Reprinted from the

Proceedings of the Linux Symposium

Volume Two

July 21th–24th, 2004
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Jes Sorensen, *Wild Open Source, Inc.*
Matt Domsch, *Dell*
Gerrit Huizenga, *IBM*
Matthew Wilcox, *Hewlett-Packard*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

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

Virtually Linux

Virtualization Techniques in Linux

Chris Wright
OSDL
chrisw@osdl.org

Abstract

Virtualization provides an abstraction layer mapping a virtual resource to a real resource. Such an abstraction allows one machine to be carved into many virtual machines as well as allowing a cluster of machines to be viewed as one. Linux provides a wealth of virtualization offerings. The technologies range in the problems they solve, the models they are useful in, and their respective maturity. This paper surveys some of the current virtualization techniques available to Linux users, and it reviews ways to leverage these technologies. Virtualization can be used to provide things such as quality of service resource allocation, resource isolation for security or sandboxing, transparent resource redirection for availability and throughput, and simulation environments for testing and debugging.

1 Introduction

Virtualization has many manifestations in computer science. At the simplest level it can be viewed as a layer of abstraction which helps delegate functionality—typically handling resource utilization. This abstraction layer often helps map a *virtual* resource to a *physical* or *real* resource. The virtual resource is then presented directly to the resource consumer obscuring the existence of the real resource. This can be implemented through hardware[1] or software [16, 21, 19], may include any subset of a machine's resources, and has a wide variety of applications. Such usages include machine emulation, hardware consolidation, resource isolation, quality of service resource allocation, and transparent resource redirection. Applications of these usage models include virtual hosting, security, high availability, high throughput, testing, and ease of administration.

It is interesting to note that differing virtualization models may have inversely correlated proportions of virtual to physical resources. For example, the method of carving up a single machine into multiple machines—useful in hardware consolidation or virtual hosting—looks quite different from a single system image (SSI) [15]—useful in clustering. This paper primarily focuses on providing multiple virtual instances of a single physical resource, however, it does cover some examples of a single virtual resource mapping to multiple physical resources.

Modern processors are sufficiently powerful to provide ample resources to more than one operating environment at a time. Of course, timesharing systems have always allowed for concurrent application execution. However, there are many ways in which these concurrent applications may effect one another. Because the

[1] For example, an MMU helps with translation of virtual to physical memory addresses.

operating system provides access to shared resources such as the CPU, memory, I/O devices, file system, network, etc., one application's use of the system's resources may effect another's. This can have negative effects on both quality of service and security. Carving a single machine into a series of independent virtual machines can eliminate the quality of service and security issues.

At the same time, modern computing systems, inclusive of both hardware and software, are subject to failures and scalability problems. The application of virtualization can hide these shortcomings by distributing computing loads across a cluster of physical systems which may present a single *virtual* interface to an application.

The remainder of this paper is organized as follows. Section 2 presents a variety of virtualization techniques. Section 3 gives a detailed comparison of some of these techniques. Section 4 presents conclusions drawn from the comparisons.

2 Virtualization Techniques

The term "virtual" is one of those horribly overloaded terms in computing. For the purpose of this paper, we will define virtualization as a technique for mapping virtual resources to real resources. These virtual resources are then used by the resource consumer, fully decoupled from any real resources that may or may not exist. As discussed in Section 1 the virtual resource may be some or all of a system's resources.

There are many virtualization techniques available to Linux users, and these techniques can be leveraged through a variety of applications. The techniques reviewed in this paper fall roughly into two categories: *complete* virtualization, Section 2.1, which provides all or nearly all of a system's resources; and *partial* virtualization, Section 2.2, which provides only a specialized subset of resources. Understanding the different techniques helps identify which technique is the best given a specific set of requirements.

2.1 Complete Virtualization

Complete virtualization techniques involve creating a fully functional and isolated virtual system which can support an OS. This instance of the OS may have no indication that it is not being run natively on real hardware, and it is often referred to as the *guest*. Host-based virtual systems run atop an existing *host* OS. Others run atop a thin supervisor which just helps multiplex resources to the virtual systems. Typically the host machine is capable of supporting many concurrent virtual systems, each with its own guest OS instance. These virtual systems can be created by simple software emulation or by more complicated methods. These types of complete virtualization techniques differ in terms of efficiency and performance, portability for either the host or the guest OS, and functional goals.

