Virtualizing Intelligent River® : A Comparative Study of Alternative Virtualization Technologies

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Masters of Science
Computer Science

by
Aravindh Sampath Kumar
December 2013

Accepted by:
Dr. Jason O. Hallstrom, Committee Chair
Dr. Amy Apon
Dr. Brian A. Malloy

Abstract

Emerging cloud computing infrastructure models facilitate a modern and efficient way of utilizing virtualized resources, enabling applications to scale with varying demands over time. The core idea behind the cloud computing paradigm is *virtualization.* The concept of virtualization is not new; it has garnered significant research attention, in a race to achieve the lowest overhead compared to bare-metal systems. The evolution has led to three primary virtualization approaches- *full-virtualization*, *para-virtualization*, and *container-based virtualization*, each with a unique set of strengths and weaknesses. Thus it becomes important to study and evaluate their quantitative and qualitative differences. The operational requirements of the Intelligent River® middleware system motivated us to compare the choices beyond the standard benchmarks to bring out the unique benefits and limitations of the virtualization approaches.

This thesis evaluates representative implementations of each approach: (i) full-virtualization - *KVM*, (ii) para-virtualization - *Xen*, and (iii) container-based virtualization - *LXC*. First, this thesis describes the scalable and resilient design of the Intelligent River® middleware system used as a reference application to evaluate the virtualization platforms. Second, this thesis describes the deployment of the application components on each of the test environments. Third, this thesis assesses the benefits and limitations of each based on their virtualization overhead, resource entitlement and isolation facilities, operational flexibility, scalability, and security.

The study presented in this thesis provides an improved understanding of available virtualization technologies. The results will be useful to architects in leveraging the best virtualization platform for a given application.

Chapter 1

Introduction

The adoption of cloud computing paradigm has changed the way we look at server infrastructure for next-generation applications. The cloud computing model has catalyzed a change from the traditional model of deploying applications on dedicated servers with limited room to scale to a new model of deploying applications on a shared pool of computing resources with (theoretically) unlimited scalability [22].

The technology backbone behind the idea of cloud computing is virtualization. Virtualization is the process of creating virtual devices that simulate the hardware which are in turn mapped to the physical resources on an underlying server. Virtualization primarily addresses the problem of under-utilization of computing resources in the dedicated server model. Virtualization maximizes the utilization of the hardware investment by running an increased number of isolated applications simultaneously on a physical server or "*host*". The isolated applications are run in an operating environment referred as *Virtual Machines (VM)* [51] or *Containers* [35]. The VMs / containers abstract the hardware interfaces required by the applications they run and utilize as much computational resources as they need (or are available). Virtualization opens the door for several added benefits:

Improved Fault-tolerance :- Virtual machines can be statefully migrated to other physical machines in the event of a hardware failure [11]. The fail-over can also be automatically triggered in some cluster-aware virtualization platforms [47]. This facility effectively enables continuous availability for critical applications, which would require application-level awareness to achieve in the dedicated server model.

Operational Flexibility :- Virtual Machines and their associated virtual devices are usually represented as a few files, which make them easy to clone, snapshot, and migrate to other physical machines. Also, individual VMs can be dynamically started or stopped without causing any outage to other VMs or the host.

New Avenues for Tuning and Customization :- Since virtualization platforms introduce an additional layer of control between applications and hardware, they enable more options to customize the environment for individual VMs. That also provide more points of control to administer the resources accessible to individual virtual machines.

Granular Monitoring :- The physical server that hosts VMs provides deeper insights and visibility into the performance, capacity, utilization and the overall health of the individual VMs. Analysis of the monitoring data facilitates resource management and control.

The benefits offered by the virtualization platforms form the core of the cloud computing ecosystem. Commercial cloud infrastructure providers have built their solutions around these benefits and offer "pay as you go" services where end-users are billed only for the resources used by the virtual machines or containers. Cloud providers pass on isolation and granular control options for the virtualization platform to end-users with a flexible interface, letting users keep complete control of their operating systems, storage environment, and networking setup without worrying about the underlying physical infrastructure. Large scale cloud computing platforms that lease their servers to virtualize applications of several end-users over a large pool of shared resources are exemplified by Amazon Elastic Compute Cloud (EC2) [cite], RackSpace [cite], and Heroku [cite].They differ by their choice of the virtualization platform. Enterprises and academic institutions also run a private cloud platform and have driven the development of open source cloud computing toolkits like OpenStack [cite], CloudStack [cite] and OpenShift [cite].

1.1 Motivation

The Intelligent River® is a large scale environmental monitoring platform that is being built by researchers at Clemson University [cite]. The Intelligent River® involves a large and distributed sensor network in the Savannah River basin and a real-time distributed middleware system hosted in Clemson University that receive, analyze and visualize the real-time environmental observation streams. The volume and unpredictable nature of the observation streams along with the increasing scale of sensor deployments demanded a distributed middleware system that is flexible, fault-tolerant and designed for scale. Architecting he Intelligent River® middleware system posed an important design question,

Given the multitude of open source virtualization platforms, KVM, Xen, and Linux Containers, and the complex operational requirements of our middleware stack, which platform to choose ?

Our quest to find the answer to the question needed a detailed analysis of the common virtualization platforms beyond the available results of the standard benchmarks [cite]. Prior research work by Andew J.Younge et al. [cite] on analysis of the virtualization platforms for HPC applications shows that KVM performed better compared to Xen on the standard HPCC benchmarks whereas another research work published by Jianhua Che et al. [cite] using the standard benchmarks claims that OpenVZ, a virtualization platform based on Linux containers performed the best while KVM performed *significantly lower* than Xen which made clear that the comparison using the standard benchmarks were not enough. This motivated us to perform a detailed study of the principles behind the three major open source virtualization platforms, KVM, Xen, and Linux Containers and evaluate them based not just on the quantitative metrics like virtualization overhead and scalability but also on the qualitative metrics like operational flexibility, isolation, resource management, operational flexibility, and security.

1.2 Contributions

This research thesis builds on the prior work on comparing the open source virtualization platforms and brings out the advantages and disadvantages of each.The contributions of this thesis include (i) Description of the scalable and resilient design of our Intelligent River® middleware system. (ii) A study on the principles of operations behind the three chosen virtualization platforms which will be useful to the architects who design their applications around them. (iii) A quantitative comparison of the virtualization overhead (and scalability ?) exhibited by KVM, Xen and Linux containers as of date of writing. (iv) A discussion on the differences among the chosen platforms in terms of qualitative factors like ease of deployment, resiliency and security. (v) A discussion on the facilities to assure the resource entitlement and control with respect to CPU,Network bandwidth, Memory and I/O which is often overlooked and an area in need of improvement from the operational perspective.

