OS distribution, as opposed to the kernel itself. Therefore, giving users the ability to use their own distribution or specialized versions of system libraries in a VM would resolve this point of pain.

In contrast to compute farms, hosting organizations tend to run many copies of the same server software, operating system distribution, and kernels in their mix of VMs. In for-profit scenarios, hosting organizations seek to benefit from an economy of scale and need to reduce the marginal cost per customer VM. Such hosting organizations are sensitive to issues of efficiency as they try to carefully oversubscribe their physical infrastructure with as many VMs as possible, without reducing overall quality of service. Unfortunately, companies are reluctant to release just how many VMs they operate on their hardware.

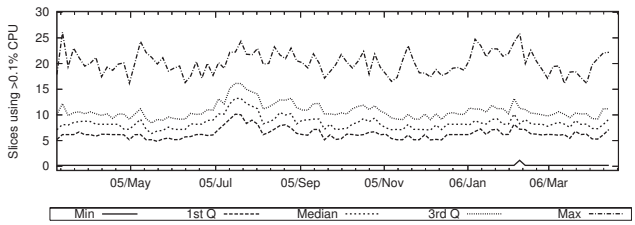
Fortunately, CoMon [24]—one of the performance-monitoring services running on PlanetLab—publicly releases a wealth of statistics relating to the VMs operating on PlanetLab. PlanetLab is a non-profit consortium whose charter is to enable planetary-scale networking and distributed systems research at an unprecedented scale. Research organizations join by dedicating at least two machines connected to the Internet to PlanetLab. PlanetLab lets researchers use these machines, and each research project is placed into a separate VM per machine (referred to as a slice). PlanetLab supports a workload consisting of a mix of one-off experiments and long-running services with its slice abstraction.

CoMon classifies a VM as *active* on a node if it contains

Exhibit 1036
Microsoft Corporation v. Edge Networking Systems, LLC



(a) Active slices, by quartile



(b) Live slices, by quartile

Figure 1: Active and live slices on PlanetLab

a process, and *live* if, in the last five minutes, it used at least 0.1% (300ms) of the CPU. Figure 1 (reproduced from [17]) shows, by quartile, the distribution of active and live VMs across PlanetLab during the past year. Each graph shows five lines; 25% of PlanetLab nodes have values that fall between the first and second lines, 25% between the second and third, and so on. We note that, in any five-minute interval, it is not unusual to see 10-15 live VMs and 60 active VMs on PlanetLab. At the same time, PlanetLab nodes are PC-class boxes; the average PlanetLab node has a 2GHz CPU and 1GB of memory. Any system that hosts such a workload on similar hardware must be concerned with overall efficiency–i.e., both performance and scalability–of the underlying virtualization technology.

2.2 Case for COS Virtualization

The case for COS virtualization rests on the observation that it is acceptable in some real-world scenarios to trade *isolation* for *efficiency*. Sections 4 and 5 demonstrate quantitatively that a COS (VServer) is more efficient than a well designed hypervisor (Xen). So, the question remains: what must be traded to get that performance boost?

Efficiency can be measured in terms of overall performance (throughput, latency, etc) and/or scalability (measured in number of concurrent VMs) afforded by the underlying VM technology. Isolation is harder to quantify than efficiency. A system provides full isolation when it supports a combination of fault isolation, resource isolation, and security isolation. As the following discussion illustrates, there is significant overlap between COS- and hypervisor-based technologies with respect to these isolation characteristics.

Fault isolation reflects the ability to limit a buggy VM from affecting the stored state and correct operation of other VMs. Complete fault isolation between VMs requires there to be no sharing of code or data. In COS- and hypervisor-based systems, the VMs themselves are fully fault isolated from each other using address spaces. The only code and data shared among VMs is the underlying system providing virtualization—i.e., the COS or hypervisor. Any fault in this shared code base can cause the whole system to fail.

Arguably the smaller code base of a hypervisor–Xen for x86 consists of roughly 80K lines of code–naturally eases the engineering task to ensure its reliability. While this may be true, a functioning hypervisor-based system also requires a *host* VM that authorizes and multiplexes access to devices. The host VM typically consists of a fully fledged Linux (millions of lines of code) and therefore is the weak link with respect to fault isolation—i.e., a fault in the host VM could cause the whole system to fail. Fraser et al. [6] propose to mitigate this problem by isolating device drivers into independent driver domains (IDDs).

While the overall Linux kernel is large due to the number of device drivers, filesystems, and networking protocols, at its core it is less than 140K lines. To improve resilience to faults (usually stemming from drivers), Swift et al. [22] propose to isolate device drivers into IDDs within the Linux kernel using their Nooks technology. Unfortunately, there exists no study that directly compares Xen+IDD and Linux+Nooks quantitatively with respect to their performance.

With respect to fault isolation, if we accept that various subsystems such as device drivers, filesystems, networking protocols, etc. are rock solid, then the principle difference between hypervisor- and COS-based systems is in the interface they expose to VMs. Any vulnerability exposed by the implementation of these interfaces may let a fault from one VM leak to another. For hypervisors there is a narrow interface to events and virtual device, whereas for COSs there is the wide system call ABI. Arguably it is easier to verify the narrow interface, which implies that the interface exposed by hypervisors are more likely to be correct.

Resource isolation corresponds to the ability to account for and enforce the resource consumption of one VM such that guarantees and fair shares are preserved for other VMs. Undesired interactions between VMs are sometimes called *cross-talk* [9]. Providing resource isolation generally involves careful allocation and scheduling of physical resources (e.g., cycles, memory, link bandwidth, disk space), but can also be influenced by sharing of logical resources, such as file descriptors, ports, PIDs, and memory buffers. At one extreme, a virtualized system that supports resource reservations 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. At the other extreme, a virtualized system might let VMs obtain cycles and bandwidth on a demand-driven (best-effort) basis. Many hybrid approaches are also possible: for instance, a system may enforce fair sharing of resources between classes of VMs, which lets one overbook available resources while preventing starvation in overload scenarios. The key point is that both hypervisors and COSs incorporate sophisticated resource schedulers to avoid or minimize crosstalk.

Exhibit 1036
Microsoft Corporation v. Edge Networking Systems, LLC

Security isolation refers to the extent to which a virtualized system limits access to (and information about) logical objects, such as files, virtual memory addresses, port numbers, user ids, process ids, and so on. In doing so, security isolation promotes (1) *configuration independence*, so that global names (e.g., of files, SysV Shm keys, etc) selected by one VM cannot conflict with names selected by another VM; and (2) *safety*, such that when global namespaces are shared, one VM is not able to modify data and code belonging to another VM, thus diminishing the likelihood that a compromise to one VM affects others on the same machine. A virtualized system with complete security 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 virtualized system that supports partial security 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. As discussed later, some COSs opt to apply access controls on shared namespaces, as opposed to maintaining autonomous and opaque namespaces via contextualization, in order to improve performance. In such a partially isolated scheme, information leaks are possible, for instance, allowing unauthorized users to potentially identify in-use ports, user names, number of running processes, etc. But, both hypervisors and COSs can hide logical objects in one VM from other VMs to promote both configuration independence and system safety.

