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 *efficiency* and *isolation*. Examples include HPC clusters, the Grid, hosting centers, and PlanetLab. We present an alternative to hypervisors that is better suited for such scenarios. This 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 VServers. This paper describes the design and implementation of Linux Vservers–as a representative instance of container-based systems–and contrasts it with Xen, both architecturally and in terms of efficiency and support for isolation.

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 point in the design space that a given system supports depends on the workload it is designed to handle. Workstation operating systems generally run multiple applications on behalf of a single user, making it natural to favor sharing over isolation. Hypervisors are often designed to let a single machine host multiple unrelated applications, which may run on behalf of independent organizations, for instance, in the case of a data center consolidating multiple physical servers onto a single machine. 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 the same operating system.

A number of emerging usage scenarios—such as HPC clusters, Grid, web/db/game hosting organizations, distributed hosting (e.g., PlanetLab, Akamai)—benefit from virtualization techniques to isolate different groups of users and their applications from each other. 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 make efficient use of system resources while maintaining a high degree of isolation between VMs. The approach synthesizes ideas from prior work on *resource containers* [3, 13] and *security containers* [7, 20, 1, 22] as applied to general-purpose, time-shared operating systems. Indeed, variants of such container-based operating systems are in production use today—e.g., Solaris 10 [17], Virtuozzo [19], and Linux VServer [11].

The paper makes two contributions. First, it is the first thorough description of the overall techniques used by 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 600+ VServer-enabled machines on PlanetLab [5].

Second, we contrast the differences between VServer with a recent generation of Xen, which architecturally has changed drastically since its original design was described by Barham et al. [4]. 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 can far out scale Xen for 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 the standard for Xen and contrasts those with what can be achieved with VServer. Section 5, describes the kinds of interference observed between VMs. Finally, Section 6 offers some concluding remarks.

2 Motivation

This section motivates the utility of container-based virtualization. For clarity, we settle on a single set of terminology, drawn from the recent virtual machine literature: we refer to the underlying system providing virtualization as a virtual machine monitor (VMM) rather than a container-based OS or hypervisor, and the isolated execution contexts running on top of a VMM as virtual machines (VMs) rather than processes, containers, or domains.

We first outline the usage scenarios of VM technology to set the context within which we compare and contrast the different approaches to virtualization. We then substantiate the case for container-based VMMs.

2.1 Usage Scenarios

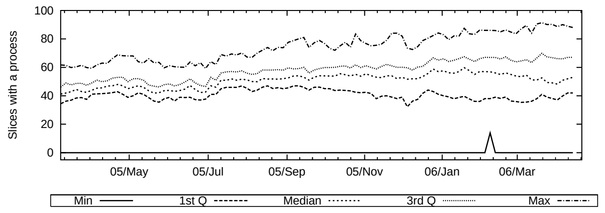
There are several innovative ideas that exploit VMs to secure work environments on laptops, detect virus attacks in real-time, determine the cause of computer break-ins, and debug difficult to track down system failures. Today, VMs are predominantly used by programmers to ease software development and testing, by IT centers to consolidate dedicated servers onto more cost effective hardware, and by traditional hosting organizations to sell virtual private servers (VPS). 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 typically realized by HPC clusters and idealized by the Grid vision, try to support many different users (and their application's specific software configurations) in a batch-scheduled manner. Experience shows 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. Giving users the ability to use their own distribution or specialized versions of system libraries in a VM would resolve this point of pain. While compute farms would 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.

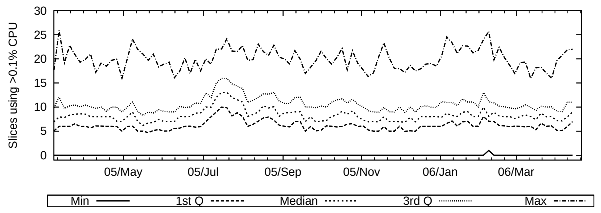
In contrast, hosting organizations tend to run many copies of the same server software, operating system distribution, and kernels in their mix of VMs. In for-profit scenarios, hosting organizations seek to benefit from an economy of scale and need to reduce the marginal cost per customer VM. Such hosting 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 [21]—one of the performance-monitoring services running on PlanetLab—publically 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 unprecendented 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 a process, and *live* if, in the last five minutes, it used at least 0.1% (300ms) of the CPU. Figure 1 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 con-



(a) Active slices, by quartile



(b) Live slices, by quartile

Figure 1: Active and live slices on PlanetLab

Features	Hyper-visor	Containers
Fault Isolation	✓	✗
Resource Isolation	✓	✓
Security Isolation	✓	✓
Multiple Kernels	✓	✗
Administrative power (root)	✓	✓
Checkpoint & Resume	✓	✓ [14, 19, 16]
Live Migration	✓	✓ [19, 16]
Live System Update	✗	✓ [16]

Table 1: Feature comparison of hypervisor and container based systems

cerned with both performance and scalability of the underlying VMM.