1.3 Thesis Organization

Chapter 2

Related Work

The core ideas of virtualization having stood the test of time have also prompted a variety of comparative studies by the academia as well as the industries. Andrew J. Younge et al. [58] compared Xen and KVM with virtual resources from the FutureGrid project [57] and claims, KVM performed significantly better than Xen, under the standard HPCC and SPEC benchmarks. Another benchmark oriented synthetic study [10] claims OpenVZ (the predecessor of Linux Containers) performed exceptionally well on almost all benchmarks. But, also claims that Xen performs significantly better than KVM contradicting the results of other publishers. A recent study by Miguel G. Xavier et al. [56] and an earlier study [52], again corroborates the fact that LXC performs significantly better than alternative virtualization platforms, Linux-VServer, OpenVZ, and Xen based on a variety of benchmarks including LINPACK, STREM, IOZONE, NETPIPE, and NPB. They also highlighted the shortcomings in isolation, and sharing in LXC which have already been addressed in the recent releases. A relatively old research work by Vincent Neri et al. [46] threw light on serious performance limitations on Xen under certain operational circumstances and also motivated the need for microbenchmarks.

Among the many new opportunities that were opened up by the relatively new and lightweight LXC, network virtualization or Software Defined Networking (SDN) attracted lots of research attention. Studies, [13], and [50] discussed the performance benefits of large scale virtual network emulation by implementing open Vswitch [54] on LXC based platforms. Mesos [17] , A clustered platform that provides fine-grained resource sharing among multiple frameworks like Hadoop [16] and MPI [12] utilizes LXC to isolate and limit the CPU, Memory, Network, and I/O resources. Also, Mircea Bardac et al. [3] showcased the scalability of LXC by deploying a solution for testing large scale Peer-to-Peer systems.

Resource affinity, Isolation and Control are discussed in detail in this thesis. Vishal Ahuja et al. [1] provides an example of exploiting CPU isolation facilities to improve overall network throughput by pinning application, and protocol processing to the same processor core. The discussions in this thesis on providing resource affinity will enable Vishal Ahuja et al.'s work to be applicable under LXC. Another interesting attempt by Zhaoling Guo and Qinfen Hao [18] made use of cgroups (the isolation mechanism used by LXC) in combination with KVM to enable CPU affinity to achieve improved performance.

!!!!!!!! Write About Docker !!!!!!!!!

CoreOS [53], in its very early stages and being actively developed at the time of writing of this thesis is a very light weight operating system (Linux Kernel+systemd) that is built to run in containerized environments attempts to lower the overhead by eliminating redundant operating system functions.

Chapter 3

Virtualization Technologies

The accelerated adoption of Linux in the cloud computing ecosystem has spurred the demand for dynamic, efficient, and flexible ways to virtualize next generation workloads. Not surprisingly, there is no single best solution to this problem. The linux community supports multiple mature virtualization platforms, leaving the choice to the end-users. This chapter provides a technical over the principles behind the operation of three representative virtualization solutions: Kernel-based Virtual Machine(KVM), Xen, and Linux Containers.

3.1 Kernel based Virtual Machines (KVM)

KVM is representative of a category of virtualization solutions known as *full-virtualization*. A full-virtualization solution, as shown in Figure 3.1, is one where a set of virtual devices are emulated over a set of physical devices with a *hypervisor* to arbitrate access from the *virtual machines*(sometimes referred to as *guests*.

A hypervisor is a critical part of a stable operating environment as it is responsible for managing the memory available to the guests, scheduling the processes, managing the network connections to and from the guests, manages the input/output facilities, and maintaining security. The KVM solution, being a relatively new entrant into the virtualization scene, chose to build upon existing utilities and features by leveraging the mature, time-proven Linux kernel to perform the role of the hypervisor.

In the KVM based approach to virtualization, majority of the work is offloaded to the Linux

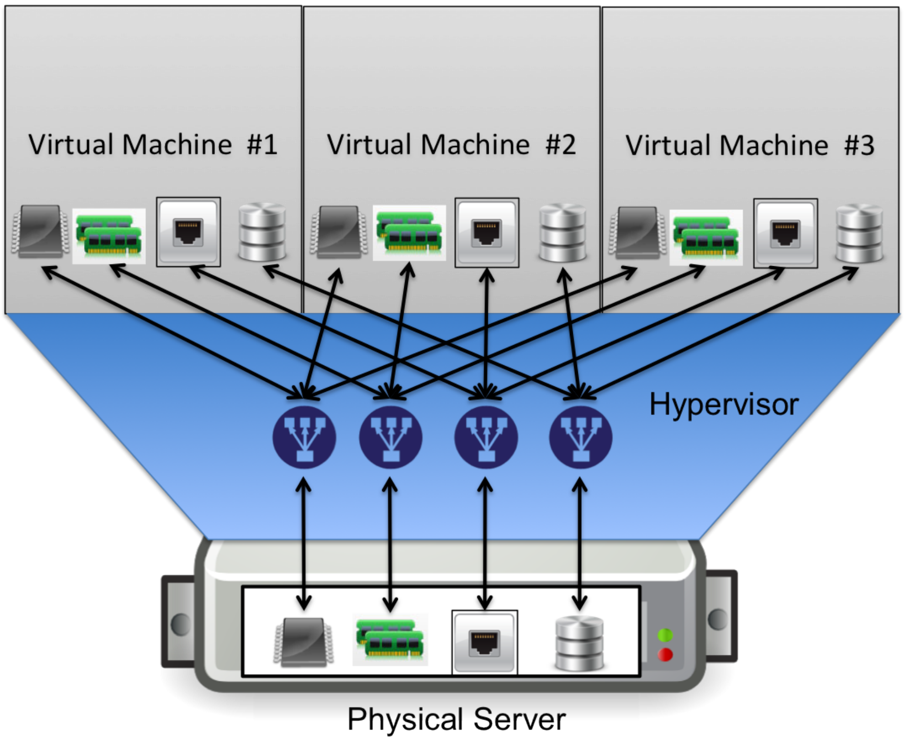


Figure 3.1: Block Diagram of a full-virtualization system

kernel, which exposes a robust, standard and secure interface to run isolated virtual machines. The virtualization facilities enabled by KVM were merged into the mainstream linux kernel since version 2.6.20 (released February 2007) [36]. KVM itself is only part of the virtualization solution. It turns the Linux kernel into a Virtual Machine Monitor (VMM) (i.e, hypervisor), which enables several virtual machines to operate simultaneously, as if they are running on their own hardware. The emulated virtual devices and the virtual machine itself are created by an independent tool known as QEMU [39]. Hence the total solution is commonly referred as QEMU-KVM. KVM is packaged as a lightweight kernel module which implements the virtual machines as regular Linux processes, and therefore leverages the linux kernel on the host for all the scheduling and device management activities. Figure 3.2 shows the architecture of a server virtualized using QEMU-KVM.