Discussion: VM technologies are often embraced for their ability to provide strong isolation as well as other value-added features. Table 1 provides a list of popular value-added features that attract users to VM technologies, which include abilities to run multiple kernels side-by-side, have administrative power (i.e., root) within a VM, checkpoint and resume, and migrate VMs between physical hosts.

Features	Hypervisor	Containers
Multiple Kernels	✓	×
Administrative power (root)	✓	✓
Checkpoint & Resume	✓	✓ [15,23,18]
Live Migration	✓	✓ [23,18]
Live System Update	×	✓ [18]

Table 1: Feature comparison of hypervisor- and COS-based systems

Since COSs rely on a single underlying kernel image, they are of course not able to run multiple kernels like hypervisors can. As well, the more low-level access that is desired by users, such as the ability to load a kernel module, the more code is needed to preserve isolation of the relevant system. However, some COSs can support the remaining features. The corresponding references are provided in Table 1. In fact, at least one solution supporting COS-based VM migration goes a step further than hypervisor-based VM migration: it enables VM migration from one kernel *version* to another. This feature lets systems administrators do a Live System Update on a running system, e.g., to release a new kernel with bug/security fixes, performance enhancements, or new features, without needing to reboot the VM. Kernel version migration is possible because COS-based solutions have explicit knowledge of the dependencies that processes within a VM have to in-kernel structures [18].

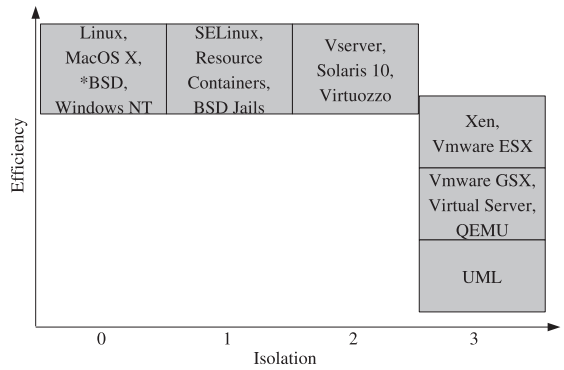


Figure 2: Summary of existing hypervisor- and COS-based technology

Figure 2 summarizes the state-of-the-art in VM technology along the efficiency and isolation dimensions. The x-axis counts how many of the three different kinds of isolation are supported by a particular technology. The y-axis is intended to be interpreted qualitatively rather than quantitatively; as mentioned, later sections will focus on presenting quantitative results.

The basic observation made in the figure is, to date, there is no VM technology that achieves the ideal of maximizing both efficiency and isolation. We argue that for usage scenarios where efficiency trumps the need for full isolation, a COS such as VServer hits the sweet spot within this space. Conversely, for scenarios where full isolation is required, a hypervisor is best. Finally, since the two technologies are not mutually exclusive, one can run a COS in a VM on a hypervisor when appropriate.

3. CONTAINER-BASED OS APPROACH

This section provides an overview of container-based systems, describes the general techniques used to achieve isolation, and presents the mechanisms with which VServer implements these techniques.

3.1 Overview

A container-based system provides a shared, virtualized OS image consisting of a root file system, a (safely shared) set of

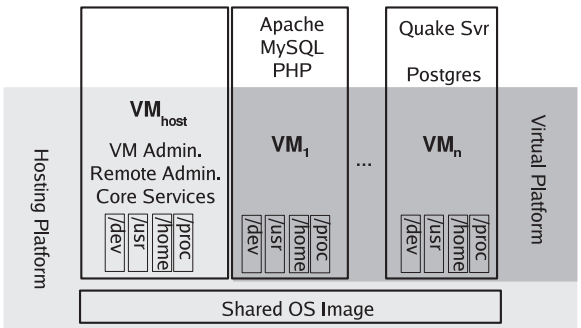


Figure 3: COS Overview

Exhibit 1036
Microsoft Corporation v. Edge Networking Systems, LLC

system libraries and executables. Each VM can be booted, shut down, and rebooted just like a regular operating system. Resources such as disk space, CPU guarantees, memory, etc. are assigned to each VM when it is created, yet often can be dynamically varied at run time. To applications and the user of a container-based system, the VM appears just like a separate host. Figure 3 schematically depicts the design.

As shown in the figure, there are three basic platform groupings. The hosting platform consists essentially of the shared OS image and a privileged *host VM*. This is the VM that a system administrator uses to manage other VMs. The virtual platform is the view of the system as seen by the *guest VMs*. Applications running in the guest VMs work just as they would on a corresponding non-container-based OS image. At this level, there is little difference between a container and hypervisor based system. However, they differ fundamentally in the techniques they use to implement isolation between VMs.

Figure 4 illustrates this by presenting a taxonomic comparison of their security and resource isolation schemes. As shown in the figure, the COS approach to security isolation directly involves internal operating system objects (PIDs, UIDs, Sys-V Shm and IPC, Unix ptys, and so on). The basic techniques used to securely use these objects involve: (1) separation of name spaces (contexts), and (2) access controls (filters). The former means that global identifiers (e.g., process ids, SYS V IPC keys, user ids, etc.) live in completely different spaces (for example, per VM lists), do not have pointers to objects in other spaces belonging to a different VM, and thus cannot get access to objects outside of its name space. Through this *contextualization* the global identifiers become per-VM global identifiers. Filters, on the other hand, control access to kernel objects with runtime checks to determine whether the VM has the appropriate permissions. For a hypervisor security isolation is also achieved with contextualization and filters, but generally these apply to constructs at the hardware abstraction layer such as virtual memory address spaces, PCI bus addresses, devices, and privileged instructions.

The techniques used by COS- and hypervisor-based systems for resource isolation are quite similar. Both need to multiplex physical resources such as CPU cycles, i/o bandwidth, and memory/disk storage. The latest generation of the Xen hypervisor architecture focuses on multiplexing the CPU. Control over all other physical resources is delegated to one or more privileged host VMs, which multiplex the hardware on behalf of the guest VMs. Interestingly, when Xen's host VM is based on Linux, the resource controllers used to manage network and disk i/o bandwidth among guest VMs are **identical** to those used by VServer. The two systems simply differ in how they map VMs to these resource controllers.

As a point of reference, the Xen hypervisor for the i32 architecture is about 80K lines of code, the paravirtualized variants of Linux require an additional 15K of device drivers, and a few isolated changes to the core Linux kernel code. In contrast, VServer adds less than 8700 lines of code to the Linux kernel, and due to its mostly architecture independent nature it has been validated to work on eight different instruction set architectures. While lightweight in terms of lines of code involved, VServer introduces 50+ new kernel files and touches 300+ existing ones—representing a non-trivial software-engineering task.