2.2 Case for Container-Based VMMs

The case for container-based VMMs rests on the assumption that for some real-world scenarios it is acceptable to trade isolation for efficiency. Sections 4 and 5 demonstrate quantitatively that a container-based VMM (VServer) is more efficient than a well designed hypervisor-based VMM (Xen). The question is, what does a VMM user have to give up to get that performance boost?

Table 1 provides a list of popular features that attract users to VM technologies. The three top rows define different kinds of isolation: *fault isolation*, *resource isolation*, and *security isolation*. A system provides full isolation if it supports all three kinds. As the following discussion illustrates, there is signifcant overlap of features between container- and hypervisor-based VMMs.

The fault isolation feature corresponds to the VMM's ability to isolate faults in one VM from affecting another VM. To ensure complete fault isolation requires that there is no direct sharing of code or data between VMs. However, the design of a container-based VMM consists of applying virtualization abstractions directly to a single, *shared* kernel (such as Linux). Therefore, a container-based VMM inherently cannot provide fault isolation from a kernel crash like a hypervisor can.

The resource isolation feature corresponds to the VMM's ability to isolate the resource consumption of one VM from that of another VM; undesired interactions between VMs are sometimes called cross-talk [9]. Providing resource isolation generally involves careful scheduling and allocation of physical machine resources (e.g., cycles, memory, link bandwidth, disk space), but can also be influenced by VMs sharing logical resources, such as file descriptors and memory buffers. At one extreme, a VMM 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 VMM might let VMs obtain cycles and bandwidth on a demand-driven (best-effort) basis. Many hybrid approaches are also possible: for instance, a VMM 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 hypervisor- and container-based VMMs incorporate sophisticated resource schedulers to avoid or minimize crosstalk.

The security isolation feature refers to the extent to which the VMM limits access to (and information about) logical objects, such as files, memory addresses, port numbers, user ids, process ids, and so on. A VMM 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 VMM 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. 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*, so one VM is not able to see or modify data and code belonging to another VM, which increases the likelihood that

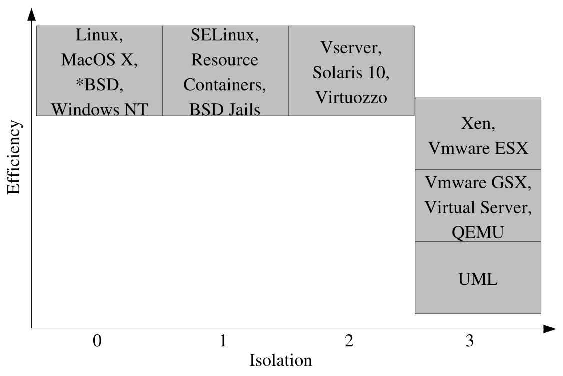


Figure 2: **Summary of existing VMM technology**

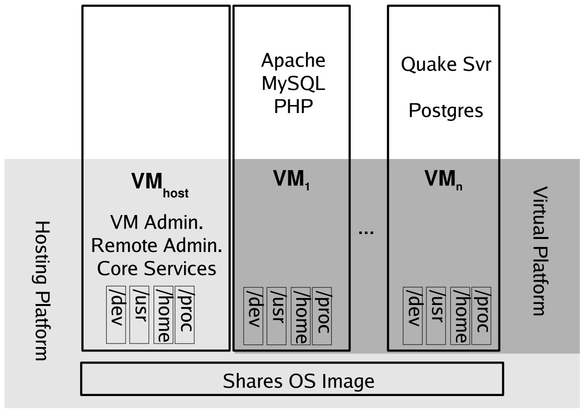


Figure 3: **Container-based VMM Overview**

a compromise to one VM does not affect others on the same machine. Both hypervisors and containers can hide logical objects in one VM from other VMs.

Finally, hypervisor-based solutions are often embraced for their ability to support the other features listed in Table 1, including the 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. Since container-based VMMs rely on a single underlying kernel image, they are of course not able to run multiple kernels like hypervisor VMMs can. However, container-based solutions support the other features—and the corresponding references are provided in the table. In fact, at least one solution supporting container-based migration goes a step further: it enables VM migration from one kernel *version* to another. This powerful 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 shutdown the VM. Kernel version migration is possible because container-based solutions have explicit knowledge of the dependencies that processes within a VM have to in-kernel structure. Such dependencies are externalized by the container-based VMM in a kernel independent manner and then internalized again when moving to a container-based VMM operating on a different kernel version [16].

