

# Linux-VServer

Resource Efficient OS-Level Virtualization

Herbert Pötzl

herbert@13thfloor.at

Marc E. Fiuczynski

mef@cs.princeton.edu

## Abstract

Linux-VServer is a lightweight virtualization system used to create many independent *containers* under a common Linux kernel. To applications and the user of a Linux-VServer based system, such a container appears just like a separate host.

The Linux-Vserver approach to kernel subsystem containerization is based on the concept of *context isolation*. The kernel is modified to isolate a container into a separate, logical execution context such that it cannot see or impact processes, files, network traffic, global IPC/SHM, etc., belonging to another container.

Linux-VServer has been around for several years and its fundamental design goal is to be an extremely low overhead yet highly flexible production quality solution. It is actively used in situations requiring strong isolation where overall system efficiency is important, such as web hosting centers, server consolidation, high performance clusters, and embedded systems.

## 1 Introduction

This paper describes Linux-VServer, which is a virtualization approach that applies *context isolation* techniques to the Linux kernel in order to create lightweight container instances. Its implementation consists of a separate kernel patch set that adds approximately 17K lines of code to the Linux kernel. Due to its architecture independent nature it has been validated to work on eight different processor architectures (x86, sparc, alpha, ppc, arm, mips, etc.). While relatively lean in terms of overall size, Linux-VServer touches roughly 460 existing kernel files—representing a non-trivial software-engineering task. Linux-VServer is an efficient and flexible solution that is broadly used for both *hosting* and *sandboxing* scenarios.

*Hosting scenarios* such as web hosting centers providing Virtual Private Servers (VPS) and HPC clusters need to isolate different groups of users and their applications from each other. Linux-VServer has been in production use for several years by numerous VPS hosting centers around the world. Furthermore, it has been in use since 2003 by PlanetLab ([www.planet-lab.org](http://www.planet-lab.org)), which is a geographically distributed research facility consisting of roughly 750 machines located in more than 30 countries. Due to its efficiency, a large number of VPSs can be robustly hosted on a single machine. For example, the average PlanetLab machine today has a 2.4Ghz x86 processor, 1GB RAM, and 100GB of disk space and typically hosts anywhere from 40-100 live VPSes. Hosting centers typically use even more powerful servers and it is not uncommon for them to pack 200 VPSes onto a single machine.

Server consolidation is another hosting scenario where isolation between independent application services (e.g., db, dns, web, print) improves overall system robustness. Failures or misbehavior of one service should not impact the performance of another. While hypervisors like Xen and VMware typically dominate the server consolidation space, a Linux-VServer solution may be better when resource efficiency and raw performance are required. For example, an exciting development is that the One Laptop Per Child (OLPC) project has recently decided to use Linux-VServer for their gateway servers that will provide services such as print, web, blog, void, backup, mesh/wifi, etc., to their \$100 laptops at school premises. These OLPC gateway servers will be based on low cost / low power consuming embedded systems hardware, and for this reason a resource efficient solution like Linux-VServer rather than Xen was chosen.

*Sandboxing scenarios* for generic application plugins are emerging on mobile terminals and web browsers to isolate arbitrary plugins—not just Java lets—downloaded by the user. For its laptops OLPC has designed a security framework, called bitfrost [1], and has decided to

utilize Linux-VServer as its sandboxing solution, isolating the various activities from each other and protecting the *main* system from all harm.

Of course, Linux-VServer is not the only approach to implementing containers on Linux. Alternatives include non-mainlined solutions such as OpenVZ, Virtuozzo, and Ensim; and, there is an active community of developers working on kernel patches to incrementally containerize the mainline kernel. While these approaches differ at the implementation level, they all typically focus in onto a single point within the overall containerization design spectrum: *system virtualization*. Benchmarks run on these alternative approaches to Linux-VServer reveal non-trivial time and space overhead, which we believe are fundamentally due to their focus on system virtualization. In contrast, we have found with Linux-VServer that using a context isolation approach to containerize critical kernel subsystems yields negligible overhead compared to a vanilla kernel—and in most cases performance of Linux-VServer and Linux is indistinguishable.