3.2 VServer Resource Isolation

This section describes in what way VServer implements resource isolation. It is mostly an exercise of leveraging existing resource management and accounting facilities already present in Linux. For both physical and logical resources, VServer simply imposes limits on how much of a resource a VM can consume.

3.2.1 CPU Scheduling: Fair Share and Reservations

VServer implements CPU isolation by overlaying a token bucket filter (TBF) on top of the standard $O(1)$ Linux CPU scheduler. Each VM has a token bucket that accumulates tokens at a specified rate; every timer tick, the VM that owns the running process is charged one token. A VM that runs out of tokens has its processes removed from the run-queue until its bucket accumulates a minimum amount of tokens. Originally the VServer TBF was used to put an upper bound on the amount of CPU that any one VM could receive. However, it is possible to express a range of isolation policies with this simple mechanism. We have modified the TBF to provide fair sharing **and/or** work-conserving CPU reservations.

The rate that tokens accumulate in a VM's bucket depends on whether the VM has a *reservation* and/or a *share*. A VM with a reservation accumulates tokens at its reserved rate: for example, a VM with a 10% reservation gets 100 tokens per second, since a token entitles it to run a process for one millisecond. A VM with a share that has runnable processes will be scheduled before the idle task is scheduled, and only when all VMs with reservations have been honored. The end result is that the CPU capacity is effectively partitioned between the two classes of VMs: VMs with reservations get what they've reserved, and VMs with shares split the unreserved capacity of the machine proportionally. Of course, a VM can have both a reservation (e.g., 10%) and a fair share (e.g., 1/10 of idle capacity).

3.2.2 I/O QoS: Fair Share and Reservations

The Hierarchical Token Bucket (HTB) queuing discipline of the Linux Traffic Control facility (tc) [10] is used to provide network bandwidth reservations and fair service among VServer. For each VM, a token bucket is created with a *reserved rate* and a *share*: the former indicates the amount of outgoing bandwidth dedicated to that VM, and the latter governs how the VM shares bandwidth beyond its reservation. Packets sent by a VServer are tagged with its context id in the kernel, and subsequently classified to the VServer's token bucket. The HTB queuing discipline then allows each VServer to send packets at the reserved rate of its token bucket, and fairly distributes the excess capacity to the VServer in proportion to their shares. Therefore, a VM can be given a capped reservation (by specifying a reservation but no share), "fair best effort" service (by specifying a share with no reservation), or a work-conserving reservation (by specifying both).

Exhibit 1036
Microsoft Corporation v. Edge Networking Systems, LLC

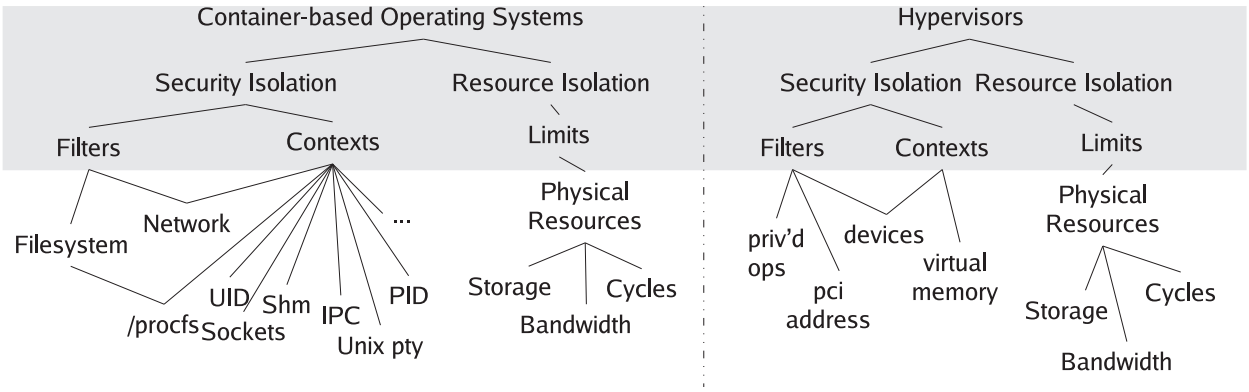


Figure 4: Isolation Taxonomy of COS and Hypervisor-based Systems

Disk I/O is managed in VServer using the standard Linux CFQ ("completely fair queuing") I/O scheduler. The CFQ scheduler attempts to divide the bandwidth of each block device fairly among the VMs performing I/O to that device.

3.2.3 Storage Limits

VServer provides the ability to associate limits to the amount of memory and disk storage a VM can acquire. For disk storage one can specify limits on the max number of disk blocks and inodes a VM can allocate. For memory storage one can specify the following limits: a) the maximum resident set size (RSS), b) number of anonymous memory pages have (ANON), and c) number of pages that may be pinned into memory using mlock() and mlockall() that processes may have within a VM (MEMLOCK). Also, one can declare the number of pages a VM may declare as SYSV shared memory.

Note that fixed upper bounds on RSS are not appropriate for usage scenarios where administrators wish to overbook VMs. In this case, one option is to let VMs compete for memory, and use a watchdog daemon to recover from overload cases—for example by killing the VM using the most physical memory. PlanetLab [17] is one example where memory is a particularly scarce resource, and memory limits without overbooking are impractical: given that there are up to 90 active VMs on a PlanetLab server, this would imply a tiny 10MB allocation for each VM on the typical PlanetLab server with 1GB of memory. Instead, PlanetLab provides basic memory isolation between VMs by running a simple watchdog daemon, called pl_mom, that resets the VM consuming the most physical memory when swap has almost filled. This penalizes the memory hog while keeping the system running for everyone else, and is effective for the workloads that PlanetLab supports. A similar technique is apparently used by managed web hosting companies.

3.3 VServer Security Isolation

VServer makes a number of kernel modifications to enforce security isolation.

3.3.1 Process Filtering

VServer reuses the global PID space across all VMs. In contrast, other container-based systems such as OpenVZ contextualize the PID space per VM. There are obvious benefits to the latter, specifically it eases the implementation of VM checkpoint, resume, and migration more easily as processes can be re-instantiated with the same PID they had at the time of checkpoint. VServer will move to this model, but for the sake of accuracy and completeness we will describe its current model.

VServer filters processes in order to hide all processes outside a VM's scope, and prohibits any unwanted interaction between a process inside a VM and a process belonging to another VM. This separation requires the extension of some existing kernel data structures in order for them to: a) become aware to which VM they belong, and b) differentiate between identical UIDs used by different VMs.

To work around false assumptions made by some user-space tools (like pstree) that the 'init' process has to exist and have PID 1, VServer also provides a per VM mapping from an arbitrary PID to a fake init process with PID 1.

When a VServer-based system boots, all processes belong to a *default host VM*. To simplify system administration, this host VM is no different than any other guest VM in that one can only observe and manipulate processes belonging to that VM. However, to allow for a global process view, VServer defines a special *spectator* VM that can peek at all processes at once.

A side effect of this approach is that process migration from one VM to another VM *on the same host* is achieved by changing its VM association and updating the corresponding per-VM resource usage statistics such NPROC, NOFILE, RSS, ANON, MEMLOCK, etc..

