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

Stephen Soltesz, Herbert Pötzl*, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson
Princeton University, {soltesz,mef,acb,llp}@cs.princeton.edu
**Linux VServer Maintainer* herbert@13thfloor.at

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.

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 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, 18, 12, 24] 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 [18], Virtuozzo [22], and Linux VServer [11].

The paper makes two contributions. First, this is the first thorough description of the techniques used by VServer for an academic audience. 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 [16].

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, Linux VServer), hosted VMs (e.g., VMware GSX), hosted 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

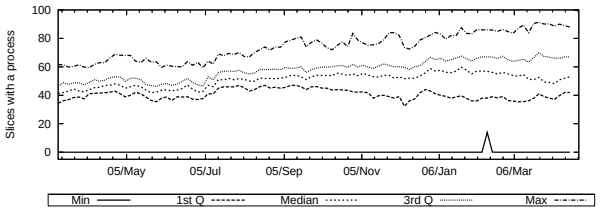
There are many innovative ideas that exploit VMs to secure work environments on laptops, detect virus attacks in real-time, determine the cause of computer break-ins, and debug difficult to track down system failures. Today, VMs are predominantly used by programmers to ease software development and testing, 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. 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.

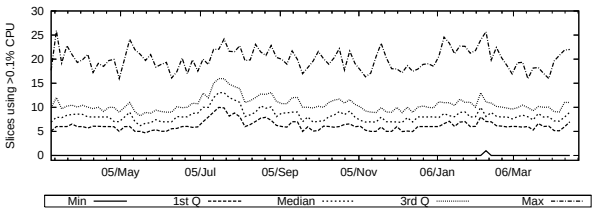
In contrast to compute farms, hosting organizations tend to run many copies of the same server software, operating system distribution, and kernels in their mix of VMs. In for-profit scenarios, hosting organizations seek to benefit from an economy of scale and need to reduce the marginal cost per customer 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 [23]—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 con-



(a) Active slices, by quartile



(b) Live slices, by quartile

Figure 1: Active and live slices on PlanetLab

tains a process, and *live* if, in the last five minutes, it used at least 0.1% (300ms) of the CPU. Figure 1 (reproduced from [16]) 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 both performance and scalability of the underlying virtualization technology.

2.2 Case for Container-Based Operating System 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 off 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. To ensure complete fault isolation requires that there is no direct sharing of code or data between VMs. 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 will 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., any fault in the host VM will 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. [21] 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.

Assuming that various subsystems such as device drivers, filesystems, networking protocols, etc. are rock solid, then with respect to fault isolation 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 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.

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) 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 ability 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, 22, 17]
Live Migration	✓	✓ [22, 17]
Live System Update	✗	✓ [17]

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, th more code is needed to preserve isolation of the relevant system. However, COSs can support the remaining features with corresponding references 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 structure [17].

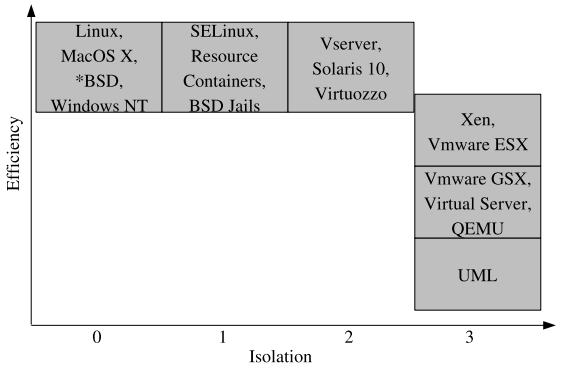


Figure 2: **Summary of existing hypervisor- and COS-based technology**

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 like 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 Operating Systems

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

3.1 Overview

A container-based system provides a shared, virtualized OS image, including a unique root file system, a (safely shared) set of system executables and libraries, and whatever resources assigned to the VM when it is created. Each VM can be booted and shut down just like a regular operating system, and rebooted in only seconds when necessary. 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