This paper has two goals: 1) serve as a gentle introduction to container-based system for the general Linux community, and 2) highlight both the benefits (and drawbacks) of our context isolation approach to kernel containerization. The next section presents a high-level overview of container-based systems and then describes the Linux-VServer approach in further detail. Section 3 evaluates efficiency of Linux-VServer. Finally, Section 4 offers some concluding remarks.

## 2 Linux-VServer Design Approach

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.

### 2.1 Container-based System Overview

At a high-level, a container-based system provides a shared, virtualized OS image, including a unique root file system, a set of system executables and libraries, and resources (cpu, memory, storage, etc.) assigned to the container when it is created. Each container can be “booted” and “shut down” just like a regular operating system, and “rebooted” in only seconds when necessary.

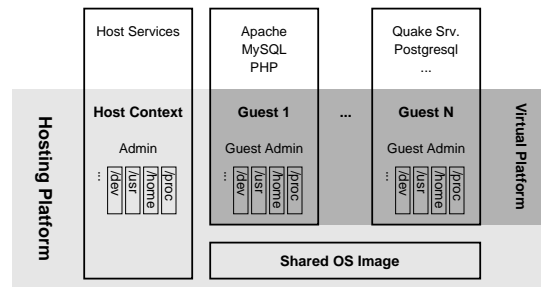


Figure 1: Container-based Platform

To applications and the user of a container-based system, the container appears just like a separate linux system.

Figure 1 depicts a container-based system, which is comprised of two basic platform groupings. The hosting platform consists essentially of the shared OS image and a privileged *host context*. This is the context that a system administrator uses to manage containers. The virtual platform is the view of the system as seen by the *guest containers*. Applications running in a guest container work just as they would on a corresponding non-container-based system. That is, they have their own root file system, IP addresses, `/dev`, `/proc`, etc.

The subsequent sections will focus on the main contributions that Linux-VServer makes, rather than being exhaustively describing all required kernel modifications.

### 2.2 Kernel Containerization

This section describes the kernel subsystem enhancements that Linux-VServer makes to support containers. These enhancements are designed to be low overhead, flexible, as well as to enhance security in order to properly confine applications into a container. For CPU scheduling, Linux-VServer introduces a novel filtering technique in order to support fair-share, work-conserving, or hard limit container scheduling. However, in terms of managing system resources such as storage space, io bandwidth, for Linux-VServer it is mostly an exercise of leveraging existing Linux resource management and accounting facilities.

#### 2.2.1 Guest Filesystems

Guest container filesystems could either be implemented using loop-back mounted images—as is typical

for qemu, xen, vmware, and uml based systems—or by simply using the native file systems and using chroot. The main benefit of using a chroot-ed filesystem over the loop-back mounted images is performance. Guests can read/write files at native filesystem speed. However, there are two drawbacks: 1) chroot() information is volatile and therefore only provides weak confinement, and 2) chroot-ed filesystems may lead to significant duplication of common files. Linux-VServer addresses both of these problems, as one of its central objectives is to support containers in a resource efficient manner that performs as well as native Linux.

### Filesystem Chroot Barrier

Because chroot() information is volatile, it is simple to escape from a chroot-ed environment, which would be bad from a security perspective when one wants to maintain the invariant that processes are confined within their containers filesystem. This invariant is nice to have when using containers for generic hosting scenarios, but clearly is required for sandboxing scenario. To appreciate how easy it is to escape conventional chroot() confinement, consider the following three simple steps: a) create or open a file and retain the file-descriptor, b) chroot into a subdirectory at equal or lower level with regards to the file, which causes the ‘root’ to be moved ‘down’ in the filesystem, and then c) use fchdir() on the file descriptor to escape from that ‘new’ root, which lets the process escape from the ‘old’ root as well, as this was lost in the last chroot() system call.

To address this problem Linux-VServer uses a special file attribute called the *chroot barrier*. The above trick does not work when this barrier is set on the root directory of a chroot-ed filesystem, as it prevents unauthorized modification and escape from the chroot confinement.

### Filesystem Unification

To appreciate the second drawback mentioned above, consider that systems with dozens or maybe even hundreds of containers based on the same Linux distribution will unnecessarily duplicate many common files. This duplication occurs simply to maintain a degree of separation between containers, as it would be difficult using conventional Linux filesystem techniques to ensure safe sharing of files between containers.

