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 discusses the design principles behind the chosen virtualization solutions. Second, this thesis evaluates them based on the overhead they impose to virtualize system resources. Finally, this thesis assesses the benefits and limitations of each based on their operational flexibility, and resource entitlement and isolation facilities.

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.

Table of Contents

List of Tables

List of Figures

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 [29].

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)* [68] or *Containers* [47]. 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 [14]. The fail-over can also be automatically triggered in some cluster-aware virtualization platforms [64]. 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. [77] compared Xen and KVM with virtual resources from the FutureGrid project [76] and claims, KVM performed significantly better than Xen, under the standard HPCC and SPEC benchmarks. Another benchmark oriented synthetic study [12] 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. [75] and an earlier study [69], 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. [63] 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, [16], and [67] discussed the performance benefits of large scale virtual network emulation by implementing open Vswitch [71] on LXC based platforms. Mesos [24] , A clustered platform that provides fine-grained resource sharing among multiple frameworks like Hadoop [23] and MPI [15] 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 [25] 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 [70], 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

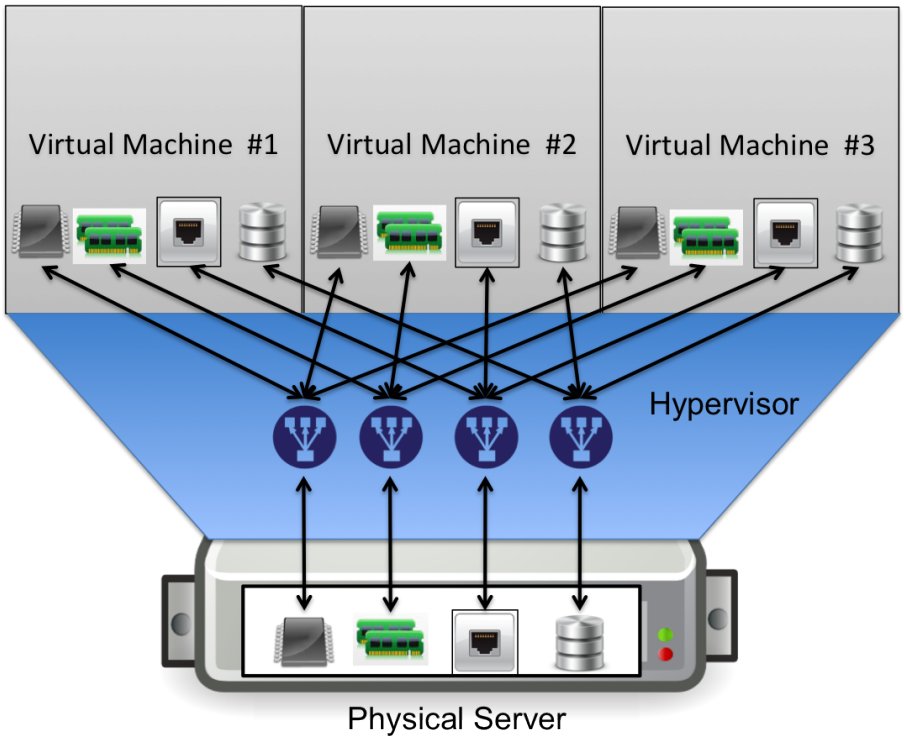


Figure 3.1: Block Diagram of a full-virtualization system

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) [48]. 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 [51]. 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. A server with a virtualization capable processor, disk storage, and network interfaces runs a standard linux operating system that contains KVM. Virtual machines co-exist with the user-space applications that may be running directly on the host. The virtual machines contain a set of virtual devices created using an user-space tool called QEMU, and run an unmodified guest operating system like Linux, Microsoft Windows.

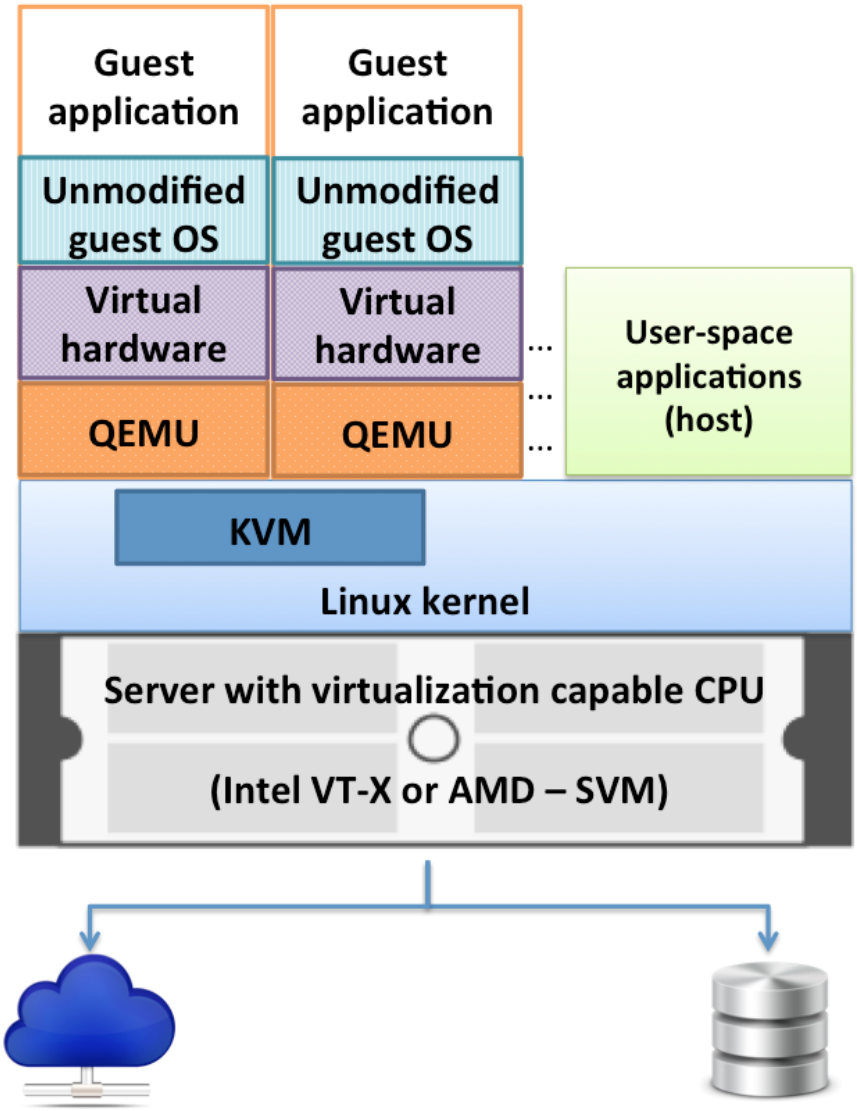


Figure 3.2: KVM - Architecture

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 [31] 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

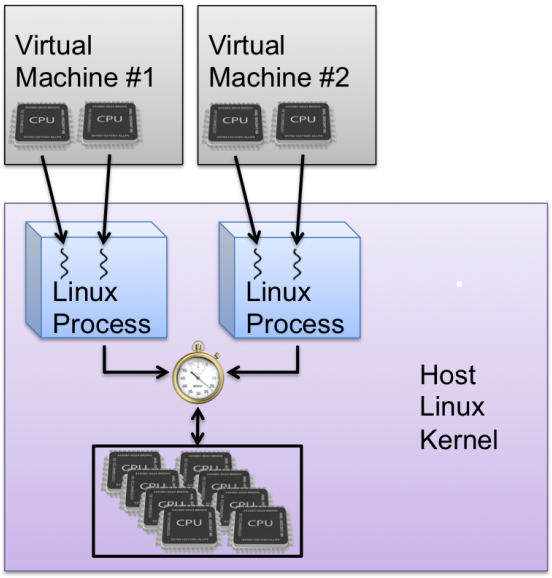


Figure 3.3: KVM - Virtual CPU management

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 [49], and Huge Pages [46] to optimize the memory allocated to the virtual machines. NUMA support and Huge Pages will be discussed in the later sections.

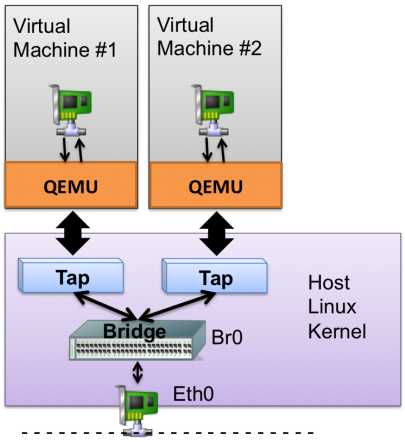
3. **Network Interfaces**:

   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, among each other, 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 [8] 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.

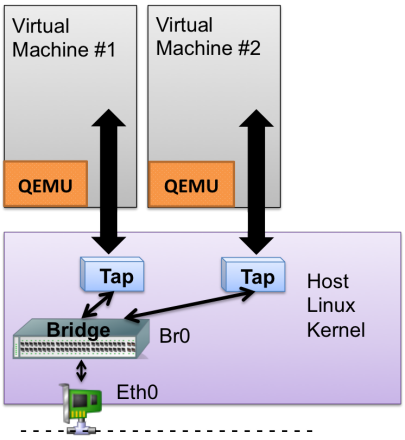
   Figure 3.4b shows a networking configuration where the guest operating system running in the virtual machine is "virtualization aware" and cooperates with the host by bypassing the device emulation by QEMU. This is known as *para-virtualized networking*. Para-virtualized networking trades compatibility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

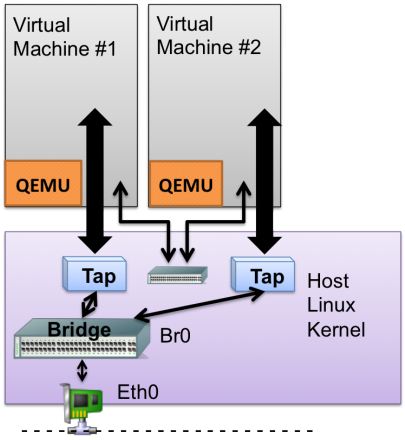
   To support the virtualization of network intensive applications, physical network interfaces attached to the host can be directly assigned to the virtual machine, bypassing the host kernel and QEMU, as shown in Figure 3.4d. This setup provides "bare-metal" performance by providing direct access to the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



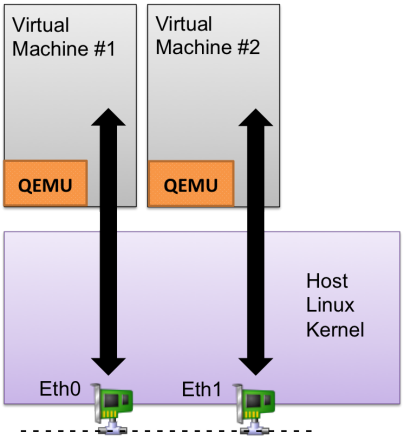
(a) Emulated Networking devices



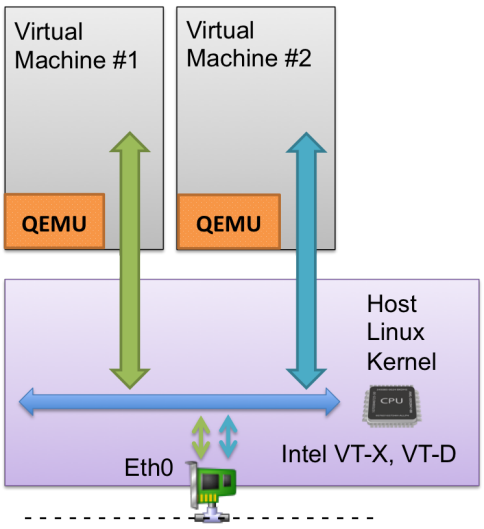
(b) Para-Virtualized Networking



(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 [33], 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) [32]. 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 [17]. 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 this problem in BSD Jails [65], Solaris Zones [56], Chroot [44], and most importantly, OpenVZ [57] - a fork of the Linux kernel maintained by Parallels. BSD Jails were designed to restrict the visibility of the host's resources from 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 solving the resource isolation problem is *"Containers"* incorporating the benefits of the above 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 imposing virtualization requirements 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.