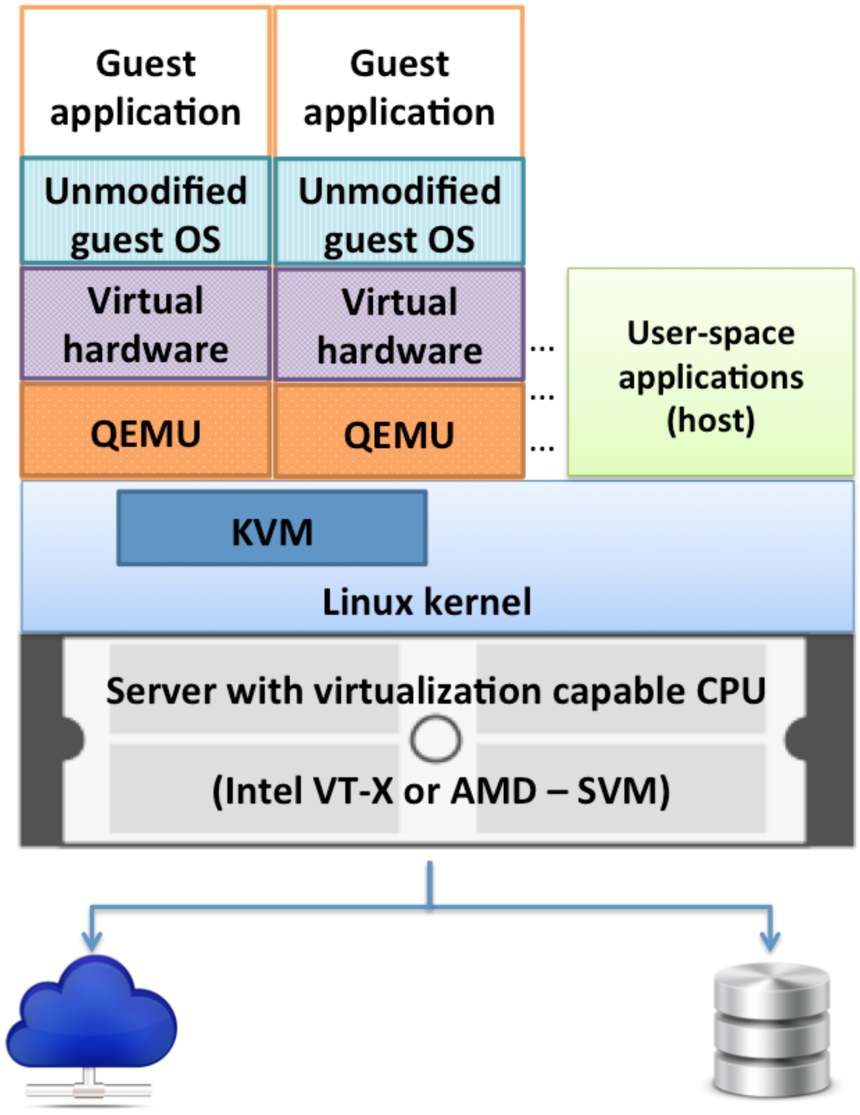


Figure 3.2: KVM - Architecture

In practice, KVM and the Linux kernel treat the virtual machines as regular Linux processes on the host and perform the mapping of virtual devices to real devices in each of the following categories.

1. **CPU**:

   KVM requires the CPU to be virtualization aware. Intel VT-X [23] and AMD-SVM [2] are the virtualization extensions provided by Intel and AMD, respectively, which are the most common CPUs used in the x86 servers. KVM relies on these facilities to isolate the instructions

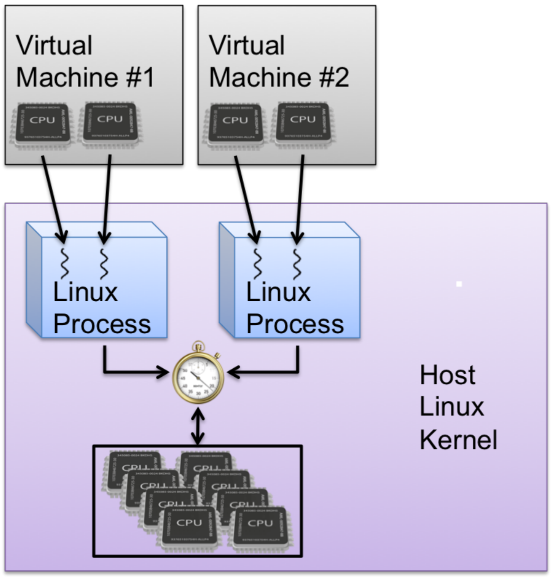


Figure 3.3: KVM - Virtual CPU management

generated by the guest operating systems from those generated by the host itself. Every virtual CPU associated with a virtual machine is created as a thread belonging to the virtual machine's process on the host as shown in Figure 3.3. Hence, enabling multiple virtual CPUs improve the virtual machine's performance by utilizing the multi-threading facilities on the host. The virtual machine's CPU requests are scheduled by the host kernel using the regular CPU scheduling policies. Improving CPU performance on the guest and ensuring fair CPU entitlement are discussed in later sections.

2. **Memory**:

   KVM inherits the memory management facilities of the Linux kernel. The memory of a virtual machine is an abstraction over the virtual memory of the standard Linux process on the host. This memory can be swapped, shared with other processes, merged, and otherwise managed by the Linux kernel. Thus, the total memory associated with all the virtual machines on a host can be greater than the physical memory available on the host. This feature is known as *memory over-commitment.* Though memory over-commitment increases overall memory utilization, it creates performance problems when all the virtual machines try to utilize their memory share at the same time, leading to swapping on the host. Since KVM offloads memory management to the Linux kernel, it enjoys the support of NUMA (Non-Uniform Memory

Access) awareness [37], and Huge Pages [34] to optimize the memory allocated to the virtual machines. NUMA support and Huge Pages will be discussed in the later sections.

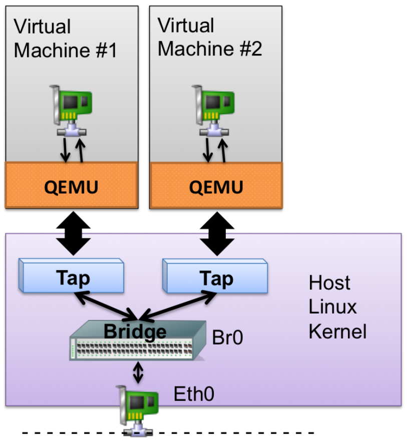
3. **Network Interfaces**:

   Networking in a virtualized infrastructure enables an additional layer of convenience and control over conventional networking practices. Virtual machines can be networked to the host, networked to other co-located virtual machines, or even participate in the same network segment as the host. Several configurations are possible, trading-off device compatibility and performance. Figure 3.4 shows the most common virtual networking options used in a KVM based infrastructure. Figure 3.4a shows a virtual networking configuration where unmodified guest operating systems use their native drivers to interact with their virtual network devices. Virtual network devices are connected to the Tap devices [7] on the host kernel. Tap interfaces are software-only interfaces existing only in the kernel; they relay ethernet frames to and from the Linux bridge. This setup trades performance to achieve superior device compatibility.