For specific usage scenarios, either hypervisors or containers may be the only VMM technology that provides the required feature set. For others, the fundamental tradeoff given current VMM technology is one of *efficiency* versus *isolation*. Figure 2 summarizes the state-of-the-art in VMM technology along these two 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 quantative results. The key observation is, to date, there is no VMM 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 container-based solution like VServer hits the sweet spot within this space. Conversely, for scenarios where full isolation is required, a hypervisor-based VMM is best.

3 Container-based VMMs

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

3.1 Overview

A container-based VMM provides a shared, virtualized OS image, including a unique root file system, a (safely shared) set of system executables and libraries, and whatever resources the root adminstrator assigns 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 the shared OS image and a priviledged *host VM* (VMhost). This is the VM that a system administrator uses to manage other VMs. The virtual platform is the

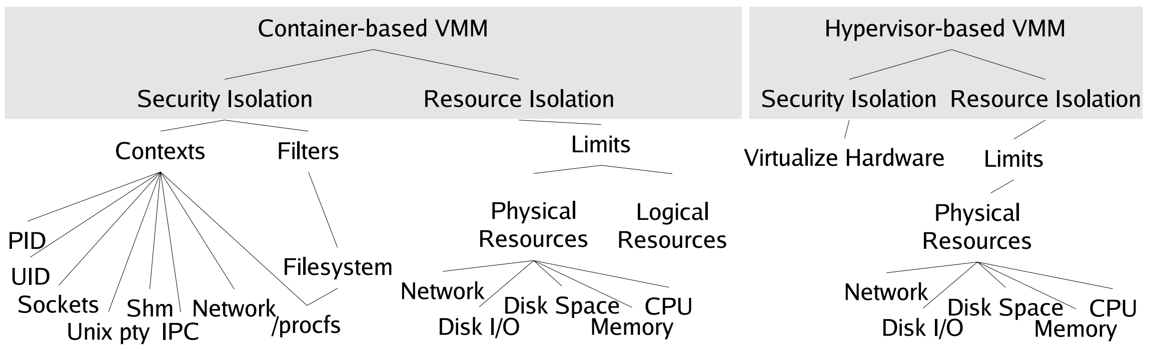


Figure 4: **Comparison of Container and Hypervisor VMM Taxonomy**

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 nearly no difference between a container and hypervisor based system. In fact, at many conceptual levels the two approaches are similar. 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. The third level of both hierarchies represents the general techniques that each VMM uses.

As shown in the figure, a container-based VMM 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 objects live in completely different spaces (for example, per VM lists), do not have pointers to objects in other spaces, and thus cannot get access to objects outside of its name space. The latter means that access to objects involve runtime checks by the VMM to determine whether the VM has the appropriate permissions. In contrast, for a hypervisor-based VMM, security isolation is achieved by virtualizing as much of the hardware as possible. Special treatment is needed for the x86 architecture, the details of which are beyond of the scope of this paper and are treated elsewhere; a simplified view of security isolation primarily involves controlling access to physical memory and devices.

The techniques for resource isolation between the two VMM systems are quite similar. Both need to multiplex physical resources. 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 priviledged 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 sharing of network bandwidth and disk i/o 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.

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

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 runqueue 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 work-conserving CPU reservations.

The rate that tokens accumulate in a VM's bucket depends on whether the VM has a *reservation* 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. On PlanetLab, for example, the default share for a VM is actually a small reservation, providing the VM with 32 tokens every second, or 3% of the total capacity.

The main difference between reservations and shares occurs when there are runnable processes but no VM has enough tokens to run: in this case, VMs with shares are given priority over VMs with reservations. First, if there is a runnable VM with shares, tokens are given out fairly to all VMs with shares (i.e., in proportion to the number of shares each VM has) until one can run. If there are no runnable VMs with shares, then tokens are given out fairly to VMs with reservations. 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.

3.2.2 Network

The Hierarchical Token Bucket (htb) queuing discipline of the Linux Traffic Control facility (tc) [10] is used to provide bandwidth reservations and fair service among VServers. 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 VServers 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).

3.2.3 Disk and Memory

VServer provides the ability to associate disk quotas and memory limits with VMs. Both limit the amount of physical resource that the VM can access and allocate. Note that when a node attempts to scale up the number of VMs, as is the case on PlanetLab [5], fixed memory allocations may not be appropriate. 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.

Disk I/O is managed in VServer using Linux's standard 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.

3.2.4 Logical Resources

VServer also limits how much logical resources a VM may consume. The logical resources include:

- RSS: Number of pages the VM's resident set can consume (the number of virtual pages resident in RAM).
- PROC: Number of processes that can be created within a VM.
- OPENFD: Aggregate number of file descriptor that can be opened by processes within a VM.
- MEMLOCK: Number of virtual memory pages that may be locked into RAM using mlock() and mlockall() by processes within a VM.
- VM: Number of virtual memory pages available to the processes within a VM (address space limit).
- LOCKS: Number of file systems locks a VM may have.
- ANON: Number of anonymous memory pages a VM may have.
- SHMEM: Number of pages that can be declared as SYSV shared memory.