Containers expand the scope of BSD jails, providing a granular operating system virtualization platform, 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 distribution of Linux that may be different from the host. For example, a host server running RedHat Enterprise Linux [28] may run containers that run Debian [18], Ubuntu [10], CentOS [11], 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.

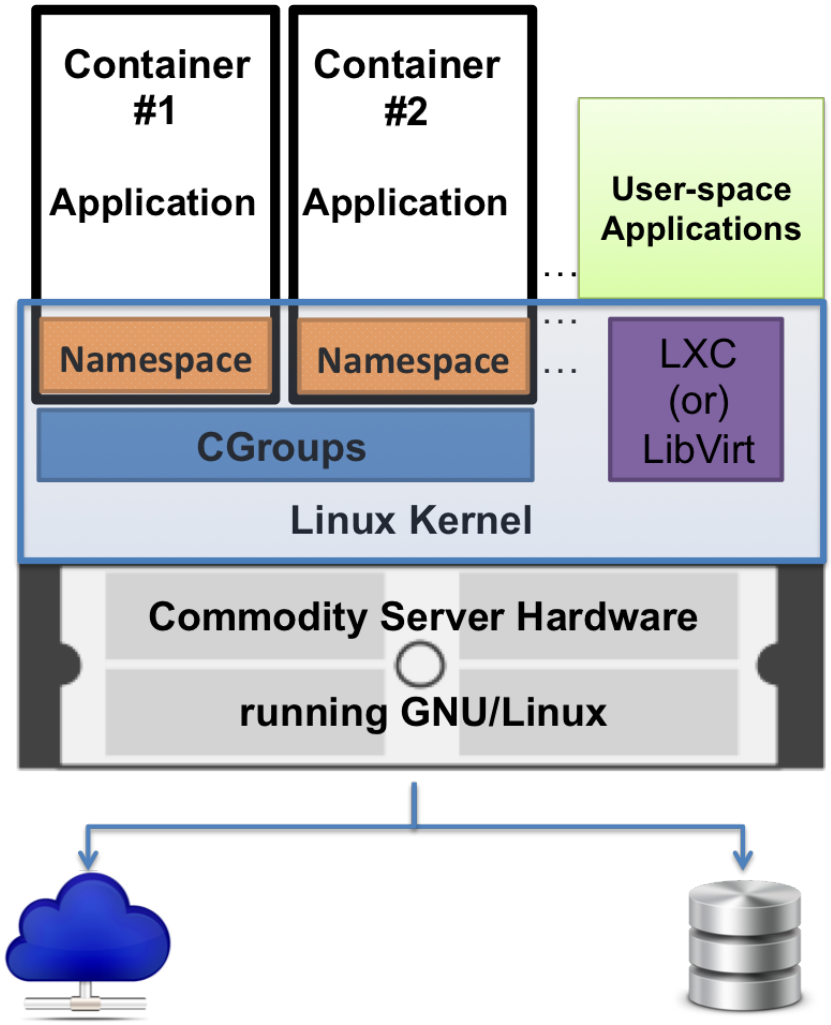


Figure 3.5: Linux Containers - Architecture

Figure 3.5 shows the architecture of a server that uses Linux containers for virtualization. Unlike KVM, Linux containers do not require any assistance from the hardware; the solution can run on any platform capable of running the mainline Linux kernel. The kernel facilitates the execution of containers by utilizing *namespaces* for resource isolation, and *cgroups* for resource management and control. Containers can be administered by using the standard libvirt API, and the LXC suite of command line tools. Since containers are viewed as regular Linux process groups by the Linux kernel, they can co-exist with any user-space applications that may be running directly on the physical server.

The following description of namespaces in the Linux kernel is based on [36], [37], [38], [34], [39], and [35]. Namespaces "wrap" a global system resource on the host and makes it appear to the processes within the namespace (containers) as though they have their own isolated instance of the global resource.

Linux implements 6 types of namespaces:

- **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 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 different locations on the filesystem tree, e.g., "/<containername1>/data", "/<containername2>/data", and so on. Of course, the same setup can be achieved using the *chroot()* command, but, chroot can be escaped with certain capabilities, including CAP_SYS_CHROOT [19]. Mount namespaces provides a secure alternative. Mount namespaces greatly improve the portability of the containers as they can retain their filesystem trees irrespective of the host's environment.

- **UTS namespaces :**

  UNIX Time-sharing System (UTS) namespaces facilitate the use of the *sethostname()* and *setdomainname()* system calls to set the container hostname and NIS domain name respectively. *uname()* returns the appropriate hostname and domain name.

- **IPC namespaces :**

  IPC namespaces isolate the System V IPC objects [52] and POSIX message queues [50] associated with individual containers.

- **PID namespaces :**

  PID namespaces in the Linux kernel facilitate the isolation of process identification numbers (PIDs) associated with processes running on the host and within its containers. Every container can have its own init process (PID 1). In general, several containers running simultaneously can have processes with identical PIDs. The Linux kernel implements the PIDs as a structure,

consisting of two PIDs, one in the container's namespace and other outside the namespace. This PID abstraction is useful in two regards. First, it isolates the containers, such that a process running in one container does not have visibility of processes running in other containers. Second, it enables the migration of containers across hosts, as the containers can retain the same PIDs.

- **Network namespaces :**

  Network namespaces provide isolation of network resources for containers, enabling the containers to have their own (virtual) network devices, IP addresses, port numbers, and IP routing tables. For example, a single host can run multiple containers, each running a web server at its own IP address over port 80.

- **User namespaces :**

  User namespaces provide isolation for user and group IDs.Each process will have two user and group IDs, one inside the container's namespace, and other outside the namespace. This enables a user to possess an UID of 0 (root privileges) inside a container, while still being treated as an unprivileged user on the host. The same applies to 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 new namespaces are being developed for future inclusion into the Linux kernel [7]:-

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

- **Security keys namespace :** This namespace aims to provide an independent security key space for containers to isolate the /proc/keys and /proc/key-users files based on the namespace [26].

- **Device namespace :** This namespace aims to provide each container its own device namespace, to enable containers to create/access devices with their own major and minor numbers so they can be seamlessly migrated across hosts.

- **Time namespace :** This namespace aims to enable containers to freeze/modify their thread and process clocks, which would support "live" host migration.

Control groups (cgroups) [45] [66] [27] are another key feature of the Linux kernel, used to allocate resources such as CPU, memory, network bandwidth, disk access bandwidth among user-defined groups of processes (containers). cgroups instrument the Linux kernel to limit, prioritize, monitor and isolate system resources. With proper usage of cgroups, hardware and software resources on the host can be efficiently allocated and shared among several containers. cgroups can be dynamically re-configured and made persistent across reboots of the host. Though cgroups are generic and apply to individual processes and process groups, the remainder of this section discusses their relavance to containers.

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

Cpu	Used to set limits on the access to the CPU for the containers
Cpuset	Used to tie containers to system subsets of CPUs and memory (memory nodes)
Cpuacct	Used to account the usage of CPU resources by containers in a cgroup.
Memory	Used to set limits on memory use by containers in a cgroup
Blkio	Used to set limits on input/output access to and from block devices
Devices	Used to allow or deny access to devices by containers in a cgroup.
Net_cls	Used to tag network packets with a class identifier (classid) that allows the Linux traffic controller (tc) to identify packets originating from a particular cgroup task.
Net_prio	Used to prioritize the network traffic to and from the containers on a per network interface basis.

Table 3.1: Cgroups - Subsystems

cgroups are organized hierarchically. There may be several hierarchies of cgroups in the host system, each associated with at least one subsystem. Process groups (in our case, containers) are assigned to cgroups in different hierarchies to be managed by different subsystems. All the containers in the Linux system are a member of the root cgroup by default and are associated with custom user-defined cgroups on an as-needed basis.