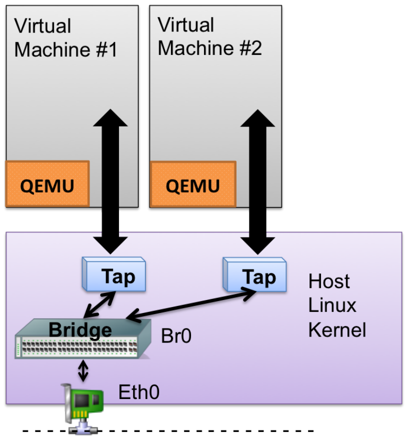
   Figure 3.4b shows a networking configuration where the guest operating system running in the virtual machine is "virtualization aware" and cooperates with the host by bypassing the device emulation by QEMU. This is known as *para-virtualized networking.* Para-virtualized networking trades compatibility to achieve near-native performance.

   Figure 3.4c shows a combination of both public and private networking. Two points are important to note. First, a virtual machine may have multiple virtual networking devices. Second, virtual machines can be privately networked using a Linux bridge that is not backed by a physical ethernet device. This setup greatly increases the inter-virtual machine communication performance as the packets need not leave the server to pass through real networking hardware. This is an ideal networking configuration for a publicly accessible virtual machine to communicate with secure private virtual machines. For example, a publicly accessible application server could then communicate with privately networked database server.

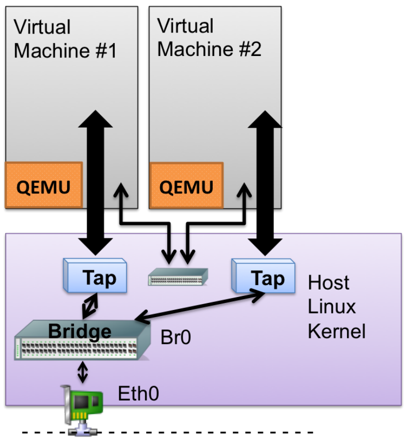
   To support the virtualization of network intensive applications, physical network interfaces attached to the host can be directly assigned to the virtual machine, bypassing the host kernel and QEMU, as shown in Figure 3.4d. This setup provides "bare-metal" performance by providing direct access to the hardware, and is applicable when individual hardware devices are available to be dedicated to the virtual machines. The key to achieving this direct device



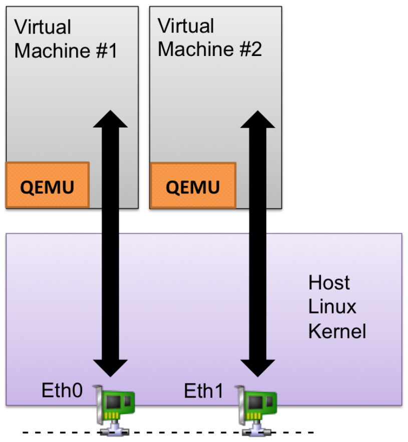
(a) Emulated Networking devices



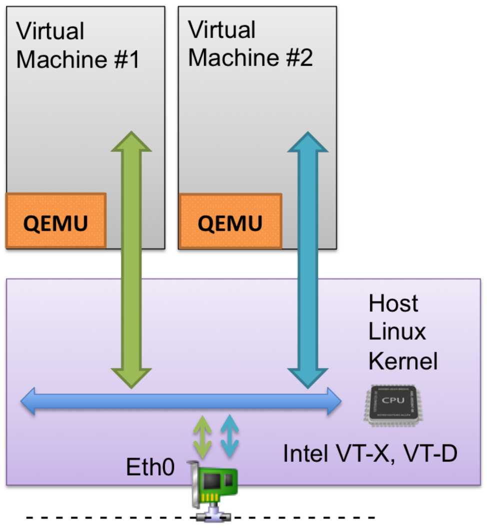
(b) Para-Virtualized Networking



(c) Public and Private Networking



(d) Direct Device Assignment



(e) Direct Device Assignment with SR-IOV

Figure 3.4: Common Networking Options for KVM

access is the hardware assistance provided through Intel VT-D [25], AMD-IOMMU [5]. Intel VT-D and AMD-IOMMU enable secure PCI-pass through to allow the guest operating system in the virtual machine to control the hardware on the host.

Figure 3.4e shows a networking setup that utilizes ethernet hardware capable of Intel Single Root Input/Output Virtualization (SR-IOV) [24]. SR-IOV takes the above configuration one step further by letting a single hardware device be accessed directly by multiple virtual machines, with their own configuration space and interrupts. This enables a capable network card to appear as several virtual devices, capable of being directly accessed by multiple virtual machines.

Advanced Networking: Virtual LANs (VLANs) provide the capability to create logically separate networks that share the same physical medium. In a virtualized environment, VLANs created for the virtual machines may share the physical network interfaces or share the bridged interface.

3.2 Linux Containers

Linux Containers are a lightweight operating system virtualization technology, and the newest entrant into the linux virtualization arena. There is an interesting perspective popularized within the Linux community that hypervisors originated due to the Linux kernel's incompetence to provide superior resource isolation and effective scalability [14]. Containers are the proposed solution. Digging deeper, hypervisors were created to isolate workloads and create virtual operating environments, with individual kernels optimally configured in accordance with workload requirements. But, the key question to be answered is, *Whether it is the responsibility of an operating system to flexibly isolate its own workloads.* If the linux kernel could solve this problem without the overhead and complexity of running several individual kernels, there would not be a need for hypervisors!

The linux community saw a partial solution to the problem in BSD Jails [48], Solaris Zones [41], Chroot [32], and most importantly OpenVZ [42] - a fork of the linux kernel by Parallels. BSD Jails were designed with an objective of restricting the visibility of the host's resources to a process. For example, when a process runs inside a jail, its root directory is changed to a sub-directory on the host thereby limiting the extent of the file system the process can access. Each process in a jail is provided its own directory sub tree, an IP address defined as an alias to the interface on the host, and optionally a hostname that resolves to the jail's own IP address. The linux kernel's approach to solve the problem was *"Containers"* incorporating the niceties from all the mentioned inspirations and more.

The core idea is to isolate only a set of processes and their resources in *"Containers"* without involving any device emulation or creating any dependency on the host hardware. Like virtual machines, several containers can run simultaneously on a single host, but all of them share the host kernel for their operation. Isolated containers run directly on the bare-metal hardware using the device drivers native to the host kernel without any intermediate relays.