3.3.2 Network Separation

Currently, VServer does not fully virtualize the networking subsystem, as is done by OpenVZ and other container-based systems. Rather, it shares the networking subsystem (route tables, IP tables, etc.) between all VMs, but only lets VMs bind sockets to a set of available IP addresses specified either at VM creation or dynamically by the default host VM. This has the drawback that it does not let VMs change their

Exhibit 1036
Microsoft Corporation v. Edge Networking Systems, LLC

route table entries or IP tables rules. However, it was a deliberate design decision to achieve native Linux networking performance at GigE+ line rates.

For VServer's *network separate* approach several issues have to be considered; for example, the fact that bindings to special addresses like IPADDR_ANY or the local host address have to be handled to avoid having one VM receive or snoop traffic belonging to another VM. The approach to get this right involves tagging packets with the appropriate VM identifier and incorporating the appropriate filters in the networking stack to ensure only the right VM can receive them. As will be shown later, the overhead of this is minimal as high-speed networking performance is indistinguishable between a native Linux system and one enhanced with VServer.

3.3.3 The Chroot Barrier

One major problem of the chroot() system used in Linux lies within the fact that this information is volatile, and will be changed on the 'next' chroot() system call. One simple method to escape from a chroot-ed environment is as follows:

- Create or open a file and retain the file-descriptor, then chroot into a subdirectory at equal or lower level with regards to the file. This causes the 'root' to be moved 'down' in the filesystem.
- Use fchdir() on the file descriptor to escape from that 'new' root. This will consequently escape from the 'old' root as well, as this was lost in the last chroot() system call.

VServer uses a special file attribute, known as the Chroot Barrier, on the parent directory of each VM to prevent unauthorized modification and escape from the chroot confinement.

3.3.4 Upper Bound for Linux Capabilities

Because the current Linux Capability system does not implement the filesystem related portions of POSIX Capabilities that would make setuid and setgid executables secure, and because it is much safer to have a secure upper bound for all processes within a context, an additional per-VM capability mask has been added to limit all processes belonging to that context to this mask. The meaning of the individual caps of the capability bound mask is exactly the same as with the permitted capability set.

3.4 VServer Filesystem Unification

One central objective of VServer is to reduce the overall resource usage wherever possible. VServer implements a simple disk space saving technique by using a simple unification technique applied at the whole file level. The basic approach is that files common to more than one VM, which are rarely going to change (e.g., like libraries and binaries from similar OS distributions), can be hard linked on a shared filesystem. This is possible because the guest VMs can safely share filesystem objects (inodes). The technique reduces the amount of disk space, inode caches, and even memory mappings for shared libraries.

The only drawback is that without additional measures, a VM could (un)intentionally destroy or modify such shared files, which in turn would harm/interfere other VMs. The approach taken by VServer is to mark the files as copy-on-write. When a VM attempts to mutate a hard linked file with CoW attribute set, VServer will give the VM a private copy of the file.

Such CoW hard linked files belonging to more than one context are called 'unified' and the process of finding common files and preparing them in this way is called Unification. The reason for doing this is reduced resource consumption, not simplified administration. While a typical Linux Server install will consume about 500MB of disk space, 10 unified servers will only need about 700MB and as a bonus use less memory for caching.

4. SYSTEM EFFICIENCY

This section explores the performance and scalability of COS- and hypervisor-based virtualization. We refer to the combination of performance and scale as the *efficiency* of the system, since these metrics correspond directly to how well the virtualizing system orchestrates the available physical resources for a given workload.

For all tests, VServer performance is comparable to an unvirtualized Linux kernel. Yet, the comparison shows that although Xen3 continues to include new features and optimizations, the overhead required by the virtual memory sub-system still introduces an overhead of up to 49% for shell execution. In terms of absolute performance on server-type workloads, Xen3 lags an unvirtualized system by up to 40% for network throughput while demanding a greater CPU load and 50% longer for disk intensive workloads.

4.1 Configuration

All experiments are run on an HP Proliant DL360 G4p with dual 3.2 GHz Xeon processor, 4GB RAM, two Broadcom NetXtreme GigE Ethernet controllers, and two 160GB 7.2k RPM SATA-100 disks. The Xeon processors each have a 2MB L2 cache. Due to reports [21] indicating that hyper-threading degrades performance for certain environments, we run all tests with hyper-threading disabled. The three kernels under test were compiled for uniprocessor as well as SMP architectures, and unless otherwise noted, all experiments are run within a single VM provisioned with all available resources. In the case of Xen, neither the guest VM nor the hypervisor include device drivers. Instead, a privileged, host VM runs the physical devices and exposes virtual devices to guests. In our tests, the host VM reserves 512MB and the remaining available is available to guest VMs.

The Linux kernel and its variants for Xen and VServer have hundreds of system configuration options, each of which can potentially impact system behavior. We have taken the necessary steps to normalize the effect of as many configuration options as possible, by preserving homogeneous setup across systems, starting with the hardware, kernel configuration, filesystem partitioning, and networking settings. The goal is to ensure that observed differences in performance are a consequence of the virtualization architectures evaluated, rather than a particular set of configuration parameters. Appendix A describes the specific configurations we have used

Exhibit 1036
Microsoft Corporation v. Edge Networking Systems, LLC

in further detail.

The hypervisor configuration is based on Xen 3.0.4, which at the time of this writing was the latest stable version available. The corresponding patch to Linux is applied against a 2.6.16.33 kernel. Prior to 3.0.4, we needed to build separate host VM and guest VM kernels, but we now use the unified, Xen paravirtualized Linux kernel that is re-purposed at runtime to serve either as a host or guest VM. Finally, we also build SMP enabled kernels, as this is now supported by the stable Xen hypervisor.

The COS configuration consists of the VServer 2.0.3-rc1 patch applied to the Linux 2.6.16.33 kernel. Our VServer kernel includes several additions that have come as a result of VServer's integration with Planetlab. As discussed earlier, this includes the new CPU scheduler that preserves the existing $O(1)$ scheduler and enables CPU reservations for VMs, and shims that let VServer directly leverage the existing CFQ scheduler to manage disk I/O and the HTB filter to manage network I/O.

4.2 Micro-Benchmarks

While micro-benchmarks are incomplete indicators of system behavior for real workloads [5], they do offer an opportunity to observe the fine-grained impact that different virtualization techniques have on primitive OS operations. In particular, the *OS* subset of McVoy's *lmbench* benchmark [13] version 3.0-a3 includes experiments designed to target exactly these subsystems.

For all three systems, the majority of the tests perform worse in the SMP kernel than the UP kernel. While the specific magnitudes may be novel, the trend is not surprising, since the overhead inherent to synchronization, internal communication, and caching effects of SMP systems is well known. For brevity, the following discussion focuses on the overhead of virtualization using a uniprocessor kernel. While the *lmbench* suite includes a large number of latency benchmarks, Table 2 shows results only for those which represent approximately 2x or greater discrepancy between VServer-UP and Xen3-UP.