3.3 VServer Security Isolation

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

3.3.1 Process Contexts

Processes are contextualize in order to hide all processes outside a VM's scope, and prohibit 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 become aware of contexts and to differentiate between identical uids used by different VMs.

It also requires the definition of a 'default' VM that is used when the host system is booted, and to work around the issues resulting from some false assumptions made by some user-space tools (like pstree) that the 'init' process has to exist and have pid '1'.

To simplify administration, processes belonging to the host VM are contextualized as well. 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 trivially achieved by changing its context identified.

3.3.2 Network Context

While Process Contexts is sufficient to isolate groups of processes, a different kind of separation, or rather a limitation, is required to confine processes to a subset of available network addresses.

Several issues have to be considered when doing so; for example, the fact that bindings to special addresses like IPADDR_ANY or the local host address have to be handled in a very special way.

Currently, VServer does not make use of virtual network devices (and maybe never will) to minimize the resulting overhead. Therefore socket binding and packet transmission have been adjusted.

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() Syscall.

One simple method to escape from a chroot-ed environment is as follows: First, 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. Next, 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() Syscall.

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

3.3.4 Upper Bound for Caps

Because the current Linux Capability system does not implement the filesystem related portions of POSIX Capabilities which 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 (bits) 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 container-based VM, which are not very likely 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 deliberately or accidentally 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 available physical resources are exposed to fulfill a given workload.

The comparison shows that although performance optimizations and the inclusion of new features continues with Xen3, the overhead required by acting as broker for the virutal memory sub-system still introduces an overhead of 45% for OS operations such as memory mapping a file, and up to 60% for process execution. In terms of absolute performance on server-type workloads, Xen3 lags an unvirtualized system by up to 70% for network I/O while requiring comparable CPU time, and 50% for disk intensive workloads. For workloads designed to stress the entire system, such as SPEC WEB99, performance again sufferes by as much as 70%. Yet, for all of these tests, VServer performance is comparable to an unvirtualized Linux kernel.

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 [**?**] indicating that hyper-threading actually degrades performance for certain environments, we run all tests with hyper-threading disabled. All systems are compiled for both uni-processor and SMP architectures, and unless otherwise noted, all experiments run within a single VM provisioned with all available resources.

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 homogenous 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 describes the specific configurations we have used in further detail.

The following systems share a common origin in the 2.6.12 Linux kernel, and because Xen3 and VServer are currently distributed as patches to the Linux kernel, an unmodified Linux kernel (Linux) will provide an unvirtualized reference to contrast the impact that hypervisor- or container-based approaches have on system efficiency.

The Xen configuration consists of Xen 3.0.1-testing [1] as well as Xen 2.0.8. Both the host VM (dom0) and guest VM (domU) are derived from the 2.6.12 Linux kernel. Also, since Xen 3.0.1 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, which is again a patch to an unmodified Linux 2.6.12 kernel. Our VServer Linux kernel (refered to as simply VServer in the text) 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 manages the disk resources; and a hierarchical token bucket packet scheduler is used to ensure fair sharing of network I/O.

[1]The pace of Xen development is so rapid, and stability so unpredictable at each incremental change that we have settled on a single release known to be stable for our hardware and testing aparatus. We have adjusted the configuration of the other systems to compensate where appropriate for missing features, such as hardware offloading of TCP operations

4.1 Micro-Benchmarks

While micro-benchmarks are incomplete indicators of system behavior for real workloads [6], 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 [12] 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. In particular, we see higher process creation and context switch times, higher communication latencies for pipes, UNIX domain and IP sockets, as well as more than a 50% reduction of local socket bandwidth in SMP kernels. 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 [4] that Xen incurs a penalty for virtualizing the virtual memory hardware. While Xen3 has optimized page table update operations in the guest kernels over Xen2, a 60% overhead remains for common operations such as process executing and memory mapping large files. Table 2 shows results for the latency benchmarks, for which there is discrepancy between Linux-UP, VServer-UP, Xen2-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 3% 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, this still induces an overhead of 60% over Linux-UP.