Linux Containers expanded the scope of BSD jails, and provides a granular operating system virtualization platform to the extent where, containers can be isolated from each other, running their own operating system yet sharing the kernel with the host. Containers are provided their own independent file system and network stack. Every container can run its own linux distribution of linux that is different from the host. For example, a host server running RedHat Enterprise Linux [21] may run containers that run Debian [15], Ubuntu [8], CentOS [9] etc. or even another copy of RedHat Enterprise Linux. This level of abstraction in the containers creates an illusion of running virtual machines as discussed earlier with KVM.

Figure 3.5 shows the architecture of a server that uses linux containers for virtualization. Unlike KVM, linux containers does not require any assistance from the hardware therefore runs on any hardware that is capable of running the mainline linux kernel. The linux kernel facilitates the execution of containers by utilizing *namespaces* for resource isolation and *cgroups* for resource management and control.

The following description of namespaces in the linux kernel is based on [28], [29], [30], [26], [31], [27] by Michael Kerrisk.

Namespaces wrap a particular global system resource on the host and makes it appear to the processes within the namespace(containers) that they have their own isolated instance of the global resource.

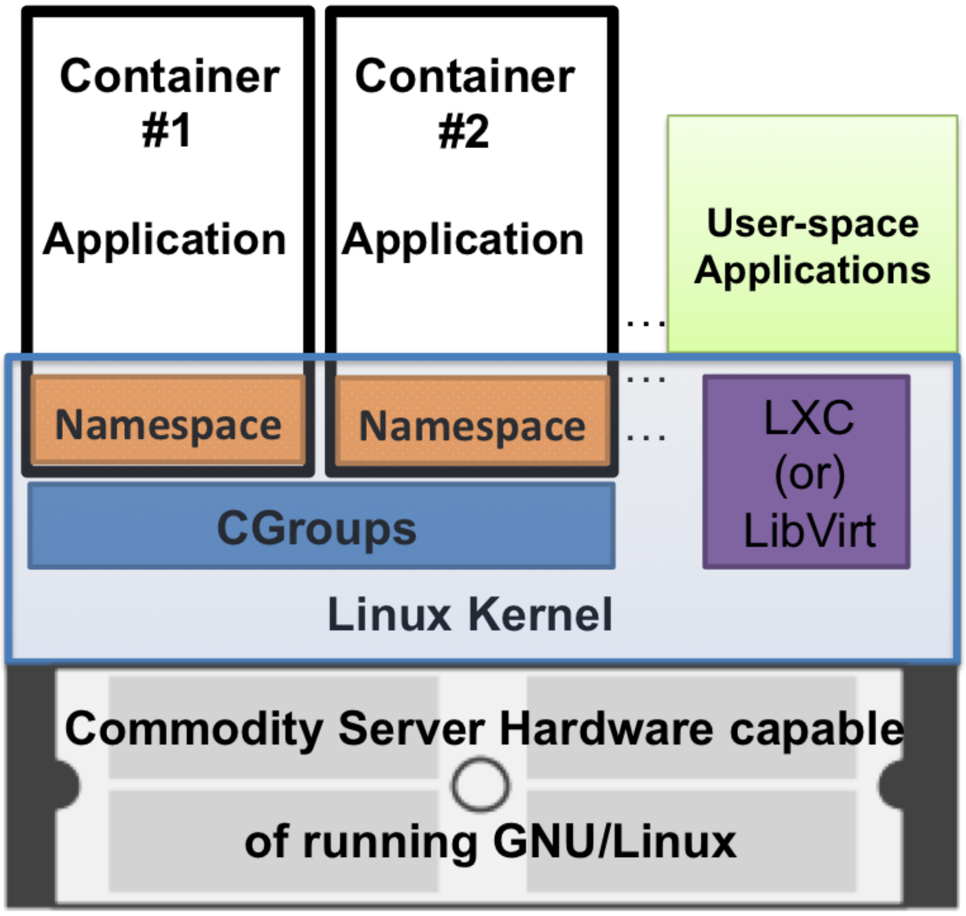


Figure 3.5: Linux Containers - Architecture

Linux implements namespaces in 6 categories :-

- **Mount namespaces :**

  Mount namespaces enable isolated file system trees to be associated with specific containers(or groups of regular linux processes). A container can create its own file system setup and the subsequent *mount()* and *unmount()* system calls issued by the process would affect only its mount namespace instead of the whole system. For example, multiple containers on the same host can issue *mount()* calls to create a mount point "/data" and access them at "/data" simultaneously. They will reside at a different location on the filesystem tree which will be appropriately seen by the host, say "/<containername1>/data", "/<containername2>/data" and so on. Ofcourse, the same setup can be achieved by using the *chroot()* command. But, chroot can be escaped with certain capabilities including CAP_SYS_CHROOT [cite]. Mount namespaces provides a secure alternative. Mount namespaces greatly improves the portability of the containers as they can retain their filesystem trees irrespective of the host's environment.

- **UTS namespaces :**

  The implementation of UTS namespaces facilitates the containers to issue *sethostname()* and *setdomainname()* system calls to set their own hostname and NIS domain name respectively. *uname()* call issued by the container returns the appropriate hostname and domain name.

- **IPC namespaces :**

  The implementation of IPC namespaces isolates the System V IPC objects [40] and POSIX Messages queues [38] associated with individual containers.

- **PID namespaces :**

  The implementation of PID namespaces in the linux kernel facilitates the isolation of Process IDentification numbers(PID) of the processes running on the host and the processes that are run inside the containers. Every container can have its own init process(PID 1). Several containers running simultaneously can have processes with same PIDs. The linux kernel implements the PIDs as a hierarchy, therefore every process on the host consists of two PIDs (one in the container's namespace and other outside the namespace). This PID abstraction does good in two perspectives, first, isolates the containers such that a process running on one container does not have visibility of processes running on other containers. Second, enables the migration of containers across hosts as the containers can retain the same PIDs.

- **Network namespaces :**

  The implementation of the network namespaces provides the most useful isolation of network resources for the containers. In other words, it enables the containers to have their own (virtual)network devices, IP addresses, port number space, IP routing tables etc. For example, a single host can run 'n' number of containers each running a web server in its own IP address transparently serving data over port 80.

- **User namespaces :**

  The implementation of the user namespaces provides the isolation for user and group ID number spaces for the processes running on the directly host and the processes running inside the containers. In other words, a process will have two user and group IDs (one inside the container's namespace and other outside the namespace). This enables an user to possess an UID of 0 (root privileges) inside the container while still being treated as an unprivileged user on the host. The same applies to the application processes that run inside the containers. This abstraction of user privileges greatly improves the security of the container based virtualization solution. It is to be noted that this feature is only available in linux kernel versions 3.8+.

Four more namespaces are being developed for *future* inclusion into the linux kernel [6]:-

- **Security namespace :** Aims to provide isolation of security policies and checks among different containers.