For the uniprocessor systems, our findings are consistent with the original report of Barham et al [3] that Xen incurs a penalty for virtualizing the virtual memory hardware. In fact, the pronounced overhead observed in Xen comes entirely from the hypercalls needed to update the guest's page table. This is one of the most common operations in a multi-user system. While Xen3 has optimized page table updates relative to Xen2, common operations such as process executing, context switches and page faults still incur observable overhead.

Configuration	Linux-UP	VServer-UP	Xen3-UP
fork process	86.50	86.90	**271.90**
exec process	299.80	302.00	**734.70**
sh process	968.10	977.70	**1893.30**
ctx (16p/64K)	3.38	3.81	**6.02**
mmap (64MB)	377.00	379.00	**1234.60**
mmap (256MB)	1491.70	1498.00	**4847.30**
page fault	1.03	1.03	**3.21**

Table 2: LMbench OS benchmark timings for uniprocessor kernels – times in μs

The first three rows in Table 2 show the performance of *fork process*, *exec process*, and *sh process* across the systems. The performance of VServer-UP is always within 1% of Linux-UP. Also of note, Xen3-UP performance has improved over that of Xen2-UP due to optimizations in the page table update code that batch pending transactions for a single call to the hypervisor. Yet, the inherent overhead is still measurable, and almost double in the case of *sh process*.

The next row shows context switch overhead between different numbers of processes with different working set sizes. As explained by Barham [3], the 2μs to 3μs overhead for these micro-benchmarks are due to hypercalls from the guest VM into the hypervisor to change the page table base. In contrast, there is little overhead seen in VServer-UP relative to Linux-UP.

The next two rows show *mmap* latencies for 64MB and 256MB files. The latencies clearly scale with respect to to the size of the file, indicating that the Xen kernels incur a $19\mu s$ overhead per megabyte, versus $5.9\mu s$ in the other systems. This is particularly relevant for servers or applications that use *mmap* as a buffering technique or to access large data sets, such as [16].

4.3 System Benchmarks

Two factors contribute to performance overhead in the Xen3 hypervisor system: overhead in network I/O and overhead in disk I/O. Exploring these dimensions in isolation provides insight into the sources of overhead for server workloads in these environments. To do so, we repeat various benchmarks used in the original and subsequent performance measurements of Xen [3, 4, 6]. In particular, there are various multi-threaded applications designed to create real-world, multi-component stresses on a system, such as Iperf, OSDB-IR, and a kernel compile. In addition, we explore several single-threaded applications such as a dd, Dbench and Postmark to gain further insight into the overhead of Xen. These exercise the whole system with a range of server-type workloads illustrating the absolute performance offered by Linux, VServer, and Xen3.

For these benchmarks there is only one guest VM active, and this guest is provisioned with all available memory. For Xen, the host VM is provisioned with 512MB of RAM and a fair share of the CPU. Each reported score is the average of 3 to 5 trials. All results are normalized relative to Linux-UP, unless otherwise stated.

4.3.1 Network Bandwidth Benchmark

Iperf is an established tool [1] for measuring link throughput with TCP or UDP traffic. We use it to measure TCP bandwidth between a pair of systems. We measure both raw throughput and the CPU utilization observed on the receiver. This is done in two separate experiments to avoid measurement overhead interfering with throughput—i.e., max throughput is lower when recording CPU utilization with *sysstat* package on VServer and Linux, and *XenMon* on Xen.

Exhibit 1036
Microsoft Corporation v. Edge Networking Systems, LLC

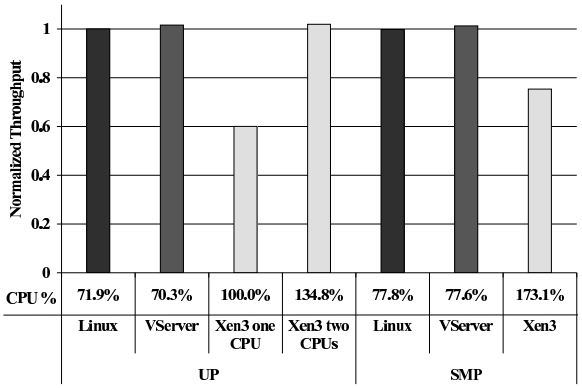


Figure 5: Iperf TCP bandwidth and CPU utilization.

Figure 5 illustrates both the throughput achieved and the aggregate percentage of CPU necessary to achieve this rate on the receiver. The first three columns are trials run with the uniprocessor kernels. Both Linux and VServer on a single processor achieve line rate with just over 70% CPU utilization as the data sink. In contrast, the Xen3-UP configuration can only achieve 60% of the line rate, because having the host VM, guest VM, and hypervisor all pinned to a single CPU saturate the CPU due to the overhead of switching between VMs and interaction with the hypervisor.

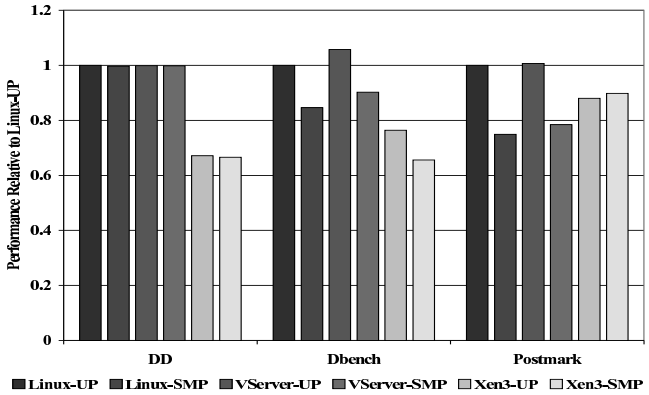
The fourth column labeled 'Xen3 two CPUs' consists of the same Xen3-UP configuration, except the host and guest VMs are pinned to separate CPUs. In this way the host and guest VMs do not interfere with each other, and can achieve line rate just as Linux and VServer. This is an improvement over prior versions of Xen using the same hardware. We attribute the improvement to switching from a safe, page-flipping data transfer model to one utilizing memory copies between VMs. Still, when compared to Linux and VServer running on a single CPU, the overall CPU utilization of the Xen3-UP configuration running on two CPUs is nearly 2x the load experienced by Linux or VServer.

The last three columns use SMP variants of the Linux kernels. Again VServer compares closely to Linux. Interestingly, Xen3 with a SMP guest VM cannot achieve line rate. It saturates the CPU it shares with the host VM. And as a result it also performs worse compared to the Linux-UP configuration.