To address this problem Linux-VServer implements a disk space saving technique by using a simple unification technique applied to whole files. The basic ap-

proach is that files common to more than one container, 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 containers can safely share filesystem objects (inodes).

The only drawback with hard linking files is that without additional measures, a container could (un)intentionally destroy or modify such shared files, which in turn would harm/interfere other containers.

This can easily be addressed by adding an immutable attribute to the file, which then can be safely shared between two containers. In order to ensure that a container cannot modify such a file directly, the Linux *capability* to modify this attribute is removed from the set of capabilities given to a guest container.

However, removing or updating a file with immutable link attribute set from inside a guest container would be impossible. To remove the file the additional “permission to unlink” attribute needs to be set. With this alone an application running inside a container could manually implement a *poor man’s* CoW system by: copying the original file, making modifications to the copy, unlinking the original file, and renaming the copy the original filename.

This technique was actually used by older Linux-VServer based systems, but this caused some incompatibilities with programs that make in-place modifications to files. To address this problem, Linux-VServer introduced *CoW link breaking* which treats shared hard-linked files as copy-on-write (CoW) candidates. When a container attempts to mutate a CoW marked file, the kernel will create a private copy of the file for the container.

Such CoW marked files belonging to more than one container are called ‘unified’ and the process of finding common files and preparing them in this way is called *Filesystem Unification*. Unification is done as an out-of-band operation by a process run in the root container—typically a cron job that intelligently walks all of the containers’ filesystems looking for identical files to unify.

The principal reason for doing filesystem unification is reduced resource consumption, not simplified administration. While a typical Linux distribution install will consume about 500MB of disk space, our experience is that after unification the incremental disk space required

when creating a new container based on the same distribution is on the order of a few megabytes.

It is straightforward to see that this technique reduces required disk space, but probably more importantly it improves memory mappings for shared libraries, reduces inode caches, slab memory for kernel structures, etc. Section 3.1 quantifies these benefits using a real world example.

### 2.2.2 Process Isolation

Linux-VServer uses the global PID space across all containers. Its approach is to hide all processes outside a container's scope, and prohibits any unwanted interaction between a process inside a container and a process belonging to another container. This separation requires the extension of some existing kernel data structures in order for them to: a) become aware to which container they belong, and b) differentiate between identical UIDs used by different containers. To work around false assumptions made by some user-space tools (like `ps`) that the `init` process has to exist and have PID 1, Linux-VServer also provides a per container mapping from an arbitrary PID to a fake `init` process with PID 1.

When a Linux-VServer based system boots, by default all processes belong to the *host context*. To simplify system administration, this host context acts like a normal Linux system and doesn't expose any details about the guests, except for a few `proc` entries. However, to allow for a global process view, Linux-VServer defines a special *spectator context* that can peek at all processes at once. Both the host and spectator context are only logical containers—i.e., unlike guest containers, they are not implemented by kernel datastructures.

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

The benefit to this isolation approach for the systems process abstraction is twofold: 1) that it scales well with a large number of contexts, 2) most critical-path logic manipulating processes and PIDs remain unchanged. The drawback is that one cannot as cleanly implement container migration, checkpoint and resume, because

it may not be possible to re-instantiate processes with the same PID. To overcome this drawback, alternative container-based systems virtualize the PID space on a per container basis.

We hope to positively influence the proposed kernel mainlining of containerized PID space support such that depending on the usage scenario it is possible to choose Linux-VServer style isolation, virtualization, or a hybrid thereof.

### 2.2.3 Network Isolation

Linux-VServer does not fully virtualize the networking subsystem. Rather, it shares the networking subsystem (route tables, IP tables, etc.) between all containers, but restricts containers to bind sockets to a subset of host IPs specified either at container creation or dynamically by the host administrator. This has the drawback that it does not let containers change their route table entries or IP tables rules. However, it was a deliberate design decision, as it inherently lets Linux-VServer containers achieve native networking performance.

For Linux-VServer's *network isolation* 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 container receive or snoop traffic belonging to another container. The approach to get this right involves tagging packets with the appropriate container identifier and incorporating the appropriate filters in the networking stack to ensure only the right container can receive them. Extensive benchmarks reveal that the overhead of this approach is minimal as high-speed networking performance is indistinguishable between a native Linux system and one enhanced with Linux-VServer regardless of the number of concurrently active containers.