The rest of this section is organized at follows. Section 2.1.1 is a look at pure software processor emulation techniques. Section 2.1.2 looks at the virtual OS approach taken by User-mode Linux. Finally, Section 2.1.3 reviews techniques using virtual machines and virtual machine monitors.

2.1.1 Processor Emulation

Processor emulation is one technique used to provide complete virtualization. In this case, the CPU is emulated entirely in software. Additionally, it is typical to find peripheral devices such as keyboard, mouse, VGA, network, timer chips, etc. supported by the emulator.

The emulation is done in user-space software, which makes it a rich environment for debugging system level software running in the emulator. Also, this technique has great advantages for portability at the cost of runtime performance. The emulator may easily run on various hardware architectures, as all emulation is done in software. Further, because these are hardware emulators, there is often little to no restriction on what OS software can be executed. However, the dynamic translation required to translate hardware instructions from the emulated processor to the native processor is pure overhead, and thus can be hundreds of times slower than native instructions [22].

An exhaustive survey of processor emulators is beyond the scope of this paper. Here we take a brief look at a few of the prevalent emulators often used to host virtual Linux instances:

- QEMU CPU emulator
- Bochs
- PearPC
- Valgrind.

QEMU [21] is a CPU emulator that does dynamic instruction translation. It maintains a translation cache for efficiency. It can be used as a user-mode emulator which will run Linux binaries compiled for the CPU that QEMU is emulating regardless of the host platform. Also, QEMU can do full system emulation, which allows one to boot an OS on the QEMU emulated CPU. While the QEMU user-mode is available for many architectures, the complete system emulation mode is only available for x86 and is in testing for PowerPC. The x86 emulator provides all the PC peripheral devices needed to boot an OS, and can easily run an unmodified Linux kernel. It also features debugger support which can be quite useful for debugging a Linux kernel.

Bochs [2] is an IA-32[2] CPU emulator. It does dynamic compilation and is often cited as being rather slow [3]. Similar to QEMU, Bochs provides full platform emulation sufficient for running an OS, and it can boot an unmodified Linux kernel. While Bochs is highly portable, it targets only the IA-32 processor.

PearPC [17] is a PowerPC CPU emulator. The generic PearPC CPU emulator can be ported and is slow. PearPC also provides a PowerPC CPU emulator that is specific to x86 hosts. This version uses dynamic instruction translation and caching techniques (similar to QEMU) which improve the speed substantially.

Valgrind [14] is worthy of mentioning as it is both a very useful tool and contains an x86-to-x86 just-in-time (JIT) compiler, thus emulating the x86 CPU. However, this tool has been historically used like Purify [10] as a memory checker, and not typically used for bringing up a virtual instance of Linux on the emulated CPU[3]. It handles user-space emulation, but not full system emulation. Valgrind is developed as an instrumentation framework around the JIT, so it can been expanded to be a general purpose “meta-tool for program supervision.” [14]

2.1.2 Virtual OS

The virtual OS is rather specific to User-mode Linux (UML) [6]. In this case, the physical machine is controlled by a normal Linux kernel. The host kernel provides hardware resources to each UML instance. The UML kernel provides virtual hardware resources to all the processes within a UML instance. The processes on a UML instance can run native code

[2] IA-32 and x86 are used interchangeably in this paper.

[3] Efforts have been made to run UML under Valgrind.

on the processor, avoiding pure emulation, and UML kernel traps all privileged needs. The UML kernel is, in fact, just an architectural port—*ARCH=um*—of the normal Linux kernel. The architecture specific code in UML is actually user-space code which uses the host Linux kernel system call interface. In other words, it is a port of the Linux kernel to the Linux kernel. This form of virtualization can be used for security[4], debugging, or virtual hosting.