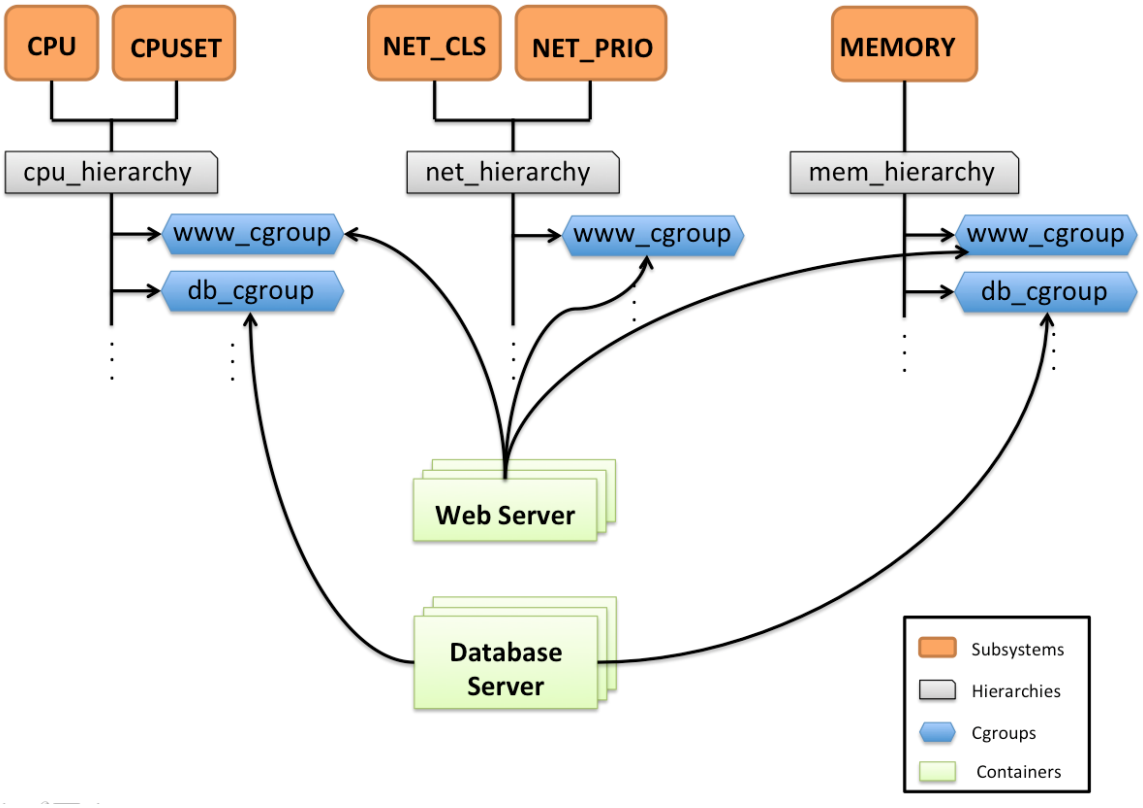


Figure 3.6: Cgroups Example Configuration

Figure 3.6 shows an example configuration of cgroups for a system running multiple web servers and database servers as containers. The objective is to limit the CPU, network bandwidth and memory available to containers based on whether they run a web server or a database server. First, a hierarchy is created for the type of subsystem(s) of interest. For example, mem_hierarchy is created with association to memory subsystem. Second, custom cgroups are created under the hierarchy based on the categories of workloads. For example www_cgroup is created define policies for all the containers that run a web server. The kernel automatically fills the cgroups directory with the settings file nodes. Third, limits are set on the created cgroup. For example, the amount of memory available to all the containers associated with this cgroup can be set to 2 Gigabytes. Finally, all the containers that are provisioned to run a web server are added to the respective cgroup (in our example, www_cgroup).

The process discussed above may be repeated for different subsystems (e.g. CPU, blkio) to limit the corresponding resources. In the example shown in Figure 3.6, all the containers running

web server are associated to the cgroup, www_cgroup, which limits their usage in terms of memory, network bandwidth, and CPU time. All containers running database server are associated to the cgroup, db_cgroup, which limits their usage in terms of memory, and CPU while allowing them to use unlimited network bandwidth.

3.3 Xen

Xen [60] is an open source virtualization platform that provides a bare-metal hypervisor, implemented as special firmware that runs directly on the hardware. The Xen project is based on para-virtualization [74] [78], a technique where the guest operating systems are modified to run on the host using an interface that is easier to virtualize. Para-virtualization significantly reduces overhead and improves performance according to [74], [4], [58].

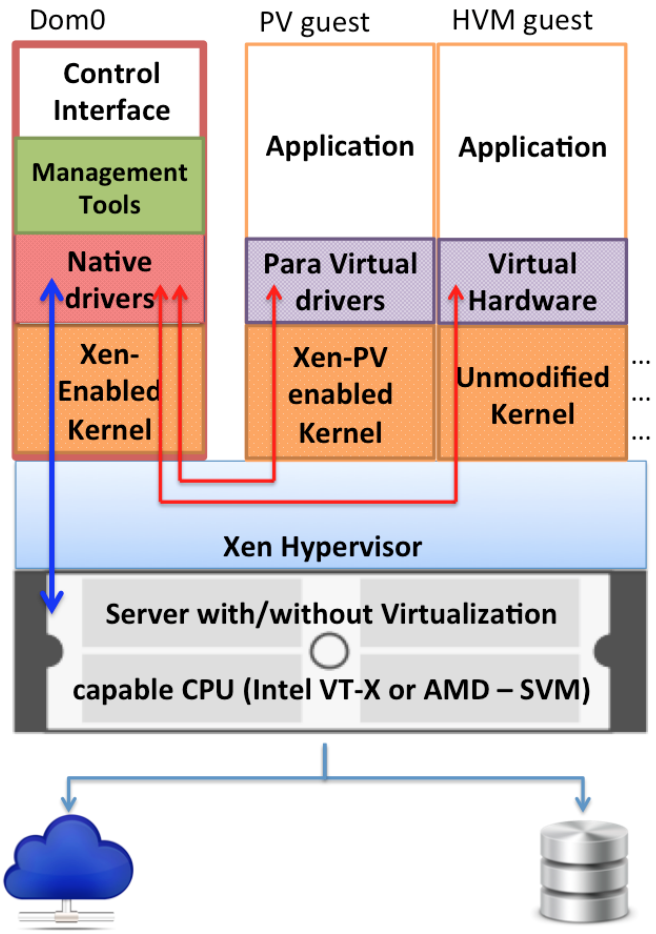


Figure 3.7: Xen - Architecture

The architecture of a system virtualized using Xen is shown in Figure 3.7. The Xen hypervisor is run directly from the bootloader. The hypervisor also referred to as the *Virtual Machine Monitor (VMM)*; it is responsible for managing CPU, memory, and interrupts. The virtual machines, referred to as *domains* or *guests* run on top of the hypervisor. A special domain referred to as *domain0* acts as a controller, containing the drivers for all the devices in the system. domain0 accesses the hardware directly, interacting with the other domains, and acts as the control interface for anyone to control the entire system. This controller domain also contains the set of tools to create, configure, and destroy virtual machines. It runs a modified version of the Linux kernel that can perform the role of the controller [59]. All other domains are totally isolated from the hardware and can use these resources only through the controller; they are referred to as *unprivileged domains (DomU)*. The DomUs can be either para-virtualized (PV) or hardware-assisted (HVM). The para-virtualized 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 do not require the CPUs to have virtualization extensions, and are usually lightweight compared to the unmodified operating systems. HVM guests can run unmodified operating systems, but require virtualization extensions on the CPU, just like KVM. The HVM guests use QEMU to emulate virtual hardware to provide the unmodified guest operating system. Both para-virtualized 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 para-virtualized drivers on a HVM guest to improve performance, combining the best of both worlds.

Chapter 4

Virtualization Overhead

Operating systems typically provide a basic set of resource virtualization facilities. These facilities enable the operating system to share physical resources among the applications through the implementation of virtual memory, CPU schedulers and process hierarchies. However, these facilities are not sufficient in a context where some applications require operating environments which differ from the needs of other applications running on the same machine. As discussed in chapter 3, several virtualization solutions address the resource isolation needs by virtualizing key system resources (CPUs, memory, disks, network interfaces). To transparently abstract system resources, most virtualization solutions perform redundant operating system functions, including CPU scheduling, memory management, network stack implementation, and disk I/O. For example, an application running on a virtual machine is scheduled on the virtual CPUs (VCPUs) by the guest operating system, while the VCPUs are in turn scheduled on the physical CPUs by the host operating system. This redundancy imposes overhead on the use of system resources. The virtualization overhead varies based on the implementation of the virtualization system. The goal of any virtualization technique is to impose as little overhead as possible, while providing the needed resource isolation.

This chapter evaluates the representative virtualization solutions introduced in chapter 3 (i.e, KVM, Linux containers, and Xen), based on the virtualization overhead they impose under varying operating conditions and workloads. First, we evaluate the solutions based on the overhead they impose in virtualizing the access to the CPUs. Second, we measure the overhead associated with memory access bandwidth available to the applications while running inside of virtual machines as

compared to applications running on the bare-metal operating system. Third, we study the impact of virtualization on network access bandwidth. Fourth, we measure the overhead associated with disk I/O operations, focusing on sequential and random file access operations, and general disk access latency. Finally, we measure the overhead exhibited at different scales of virtualization by running several instances of the test application suite in virtual machines simultaneously, and comparing their performance metrics with those collected from execution on bare-metal host operating system.

4.1 Experimental Conditions

To evaluate the three representative virtualization solutions independently, and to be able to compare the results with the performance on an independent bare-metal host, we used four identical Dell Optiplex 990 machines with the following configuration:

Processor	Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz
Chipset	Intel(R) Q67 Express Chipset
Memory	8 GB, Non-ECC Dual-Channel 1333 MHz DDR3
Network	Intel (R) 82579LM Ethernet LAN 10/100/1000
Hard disk	1TB 7200 RPM SATA 3.0Gb/s

Table 4.1: Experimental setup - Hardware configuration

To ensure fair comparison, we preserve the default operating system and hypervisor configurations. Modification of any default settings required to evaluate a specific scenario are documented in place. The default operating system, kernel and hypervisor settings are as follows:

Operating system	Ubuntu Server 12.04.3 LTS
Kernel	GNU/Linux 3.8.0-29-generic x86_64
Lxc version	0.7.5
QEMU-KVM version	1.0
Xen version	xen-3.0-x86_64 hvm-3.0-x86_64

Table 4.2: Experimental setup - Software configuration

4.2 CPU Overhead

Most virtualization solutions rely on the creation of virtual CPUs, with the hypervisor responsible for scheduling the virtual CPUs over the physical CPUs. This creates an additional layer of control, which makes it possible to create virtual CPUs that are in turn backed by more than one physical CPU. Alternatively, it also enables creation of more virtual CPUs than physical CPUs available on the system. The CPU capacity available to the applications in a virtualized environment is influenced by two parameters, (i) the raw performance of the CPU in executing primitive tasks and (ii) the performance of the scheduler in scheduling multiple threads.

To evaluate CPU overhead including both the parameters mentioned above, we created a sample application (w1) representing a diverse set of CPU-oriented tasks:

1. Repeated calculation of large prime numbers using varying numbers of threads.

2. Measuring the speed and efficiency of floating point operations including sin, cos, sqrt, exp, log, array accesses, conditional branches, and procedure calls by executing the whetstone workload from the unixbench suite of tests [20].

3. Execution of a large number of threads competing for a set of mutexes, in order to measure the scheduler performance for multi-threaded applications [43].

4. Generation of 500 RSA private keys using openssl [22].

The objective of this test is to measure the execution time of the sample application and its individual tasks as described above. The test was performed on a virtual machine created on each of the three virtualized servers, and also on an independent bare-metal system used as a baseline.