In contrast, the best network L2 or L3 virtualization approaches as implemented in alternative container-based systems impose significant CPU overhead when scaling the number of concurrent, high-performance containers on a system. While network virtualization is a highly flexible and nice feature, our experience is that it is not required for all usage scenarios. For this reason, we believe that our network isolation approach should be a feature that high-performance containers should be permitted to select at run time.

Again, we hope to positively influence the proposed kernel mainlining of network containerization support such that depending on the usage scenario it is possible to choose Linux-VServer style isolation, virtualization, or a hybrid thereof.

#### 2.2.4 CPU Isolation

Linux-VServer implements CPU isolation by overlaying a *token bucket* scheme on top of the standard Linux CPU scheduler. Each container has a token bucket that accumulates tokens at a specified rate; every timer tick, the container that owns the running process is charged one token. A container that runs out of tokens has its processes removed from the run-queue until its bucket accumulates a minimum amount of tokens. This token bucket scheme can be used to provide fair sharing **and/or** work-conserving CPU reservations. It can also enforce hard limits (i.e., an upper bound), as is popularly used by VPS hosting centers to limit the number of cycles a container can consume—even when the system has idle cycles available.

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

#### 2.2.5 Network QoS

The Hierarchical Token Bucket (`htb`) queuing discipline of the Linux Traffic Control facility (`tc`) [2] can be used to provide network bandwidth reservations and fair service. For containers that have their own IP addresses, the `htb` kernel support just works without modifications.

However, when containers share an IP address, as is done by PlanetLab, it is necessary to track packets in order to apply a queuing discipline to a containers flow of network traffic. This is accomplished by tagging packets sent by a container with its context id in the kernel. Then, for each container, a token bucket is created with a *reserved rate* and a *share*: the former indicates the amount of outgoing bandwidth dedicated to that container, and the latter governs how the container shares bandwidth beyond its reservation. The `htb` queuing discipline then allows each container to send packets at the reserved rate of its token bucket, and fairly distributes the excess capacity to other containers in proportion to their shares. Therefore, a container 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).

#### 2.2.6 Disk QoS

Disk I/O is managed in Linux-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 containers performing I/O to that device.

#### 2.2.7 Storage Limits

Linux-VServer provides the ability to associate limits to the amount of memory and disk storage a container can acquire. For disk storage one can specify limits on the maximum number of disk blocks and inodes a container can allocate. For memory, a variety of different limits can be set, controlling the Resident Set Size and Virtual Memory assigned to each context.

Note that fixed upper bounds on RSS are not appropriate for usage scenarios where administrators wish to overbook containers. In this case, one option is to let containers compete for memory, and use a watchdog daemon to recover from overload cases—for example by killing the container using the most physical memory. PlanetLab [3] 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 containers on a PlanetLab server, this would imply a tiny 10MB allocation for each container

on the typical PlanetLab server with 1GB of memory. Instead, PlanetLab provides basic memory isolation between containers by running a simple watchdog daemon, called `pl_mom`, which resets the container 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 Evaluation

In a prior publication [5], we compared in further detail the performance of Linux-VServer with both vanilla Linux and Xen 3.0 using `lmbench`, `iperf`, and `dd` as microbenchmarks and `kernel compile`, `dbench`, `postmark`, `osdb` as synthetic macrobenchmarks. Our results from that paper revealed that Linux-VServer has in the worst case a 4 percent overhead when compared to an unvirtualized, vanilla Linux kernel; however, in most cases Linux-VServer is nearly identical in performance and in a few lucky cases—due to gratuitous cache effects—Linux-VServer consistently outperforms vanilla kernel. Please consult our other paper [5] for these details.

This section explores the efficiency of Linux-VServer. We refer to the combination of scale and performance 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. All experiments are run on HP Proliant servers with dual core processors, 2MB caches, 4GB RAM, and 7.2k RPM SATA-100 disks.

#### 3.1 Filesystem Unification

As discussed in Section 2.2.1, Linux-VServer supports a unique filesystem unification model. The reason for doing filesystem unification is to reduce disk space consumption, but more importantly it reduces system resource consumption. We use a real world example to demonstrate this benefit from filesystem unification.