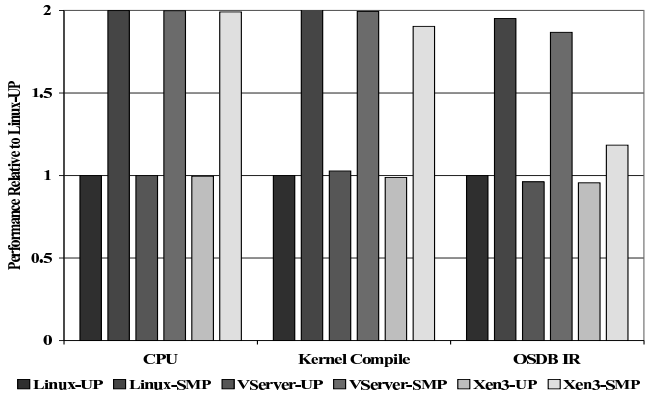
4.3.2 Macro Benchmarks

This section evaluates a number of benchmarks that are CPU and/or disk I/O intensive. The results of these benchmarks are shown in Figures 6(a) and 6(b), which summarizes the performance between VServer and Xen3 normalized against Linux.

The DD benchmark writes a 6GB file to a scratch device. Linux and VServer have identical performance for this benchmark, as the code path for both is basically identical. In contrast, for Xen3 we observe significant slow down for both UP and SMP. This is due to additional buffering, copying, and synchronization between the host VM and guest VM to write blocks to disk.



(a) Disk performance



(b) Performance of CPU and memory bound benchmarks

Figure 6: Relative performance of Linux, VServer, and XenU kernels.

DBench is derived from the industry-standard NetBench filesystem benchmark and emulates the load placed on a file server by Windows 95 clients. The DBench score represents the throughput experienced by a single client performing around 90,000 file system operations. Because DBench is a single-threaded application, the Linux-SMP and VServer-SMP results show reduced performance due to inherent overhead of SMP systems. Accordingly, the Xen3-UP performance is modestly greater than that of Xen3-SMP, but again, both have performance that is 25-35% less than Linux-UP, while VServer-UP slightly exceeds the Linux performance.

Postmark [8] is also a single-threaded benchmark originally designed to stress filesystems with many small file operations. It allows a configurable number of files and directories to be created, followed by a number of random transactions on these files. In particular, our configuration specifies 100,000 files and 200,000 transactions. Postmark generates many small transactions like those experienced by a heavily loaded email or news server, from which it derives the name 'postmark'. Again, the throughput of Xen3 is less

Exhibit 1036
Microsoft Corporation v. Edge Networking Systems, LLC

than both Linux and VServer, as there is more overhead involved in pushing filesystem updates from the guest VM via the host VM to the disk device.

Figure 6(b) demonstrates the relative performance of several CPU and memory bound activities. These tests are designed to explicitly avoid the I/O overhead seen above. Instead, inefficiency here is a result of virtual memory, scheduling or other intrinsic performance limits. The first test is a single-threaded, CPU-only process. When no other operation competes for CPU time, this process receives all available system time. But, the working set size of this process fits in processor cache, and does not reveal the additive effects of a larger working set, as do the second and third tests.

The second test is a standard kernel compile. It uses multiple threads and is both CPU intensive as well as exercising the filesystem with many small file reads and creates. However, before measuring the compilation time, all source files are moved to a RAMFS to remove the impact of any disk effects. The figure indicates that performance is generally good for Xen relative to Linux-UP, leaving overheads only in the range of 1% for Xen3-UP to 7% for Xen3-SMP.

Finally, the Open Source Database Benchmark (OSDB) provides realistic load on a database server from multiple clients. We report the Information Retrieval (IR) portion, which consists of many small transactions, all reading information from a 40MB database, again cached in main memory. The behavior of this benchmark is consistent with current web applications. Again, performance of VServer-UP is comparable to that of Linux-UP within 4%, but Xen3-SMP suffers a 39% overhead relative to Linux-SMP. Not until we look at the performance of this system at scale do the dynamics at play become clear.

4.4 Performance at Scale

This section evaluates how effectively the virtualizing systems provide performance at scale. Barham et al. point out that unmodified Linux cannot run multiple instances of PostgreSQL due to conflicts in the SysV IPC namespace. However, VServer's mechanisms for security isolation contain the SysV IPC namespace within each context. Therefore, using OSDB, we simultaneously demonstrate the security isolation available in VServer that is unavailable in Linux, and the superior performance available at scale in a COS-based design.

The Information Retrieval (IR) component of the OSDB package requires memory and CPU time. If CPU time or system memory is dominated by any one VM, then the others will not receive a comparable share, causing aggregate throughput to suffer. As well, overall performance is calculated as a function of the finish-time of all tests. This methodology favors schedulers which are able to keep the progress of each VM in sync, such that all tests end simultaneously. Figure 7 shows the results of running 1, 2, 4, and 8 simultaneous instances of the OSDB IR benchmark. Each VM runs an instance of PostgreSQL to serve the OSDB test.

A virtualization solution with strong isolation guarantees would partition the share of CPU time, buffer cache, and memory bandwidth perfectly among all active VMs and

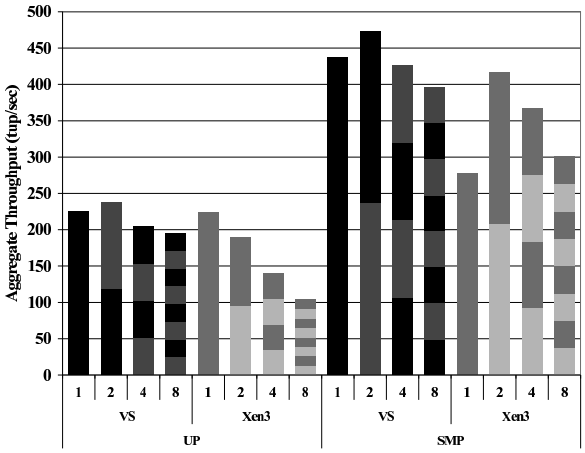


Figure 7: OSDB-IR at Scale. Performance across multiple VMs

maintain the same aggregate throughput as the number of active VMs increased. However, for each additional VM, there is a linear increase in the number of processes and the number of I/O requests. Since it is difficult to perfectly isolate all performance effects, the intensity of the workload adds increasingly more pressure to the system and eventually, aggregate throughput diminishes. Figure 7 illustrates that after an initial boost in aggregate throughput at two VMs, all systems follow the expected diminishing trend.

We observe two noteworthy deviations from this trend. First, Xen3-UP does not improve aggregate throughput at two VMs. Instead, the Xen3-UP performance quickly degrades for each test. In part this is due to the increased load on the single CPU. It must simultaneously host as many as 8 guest VMs as well as the host VM. Because the host VM is explicitly scheduled just as the guest VMs, for larger tests it is given more hosts to serve and correspondingly less time is available for each guest. This pressure is worst when the load reaches eight simultaneous VMs, where the performance is 47% less than VServer-UP.

Second, the significant performance jump between one and two VMs for Xen3-SMP is very large. This behavior is due in part to the credit scheduler unequally balancing the Xen3-SMP kernel across both physical CPUs, as evidenced by monitoring the CPU load on both processors. As a result, the single Xen3-SMP case does not have the opportunity to benefit from the full parallelism available. Not until this situation is duplicated with two Xen3-SMP kernels is greater utility of the system achieved. Of course, VServer-SMP outperforms the Xen3-SMP system. In particular, the total performance in the VServer, eight VM case is within 5% of Xen3-SMP with 2 VMs and greater than any of the other Xen3-SMP tests.