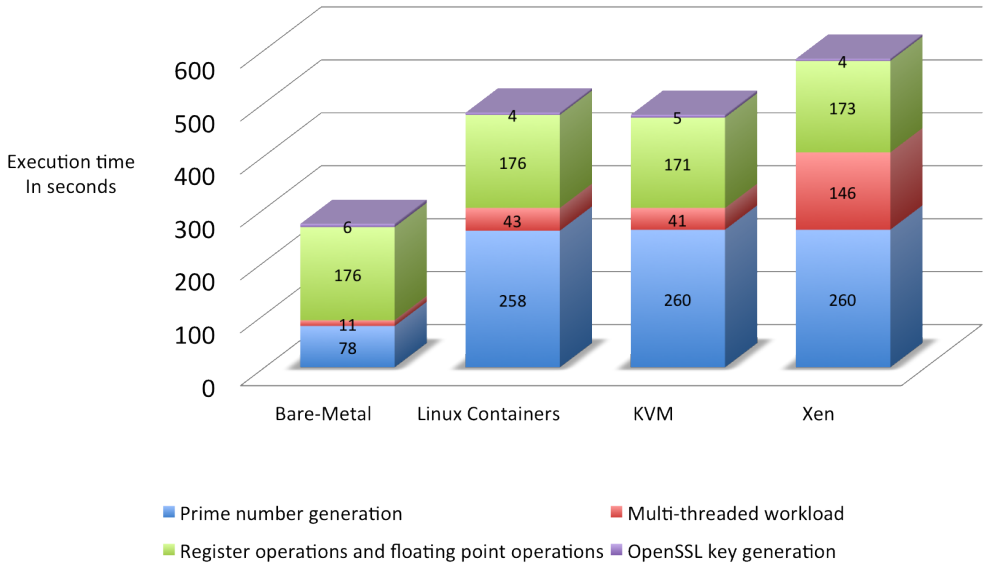


Figure 4.1: CPU virtualization overhead - 1 virtual CPU

A virtual machine was created on each of the three physical servers, representing the virtualization solutions with one virtual CPU, 300 GB of disk storage and 7680 MB of memory. Figure 4.1 shows the overhead caused by the representative virtualization platforms while virtualizing a single VCPU over one of the eight physical CPUs. The X-axis represents the virtualization platform, and the Y-axis represents the stacked execution time of the sample application (w1). It is evident from Figure 4.1 that virtual machines created using Linux containers and KVM executed the sample application faster than the virtual machine created using Xen. Xen, among the three virtulization solutions yielded higher overhead with respect to access to the CPU. In particular, Xen took longer to complete the task involving multiple threads completing for a set of mutexes, indicating the poor relative performance of the CPU scheduler in the Xen hypervisor.

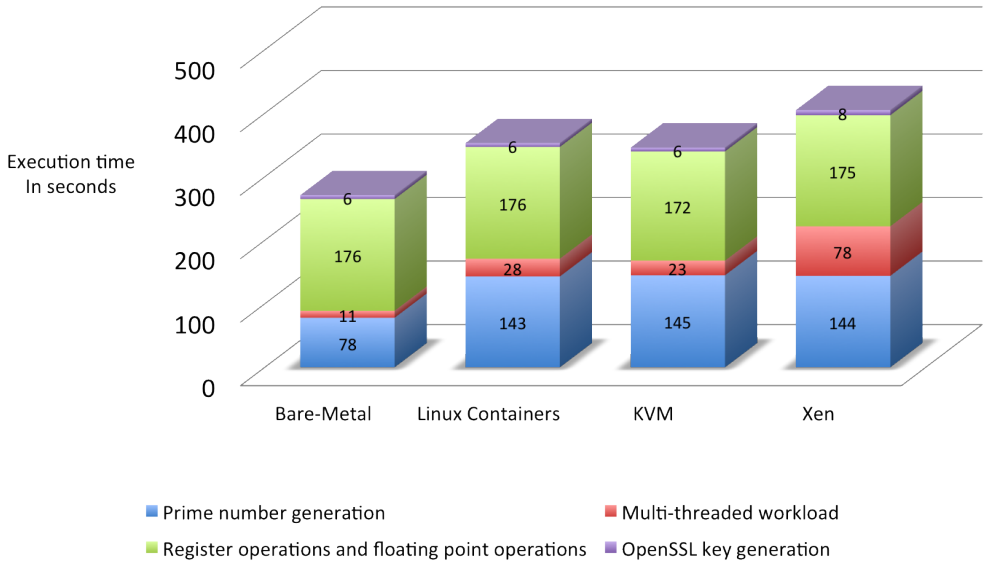


Figure 4.2: CPU virtualization overhead - 2 virtual CPUs

The same test was repeated with the virtual machines configured with two virtual CPUs mapped to two of the physical CPUs, 300 GB of disk storage and 7680 MB of memory. The execution times with the modified configuration are shown in Figure 4.2. The single threaded tasks were not affected by the availability of an additional core and were executed in almost the same time durations as the previous test. It is to be noted that the execution times for the multi-threaded tasks are lower compared to the results in Figure 4.1 owing to the availability of one additional physical core. The results once again bring out the poor scheduler performance of Xen hypervisor.

The same test was repeated with the virtual machines configured with four virtual CPUs mapped to four of the physical CPUs, 300 GB of disk storage and 7680 MB of memory. The execution times with the modified configuration are shown in Figure 4.3. The results once again confirm our hypothesis from the previous tests about the poor performance of the Xen scheduler for multi-threaded workloads.

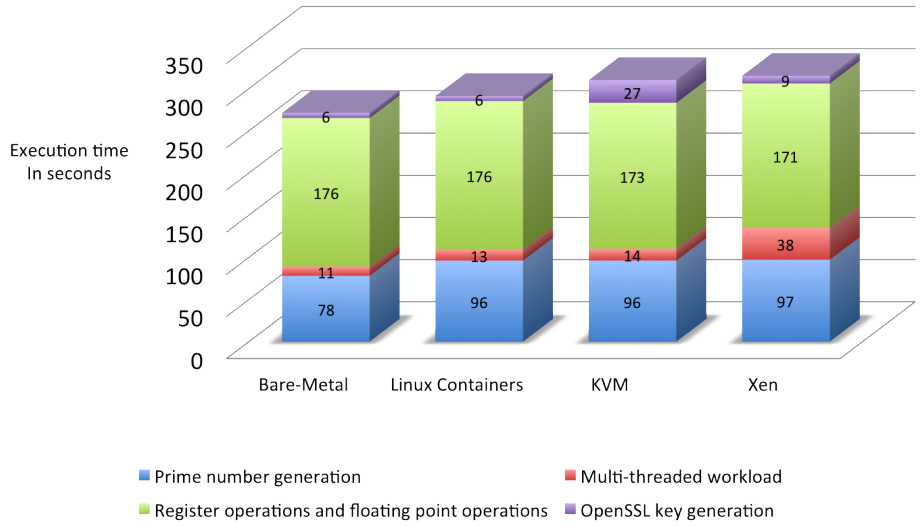


Figure 4.3: CPU virtualization overhead - 4 virtual CPUs

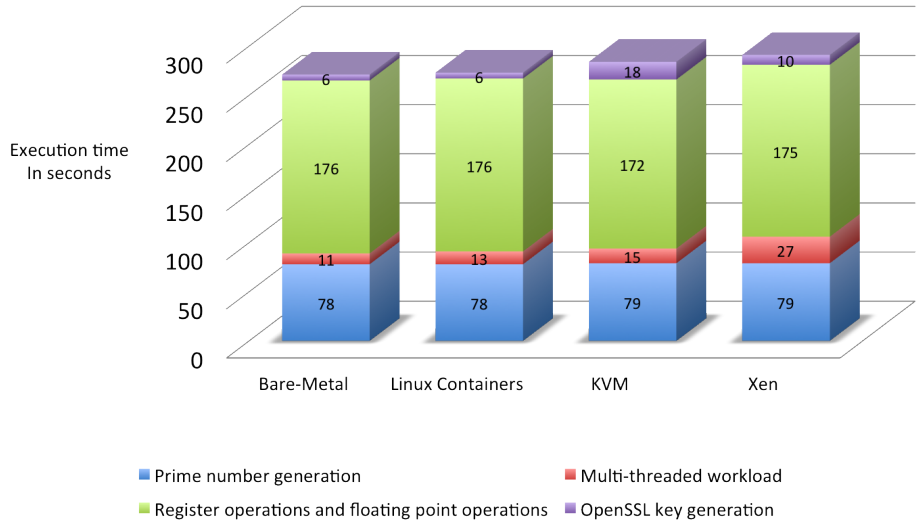


Figure 4.4: CPU virtualization overhead - 8 virtual CPUs

To bring out the direct overhead caused by virtualizing the CPUs, we ran the test with the virtual machines having eight virtual CPUs mapped to eight physical CPUs, 300 GB of disk storage and 7680 MB of memory. The resources available to the virtual machines are identical to the bare-metal host with the only exception of passing through the hypervisor. The difference in execution times between the virtual machines representing Linux containers, KVM, and Xen and the bare-metal host can be correlated to the overall CPU overhead. Both Linux containers, and KVM exhibit the least overhead, followed by Xen.

Based on the test results discussed above, it can be concluded that Linux Containers and KVM perform the best for both single threaded and multi-threaded workloads by exhibiting the least overhead compared to the bare-metal performance. Though Xen performed identical to the rest for single threaded workloads, it exhibited relatively bad performance with respect to scheduling multi-threaded workloads.

4.3 Memory Overhead

KVM, Xen, and Linux containers manage virtual machine memory differently. Xen, being a para-virtualized platform, manages virtual machine memory in a collaborative fashion. The Xen hypervisor allocates the required memory on the host, holds a section of the virtual machine's address space, and then decouples the virtual machine for any unprivileged memory accesses. Any sensitive instructions originating from the virtual machines are directed to the hypervisor. The virtual machines manage their own page tables. KVM, as a full virtualization solution, allocates the required memory on the host, and then decouples its virtual machines, allowing each to manage its own memory. The KVM kernel module provides each virtual machine its own address space in the host kernel. The memory allocated for each virtual machine is allocated in the virtual memory of the host. This design creates an interesting possibility where virtual machines can be created with total memory that exceeds the physical memory of the host. If a virtual machine attempts to use more memory than the host can provide, the host kernel starts its swapping process. The virtual machines are responsible for their own page tables. Linux containers, as an operating system virtualization solution, rely on the simplest memory management mechanism. Since containers share the kernel with the host, they do not maintain a separate page table. The containers are represented as a group of processes in the host kernel. Hence, the memory associated with the containers is just the

virtual memory allocated for the corresponding process groups in the kernel.

To evaluate the virtualization overhead associated with memory access, we created a sample application (w2) that uses mbw [30] to allocate two arrays of a given size, and then copies data from one to another. The reported "bandwidth" is the amount of data copied, over the time required for the operation to complete. The objective of this test is to measure the memory access bandwidth as available from the virtual machine created on each of the three virtualized servers, and compare against the bandwidth observed on an independent bare-metal host.