We configure guest containers with Mandriva 2007. The disk footprint of a non-unified installation is 150MB per guest. An activated guest runs a complement of daemons and services such as `syslog`, `crond`, `sshd`, `apache`, `postfix` and `postgres`—a typical configuration used in VPS hosting centers.

We evaluate the effectiveness of filesystem unification using two tests. The first consists of starting 200 separate guests one after the other measuring memory consumption. The second test is identical to the first, except before all guests are started, their filesystems are unified. For the latter test, the disk footprint of each unified guest reduces from 150MB to 10MB, resulting in 140MB of common and thus shared data on disk.

Tables 1 and 2 summarize the results for this test, categorizing how much time it takes for the guest to start up and how much memory it consumes categorized by memory type such as active, buffer, cache, slab, etc. What these columns in the two tables reveal is that the kernel inherently shares memory for shared libraries, binaries, etc. due to the unification (i.e., hard linking) of files.

Table 3 compares the rows with the 200<sup>th</sup> guest, which shows the difference and overhead percentage in consumed memory as well as the time required to start the same number of guest container. These differences are significant!

Of course, techniques exist to recoup redundant memory resources (e.g., VMware's content-based memory sharing used in its ESX product line [6]). However, such techniques require the system to actively seek out redundant memory by computing hash keys page data, etc., which introduces non-trivial overhead. In contrast, with the filesystem unification approach we inherently obtain this benefit.

As a variation of our prior experiment, we have also measured starting the 200 guest containers in parallel. The results for this experiment are shown in Figure 2. The first thing to note in the figure is that the parallel startup of 200 separate guests causes the machine to spend most of the very long startup (approx. 110min) paging in and out data (libraries, executeables, shared files), which causes the cpu to hang in `iowait` most of the time (light blue area in the cpu graphs) rendering the system almost unresponsive. In contrast, the same startup with 200 unified guests is rather fast (approx. 20min), and most of the startup time is spent on actual guest processes.

#### 3.2 Networking

We also evaluated the efficiency of network operations by comparing 2.6.20 based kernels, one unmodified kernel, and one patched with Linux-VServer 2.2. Two sets

Guest	Time	Active	Buffers	Cache	Anon	Mapped	Slab	Recl.	Unrecl.
001	0	16364	2600	20716	4748	3460	8164	2456	5708
002	7	30700	3816	42112	9052	8200	11056	3884	7172
003	13	44640	4872	62112	13364	12872	13248	5268	7980
...	...	...	...	...	...	...	...	...	...
198	1585	2093424	153400	2399560	849696	924760	414892	246572	168320
199	1593	2103368	151540	2394048	854020	929660	415300	246324	168976
200	1599	2113004	149272	2382964	858344	934336	415528	245896	169632

Table 1: Memory Consumption—Separate Guests

Guest	Time	Active	Buffers	Cache	Anon	Mapped	Slab	Recl.	Unrecl.
001	0	16576	2620	20948	4760	3444	8232	2520	5712
002	10	31368	4672	74956	9068	8140	12976	5760	7216
003	14	38888	5364	110508	13368	9696	16516	8360	8156
...	...	...	...	...	...	...	...	...	...
198	1304	1172124	88468	2492268	850452	307596	384560	232988	151572
199	1313	1178876	88896	2488476	854840	309092	385384	233064	152320
200	1322	1184368	88568	2483208	858988	310640	386256	233388	152868

Table 2: Memory Consumption—Unified Guests

Attribute	Difference	Overhead
Time	277 s	21.0 %
Active	928848 k	79.5 %
Buffers	60724 k	70.7 %
Cache	100012 k	-4.2 %
Anon	632 k	0.0 %
Mapped	623680 k	203.0 %
Slab	29340 k	7.8 %
Recl.	12572 k	5.4 %
Unrecl.	16768 k	11.4 %