The next three rows show context switch overhead between different numbers of processes with different working set sizes. As explained by Barham [4], the 1μs to 3μs overhead for these micro-benchmarks are due to hypercalls from XenoLinux into the VMM to change the page table base. In contrast, there is little overhead be-

Config	Linux-UP	VServer-UP	Xen2-UP	Xen3-UP
fork process	103.70	105.20	302.70	249.70
exec process	365.10	367.40	812.30	695.00
sh process	1086.00	1095.70	2128.30	1735.10
ctx (2p/ 0K)	1.78	1.85	3.46	3.43
ctx (16p/16K)	2.52	2.63	3.94	3.85
ctx (16p/64K)	4.06	4.23	5.89	6.49
mmap (64MB)	588.00	585.00	804.00	874.00
mmap (256MB)	2341.00	2328.00	3162.00	3391.00
prot fault	0.58	0.55	0.87	0.86
page fault	1.54	1.54	2.17	2.27

Table 2: lmbench latency OS benchmarks for Uniprocessor kernels – times in μs

tween Linux-UP and VSever-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 $3.3\mu s$ overhead per megabyte, or nearly 33% additional processing time compared to the other systems. This is particularly relevant for servers or applications that use *mmap* as a buffering technique or convenient access to large data sets, such as [15].

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 occured in Xen3, this fundamental overhead is still measurable.

4.2 System Benchmarks

We repeat a number of the single VM benchmarks that have become familiar metrics for validation of the virtualization techniques of Xen [4, **?**]. They exercise the whole system with a range of server-type workloads to illustrate the absolute performance offered by Linux, VServer, and Xen3.

Figure 5 includes tests run against Uniprossor and SMP kernels for the three systems we are evaluating. In particular, there are various multi-threaded applications designed to create real-world, multi-component stresses on a system, such as Iperf, OSDB-IR, a kernel compile, and SPEC WEB99. In addition, we explore several single-threaded applications such as a Dbench and Postmark to gain further insight into the overhead of Xen.

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

The first test SPEC WEB99, an industry standard benchmark for webserver platforms, is designed specifically to be sensitive to small degrees of system overhead. In particular, achieving a good score requires both high throughput and bounded latency. When a client request gets stalled or badly delayed due to a slow disk read or saturated network, the connection will be classified as *non-conforming* and will not contribute to the overall score. Figure 5 shows the results of running SPEC WEB99 on all platforms. Clearly, the scores of Linux and VServer are very comparable for both UP and SMP kernels. However, the achievable score of Xen3-UP suffers a reduction of 64% over the non-virtualized Linux-UP, and for the Xen3-SMP kernel, performance is reduced only to 47% of Linux-SMP, still below the score for the Linux and VServer UP kernels.

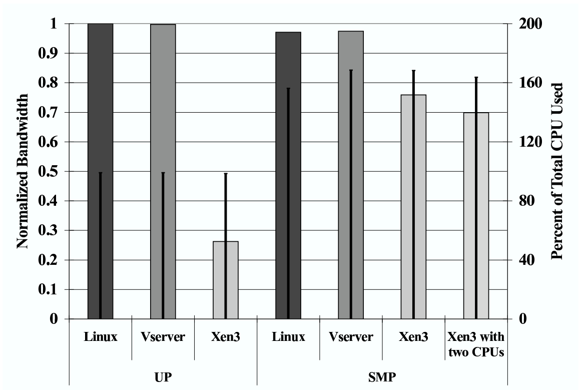


Figure 6: **CPU utilization during Network I/O.**

This degree of achievable performance in the Xen3 kernels is due to two factors: overhead in network I/O and overhead in disk I/O. The SPEC WEB99 workload

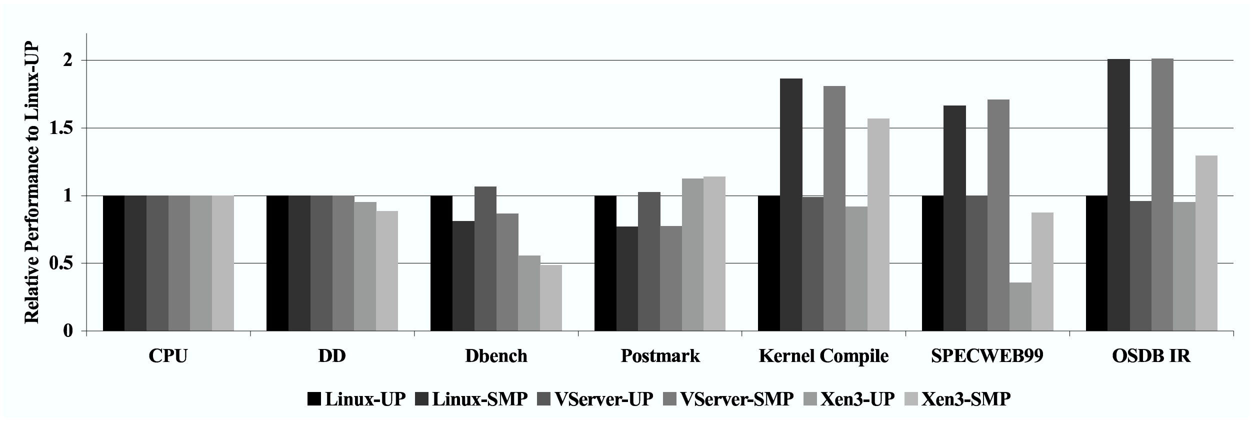


Figure 5: **Relative performance of Linux, VServer, and XenU kernels.** Measurements for various benchmarks are shown for each system - ULK-UP is the uniprocessor kernel of Unmodified Linux. VServer-UP is the uniprocessor kernel of VServer. Xen2-domU and Xen3-domU are uniprocessor kernels working in tandem with dom0 on a single CPU.

exercises all aspects of the system, CPU, network and disk. Looking at these elements in isolation provides insight into the SPEC WEB99 overhead.