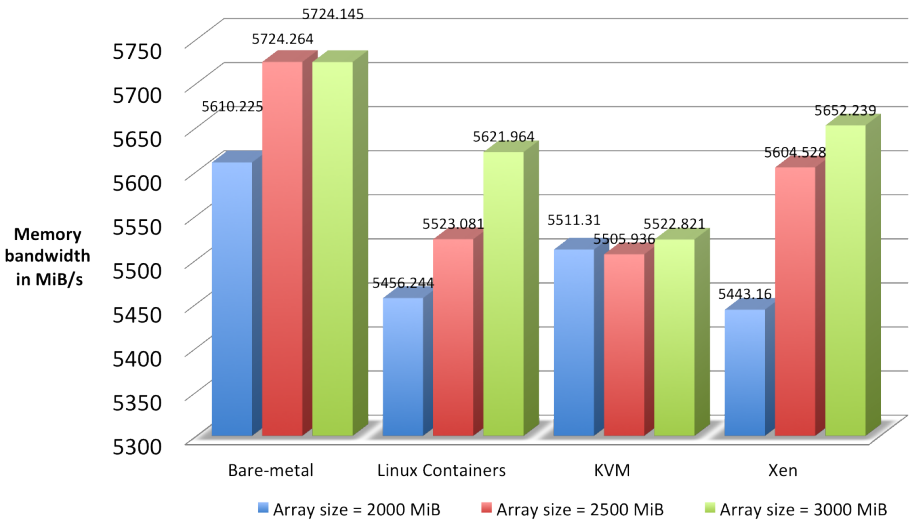


Figure 4.5: Memory virtualization overhead - (virtual machine's memory = Host's memory = 8 GB)

A virtual machine was created on each of the three physical servers, representing the virtualization solutions with eight virtual CPUs mapped to eight physical CPUs, 300 GB of disk storage and 7680 MB of memory. In this case, since the virtual machine memory is equal to the host memory, the difference in memory access bandwidth between the virtual machines and the bare-metal host corresponds to the memory overhead caused by the virtualization platforms. The results are shown in Figure 4.5, with X-axis representing the virtualization platforms, and Y-axis representing the observed bandwidth in mebibyte per second (MiB/s) for different array sizes. It is evident from the results shown in Figure 4.5 that each of the virtualization platform exhibit a different pattern

in causing memory overhead. First, Xen exhibits high overhead for smaller array sizes indicating a less than average performance in the normal scenario. However, Xen performs relatively better for larger array sizes. This behavior is explained by the decoupling of memory management to the virtual machines themselves by the Xen Hypervisor. However, KVM showed an average memory overhead irrespective of the array size. This behavior is explained by the isolation of virtual machine memory in its own guest address space in the host and managing a mapping between the host address space and the guest address space. While the host is managing the memory allocated to a KVM guest, the guest kernel is simultaneously managing the same memory. To make optimal memory decisions and re-use identical pages among the virtual machines, the host kernel and the guest kernel collaborates using a feature called "Kernel Same page Merging" [54] causing an overhead in low utilization scenarios.

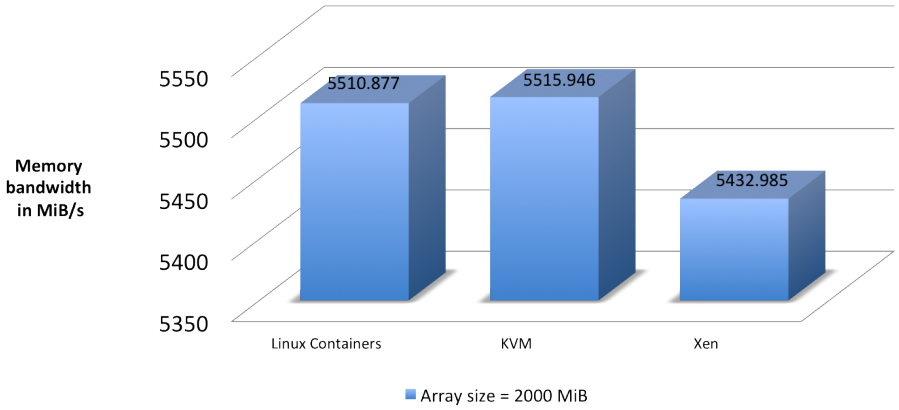


Figure 4.6: Memory virtualization overhead (virtual machine memory(5 GB) <Host memory(8 GB))

To bring out the behavior of the virtualization platforms when all of virtual machine memory can be accommodated by the host, we repeated the same test with the virtual machines configured with 5 GB of memory, 8 virtual CPUs mapped to 8 physical CPUs and 300 GB of disk storage. As evident from the results shown in Figure 4.6 Linux containers and KVM yielded higher memory bandwidth than Xen.

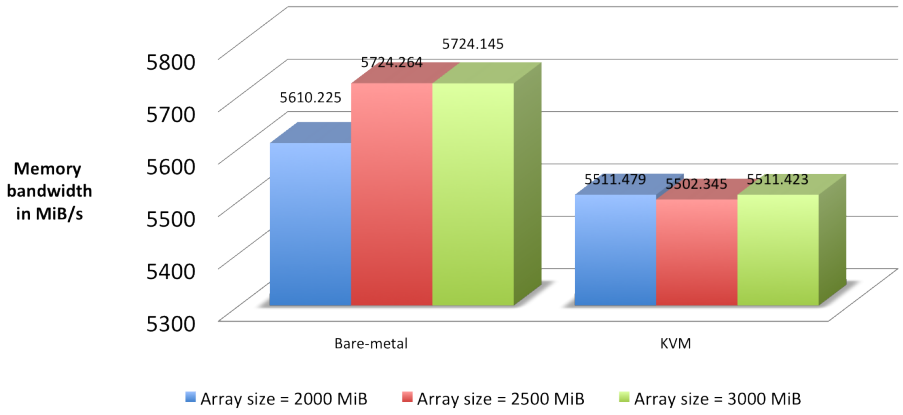


Figure 4.7: Virtualization overhead - Virtual machine memory(10 GB) >Host memory(10 GB)

KVM, allowing the virtual machines to run their own kernel and yet managing the virtual machine memory from the host linux kernel, enjoys the ability to leverage the memory over commitment facilities by design. Memory pages requested by a virtual machine are not allocated until they are actually used. The host kernel can free up memory by swapping less frequently used pages to disk. While not perfect, these techniques can be used to create extremely over-committed environments. To evaluate the behaviour of KVM in such an over-committed environment, we created a virtual machine with 10 GB of memory virtualized on 8 GB of host memory, and observed the memory access bandwidth. The results as shown in Figure 4.7 confirm our understanding that KVM performs the same even in an over-committed environment until all the virtual machines attempt to use their share of memory at the same time.

4.4 Network overhead

KVM, Xen and Linux containers provide a way to transparently virtualize the network interface on the host to the virtual machines so that, the virtual machines can build an independent network stack for themselves over their virtual network interfaces. KVM, uses QEMU to emulate a virtual network devices over the physical network interfaces. Xen, enables the virtual machines to use para-virtual drivers to access the network interfaces on the host. Linux containers uses network namespaces to virtualize the networks of containers. The virtual machines use their virtual

networking facilities to communicate with external networks, as well as other virtual machines running on the same host. The network access available to the virtual machine can be : (i) Network Address Translated (NAT), where the host proxies the communications from the virtual machine, or (ii) Bridged, where the virtual machines are allowed to participate in the same LAN segment as the host. To measure any overhead caused by the virtualizing the networking interface, we measured the network bandwidth available to a virtual machine in both NAT and bridged configurations and compared them with the bandwidth available to an independent bare-metal host. To ensure fair comparison, we set up an Iperf [21] server on a machine outside of the network of the physical servers being tested and measured the network bandwidth by running an iperf client on each of the virtual machines. The test was conducted on both NAT and bridged configurations, and the results are shown in Figure 4.8, where X-axis represents the virtualization platforms and the Y-axis represents the observerved network bandwidth in MBytes/sec.

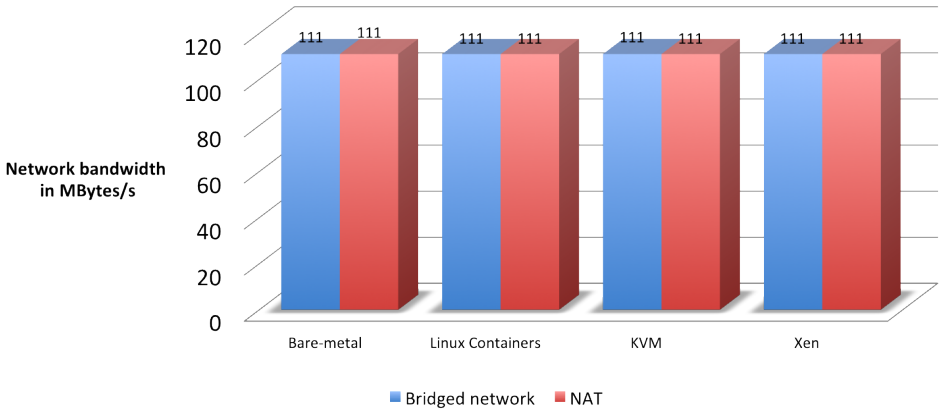


Figure 4.8: Network virtualization overhead - Network bandwidth

It is evident from the results that there is no observable overhead caused by the virtualizing the network interfaces using KVM, Xen and Linux containers.

4.5 Disk I/O overhead

KVM, Xen, and Linux containers handle disk I/O differently. Xen, being a para-virtualized platform, performs disk I/O with the help of para-virtual drivers. By design, Xen does not expose

the disk devices to the virtual machines. When the system boots, dom0 (privileged domain) which has privileged access to the hardware, sets up the driver domains, which then serve as the hardware interfaces for all virtual machines that require disk access. Communication between the virtual machines and the driver domains occurs in memory. The software layer between the virtual machines and the physical layer driver leaves room for interesting optimization strategies in terms of shared memory, virtual interrupts, and grant tables, thereby reducing disk I/O overhead.

KVM, as a full-virtualization platform, uses QEMU to create virtual devices that map to physical hardware. This mapping introduces additional overhead, as disk access is dependent on QEMU, which is single threaded, presenting scalability limitations. The "Big QEMU lock" as referred in the Linux community is an expensive mechanism in the path to the disk [6]. More recently, development efforts have shifted to para-virtual drivers for KVM. Currently, KVM supports two para-virtual drivers, virtio-blk [61], and virtio-scsi [62]. Virtio drivers could also be used to access file-backed storage. For example, a virtual machine can access a file on the host operating system as a virtual disk using the virtio drivers. At the time of writing, development efforts are focused on "virtio-blk-data-plane", which provides an accelerated data path for para-virtual drivers using dedicated threads and Linux Aio and entirely bypassing the QEMU block driver [41]. The system under evaluation for this thesis runs the stable version of QEMU and uses virtio drivers to access a file-backed virtual disk.

Linux containers take the simplest route to implement disk I/O. A container's storage is represented as another directory on the host file system. Hence each container uses the host's file system in an isolated fashion. The containers under evaluation for this thesis utilize a separate logical volume on the host to create its root file system.

To evaluate the overhead caused by virtualizing disk I/O, we created a sample application (w4) that uses sysbench [43], ioping [40], and dd to perform sequential and random disk I/O. The objective of this test is to observe the execution time of the sample application and its individual tasks as mentioned above. The test was performed on a virtual machine created on each of the three virtualized servers, and also on an independent bare-metal system used as a baseline. The virtual machine was configured with 8 virtual CPUs mapped to 8 physical CPUs, 300 GB of disk storage and 7680 MB of memory.