2.1.3 Virtual Machine

The virtual machine (VM) has been studied for well over thirty years [8, 9]. It is a powerful abstraction that gives the illusion of running on dedicated real hardware without such physical requirements. In its early incarnations it provided a safe and convenient way to share expensive hardware resources. The well-known IBM VM/370 [5] simulated the System/390 hardware, presenting multiple independent VM's to the user. The VM/370 was aided by the System/370 hardware design, a luxury which is often not available to the modern world of low-priced, powerful commodity processors based on the x86 architecture [23]. However, it is precisely this type of environment which can benefit from consolidating multiple hardware servers to a single amply powered machine.

The typical architecture includes a physical platform which runs a virtual machine monitor (VMM). This monitor carves up the physical resources and makes them available to each virtual machine. In some cases, the VMM is *host-based* requiring a host OS, host specific drivers and user-space code to launch a VM [13, 7]. As with processor emulation in Section 2.1.1, it is beyond the scope of this paper to give an exhaustive survey of virtual machine technologies. Here we take a brief look at a few of the prevalent projects which can be used to run Linux in a virtual machine:

- Plex86
- VMware[5]
- Xen

Plex86 [18] is one project that provides an x86 virtual machine. This project provides a hosted virtual machine monitor, requiring a host OS to run the plex86 VMM. Plex86 is quite specific to Linux. The host OS may be Linux (although other host kernels are supported) and requires a kernel module to help implement the VMM. It also makes some key assumptions regarding usage of the virtual x86 hardware and patches the guest Linux kernel to conform to these assumptions. Plex86 does very little to virtualize hardware I/O. Instead, Plex86 uses a Hardware Abstraction Layer (HAL) to handle virtual I/O to the hardware devices. This eliminates the need to provide any kind of virtual devices in the VM, and being host-based eliminates the need for the VMM to understand all the possible hardware on the host. I/O which is started in the guest OS is passed through the HAL using fairly simple guest kernel drivers which issue an `int $0xff`—which must not be used for other purposes on the host OS. The host VMM traps that software interrupt and handles the request accordingly. As noted by the project's author, Plex86 is still in a prototype state, and not really ready for meaningful benchmarking yet.

VMware [7] is worthy of mention, despite the fact that it is a commercial product. VMware™ Workstation [12] provides an x86 virtual machine and is in some ways similar to Plex86. It is a hosted virtual machine monitor, however,

[4] To be secure, UML must run in `skas` mode which requires a small patch to the host kernel

[5] VMware is a commercial product.

the goals of VMware Workstation include the ability to run a complete x86 OS without making any modifications. Therefore, it makes no assumptions about the guest OS. By emulating very standard hardware such as the PS/2 keyboard and mouse, the AMD PCnet™ network interface card or the Soundblaster 16 sound card the VM provides virtual hardware devices that can be run by standard guest OS drivers. Another x86 virtual machine from VMware is the ESX Server [26]—a pure virtual machine monitor that is not host-based. This method eliminates some of the overhead involved with running atop a host OS at the cost of requiring more hardware support in the VMM itself. As with Workstation, ESX requires no modifications to the guest OS. The lower overhead of ESX makes it a contender for a data center virtual hosting environment, where it could easily run multiple VM's on a single physical system.

Xen [16] is an x86 virtual machine monitor that provides a virtual hardware interface to the virtual machine. Typically, the virtual machine provides a hardware interface which is identical to the underlying hardware. However, the Xen VM hardware abstraction is similar but not identical to the underlying x86 hardware. This allows the VMM to overcome some of the shortcomings of the x86 architecture which make it difficult to virtualize [23]. A similar method was used for the Denali [1] isolation kernel. However, unlike Denali, the Xen VM supports a notion of a virtual address space. So the guest OS and applications may share resources just like a normal OS environment. In addition, guest kernels running in a Xen VM preserve the ABI to their applications. So, while there is a need to port the guest OS kernel to the Xen VM virtual hardware abstraction, the porting effort ends there. Further, given the similarity to the x86 architecture, the effort to port to Xen results in a very small amount of new OS code—well below 2% of the OS code base [16]. This method has proven to be quite effective when considering the minimal porting effort coupled with the impressive performance benchmarks [16].