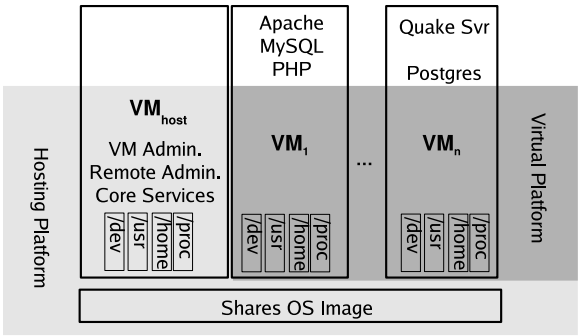


Figure 3: **COS Overview**

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 Linux 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 view isolated changes to the core Linux kernel code. In contrast, the VServer adds less than 8700 lines of code to the Linux kernel, and due to its mostly architecture independent nature it has been

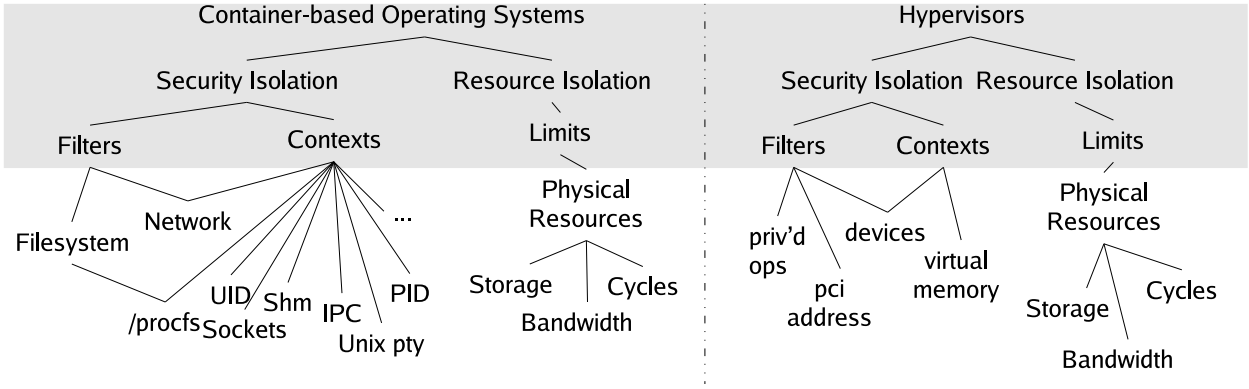


Figure 4: **Isolation Taxonomy of COS and Hypervisor-based Systems**

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 Linux 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

Linux 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).

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 [16] 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 simply 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 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 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 container- 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 72% for shell execution. In terms of absolute performance on server-type workloads, Xen3 lags an unvirtualized system by up to 38% for network throughput while demanding a comparable CPU load, and 25% for disk intensive workloads.

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. Due to reports [20] indicating that hyper-threading actually 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 run within a single VM provisioned with all available resources. In the case of Xen, the hypervisor does not include any device drivers. This requires a privileged host VM that exposes devices to guests. In our tests, the host VM used an additional 512MB, leaving 3000 MB for guests.

Linux and its derived systems 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 software architectures evaluated, rather than a particular set of configuration options. Appendix A [1] describes the specific configurations we have used in further detail.

The Xen configuration consists of Xen 3.0.2-testing [2]. The kernel configuration now allows a unified kernel, where both the host and guest VM are the same binary, but re-purposed at runtime. This kernel is derived from the 2.6.16 Linux kernel. Also, since Xen 3.0.2 allows guest VMs to leverage multiple virtual CPUs, our tests also evaluate an SMP-enabled guest.

The VServer configuration consists of VServer 2.0.1 and Fedora Core 4 security patches, applied to a Linux 2.6.17 kernel. Our VServer kernel includes several additions that have come as a result of VServer's integration with Planetlab. As discussed earlier, these include the new fair share CPU scheduler that preserves the existing $O(1)$ scheduler, and enables CPU reservations for VMs; a CFQ-based filter, which manages the disk resources based on context id; and a hierarchical token bucket packet scheduler, used to ensure fair sharing of network I/O.