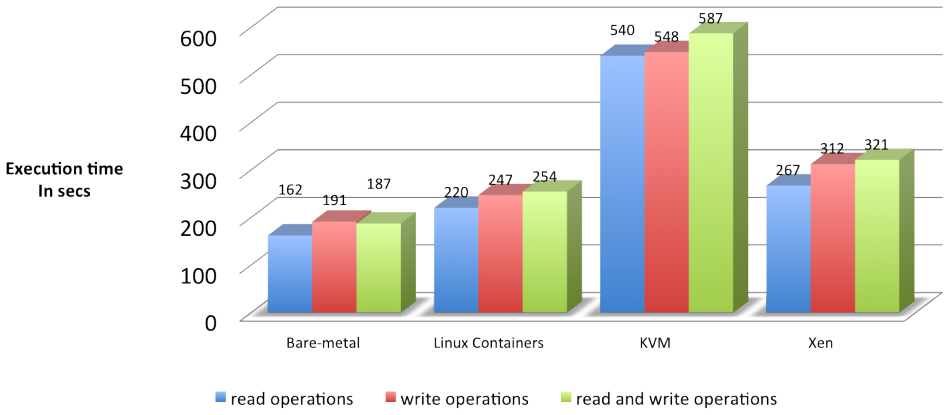


Figure 4.9: Virtualization overhead - sequential file I/O

Figure 4.9 summarizes the execution time for performing 10,000 sequential reads, sequential writes, and mixed sequential reads and sequential writes. It is evident from the results that KVM exhibits a significant overhead in virtualizing sequential I/O operations. Linux containers exhibit the least overhead as it does not involve any device emulation or additional drivers.

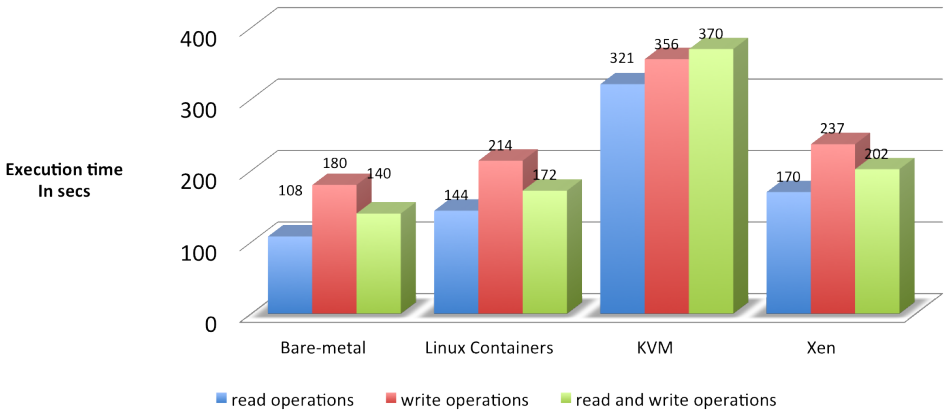


Figure 4.10: Virtualization overhead - random file I/O

Figure 4.10 summarizes the execution time for performing 10000 each of random reads, random writes, and mixed random reads and random writes. Though KVM exhibits an improved

performance for random I/O operations than sequential I/O operations, it is still the worst performer in among the solutions being evaluated. Once again, Linux containers exhibited the least overhead in performing random I/O operations.

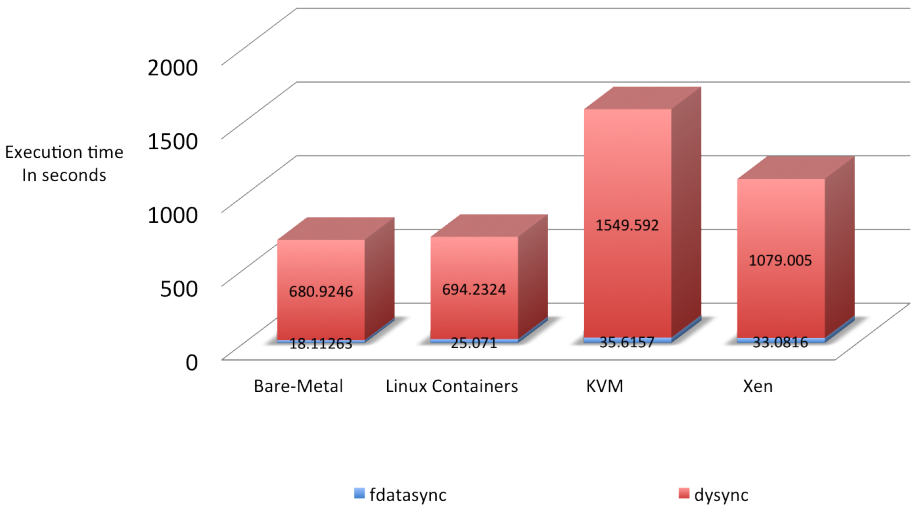


Figure 4.11: Virtualization overhead - Disk copy performance

To evaluate the overhead caused by virtualization in copying files within the disk, we used the dd tool to copy a set of 1024 MB files from one location to another with fdatasync flag (physically write output file data before finishing) and dsync flag (use synchronized I/O for data). The results as summarized in Figure 4.11, conform to our earlier results that KVM exhibits highest overhead and Linux containers exhibits the least. Based on all the results discussed above, it can be concluded that Linux containers perform the best with respect to virtualizing disk I/O operations.

Chapter 5

Operational Flexibility

The choice of the virtualization platform is important, as it influences the design of the application and service management infrastructure. In the previous chapters, we discussed the design of the three representative virtualization platforms and quantitatively analyzed the platforms based on their raw performance, as well as the overhead they impose. However, there are other dimensions that are harder to quantify, but deserve careful consideration to determine if a platform matches the operational needs of an application infrastructure.

This chapter provides a qualitative analysis of KVM, Xen, and Linux containers by answering several questions. We believe these questions are crucial; the responses must be factored in when making the final decision of which virtualization platform to use.

1. How do the representative virtualization platforms compare with respect to the time they require to perform tasks like virtual machine provisioning, cloning, boot, and reboot ?
2. What are the features and operational constraints associated with each of the platforms ?
3. What are the management facilities and tool sets available ?
4. How mature and time-proven are the platforms ?
5. How complex is the design of the virtualization solution from the perspective of setting up and customizing to specific needs ?

5.1 Operational Metrics

A key characteristic of any virtualization solution is its operational performance for common execution tasks, as well as the flexibility to customize them. Let us consider two real-world examples of what enterprises expect out of the virtualization systems to base our analysis to answer their needs. First, a Platform-As-A-Service(PAAS) vendor who leases their computing capacity to their customers based on demand, needs the virtualization solution to be extra-ordinarily dynamic. Being able to dynamically instantiate new virtual machines, bringing them down, and the ability to scale a virtual machine's resources up and down are very important. Second, an e-commerce company that favors more reliable, standard appliances cares more about resource isolation, security, and device compatibility than the dynamic re-configuration capabilities.

To evaluate the operational dynamics of KVM, Xen, and Linux containers, we measured the amount of time each platform takes to perform the following tasks: (i) Provision a new virtual machine with a pre-defined configuration and install Ubuntu 12.04.3 server operating system on it using an automated installation procedure [73]. (ii) Start the execution of a virtual machine and boot the installed operating system completely. (iii) Reboot a virtual machine that is currently in operation. (iv) Clone a virtual machine and all its resources to create a new virtual machine.

Virtualization solution	Time taken to provision a new virtual machine and perform an automated installation of Ubuntu 12.04.3 server operating system (seconds)
KVM	532
Xen	718.5
Linux containers	27.502

Table [] shows that provisioning a Linux container is several times faster than provisioning a virtual machine through Xen or KVM. As discussed in chapter 3, Linux containers share the kernel with the host operating system. Hence, provisioning a new container does not involve the installation of a new kernel. By design, the root file system of the guest operating system is cached on the host and is copied whenever a new container is created. Unlike KVM and Xen, Linux containers does not create virtual devices for the containers, but only initialize their own name space in the host kernel and use cgroups for resource management. The unique design of the Linux containers gives it the advantage of being very fast in creating a new container. Though both KVM and Xen creates

virtual devices along with the virtual machine, and installs the operating system from the scratch, we observed KVM to be faster than Xen in completing the installation process and turning operational.

Virtualization solution	Time taken for cold start (seconds)	Time taken to reboot (seconds)
KVM	5.86	38
Xen	5.34	11
Linux containers	0.053	10

Table [] shows our observations of the time each platform takes to complete the guest operating system boot process. Our results show that a Linux container starts up and becomes available for use in a fraction of the time taken by KVM, and Xen. Since the container does not have a kernel to boot, it just spawns as a group of processes on the host kernel and becomes available very quickly. Similarly, the reboot process of a container is very quick since, it only has to de-allocate its resources on the host operating system and start afresh.

Virtualization solution	Time taken to clone a virtual machine from a disk image (seconds)
KVM	35
Xen	215
Linux containers	0.553

In real world use-cases, it is common practice to create a standard virtual machine (also referred as golden-image) with all the site-specific customization, and clone the golden-image whenever a new virtual machine is provisioned. Table [] shows the time taken by each of the virtualization platforms to create new virtual machine by cloning the golden-image. The results once again favor the Linux containers by a big margin. The big difference between the results of KVM and Xen is due to the format of the storage used by KVM. KVM used a copy-on-write (QCOW2) format as the default storage resulting in smaller disk footprint, whereas Xen used a raw disk image by default resulting in occupying the entire provisioned disk size.

The results mentioned above supports our earlier discussion that Linux containers trades the isolation achieved by running an individual kernel for each of the virtual machines to provide operational benefits. Hence, Linux containers are most suitable for situations demanding dynamic creation and deletion of virtual machines. KVM and Xen would be more appropriate for environ-

ments that require superior isolation and run for long time without needing reboots. It is imperative that only KVM and Xen can virtualize the applications that require kernel customization as a linux container cannot modify the host kernel individually.

Another important factor from the paas vendor example is the resource footprint of the virtual machines. The lighter the foot print of the virtual machines are, more virtual machines can be accommodated on a single server leading to effective resource utilization and lower costs. The disk storage on all of KVM, Xen, and Linux containers are either file-backed or device backed which does not vary by the virtualization platform. However, the memory footprint of the virtual machines and the containers are a key factor. In order to provide superior isolation between the virtual machines, KVM, and Xen provides for them to run their own kernel, paging, and caching mechanisms. Since the virtual machines behave like an independent systems themselves, they create a considerable memory footprint on the host. But, Linux containers are just a set of regular linux processes and cause almost negligible memory footprint when the containers are idle. This feature of the Linux containers, makes it an ideal platform of choice for the web hosting providers, who aim to serve as many individual websites as possible on a single physical server. Though not as light as the containers, KVM allows for memory over-commitment, which makes it simpler to accommodate the dynamic memory requirements of the virtual machines.