Table 3: Overhead Unified vs. Separate

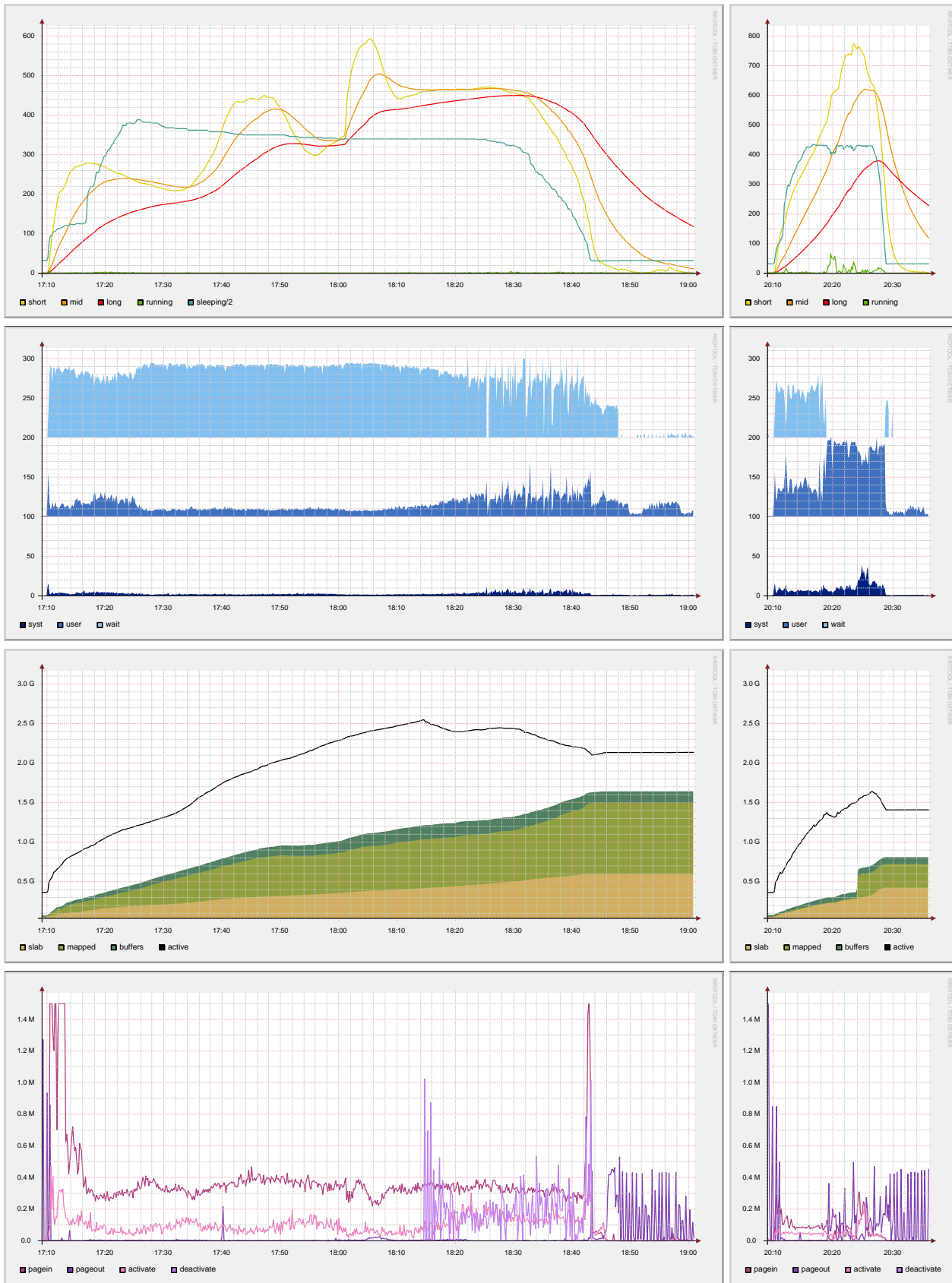


Figure 2: Parallel Startup of 200 Guests—Separate (left) vs. Unified (right)

of experiments were conducted: the first measuring the throughput of packets sent and received over the network for a CPU bound workload, and the second assessing the cost of using multiple heavily loaded Linux-VServer containers concurrently. While the former evaluates the ability of a single container to saturate a high-speed network link, the latter measures the efficiency of Linux-VServer's utilization of the underlying hardware. In both experiments, fixed-size UDP packets were exchanged between containers and a remote system on the local network. HP Proliant servers were used for both the sender and receiver, connected through a Gigabit network, equipped with 2.4Ghz Xeon 3060.