4.1 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.

For the uniprocessor systems, our findings are consistent with the original report of Barham et al [3] that

[1] While not included due to space constraints, the appendix is available at http://www.cs.princeton.edu/ŝoltesz/eurosys07/appendixA.pdf

[2] The pace of Xen development is very rapid. We have settled on a single release known to be stable for our hardware and testing apparatus.

Config	Linux-UP	VServer-UP	Xen3-UP
fork process	103.2	103.4	282.7
exec process	252.9	253.6	734.4
sh process	1054.9	1047.1	1816.3
ctx (2p/ 0K)	2.254	2.496	3.549
ctx (16p/16K)	3.008	3.310	4.133
ctx (16p/64K)	5.026	4.994	6.354
page fault	1.065	1.070	2.245

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

Xen incurs a penalty for virtualizing the virtual memory hardware. While Xen3 has optimized page table update operations in the guest kernels over Xen2, common operations such as process executing, context switches and page faults still incur observable overhead. Table 2 shows results for the latency benchmarks, for which there is discrepancy between Linux-UP, VServer-UP, and Xen3-UP. The performance of the results not included does not vary significantly; i.e., they are equal within the margin of error.

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. However, the effect is still observable.

The next three rows show context switch overhead between different numbers of processes with different working set sizes. As explained by Barham [3], the $1\mu s$ to $2\mu s$ 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 difference across these micro-benchmarks comes directly from the Xen hypercalls needed to update the guest's page table. This is one of the most common operations in a multi-user system. While, optimizations for batching updates have occurred in Xen3, this inherent overhead is still measurable.

4.2 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 elements of a server workload in isolation provides insight into the sources of overhead. As well, we repeat various bench-

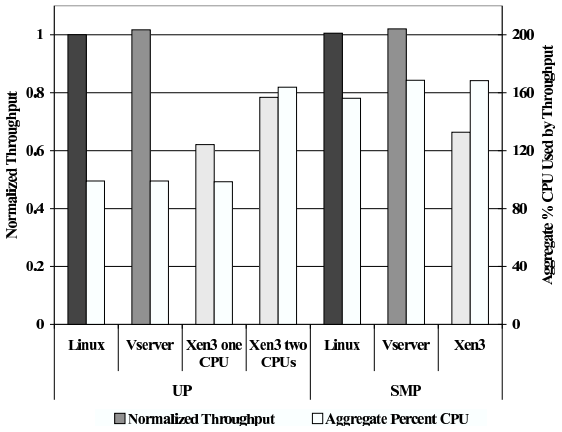


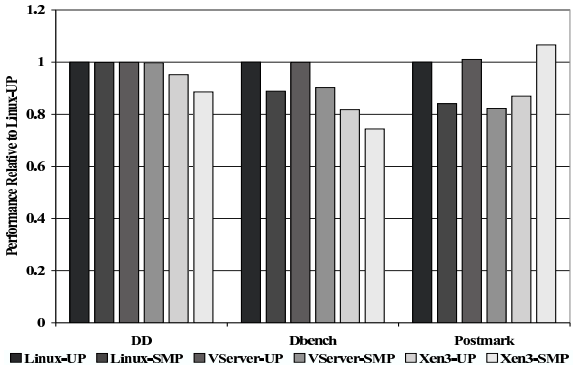
Figure 5: **CPU utilization during Network I/O.**

marks used in the original and subsequent performance measurements of Xen [3, 4, 6]. They exercise the whole system with a range of server-type workloads to illustrate the absolute performance offered by Linux, VServer, and Xen3.