5.2 Operational features and constraints

Feature/constraint	**Linux Containers**	**KVM**	**Xen**
Virtual machines running an operating system different than the host	**No**. (Different distribution of GNU Linux may be run, but the kernel must be shared with the host)	**Yes**. (Several guest operating systems are supported, including Microsoft Windows)	**Yes.** (Several guest operating systems are supported, including Microsoft Windows)
Live migration of virtual machines between physical servers	**Yes.** (A running container can be checkpointed to files and restored on a different machine [55])	**Yes.**	**Yes**.
Memory over commitment (configure more memory for virtual machines than is available on the host)	**Yes**. (backed by virtual memory on the host)	**Yes**. (backed by virtual memory on the host)	**No**. (para-virtual virtual machines)

Feature/constraint	**Linux Containers**	**KVM**	**Xen**
CPU over commitment	**Yes**. (CPUs are shared by default, controlled by cgroups)	**Yes**. (limited by the max-vcpus configuration)	**Yes**. (limited by configuration parameters)
Leverage hardware extensions in CPU	**No**.	**Yes**. (Required)	**No** (para-virtualized VMs)
Memory density	**High** (Lowest memory foot print. More containers can run simultaneously)	**Low** (Higher memory footprint, Kernel + page tables)	**Low** (Higher memory footprint, Kernel + page tables)
Isolated kernel updates	**No.** (Updates to the host kernel impacts all the containers)	**Yes.** (host kernel and guest kernel are fully isolated and can be individually updated)	**Yes**. (host kernel and guest kernel are fully isolated and can be individually updated)
Code base being part of the supported mainline linux kernel	**Yes.**	**Yes.**	**Yes**. (Linux with mainline kernel can run as Dom0 and DomU. But the hypervisor is still separate)

5.3 Management facilities

KVM, being part of the mainline kernel for a long time, is seen as a standard virtualization solution from the Linux community. The KVM system exposes a standard virtualization interface by conforming to the libvirt API. There are a plethora of management tools and cloud solutions already developed around the libvirt API. The most common management tools that are used to manage a KVM based server farm are, OVirt, Virt-manager, ConVirt, Apache CloudStack and OpenStack. Xen, also supports the libvirt API, therefore manageable through the standard tools mentioned earlier. A commerical management tool specific to Xen is XenCenter [13]. Linux containers can be managed through a set of command line tools known as lxc, or through the libvirt API. Cloud solutions like OpenStack and CloudStack has recently added support for linux containers to be managed just like the virtual machines. A simple web-based management tool, lxc-web [79] can be used to manage small and medium sized installations of linux containers.

5.4 Maturity and commercial support

The most mature and time-tested of the three virtualization solutions, is Xen, with its first stable release made in 2003. Though Xen enjoys the support of several major hardware and software vendors, Citrix acquired XenSource Inc and now provides commercial support to several Xen based virtualization solutions. KVM, has been part of the Linux kernel since version 2.6.20 and enjoys close integration with the linux kernel and the testing and support offered by the Linux community at large. KVM is part of most of the commercially supported linux distributions including RedHat Enterprise Linux [28], Ubuntu, and Suse Linux Enterprise Server. KVM is also notably forms the base of the commercial product RedHat Enterprise Virtualization(RHEV). Linux containers, as part of the kernel is supported by most of the commercial Linux distributions. The containers are the newest of the three virtualization solutions and is maturing very fast at the time of writing. Linux containers are notably behind the technical platform of PAAS vendor, Heroku and is also extensively used in the commmercially supported open source product, OpenShift, by RedHat.

Chapter 6

Resource Entitlement

In many ways, a virtualized server is analogous to an apartment building. Several virtual machines are provisioned in a physical machine, just like several apartments are owned or rented in an apartment building. The hallways and the other common spaces are compactly designed under the assumption that not all of the residents would use them at the same time. Similarly, in a virtualized system, the core hardware resources including the CPU, memory, network bandwidth, and the disk I/O are shared among the virtual machines under the assumption that not all virtual machines would need all the mentioned resources at the same time. Virtualization of the critical system resources leads to effective utilization of hardware and improving the cost-effectiveness. A sample virtualized system running heterogeneous applications is shown in Figure 6.1 to represent the scheme of higher utilization. The virtual machines, which would have under utilized a host's CPU individually, effectively share the CPU in a virtualized environment leading to higher utilization of the CPU.

Drawing from our analogy, every apartment in the apartment building, expects a certain degree of privacy and isolation. The presence of over-consuming or noisy neighbors negatively impacts the peace and harmony of the building. Similarly, every virtual machine, requires a degree of isolation from other virtual machines in order to fulfill its own service requirements. For example, consider a system in which four physical CPUs are shared among four virtual machines. If one of the virtual machines turns into a CPU hog and tries to utilize all the four CPUs accessible to it, the other virtual machines start to starve for CPU even though they hold other resources like memory, network and disk I/O. Such situations must be avoided and the individual machines must

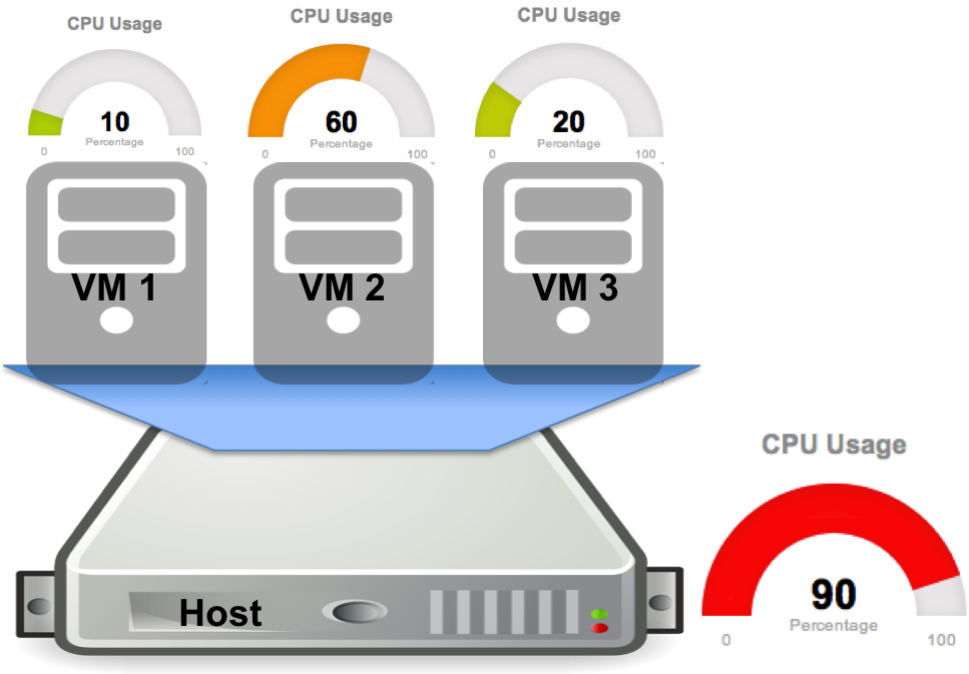
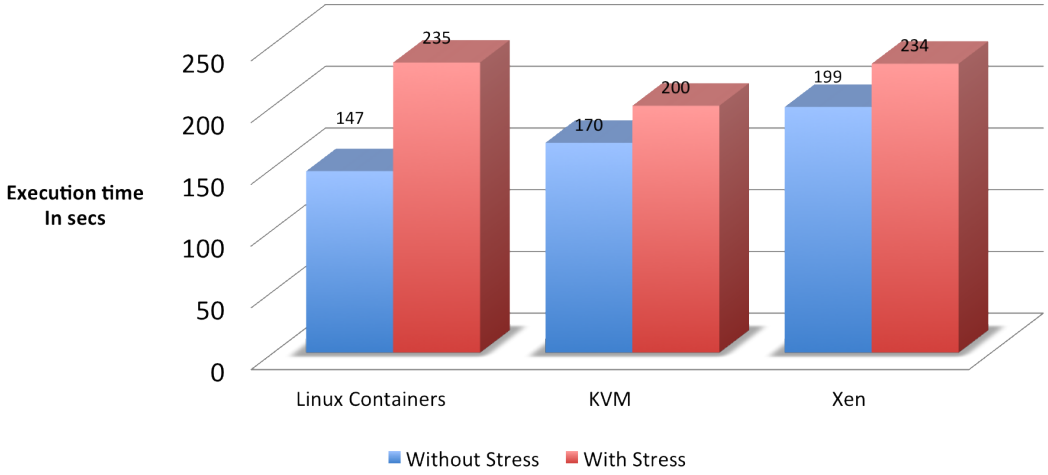


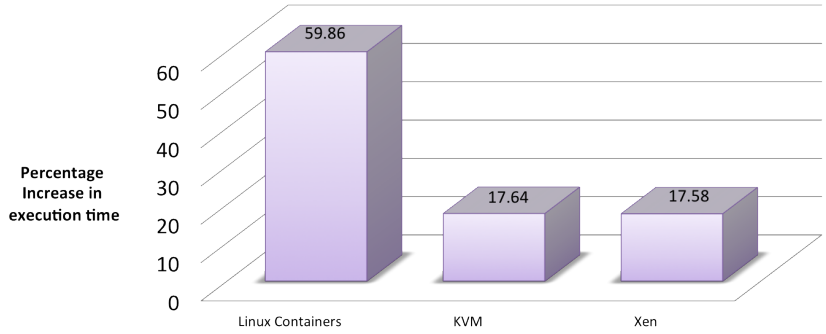
Figure 6.1: Virtualization - CPU utilization perspective

be guaranteed their minimum entitled share of the resources at all times with the implementation of a proper resource sharing strategy. This chapter analyzes the isolation effectiveness available by default on all the representative virtualization solutions and also describes the available resource entitlement mechanisms that can be used.

To evaluate the resource isolation efficiency of KVM, Xen and Linux Containers, we wanted to measure the impact of an over-stressed virtual machine on other virtual machines running on the same host. We created two identical virtual machines on each of the virtualized servers, sharing all of the host's resources among them, and then measured the change in execution times of the sample applications (w1, w2, w3, and w4) running on a virtual machine, when stress was created on another virtual machine using stress [**?**] tool.