A single-threaded, CPU-bound process demonstrates that all platforms introduce no observable overhead. When no other operation competes for CPU time, this process receives all available system time. This validates all the virtualizing techniques as able to achieve native performance when there is no I/O.

Iperf is an established tool [2] for measuring bandwidth with TCP or UDP traffic. We use it in this environment to exercise the networking subsystem of both virtualizing systems. Figure 6 illustrates both the bandwidth achieved across the nodes, and the percent of total CPU utilization needed to achieve this amount. The wide bars correspond to the total bandwidth, while the thin bars represent all CPU not reported as idle by the system performance monitors used during testing [2]. Performance here is worse than the Xen3-UP kernels by 70%. When two CPUs are used in the SMP configuration, the overall CPU utilization increases, but still maintains a 24-30% reduction compared to the full throughput of Linux-UP.

While both versions of Xen tested did not support TCP offloading, this feature was also disabled in the unmodified Linux and VServer kernels. This certainly contributes to the high degree of CPU time observed. A further contributor is that all I/O interrupts are received by the hosting VMM, and then delivered to the guest via a virtual interrupt. Therefore, packets are effectively handled by two operating systems, both on the incoming and outgoing path.

This scenario of double-handling is similar for data read from or writte to disk. Processes in the guest initiate I/O. The guest commits transactions to the virtual disk device, after which the host (dom0) receives the data before finally committing it to the physical device.

DBench and the compilation of a standard kernel highlight the overhead derived from this I/O model. Here, dbench is strictly I/O bound, so the longer the code path is from client to disk the more delay will accumulate over time. This is observed in the measurements. The Linux-SMP and VServer-SMP kernels have additional overhead due to dbench being a single threaded process and the overhead inherent in SMP systems. Accordingly, the Xen3-UP performance is modestly greater than that of Xen3-SMP, but again, both have performance that is at least 44% less than Linux-UP.

Next, a standard kernel compile uses multiple threads and is both CPU intensive as well as exercising the filesystem with many small file reads and creates. The figure indicates that there is generally good performance for Xen relative to Linux-UP, and overheads are no more than 19% for Xen3-SMP and 9% for Xen3-UP. This suggests that compilation is largely CPU bound, and the overhead observed in dbench due to disk I/O delay, has either modest impact on this test or is amortized over time.

Postmark [8] is a single-threaded benchmark originally designed to stress filesystems. It allows a configurable number of files and directories to be created, and followed by a number of random transactions on these files. In particular, it generates many small transactions like those experienced by a heavily loaded email or news

[2]For testing on Linux and VServer, *sar* of the *sysstat* package was used, while for Xen we used the *XenMon* package

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 VMs use its dedicated partition. The improved performance of Xen over Linux and VServer is due to the fact that I/O committed to the virtual disk are batched within the hosting VMM beforing being committed to the physical disk. This batching is easily observable 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 domU to the virtual device as issued to the physical device by dom0. Thus, while overall throughput is increased by batching transactions before committing them to disk, there are unanswered questions in regard to data and filesystem integrity for common or database applications. The guest VM believes that every transaction to the virtual device is committed permanently, but this assumption is violated by the backend driver.

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 the database, a behavior consistent with current web applications. The performance is quite comparable to that of Linux-UP and VServer-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 of the system become clear.

4.3 Performance at Scale

This section evaluates how effectively the systems provide performance at scale, and in particular, running multiple instances of OSDB-IR. 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 container-based design.

Barham et al. point out that unmodified Linux cannot run multiple instances of PostgreSQL due to conflicts in the SysV IPC namespace. However, the mechanisms present in VServer for security isolation contain the SysV IPC namespace within each VServer context. Consequently, a framework now exists to directly compare the scalabilty of Xen to VServer.

The Information Retrieval component of the OSDB package requires both high disk throughput and fair resource sharing. If disk I/O 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.