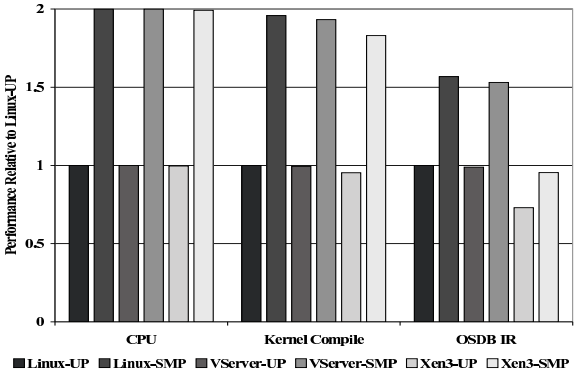
Figures 5, 6(a) and 6(b) include evaluations of all combinations of uniprocessor and SMP kernels and virtualizing system. In particular, there are various multi-threaded applications designed to create real-world, multi-component stresses on a system, such as Iperf, OSDB-IR, and a kernel compile. In addition, we explore several single-threaded applications such as a dd, Dbench and Postmark to gain further insight into the overhead of Xen.

Since we are running one VM per system, each VM is provisioned with all available memory, minus that required by the virtualized system. Each reported score is the median of 5 or more trials. All results are normalized relative to Linux-UP, unless otherwise stated. The data demonstrate that IO-bound applications suffer within a Xen VM. However, there are some interesting details.

Iperf is an established tool [1] for measuring link throughput with TCP or UDP traffic. We use it in this environment to exercise the networking subsystem of both virtualizing systems. Figure 5 illustrates both the throughput achieved and the aggregate percentage of CPU utilized to achieve this rate. First, maximum throughput was recorded, and then the aggregate CPU utilization was recorded independently, to avoid measurement overhead interfering with throughput. Bars on the left of each system correspond to the aggregate throughput, while bars to the right represent aggregate CPU utilization not reported idle by system performance monitors (a maximum of 200%, from 100% on two



(a) Disk performance



(b) Performance of CPU and memory bound benchmarks

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

CPUs) [3].

The first three columns are trials run with the uniprocessor kernels. The Xen3-UP kernel is pinned to the same CPU as the host VM and hypervisor, to reflect performance on a uniprocessor machine. For the Xen3-UP kernel, performance suffers a 38% overhead, while VServer-UP performance is within 2% of Linux-UP. The fourth column, is labeled 'Xen3 two CPUs'. In this case, the uniprocessor kernels are still used, but rather than pinning both kernels to a single CPU, the host and guest VM are placed on opposing processors. The increased burden of synchronization between CPUs (evidence by increased aggregate CPU utilization) falls to the Xen hypervisor. Though this increased CPU usage narrows the gap only to a 21% overhead. The last three columns are trials of the SMP kernels. Interestingly, all trials using

[3]Due to the different architecture of VServer and Xen, total CPU utilization was recorded using the *sysstat* package on VServer and *XenMon* on Xen.

two CPUs, either SMP or opposed uniprocessor kernels, consume similar total CPU time, within 8%. As well, the overall throughput increases for Xen-SMP over Xen-UP, but still maintains a high overhead.

To provide an intuition for the origin of this overhead, we observe that all I/O interrupts are first received by the Xen host VM, since it is privileged to interact with the physical devices. Then, a second virtual interrupt is delivered to and handled by the guest. As a result, packets are effectively handled by two operating systems, on both the incoming and outgoing path. While other work has sought to optimize the communication channel, as long as Xen uses the architecture of independent driver domains, some overhead will remain.

This scenario of double-handling in Xen is similar for data read from or written to virtual disks. Processes in the guest initiate I/O and the guest VM commits transactions to the virtual disk device. After this, the host VM receives the data before finally committing it to the physical device.

Figure 6(a) demonstrates the relative performance of workloads with differing working set sizes. DD highlights the overhead incurred by disk activity using this I/O model. Here, we write a 6GB file to a scratch device. Because dd is strictly I/O bound, the longer the code path is from client to disk the more delay will accumulate over time. This is observed in the 5-12% reduction in throughput seen in the measurements of Xen3.