2.2 Partial Virtualization

Partial virtualization techniques create virtualized resources that are a specialized subset of a complete system's resources rather than a complete virtual machine. These methods are typically used to present a virtual interface to clients or applications when limited isolation or virtualization is sufficient. Partial virtualization can have very different applications depending on the resource which is being virtualized. These techniques vary widely in the problems they solve, and in some cases can be used with alongside of complete virtualization. The remainder of this Section reviews these techniques.

2.2.1 Linux-Vserver

The Linux-Vserver [20] project takes some of the basic ideas of isolation from a virtual machine and implements them in a single host OS. The Linux kernel is patched to allow for multiple concurrent execution contexts, often called Virtual Private Servers[6] (VPS). This method eliminates any overhead associated with running multiple operating systems, multiple VM's and the supervisor VMM. Each context can have its own file system, its own network addresses, its own set of Linux Capabilities [25], and its own set of resource limits. With this level of software isolation, it is possible to run two concurrent contexts that are unable to interact with each other directly. It may still be possible to generate some indirect QoS degradation from *crosstalk* [24], however these effects should be largely mitigated by

[6] This is also the name given to Ensim's commercial product [4].

proper setting of each context's resource limits. While this solution does require a reasonably large kernel patch (a 337K patch against Linux 2.6.6), it is a very thin virtualization layer that efficiently isolates execution contexts.

2.2.2 Linux Virtual Server

The Linux Virtual Server Project [11] takes a very different view of server virtualization from Linux-Vserver, Section 2.2.1. Rather than creating a virtual operating environment for each server, it behaves as a network load balancer. The Linux Virtual Server, also referred to as IP Virtual Server (IPVS), presents a single network address for the network service and distributes client requests transparently to a hardware cluster of network servers. With IPVS, the client can be redirected to the next available resource using a variety of algorithms such as round robin and least connected. This is an example of virtualization used to provide enhanced availability throughput, or scalability. Further, this project in contrast with Linux-Vserver helps illustrate the difficulty in defining a "Virtual Server."

2.2.3 File system and Disks

The UNIX file system provides the basic namespace that applications use to interact with significant portions of the system. The root of a file system can be relocated in Linux using `chroot()`. This may be a stretch of the definition of virtualization, but this technique does allow a single server to give different views into the system global namespace. Tools like `chroot()` or the BSD `jail()` system[7] allow multiple applications to have completely private file system namespaces, which becomes an effective tool towards system virtualization. In fact, Linux-Vserver, Section 2.2.1, makes use of `chroot()` as key to its file system isolation. Linux has native support for per process private namespaces. This gives each process its own virtual or logical view of the system's global namespace, in a more powerful, flexible and secure manner than `chroot()`. Linux-Vserver is considering moving to namespaces as a replacement for `chroot()` isolation [20]. It would not be surprising to find other virtualization systems using the same technique for file system isolation.

Another layer of virtualization can be found in the disk or block device layer of the Linux kernel. The device-mapper allows administrators to create a virtual block device which is backed by one or more physical block devices. This type of virtualization is typically used for ease of administration.

3 Comparisons

Having reviewed a variety of virtualization techniques in Section 2, it is now useful to pick a representative subset and see how they compare with one another. For the sake of comparison, this paper will focus on QEMU, Usermode Linux, Xen, and Linux-Vserver. All four of these technologies can provide a virtual execution environment comprehensive enough to run either a complete OS, or at a minimum user-space applications.

3.1 QEMU

Pros:

- Portable to numerous architectures.
- Can be used to cross platforms.
- Can run guest OS unmodified.
- Can run on unmodified host OS.

[7] An implementation of BSD jail has been ported to Linux.

- Flexible, can run a full system or just isolated user-space programs.
- Very easy to debug system software.
- Security through isolation.

Cons:

- Processor emulation is much slower than virtualization.

3.2 User-mode Linux

Pros:

- Portable to numerous architectures.
- Can run on unmodified host OS.
- Efficient enough to run multiple instances on single host in virtual hosting environment.
- Very easy to debug system software.
- Security through isolation.