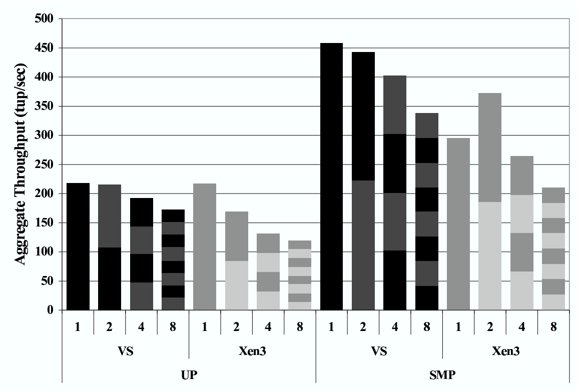


Figure 7: **OSDB-IR at Scale.** Performance across multiple VMs

A perfect virtualization would partition the share of resources perfectly among all active VMs, and maintain the 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 single VM case (1) for VServer performs comparably to Xen3-UP, but after the load approaches eight (8) simultanious VMs, the performance disparity increases up to 30-38%. VServer-SMP outperforms the Xen3-SMP equally well.

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. This suggests an inherent throttle in the Xen3-SMP system that prevents maximum utilization of the Xen system until multiple VMs are active. As a result, full system capacity is not realized until additional VMs are added to effectively ’fill in’ the under-utilized resources. While the total performance in this case is greater than of the single-VM, the improved aggregate throughput is still 15% less than VServer-SMP at the same scale.

In summary, 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 left to serve clients at increasing scale.

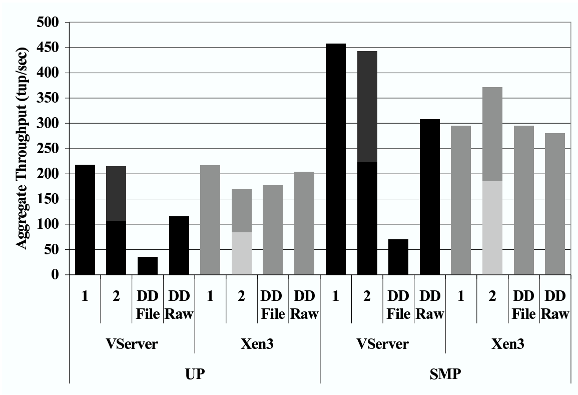


Figure 8: **Database performance with competing VMs**

5 Resource Isolation

Both security and resource isolation are necessary to protect VMs from one another. Earlier sections have discussed the techniques used to address security isolation in container-based OS virtualization. This section validates this approach as providing a comparable degree of flexibility and precision with respect to resource isolation.

While there are potentially many different ways to define the quality of resource isolation, we focus on two aspects. First, the VMM should multiplex a device or resource fairly between multiple users. No VM, no matter how aggressively it uses a resource should interfere with any other VMs fair share. When the resource in contention is the same for both VMs we call this single-dimensional isolation. This also includes reservations or guarantees of a certain amount. Second, one VM should have limited ability to affect the activity of another using an *unrelated* resource. For instance, when running a network-intensive file transfer at the same time another VM runs a CPU intensive application, the overhead should be bounded so that each still receives a fair share or the resources available. Since the resources in contention are now different, we call this multi-dimensional isolation.

5.1 Single-dimensional Isolation

As noted earlier, traditional time-sharing UNIX systems have a legacy of vulnerability to layer-below attacks. To investigate whether Linux VServer is still susceptible to single-dimensional resource 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 on a disk common to both VMs.

Figure 8 shows that OSDB on VServer suffers when competing with an active dd. This is the result of dd writing to a *file* rather than to a raw device. Since the block cache maintained by the kernel is both global and not accounted to the originating VServer, the consequence is that dd pollutes the block cache. Therefore, less system memory is available for other processes or VMs, and the performance of OSDB suffers.

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. A second experiment writes to a raw device. Now, the block cache is not involved, and it is up to the CFQ disk I/O scheduler to correctly schedule disk activity based on the associated VServer for the process performing the request. In this case, performance returns to the expected level. This difference highlights one aspect of container-based OS virtualized systems that remain open to sharing. Other resource container implementations may address the issue of a global block cache differently.

5.2 Resource Guarantees

To investigate both isolation across different resources and resource guarantees, we use a combination of iperf and hourglass, a synthetic real-time application useful for investigating scheduling behavior at microsecond granularity [18]. It is CPU-bound and involves no I/O. Since TCP offloading is disabled for all systems, network I/O activity should still interfere minimally with a CPU-bound workload.

Again, we instantiate two opposing VMs. The first acts as an iperf server, receiving connections from six clients across two network interfaces. The second runs hourglass as a CPU bound thread that records contiguous periods of time that it is scheduled. Because hourglass has no I/O, we may infer from the gaps in its time-line that either the second VM is running or the VMM is running on behalf of the second VM. In the case of network I/O, therefore, gaps refer to time spent by the driver VM receiving packets in the interrupt handler or the second VM communicating with the kernel.