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 kernels have additional overhead due to inherent overhead in SMP systems. Accordingly, the Xen3-UP performance is modestly greater than that of Xen3-SMP, but again, both have performance that is 19-25% less than Linux-UP, while VServer-UP reflects 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'. For the first time, Xen guests appear to perform better than the other systems under these loads.

Each VM uses a dedicated partition. The improved performance of Xen over Linux and VServer is due to the fact that I/O requests committed to the virtual disk are batched within the VM hosting the device before being committed to the physical disk. This batching is easily observed with $iostat$. While the amount of data written by both the guest and host is equal, there are 8x the number of transactions issued by the guest VM to the virtual device as issued to the physical device by the host VM.

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, since compilation is largely CPU bound, before measuring the compilation time, all source files are moved to a RAMFS. Therefore, performance times do not reflect disk related I/O effects. The figure indicates that while performance is generally good for Xen relative to Linux-UP, and overheads still range between 17% for Xen3-SMP and 5% for Xen3-UP.

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. This behavior is consistent with current web applications. Again, performance of VServer-UP is comparable to that of Linux-UP, but we do see a 35% reduction of throughput for Xen3-SMP relative to Linux-SMP. Not until we look at the performance of this system at scale do the dynamics at play become clear.

4.3 Performance at Scale

This section evaluates how effectively the virtualizing systems provide performance at scale.

4.3.1 OSDB

Barham et al. point out that unmodified Linux cannot run multiple instances of PostgreSQL due to conflicts in the

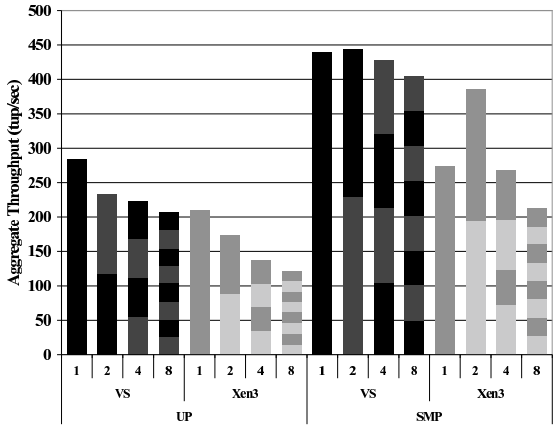


Figure 7: **OSDB-IR at Scale.** Performance across multiple VMs

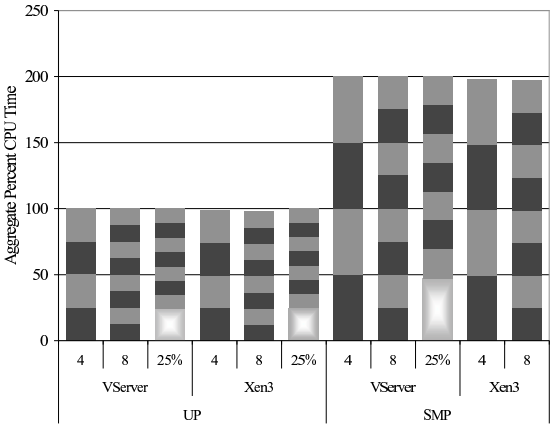


Figure 8: **Aggregate CPU-time received in a fair-share and 25% guarantee**

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 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. 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 perfect virtualization would partition the share of CPU time, buffer cache, and memory bandwidth perfectly among all active VMs and maintain the same aggregate throughput as the number of active VMs increased. However, for each additional VM, there is an arithmetic increase in the number of processes and the number of I/O requests. The diminishing trend observed in Figure 7 for VServer-UP and VServer-SMP illustrates the result. Since no virtualization is perfect, the intensity of the workload adds increasingly more pressure to the system and aggregate throughput diminishes.

Note that in Figure 7, the performance of all systems diminishes as the scale increases. But, as the load for Xen-UP approaches eight (8) simultaneous VMs, the performance disparity of relative to VServer-UP grows to 45%.