- **Security keys namespace :** Aims to provide an independent security key space for the containers. In other words, isolate the /proc/keys and /proc/key-users based on namespace of the container [19].

- **Device namespace :** Aims to provide the containers their own device namespace. In other words, enable the containers to create/access devices with their own major, minor numbers so they can be seamlessly migrated to any host.

- **Time namespace :** Aims to enable the containers to freeze/modify the thread and process clocks which would greatly help when the container migrates to another host and continues execution.

Control groups (cgroups) [33] [49] [20] is another key feature of the linux kernel that is used for resource management of the containers. Cgroups instruments the kernel to limit, prioritize, account and isolate resource usage of several process groups. By proper usage of cgroups, hardware and software resources on the host can be smartly divided up and shared among several containers. Though cgroups are generic and applies to any individual or group of processes, the rest of this section discusses their relavance to containers as the group of processes.

The implementation of cgroups categorizes the manageable system resources into the following subsystems :

1. **blkio** : Allows to set limits on input/output access to and from block devices such as physical drives.
2. **cpu** : Allows to set limits on the access to the CPU for the tasks (Containers).
3. **cpuacct** : Reporting the usage of CPU resources by tasks/containers in a cgroup.
4. **cpuset** : Enables assignment of individual CPUs (on a multicore system) and memory nodes to tasks/containers in a cgroup.
5. **devices** : Allows or denies access to devices by tasks/containers in a cgroup.
6. **freezer** : This subsystem suspends or resumes tasks in a cgroup.
7. **memory** : Allows setting limits on memory use by tasks/containers in a cgroup, and generates automatic reports on memory resources used by them.
8. **net_cls** : This subsystem tags network packets with a class identifier (classid) that allows the Linux traffic controller (tc) to identify packets originating from a particular cgroup task.
9. **net_prio** : Enables prioritizing the network traffic to and from the tasks/containers on a per network interface basis.
10. **ns** : The namespace subsystem.

Cgroups are organized hierarchically. There may be several hierarchies of cgroups in the whole system each associated with atleast one subsystem. Process groups (In our case, containers) are assigned to cgroups in different hierarchies to be managed by different subsystems. All the processes in the linux system are a member of the root cgroup by default and are associated with custom cgroups on a need basis.

The configuration of cgroups are governed by a basic set of rules :-

- A single hierarchy can have one or more subsystems attached to it.
- A subsystem can have several hierarchies that are not already associated with any other subsystems.
- A container can be assigned to several cgroups but, cannot belong to more than one cgroup in the same hierarchy.
- The future children of all the processes within a container will inherit the same cgroup associations from their parent process. However their cgroup associations may be changed independantly while in execution.

Adhering to the above mentioned rules makes sure that, there is exactly one way a container is controlled by a single subsystem.

Figure 3.6 shows an example to illustrate the configuration of cgroups. Containers running web servers are assigned the cgroup that specifies appropriate limits on the resources(CPU, network and memory) it requires. The cgroups are in turn assigned to individual hierarchies which are with the appropriate subsystems (type of resource that is to be managed). Containers running as database servers are limited/managed only for CPU and memory, while using network resources like any other process in the system.

Chapter ¡??¿ describes cgroups in more detail as part of the resource entitlement discussion.

3.3 Xen

Xen [45]is an open source virtualization platform that provides a bare-metal hypervisor (special firmware that runs directly on the hardware). The xen project based its core principles on para-virtualization [55] [59], a technique where the guest operating systems are modified to run on the host using an interface that is easier to virtualize. Paravirtualization significantly reduced the overhead and improved performance according to [55], [4], [43].