Two factors contribute to the the higher average performance of VServer: lower overhead imposed by the COS approach and a better CPU scheduler for keeping competing VMs progressing at the same rate. As a result, there is simply more CPU time left to serve clients at increasing scale.

Exhibit 1036
Microsoft Corporation v. Edge Networking Systems, LLC

Fraction of Host Requested	VS-UP Achieved	VS-SMP Achieved	Xen3-UP Achieved	Xen3-SMP Achieved
Weight $1/4^{th}$	25.16%	49.88%	**15.51%**	**44.10%**
Cap $1/4^{th}$	n/a	n/a	24.35%	**46.62%**

Table 3: Percent of time achieved from a one quarter CPU reservations using weights and caps. Deviations are highlighted.

5. ISOLATION

VServer complements Linux's existing per process resource limits with per VM limits for logical resources such as shared file descriptors, number of process limits, shared memory sizes, etc., which are shared across the whole system. In this way VServer can effectively isolate VMs running fork-bombs and other antisocial activity through the relevant memory caps, process number caps, and other combinations of resource limits. In contrast, Xen primarily focuses on CPU scheduling and memory limits, while i/o bandwidth management is left to the host VM. As mentioned, for both Xen and VServer the disk and network i/o bandwidth management mechanism (CFQ and HTB, respectively) are largely identical—differing only in the shims that map per VM activities into CFQ and HTB queues. What differs significantly between Xen and VServer are their CPU schedulers. The remainder of this section first focuses on how well their schedulers can fairly share and reserve the CPU among VMs, and then evaluates at a macro level the impact of introducing an antisocial process contained in one VM on another VM running the OSDB benchmark.

5.1 CPU Fair Share and Reservations

To investigate both isolation of a single resource and resource guarantees, we use a combination of CPU intensive tasks. Hourglass is a synthetic real-time application useful for investigating scheduling behavior at microsecond granularity [20]. It is CPU-bound and involves no I/O.

Eight VMs are run simultaneously. Each VM runs an instance of hourglass, which records contiguous periods of time scheduled. Because hourglass uses no I/O, we may infer from the gaps in its time-line that either another VM is running or the virtualized system is running on behalf of another VM, in a context switch for instance. The aggregate CPU time recorded by all tests is within 1% of system capacity.

We evaluated two experiments: 1) all VMs are given the same fair share of CPU time, and 2) one of the VMs is given a reservation of $1/4^{th}$ of overall CPU time. For the first experiment, VServer and Xen for both UP and SMP systems do a good job at scheduling the CPU among the VMs such that each receive approximately one eights of the available time.

Table 3 reports the amount of CPU time received when the CPU reservation of one VM is set to one fourth of the system. For a two-way SMP system with an aggregate of 200% CPU time, a one fourth reservation corresponds to 50% of available resources. The CPU scheduler for VServer achieves this within 1% of the requested reservation for both UP and SMP configurations.

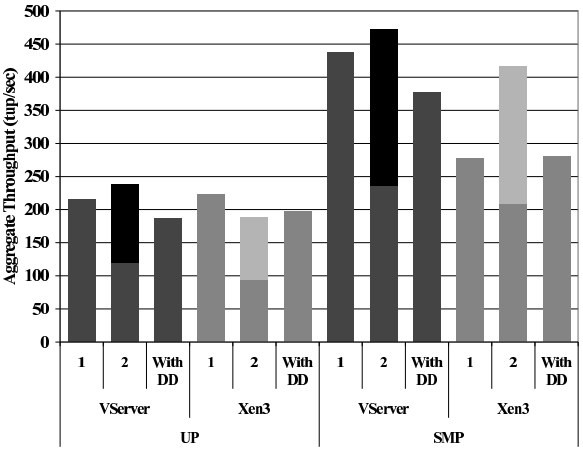


Figure 8: Database performance with competing VMs

In contrast, Xen is off by up to 6% of the requested reservations in the worst case. The reason for this is because Xen does not offer an explicit means to specify a reservation of CPU time. Instead, it provides only two CPU time allocation parameters: relative weights or performance caps (i.e., hard limits). The inherent problem with relative weights is that there is no 1:1 relationship between a VM's weight and the minimum percentage of CPU allotted to it under load. Performance caps, on the other hand, only let one express the maximum fraction of the system a VM can consume, but not the minimum it should obtain. As a result, to express a reservation in terms of weights or performance caps can at best be approximated, which the results shown in Table 3 demonstrate.

5.2 Performance Isolation

Traditional time-sharing UNIX systems have a legacy of vulnerability to layer-below attacks, due to unaccounted, kernel resource consumption. To investigate whether VServer is still susceptible to such interference, we elected to perform a variation of the multi-OSDB database benchmark. Now, instead of all VMs running a database, one will behave *maliciously* by performing a continuous *dd* of a 6GB file to a separate partition of a disk common to both VMs.

Figure 8 shows that the performance of OSDB on VServer is impacted between 13-14% for both UP and SMP when competing with an active *dd*. This, despite the fact that the VServer block cache is both global (shared by all VMs) and not directly accounted to the originating VM. Earlier kernels, such as 2.6.12 kernel, experienced crippling performance penalties during this test while the swap file was enabled. While the ultimate cause of this overhead in older kernels is still not known, these results for modern kernels are very a promising improvement. Clearly, though, additional improvements can be made to account more completely for guest block cache usage and are a subject for future research.

Xen, on the other hand, explicitly partitions the physical memory to each VM. As a result, since the block cache is maintained by each kernel instance, the guests are not vul-

Exhibit 1036
Microsoft Corporation v. Edge Networking Systems, LLC

nerable to this attack. Yet, Xen-UP sees a 12% decrease for and Xen3-SMP actually gets a 1% boost. Given earlier results these are not surprising. Any additional activity by the *dd* VM or the host VM acting on its behalf, is expected to take processing time away from the Xen3-UP, OSDB VM. As for the Xen3-SMP case, the original poor performance of the single VM was due to poor scheduling by the credit scheduler, which allowed an uneven balancing across the physical CPUs. Given this scenario, the lighter loaded CPU runs the *dd* VM, which requires little CPU time and consequently has little impact on the mostly OSDB VM.

6. CONCLUSION

Virtualization technology benefits a wide variety of usage scenarios. It promises such features as configuration independence, software interoperability, better overall system utilization, and resource guarantees. This paper has compared two modern approaches to providing these features while they balance the tension between complete isolation of co-located VMs and efficient sharing of the physical infrastructure on which the VMs are hosted.

We have shown the two approaches share traits in their high-level organization. But some features are unique to the platform. Xen is able to support multiple kernels while by design VServer cannot. Xen also has greater support for virtualizing the network stack and allows for the possibility of VM migration, a feature that is possible for a COS design, but not yet available in VServer. VServer, in turn, maintains a small kernel footprint and performs equally with native Linux kernels in most cases.