More surprising is the observed performance of Xen3-SMP. Here, the aggregate throughput of the system increases when two VMs are active, followed by the expected diminishing performance. Of course, VServer-SMP outperforms the Xen3-SMP system, but the total performance in the VServer, eight VM case is greater than any of the other Xen3-SMP tests. Therefore, the higher absolute performance of VServer is primarily due to the lower overhead imposed by the OS-virtualized approach. As a result, there is simply more CPU time left to serve clients at increasing scale.

5 Resource Isolation

VServer borrows the resource limits already available in Linux. Therefore, VServer can easily isolate VMs running fork-bombs and other antisocial activity through memory caps, process number caps, and other combinations of resource limits.

While there are potentially many ways to define the quality of resource isolation, we focus on two aspects. First, the virtualized system should multiplex a device or resource fairly when shared between multiple VMs. No VM, no matter how aggressively it uses a resource should interfere with the fair share of another VM. Whether resource allocations are fair-share or guaranteed, when the resource in contention is the same for both VMs, we call this single-dimensional isolation. Second, one VM should have limited ability to affect the activity of another using an *unrelated* resource. For instance, network activity while another VM runs a CPU-only application, cache hungry applications, disk I/O, etc, the overhead should be bounded so that each still receives a fair share of the resources available. Since the resources in contention are now different, we call this multi-dimensional

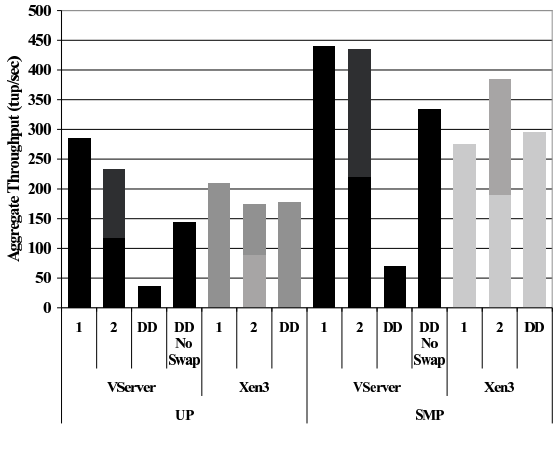


Figure 9: **Database performance with competing VMs**

isolation.

5.1 Single-dimensional Isolation

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 [19]. It is CPU-bound and involves no I/O.

As shown in figure 8, we run either four or eight VMs 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. The aggregate in each case is nearly 100% of all scheduled time, with minor overheads in Xen, due to multiple guest kernels.

In the four (4) and eight (8) VM case, each VM competes for a fair share of available time, and each receives approximately one fourth or one eighth. However, if the amount granted by a fair-share of the system is not adequate for a particular application, then the ability to make guarantees through resource reservation is also an option. The third column of figure 8 reports the amount of CPU time received when the CPU reservation of one VM is set to 25% of the system. As expected, 25% is delivered by VServer-UP, -SMP, and Xen-UP.

Unfortunately, the Xen, sedf scheduler has known issues with scheduling on SMP systems. As a result, priorities can be assigned to VMs, but they are not scheduled appropriately. Therefore, the final column for Xen-SMP is not available.

5.2 Multi-dimensional Isolation and Resource Guarantees

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 multi-dimensional 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 9 shows that OSDB on VServer suffers when competing with an active dd when a swap drive is enabled. Because the Linux kernel block cache is both global (shared by all VMs) and not accounted to the originating VM, the consequence is that dd adds significant pressure to available memory, polluting the block cache. Therefore, less memory is available for other VMs, and the performance of OSDB, which had run from a cached database, suffers because it is forced to re-read from disk.

This vulnerability is not present in Xen since the block cache is maintained by each kernel instance. Moreover, each virtual block device is mapped to a unique thread in the driver VM, allowing it to be governed by existing CFQ priorities. However, when the swap file for VServer is disabled, as in column four, we see OSDB performance returns to reasonable levels.