The first set of experiments demonstrated that the performance of Linux-VServer is equivalent to that of vanilla Linux. This is in line with our prior iperf-based TCP throughput results [5].

The second set of experiments involved continually increasing the number of clients confined in Linux-VServer based containers sending UDP packets as in the previous experiment. We made two observations: 1) the CPU utilization of Linux-VServer containers for packet sizes with which the network was saturated was marginally higher (77% as opposed to 72%), and 2) the increase in the CPU utilization of Linux-VServer, and the threshold beyond which it saturated the CPU was identical to that of Native Linux as containers were added.

These experiments suggest that for average workloads, the degradation of performance using the Linux-VServer network isolation approach is marginal. Furthermore, Linux-VServer can scale to multiple concurrent containers exchanging data at high rates with a performance comparable to native Linux.

### 3.3 CPU Fair Share and Reservations

To investigate both CPU 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 [4]. It is CPU-bound and involves no I/O.

Eight containers are run simultaneously. Each container 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 container is running or the virtualized system is running on behalf of another container, in a context switch for instance. The aggregate CPU time recorded by all tests is within 1% of system capacity.

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

For the second experiment we observe that the CPU scheduler for Linux-VServer achieves the requested reservation within 1%. Specifically, the container having requested  $1/4^{th}$  of overall CPU time receives 25.16% and 49.88% on UP and SMP systems, respectively. The remaining CPU time is fairly shared amongst the other seven containers.

## 4 Conclusion

Virtualization technology in general benefits a wide variety of usage scenarios. It promises such features as configuration independence, software interoperability, better overall system utilization, and resource guarantees. This paper described the Linux-VServer approach to providing these features while balancing the tension between strong isolation of co-located containers with efficient sharing of the physical resources on which the containers are hosted.

Linux-VServer maintains a small kernel footprint, but it is not yet feature complete as it lacks support for true network virtualization and container migration. These are features that ease management and draw users to hypervisors such as Xen and VMware, particularly in the server consolidation and hosting scenarios. There is an active community of developers working towards adding these features to the mainline Linux kernel, which we expect will be straightforward to integrate with Linux-VServer.

In the mean time, for managed web hosting, PlanetLab, the OLPC laptop and gateway server, embedded systems, etc., the trade-off between isolation and efficiency is of paramount importance. We believe that Linux-VServer hits a sweet spot in the containerization design

space, as it provides for strong isolation and it performs equally with native Linux kernels in most cases.

## References

- [1] Ivan Krstic. System security on the One Laptop per Child's XO laptop: the Bitfrost security platform. <http://wiki.laptop.org/go/Bitfrost>.
- [2] Linux Advanced Routing and Traffic Control. <http://lartc.org/>.
- [3] Larry Peterson, Andy Bavier, Marc E. Fiuczynski, and Steve Muir. Experiences building planetlab. In *Proceedings of the 7th USENIX Symposium on Operating System Design and Implementation (OSDI '06)*, Seattle, WA, November 2006.
- [4] John Regehr. Inferring scheduling behavior with hourglass. In *In Proceedings of the Freenix Track of the 2002 USENIX Annual Technical Conference*, June 2002.
- [5] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proc. 2nd EUROSYS*, Lisboa, Portugal, March 2007.
- [6] Carl Waldspurger. Memory resource management in vmware esx server. In *Proc. 5th OSDI*, Boston, MA, Dec 2002.

# Proceedings of the Linux Symposium

Volume Two

June 27th–30th, 2007  
Ottawa, Ontario  
Canada

## **Conference Organizers**

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium,*  
*Thin Lines Mountaineering*

C. Craig Ross, *Linux Symposium*

## **Review Committee**

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium,*  
*Thin Lines Mountaineering*

Dirk Hohndel, *Intel*

Martin Bligh, *Google*

Gerrit Huizenga, *IBM*

Dave Jones, *Red Hat, Inc.*

C. Craig Ross, *Linux Symposium*

## **Proceedings Formatting Team**

John W. Lockhart, *Red Hat, Inc.*

Gurhan Ozen, *Red Hat, Inc.*

John Feeney, *Red Hat, Inc.*

Len DiMaggio, *Red Hat, Inc.*

John Poelstra, *Red Hat, Inc.*