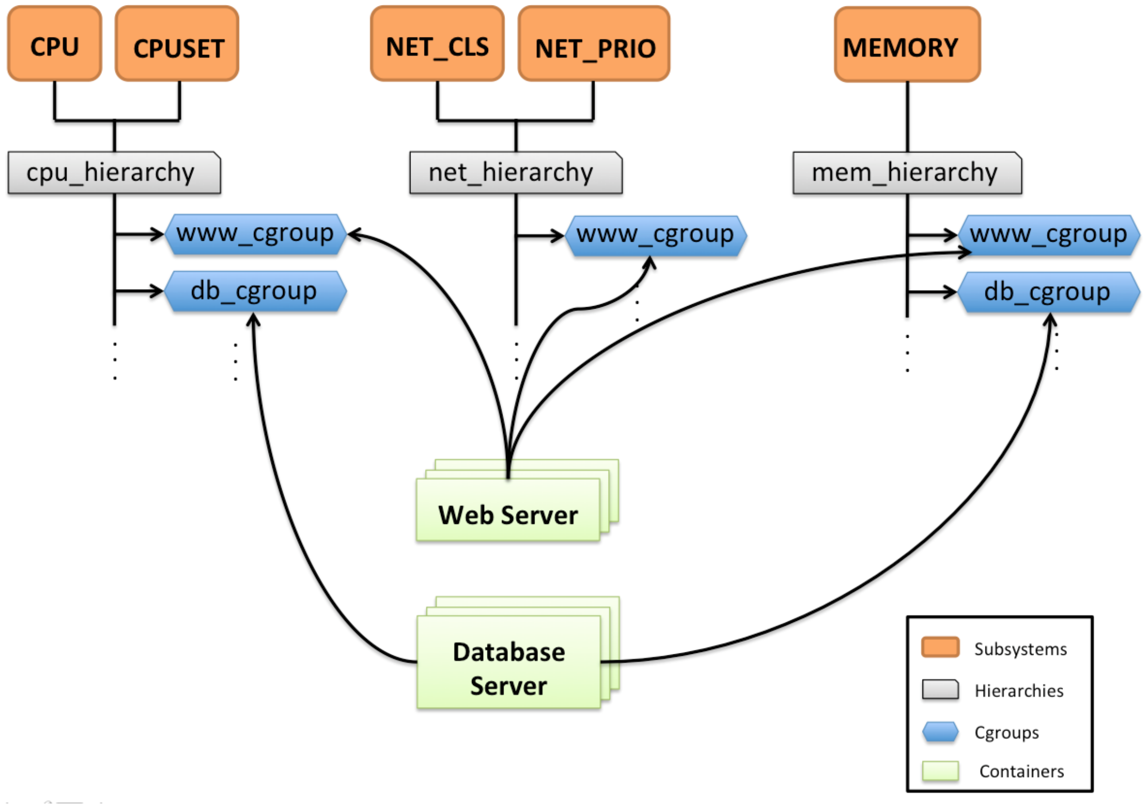


Figure 3.6: Cgroups configuration example

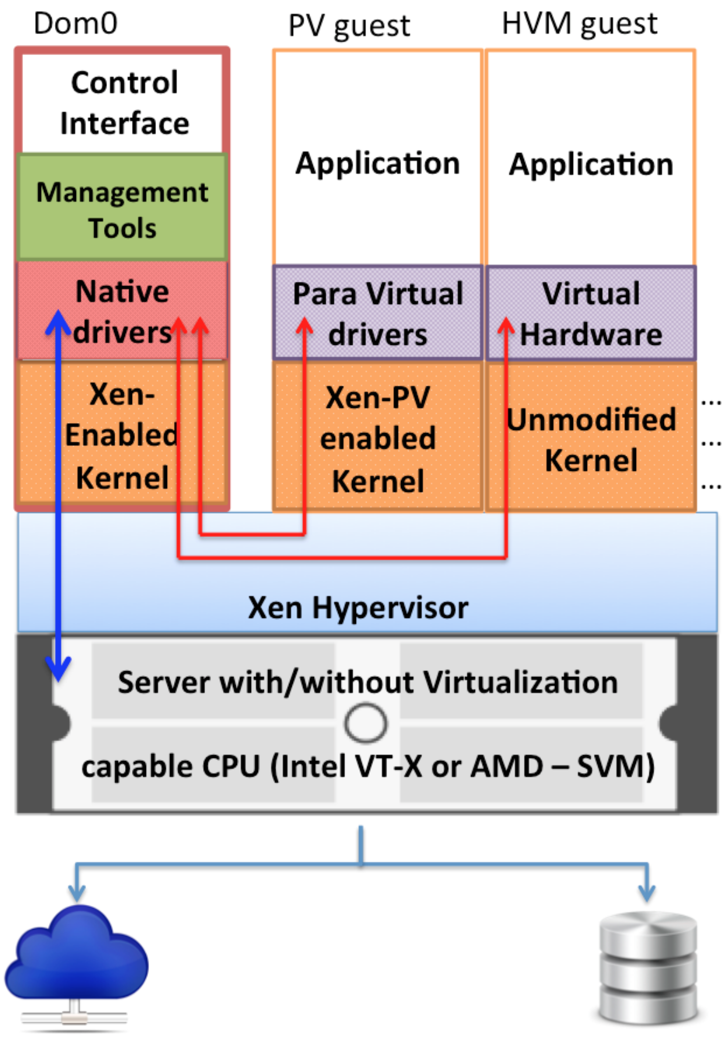


Figure 3.7: Xen - Architecture

The architecture of a system virtualized using Xen is shown in Figure 3.7. The Xen hypervisor is run directly from the bootloader. The Xen Hypervisor also referred as the Virtual Machine Monitor(VMM) is responsible for managing CPU, memory, and interrupts. The virtual machines termed as domains or guests run on top of the hypervisor. A special domain referred as domain0 acts as a controller and contains the drivers for all the devices in the system. The domain0 accesses the hardware directly, interacts with the other domains, and acts as the control interface for anyone to control the entire system. The controller domain also contains the set of tools to create, configure, and destroy virtual machines. The domain0 runs a modified version of the linux kernel that can perform the role of the controller [44]. All other domains created are totally isolated from the hardware and can use them only through the controller, hence referred as unprivileged domains (DomU). The DomUs can be either paravirtualized(PV) or hardware-assisted(HVM). The paravirtualized guests can run only modified operating systems as they require Xen-PV-enabled kernel and PV drivers which make them aware of the hypervisor. As an upside, para-virtualized guests does not require the CPUs to have virtualization extensions and are usually light weight compared to the unmodified operating systems. HVM guests can run unmodified operating systems but require virtualization extensions on the CPU, Intel-VT or AMD-V just like KVM. The HVM guests uses QEMU to emulate virtual hardware to provide the unmodified guest operating system. Both paravirtualized guests and hardware assisted guests can run on a single system at the same time. Recent work on the xen project also attempts to utilize the paravirtualized drivers on a HVM guest to improve performance and combine the best of both worlds.

The key differences in ideology of Xen as against the earlier discussed KVM, are :-

- Ability to run fully para-virtualized guests that have been optimized to run as a virtual machine.
- Ability to run para-virtualized guests even on CPUs without any hardware assistance.
- Lightweight hypervisor compared to the full linux kernel being used as hypervisor with KVM.
- Claims of better security as (i) The drivers run on a virtual machine instead of the hypervisor (ii) Reduced attack surface due to the smaller footprint of the hypervisor compared to the linux kernel.
- Though not widely used, Operating systems other than linux, NetBSD and OpenSolaris can

be used to run the controller VM (Dom0).

Bibliography

[1] Vishal Ahuja, Matthew Farrens, and Dipak Ghosal. Cache-aware affinitization on commodity multicores for high-speed network flows. In *Proceedings of the eighth ACM/IEEE symposium on Architectures for networking and communications systems*, ANCS '12, pages 39–48, New York, NY, USA, 2012. ACM.

[2] AMD. Amd-v virtualization extensions to x86. `http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf`, October 2013 (Last Accessed).

[3] M. Bardac, R. Deaconescu, and A.M. Florea. Scaling peer-to-peer testing using linux containers. In *Roedunet International Conference (RoEduNet), 2010 9th*, pages 287–292, 2010.

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003.

[5] Muli Ben-Yehuda, Jon Mason, Jimi Xenidis, Orran Krieger, Leendert Van Doorn, Jun Nakajima, Asit Mallick, and Elsie Wahlig. Utilizing iommus for virtualization in linux and xen. In *OLS06: The 2006 Ottawa Linux Symposium*, pages 71–86. Citeseer, 2006.

[6] Eric Biederman and Linux Networx. Multiple instances of the global linux namespaces. In *Proceedings of the Linux Symposium*. Citeseer, 2006.

[7] Davide Brini. Coreos. `http://backreference.org/2010/03/26/tuntap-interface-tutorial/`, October 2013 (Last Accessed).

[8] Canonical. Ubuntu server. `http://www.ubuntu.com/download/server`, October 2013 (Last Accessed).

[9] CentOS. Community enterprise operating system. `www.centos.org`, October 2013 (Last Accessed).

[10] Jianhua Che, Yong Yu, Congcong Shi, and Weimin Lin. A synthetical performance evaluation of openvz, xen and kvm. In *Services Computing Conference (APSCC), 2010 IEEE Asia-Pacific*, pages 587–594, 2010.

[11] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[12] MPI Commitee. Message passing interface. `http://www.mcs.anl.gov/research/projects/mpi/`, August 2013 (Last Accessed).

[13] C.N.A. Correa, S.C. de Lucena, D. de A.Leao Marques, C.E. Rothenberg, and M.R. Salvador. An experimental evaluation of lightweight virtualization for software-defined routing platform. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 607–610, 2012.

[14] Glauber Costa. Resource isolation: The failure of operating systems and how we can fix it - glauber costa. `http://linuxconeurope2012.sched.org/event/bf1a2818e908e3a534164b52d5b85bf1?iframe=no&w=900&sidebar=yes&bg=no#.UKPuE3npvNA`, October 2013 (Last Accessed).