6 Conclusion

Virtualization technology brings benefits to a wide variety of usage scenarios. It provides many different techniques to enhance system impregnability through isolation, configuration independence and thereby software interoperability, better system utilization, and accountable and predictable performance. In terms of isolation and efficiency, we expect ongoing efforts to improve hypervisor- and container-based virtualization technologies along both dimensions.

In the mean time, for managed web hosting, PlanetLab, etc., the trade-off between isolation and efficiency is of paramount importance. Experiments indicate that container-based systems provide up to 2x the performance of hypervisor-based systems for server-type workloads. A number of different container- and hypervisor-based technologies exist, and choosing one for a particular system is clearly motivated by the set of virtualization features it provides. However, we expect container-based systems like VServer to compete strongly against hypervisor systems like Xen.

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] BANGA, G., DRUSCHEL, P., AND MOGUL, J. C. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proc. 3rd OSDI* (New Orleans, LA, Feb 1999), pp. 45–58.

[3] 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).

[4] CLARK, B., DESHANE, T., DOW, E., EVANCHIK, S., FINLAYSON, M., HERNE, J., AND MATTHEWS, J. Xen and the art of repeated research. In *USENIX Technical Conference FREENIX Track* (June 2004).

[5] 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.

[6] FRASER, K., HAND, S., NEUGEBAUER, R., PRATT, I., ELD, A. W., AND WILLIAMSON, M. 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] KAMP, P.-H., AND WATSON, R. N. M. Jails: Confining the Omnipotent Root. In *Proc. 2nd Int. SANE Conf.* (Maastricht, The Netherlands, May 2000).

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

[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 ADVANCED ROUTING AND TRAFFIC CONTROL. `http://lartc.org/`.

[11] LINUX-VSERVER PROJECT. `http://linux-vserver.org/`.

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

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

[14] NABAH, S., FRANKE, H., CHOI, J., SEETHARAMAN, C., KAPLAN, S., SINGHI, N., KASHYAP, V., AND KRAVETZ, M. Class-based prioritized resource control in Linux. In *Proc. OLS 2003* (Ottawa, Ontario, Canada, Jul 2003).

[15] OSMAN, S., SUBHRAVETI, D., SU, G., AND NIEH, J. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proc. 5th OSDI* (Boston, MA, Dec 2002), pp. 361–376.

[16] PETERSON, L., BAVIER, A., FIUCZYNSKI, M. E., AND MUIR, S. Experiences building planetlab. In *Proceedings of the 7th USENIX Symposium on Operating System Design and Implementation (OSDI '06)* (Seattle, WA, November 2006).

[17] POTTER, S., AND NIEH, J. 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).

[18] PRICE, D., AND TUCKER, A. Solaris zones: Operating system support for consolidating commercial workloads. In *Proceedings of the 18th Usenix LISA Conference.* (2004).

[19] REGEHR, J. Inferring scheduling behavior with hourglass. In *In Proceedings of the Freenix Track of the 2002 USENIX Annual Technical Conference* (June 2002).

[20] RUAN, Y., PAI, V. S., NAHUM, E., AND TRACEY, J. M. 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* (New York, NY, USA, 2005), ACM Press, pp. 315–326.

[21] SWIFT, M. M., BERSHAD, B. N., AND LEVY, H. M. Improving the reliability of commodity operating systems. *ACM Trans. Comput. Syst. 23*, 1 (2005), 77–110.

[22] SWSOFT. Virtuozzo Linux Virtualization. `http://www.virtuozzo.com`.

[23] VIVEK PAI AND KYOUNGSOO PARK. CoMon: A Monitoring Infrastructure for PlanetLab. `http://comon.cs.princeton.edu`.

[24] WRIGHT, C., COWAN, C., SMALLEY, S., MORRIS, J., AND KROAH-HARTMAN, G. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceedings of the 11th USENIX Security Symposium* (San Francisco, CA, Aug 2002).