Cons:

- Still slower than a virtual machine.
- The guest OS kernel is not the same as a native one.

3.3 Xen

Pros:

- True virtual machine monitor for best performance.
- The guest OS user-space applications are binary compatible.
- No host OS, very clean virtual machine separation.
- Security through isolation.
- Ideal for virtual hosting environment, can scale up to 100 virtual machines.

Cons:

- The guest OS kernel must be ported to Xen virtual hardware architecture.

3.4 Linux-Vserver

Pros:

- Highly efficient way to isolate resources.
- Can conserve on disk and memory by sharing basic resources like shared libraries.
- Security through context separation.

Cons:

- Only one kernel instance, so quality of service may be hard to guarantee.

4 Conclusions

Virtualization is an old yet resurging technology. Virtual machine research is alive and well, and Linux provides a great testbed for new virtualization technologies. With a wealth of choices, Linux users are sure to find a virtualization technique that suits their requirements. From running as a guest OS on a virtual machine, to providing thin isolation environments for applications, to single system image clusters, Linux is thriving in this virtual reality.

References

[1] M. Shaw A. Whitaker and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[2] Bochs IA-32 Emulator Project. `http://bochs.sourceforge.net/`.

[3] Bochs FAQ. `http://bochs.sourceforge.net/doc/docbook/user/faq.html#AEN273`.

[4] Ensim Corporation. Ensim Virtual Private Server, June 2004. `http://www.ensim.com/products/privateservers/index.html`.

[5] R. J. Creasy. The Origin of the VM/370 Time-Sharing System. *IBM Journal of Research and Development*, 25(5):483–490, September 1981.

[6] Jeff Dike et al. User-Mode Linux, July 2000. `http://user-mode-linux.sourceforge.net/`.

[7] Mendel Rosenblum et al. VMWare. `http://www.vmware.com/`, February 1998.

[8] R. Goldberg. *Architectural Principles for Virtual Computer Systems*. PhD thesis, Harvard University, 1972.

[9] R. P. Goldberg. Architecture of Virtual Machines. In *Proceedings of the Workshop on Virtual Computer Systems*, pages 74–112, 1973.

[10] Reed Hastings and Bob Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Winter USENIX Conference*, 1992. Also available at `http://www.rational.com/support/techpapers/fast_detection/`.

[11] The Linux Virtual Server Project. `http://www.linuxvirtualserver.org/`.

[12] Ganesh Venkitachalam Jeremy Sugerman and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *USENIX Annual Technical Conference*, June 2001.

[13] Mac-on-Linux. `http://www.maconlinux.org/`.

[14] Julian Seward Nicholas Nethercote. Valgrind: A Program Supervision Framework. In *Electronic Notes in Theoretical Computer Science 89 No. 2*, 2003.

[15] Single System Image Clusters (SSI) for Linux. `http://openssi.org`.

[16] K. Fraser S. Hand T. Harris A. Ho R. Neugebauer I. Pratt P. Barham, B. Dragovic and A. Warfield. Xen and the Art of Virtualization. In *Symposium on Operating Systems Principles (SOSP)*, 2003.

[17] PearPC - PowerPC Architecture Emulator. `http://pearpc.sourceforge.net/`.

[18] Plex86 x86 Virtual Machine Project. `http://plex86.sourceforge.net/`.

[19] Herbert Poetzl. Linux-VServer Project, October 2001. `http://www.linux-vserver.org`.

[20] Herbert Poetzl. Linux-VServer Technology. In *LinuxTAG 2004*, June 2004.

[21] QEMU CPU Emulator. `http://fabrice.bellard.free.fr/qemu/`.

[22] QEMU Benchmarks. `http://fabrice.bellard.free.fr/qemu/benchmarks.html`.

[23] J. S. Robin and C. E. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *9th USENIX Security Symposium*, pages 129–144, August 2000.

[24] D. L. Tennenhouse. Layered multiplexing considered harmful. In

Rudin and Williamson, editors, *Protocols for High Speed Networks*. May 1989.

[25] Winfried Trumper. Summary about POSIX.1e. `http://wt.xpilot.org/publications/posix.1e`, July 1999.

[26] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.