Figure 9 displays the percentage of CPU time received when competing with iperf. In each case, the percentage is roughly 50%, or a fair share of available cpu time. The 3% missing from VServer is attributable to the higher clock-rate of the system. Also, it is not surprising that

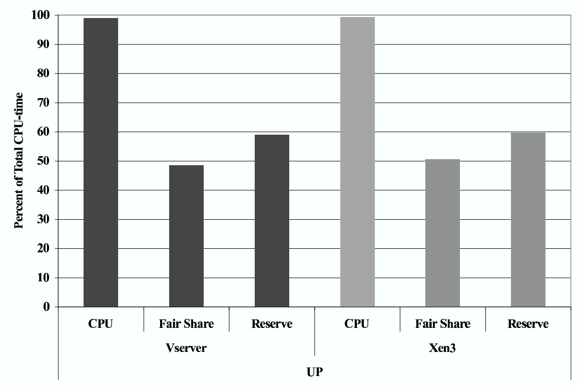


Figure 9: **Percent of CPU-time received with and without a network workload**

network throughput suffers when competing with a CPU intensive process, given the values reported earlier for the CPU load generated to achieve full capacity communication.

However, if the amount granted by a fair-share of the system is not adequate for a particualr application, than the ability to make guarantees through resource reservation is also an option. The third column of figure 9 reports the amount of CPU time received by hourglass when the CPU reservation is set to 60% of the system. As expected, 60% is delivered in each case.

Finally, VServer can easily isolate VMs running fork-bombs and other antisocial activity through memory caps, process number caps, and other combinations of resource limits.

6 Conclusion

Virtualization technology brings benefits to a wide variety of usage scenarios. For some, like PlanetLab, the fundamental tradeoff that VMMs make between isolation and efficiency is of paramount importance. Experiments indicate that container-based VMMs provide up to 2x the performance of hypervisor-based systems for certain workloads. A number of different VMM technologies exist, and the choice of VMM for a particular system is clearly motivated by the set of virtualization features it provides. However, we expect container-based VMMs to compete strongly against hypervisor systems like Xen.

References

[1] Security-enhanced linux. `http://www.nsa.gov/selinux/`.

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

[3] 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.

[4] 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).

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

[6] 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.

[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 VServers Project. `http://linux-vserver.org/`.

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

[13] 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).

[14] 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.

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

[16] 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).

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

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

[19] SWSoft. Virtuozzo Technology. `http://www.sw-soft.com`.

[20] Tucker, A., and Comay, D. Solaris Zones: Operating System Support for Server Consolidation. In *3rd Virtual Machine Research and Technology Symposium Works-in-Progress* (San Jose, CA, May 2004).

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

[22] 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).

A Normalized Configuration

A significant aspect of this effort is ensuring that the experiments are fair. We report the configuration details of the various subsystems.

A.1 Filesystem

The distribution populating all filesystems is the latest release of Fedora Core 2, but the read and write performance of platter-based hard drives naturally diminishes across the extent of the device. To account for this variation, each virtual machine is allocated a dedicated LVM partition, that is used solely by this VM, irrespective of which VMM is currently active.

This configuration departs from a traditional VServer system. As noted earlier, VServer can use a file-level copy-on-write technique to replicate the base environment without unnecessary redundancy. However, the decision to depart from this convention allows the tests to reflect the differences of performance at a lower level without introducing particular optimizations that aim to increase the sharing available in the system.

A.2 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 portspace 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.

All kernels, boot with the following sysctl.conf:

Listing 1: Non-default kernel configuration

```
# All VMs
net.core.rmem_max=1048576
net.core.rmem_default=1048576
net.core.wmem_max=1048576
net.core.wmem_default=1048576
net.ipv4.tcp_max_syn_backlog=20480
net.ipv4.tcp_timestamps=0
net.ipv4.tcp_max_tw_buckets=2000000
net.ipv4.tcp_tw_recycle=1
net.ipv4.tcp_tw_reuse=1
net.core.netdev_max_backlog=20000
net.core.somaxconn=20480
```

In all test cases, these settings effect networking buffers. They increase the system default and are necessary to give optimial performance for http-based benchmarks where numerous connections are created and destroyed in less time than $TIME\_WAIT$. Moreover, they specify buffer sizes that optimize throughput for the local link between client and server.

A.3 Client Configuration

Three client machines are attached to each Ethernet port on the system under test through a 1 Gbps Netgear switch. Each client is a 1.3GHz AMD Duron PC equiped with a AC9100 Gigabit Ethernet, and running RedHat 9.0.

A.4 System Clock

The XenoLinux system clock is 100Hz whereas VServer and Linux use a 1000HZ clock. In future work this difference will be eliminated, as it is unclear how much additional overhead is incurred from 10x the number of context switches in Linux and VServer, or the benefit realized by allowing longer quanta in Xen.