Unfortunately, there is no one-size solution. As our tests have shown, i/o related benchmarks perform worse on Xen when compared to VServer. This is an artifact of virtualizing i/o devices via a proxy host VM. VMware partially addresses this issue by incorporating device drivers for a small set of supported high performance I/O devices directly into its ESX hypervisor product line. In contrast, this issue is non-issue for COS based systems where all I/O operates at native speeds. Despite these weaknesses we expect ongoing efforts to continue to improve Xen-based hypervisor solutions.

In the mean time, for managed web hosting, PlanetLab, etc., the trade-off between isolation and efficiency is of paramount importance. Our experiments indicate that container-based systems provide up to 2x the performance of hypervisor-based systems for server-type workloads and scale further while preserving performance. And, we expect that container-based systems like VServer will incorporate more of the feature set that draws users to hypervisors (e.g., full network virtualization, migration, etc.), and thereby continue to compete strongly against hypervisor systems like Xen for these usage scenarios.

7. REFERENCES

[1] Ajay Tirumala, Feng Qin, Jon Dugan, Jim Ferguson, and Kevin Gibbs. Iperf version 1.7.1. `http://dast.nlanr.net/Projects/Iperf/`.

[2] G. Banga, P. Druschel, and J. C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proc. 3rd OSDI*, pages 45–58, New Orleans, LA, Feb 1999.

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

[4] B. Clark, T. Deshane, E. Dow, S. Evanchik, M. Finlayson, J. Herne, and J. Matthews. Xen and the art of repeated research. In *USENIX Technical Conference FREENIX Track*, June 2004.

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

[6] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. W. eld, and M. Williamson. Safe Hardware Access with the Xen Virtual Machine Monitor. In *First Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure (OASIS)*, Oct 2004.

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

[8] J. Katcher. Postmark: a new file system benchmark. In *TR3022. Network Appliance*, October 1997.

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

[10] Linux Advanced Routing and Traffic Control. `http://lartc.org/`.

[11] Linux-VServer Project. `http://linux-vserver.org/`.

[12] B. McCarty. *SELINUX: NSA's open source Security Enhanced Linux*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, USA, 2005.

[13] L. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. In *Proc. USENIX '96*, pages 279–294, Jan 1996.

[14] S. Nabah, H. Franke, J. Choi, C. Seetharaman, S. Kaplan, N. Singhi, V. Kashyap, and M. Kravetz. Class-based prioritized resource control in Linux. In *Proc. OLS 2003*, Ottawa, Ontario, Canada, Jul 2003.

[15] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proc. 5th OSDI*, pages 361–376, Boston, MA, Dec 2002.

[16] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, 1999.

[17] L. Peterson, A. Bavier, M. E. Fiuczynski, and S. Muir. Experiences building planetlab. In *Proceedings of the 7th USENIX Symposium on Operating System Design and Implementation (OSDI '06)*, Seattle, WA, November 2006.

[18] S. Potter and J. Nieh. Autopod: Unscheduled system updates with zero data loss. In *Abstract in Proceedings of the Second IEEE International Conference on Autonomic Computing (ICAC 2005)*, June 2005.

[19] D. Price and A. Tucker. Solaris zones: Operating

Exhibit 1036
Microsoft Corporation v. Edge Networking Systems, LLC

system support for consolidating commercial workloads. In *Proceedings of the 18th Usenix LISA Conference.*, 2004.

[20] J. Regehr. Inferring scheduling behavior with hourglass. In *In Proceedings of the Freenix Track of the 2002 USENIX Annual Technical Conference*, June 2002.

[21] Y. Ruan, V. S. Pai, E. Nahum, and J. M. Tracey. Evaluating the impact of simultaneous multithreading on network servers using real hardware. In *SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 315–326, New York, NY, USA, 2005. ACM Press.

[22] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Trans. Comput. Syst.*, 23(1):77–110, 2005.

[23] SWSoft. Virtuozzo Linux Virtualization. `http://www.virtuozzo.com`.

[24] Vivek Pai and KyoungSoo Park. CoMon: A Monitoring Infrastructure for PlanetLab. `http://comon.cs.princeton.edu`.

[25] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, CA, Aug 2002.

APPENDIX

A. NORMALIZED CONFIGURATION

A significant aspect of this work involved ensuring that the experiments were fair. We report the configuration details of the various subsystems.

A.1 Hardware

All experiments are run on an HP DL360 Proliant with dual 3.2 GHz Xeon processor, 4GB RAM, two Broadcom NetXtreme GigE Ethernet controllers, and two 160GB 7.2k RPM SATA-100 disks. The Xeon processors each have a 2MB L2 cache. All tests are run with hyper-threading disabled.

A.2 Kernel Configuration

All kernels were based on the 2.6.16.33 Linux kernel. The kernel configuration options were normalized across all platforms. The only differences between the kernel config files that remain come from specific options available for the given platform, i.e. VServer or Xen specific, for which there is no comparable option available in the other versions.

A.3 System Clock

The Linux, VServer and unified XenoLinux kernels are configured to run with a 250Hz system clock. This is a deviation from the default configuration of both VServer and Xen, whose defaults are 1000Hz and 100Hz respectively. However, the ten fold difference between the two was in earlier tests shown to contribute to latency measurements, and context switch overheads. Keeping the system clock equal puts both on equal footing.

A.4 Filesystem

The host VM is a Fedora Core 5 distribution with current updates. This is the environment into which the host VMs boot up. The distribution populating the guest VMs is Fedora Core 2 with the all current updates.

Each guest partition is a 2GB, LVM-backed, ext3 filesystem with the following features: has_journal, filetype, and sparse_super. No other fs level features are enabled. The default journal size is created by mke2fs. Due to the size of our disk and the chosen partition size we are limited to approximately 80 VMs. Sixty-four are currently available. The remaining space is used to host larger scratch space for particular tests, DD for instance.

We account for the natural, diminishing read and write performance across the extent of platter-based hard drives by assigning each virtual machine a dedicated LVM partition. This partition is used exclusively by one VM, irrespective of which virtualizing system is currently active. This configuration departs from a traditional VServer system, where a file-level copy-on-write technique replicates the base environment for additional storage savings.

A.5 Networking

The physical host has two Ethernet ports, so both Linux and VServer share two IPv4 IP addresses across all VMs. As a consequence, the port space on each IP address is also shared between all VMs. The Xen configuration, on the other hand, differs by virtue of running an autonomous kernel in each VM which includes a dedicated TCP/IP stack and IP address. The Xen network runs in bridged mode for networking tests. We also use two client machines attached to each Ethernet port on the system under test through a 1 Gbps Netgear switch. Each client is a 3.2GHz HP DL320g5 server equipped with a Broadcom Gigabit Ethernet running Fedora Core 5.

Exhibit 1036
Microsoft Corporation v. Edge Networking Systems, LLC