[15] Debian. Debian - the universal operating system. `http://www.debian.org/`, October 2013 (Last Accessed).

[16] Apache Software Foundation. hadoop. `http://hadoop.apache.org/`, August 2013 (Last Accessed).

[17] Apache Software Foundation. Mesos. `http://mesos.apache.org/`, August 2013 (Last Accessed).

[18] Zhaoliang Guo and Qinfen Hao. Optimization of kvm network based on cpu affinity on multi-cores. In *Information Technology, Computer Engineering and Management Sciences (ICM), 2011 International Conference on*, volume 4, pages 347–351, 2011.

[19] Serge E. Hallyn. Discussion on security key namespaces. `http://lkml.indiana.edu/hypermail/linux/kernel/0902.3/01529.html`, October 2013 (Last Accessed).

[20] Red Hat. cgroups - red hat enterprise linux - resource management guide. `https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/ch01.html`, October 2013 (Last Accessed).

[21] Red Hat. Red hat enterprise linux. `http://www.redhat.com/products/enterprise-linux/server/`, October 2013 (Last Accessed).

[22] Todd Hoff. Building super scalable systems. `http://highscalability.com/blog/2009/12/16/building-super-scalable-systems-blade-runner-meets-autonomic.html`, August 2013 (Last Accessed).

[23] Intel. Hardware-assisted virtualization technology. `http://www.intel.com/content/www/us/en/virtualization/virtualization-technology/hardware-assist-virtualization-technology.html`, October 2013 (Last Accessed).

[24] Intel. Intel sr-iov primer. `http://www.intel.com/content/dam/doc/application-note/pci-sig-sr-iov-primer-sr-iov-technology-paper.pdf`, October 2013 (Last Accessed).

[25] Intel. Intel virtualization technology for directed i/o: Spec. `http://www.intel.com/content/www/us/en/intelligent-systems/intel-technology/vt-directed-io-spec.html`, October 2013 (Last Accessed).

[26] Michael Kerrisk. More on pid namespaces. `https://lwn.net/Articles/532748/`, October 2013 (Last Accessed).

[27] Michael Kerrisk. More on user namespaces. `https://lwn.net/Articles/540087/`, October 2013 (Last Accessed).

[28] Michael Kerrisk. Namespaces - overview. https://lwn.net/Articles/531114/#series_index, October 2013 (Last Accessed).

[29] Michael Kerrisk. Namespaces - the api. https://lwn.net/Articles/531381/, October 2013 (Last Accessed).

[30] Michael Kerrisk. Pid namespaces. https://lwn.net/Articles/531419/, October 2013 (Last Accessed).

[31] Michael Kerrisk. User namespaces. https://lwn.net/Articles/532593/, October 2013 (Last Accessed).

[32] Linux. chroot - change root directory. http://man7.org/linux/man-pages/man2/chroot.2.html, October 2013 (Last Accessed).

[33] Linux. Control groups. https://lwn.net/Articles/524935/, October 2013 (Last Accessed).

[34] Linux. Kernel documentation - summary of hugetlbpage support. https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt, October 2013 (Last Accessed).

[35] Linux. Linux containers. http://sourceforge.net/projects/lxc/, August 2013 (Last Accessed).

[36] Linux. Linux kernel virtualization support with kvm. http://kernelnewbies.org/Linux_2_6_20#head-bca4fe7ffe454321118a470387c2be543ee51754, October 2013 (Last Accessed).

[37] Linux. Overview of non-uniform memory architecture. http://man7.org/linux/man-pages/man7/numa.7.html, October 2013 (Last Accessed).

[38] Linux. Posix message queues. http://man7.org/linux/man-pages/man7/mq_overview.7.html, October 2013 (Last Accessed).

[39] Linux. Qemu. http://wiki.qemu.org/Main_Page, October 2013 (Last Accessed).

[40] Linux. System v inter process communication mechanisms. http://man7.org/linux/man-pages/man7/svipc.7.html, October 2013 (Last Accessed).

[41] Oracle. Oracle solaris zones. http://docs.oracle.com/cd/E18440_01/doc.111/e18415/chapter_zones.htm, October 2013 (Last Accessed).

[42] Parallels. Openvz. http://openvz.org/Main_Page, October 2013 (Last Accessed).

[43] Ian Pratt. Xen virtual machine monitor performance. http://www.cl.cam.ac.uk/research/srg/netos/xen/performance.html, October 2013 (Last Accessed).

[44] Xen Project. Dom0 - kernels. http://wiki.xenproject.org/wiki/Dom0_Kernels_for_Xen, October 2013 (Last Accessed).

[45] Xen Project. Xen - overview. http://wiki.xen.org/wiki/Xen_Overview, October 2013 (Last Accessed).

[46] Benjamin Qutier, Vincent Neri, and Franck Cappello. Scalability comparison of four host virtualization tools. *Journal of Grid Computing*, 5(1):83–98, 2007.

[47] Inc Red Hat. Automatic virtual machine migration. https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Virtualization/3.0/html/Administration_Guide/Tasks_RHEV_Migration_Automatic_Virtual_Machine_Migration.html, August 2013 (Last Accessed).

[48] FreeBSD Matteo Riondato. Jails - freebsd. http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/jails.html, October 2013 (Last Accessed).

[49] Rami Rosen. Namespaces and cgroups. http://media.wix.com/ugd//295986_d73d8d6087ed430c34c21f90b0b607fd.pdf, October 2013 (Last Accessed).

[50] L. Sarzyniec, T. Buchert, E. Jeanvoine, and L. Nussbaum. Design and evaluation of a virtual experimental environment for distributed systems. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 172–179, 2013.

[51] J.E. Smith and R. Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, 2005.

[52] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3):275–287, March 2007.

[53] Open Source. Coreos. http://coreos.com/, August 2013 (Last Accessed).

[54] Open Source. Open vswitch. http://openvswitch.org/, August 2013 (Last Accessed).

[55] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the denali isolation kernel. *SIGOPS Oper. Syst. Rev.*, 36(SI):195–209, December 2002.

[56] M.G. Xavier, M.V. Neves, F.D. Rossi, T.C. Ferreto, T. Lange, and C.A.F. De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 233–240, 2013.

[57] XSEDE. Futuregrid. https://portal.futuregrid.org/, August 2013 (Last Accessed).

[58] Andrew J Younge, Robert Henschel, James T Brown, Gregor von Laszewski, Judy Qiu, and Geoffrey C Fox. Analysis of virtualization technologies for high performance computing environments. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 9–16. IEEE, 2011.

[59] Lamia Youseff, Rich Wolski, Brent Gorda, and Chandra Krintz. Paravirtualization for hpc systems. In *Frontiers of High Performance Computing and Networking–ISPA 2006 Workshops*, pages 474–486. Springer, 2006.