(a) Impact of stress on sample application w1

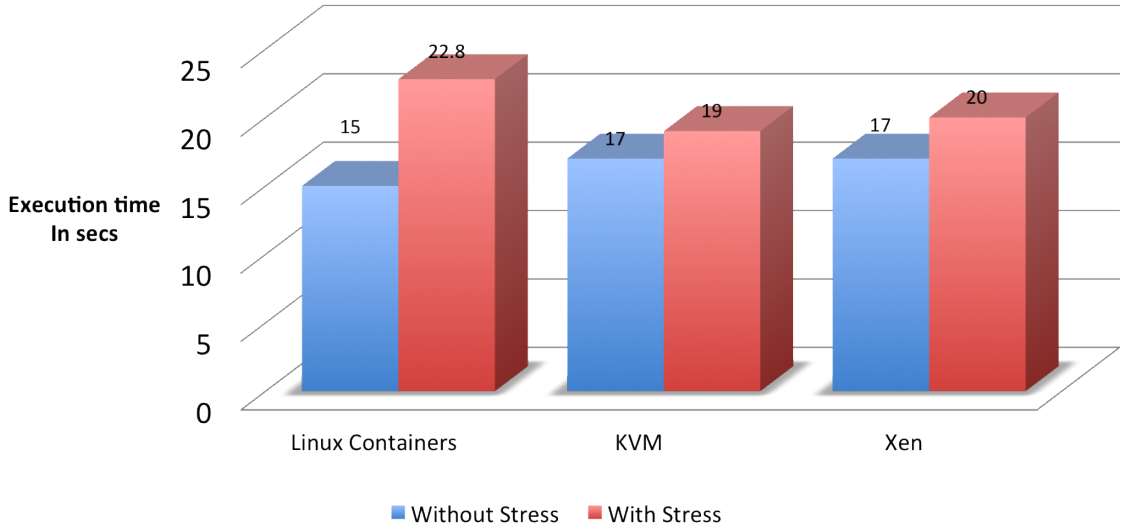


(b) Impact of stress in percentage increase

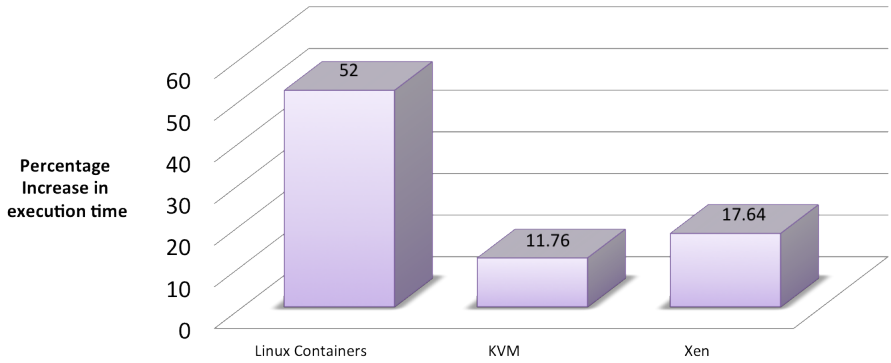
Figure 6.2: CPU Isolation Efficiency

Figure 6.2a summarizes the change in execution times of the sample application w1 (performing CPU intensive tasks), on each of the virtualization platforms. X-axis represents the virtualization platforms, and Y-axis represents the execution times of the application before and after stress was applied. Based on the results shown in Figure 6.2a, percentage increase in execution times was calculated for each platform and summarized in Figure 6.2b. The percentage increase in execution time is inversely proportional to the CPU isolation efficiency offered by the virtulaization platform.

In other words, In an effectively isolated environment, the increase in execution time will be the minimal. Hence, it is evident from the results that KVM and Xen offer superior CPU isolation for the virtual machines, while Linux containers are poorly isolated with respected to CPU.



(a) Impact of stress on sample application w2

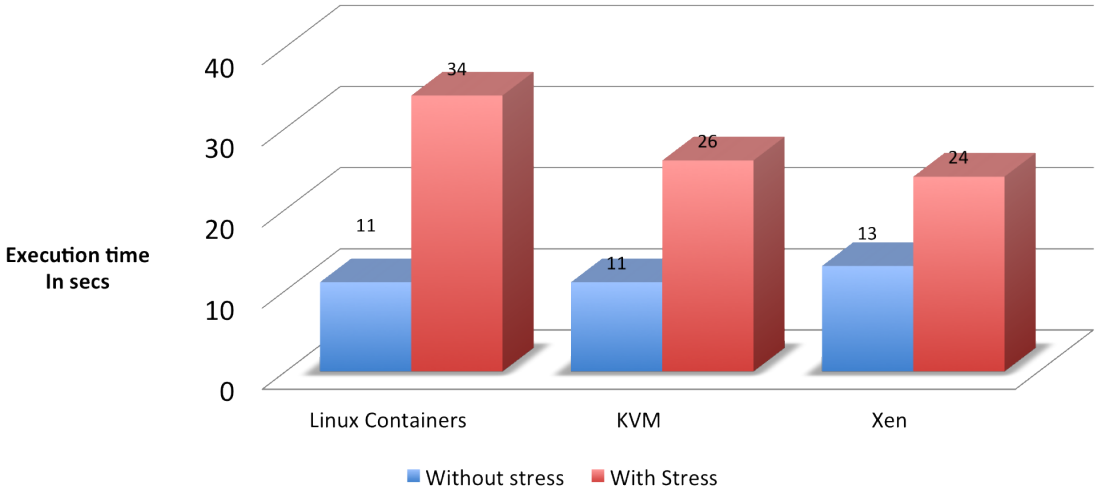


(b) Impact of stress in percentage increase

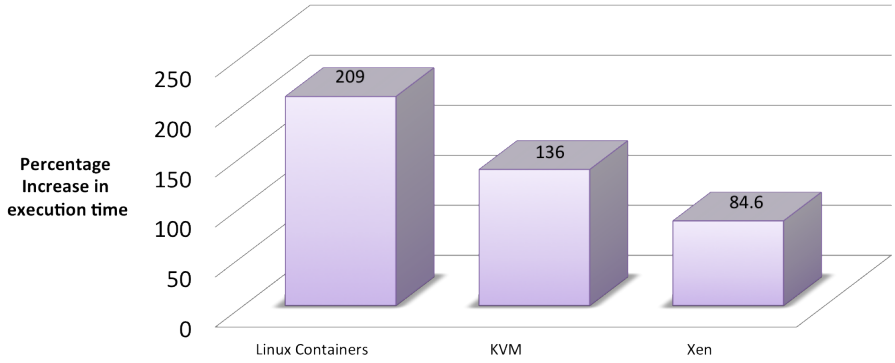
Figure 6.3: Memory Isolation Efficiency

Figure 6.3a summarizes the change in execution times of the sample application w2 (performing memory intensive tasks), on each of the virtualization platforms. X-axis represents the virtualization platforms, and Y-axis represents the execution times of the application before and after stress was applied. Based on the results shown in Figure 6.3a, percentage increase in execution times was calculated for each platform and summarized in Figure 6.3b. The percentage increase in execution time is inversely proportional to the memory isolation efficiency offered by the virtulaiza-

tion platform. In other words, In an effectively isolated environment, the increase in execution time will be the minimal. Hence, it is evident from the results that KVM offers superior memory isolation for the virtual machines, while Linux containers are poorly isolated with respected to memory.



(a) Impact of stress on sample application w3

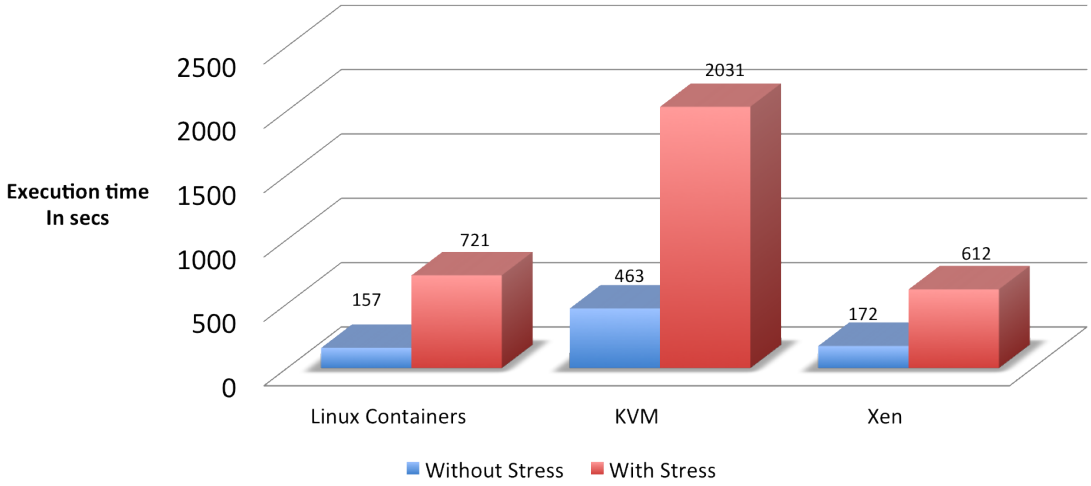


(b) Impact of stress in percentage increase

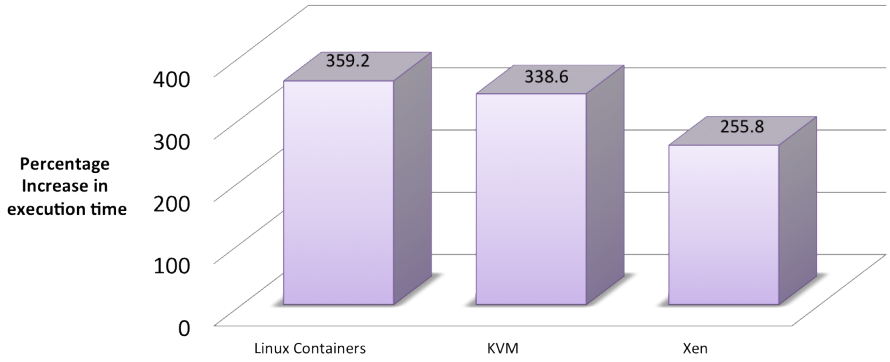
Figure 6.4: Network Isolation Efficiency

Figure 6.4a summarizes the change in execution times of the sample application w2 (performing network I/O operations), on each of the virtualization platforms. X-axis represents the virtualization platforms, and Y-axis represents the execution times of the application before and after stress was applied. Based on the results shown in Figure 6.4a, percentage increase in execution times was calculated for each platform and summarized in Figure 6.4b. The percentage increase in execution time is inversely proportional to the isolation efficiency of network I/O operations offered by the virtualization platform. In other words, In an effectively isolated environment, the increase

in execution time will be the minimal. Hence, it is evident from the results that Xen offers superior network isolation for the virtual machines, while Linux containers are poorly isolated with respected to network bandwidth.



(a) Impact of stress on sample application w4



(b) Impact of stress in percentage increase

Figure 6.5: Disk I/O Isolation Efficiency

Figure 6.5a summarizes the change in execution times of the sample application w4 (performing large number of disk I/O operations), on each of the virtualization platforms. X-axis represents the virtualization platforms, and Y-axis represents the execution times of the application before and after stress was applied. Based on the results shown in Figure 6.5a, percentage increase in execution times was calculated for each platform and summarized in Figure 6.5b. The percentage increase in execution time is inversely proportional to the efficiency of isolating disk I/O operations, offered by the virtulaization platform. In other words, In an effectively isolated environment, the increase in

execution time will be the minimal. Hence, it is evident from the results that Xen offers superior isolation of disk I/O operations for the virtual machines, and once again, Linux containers are poorly isolated with respected to disk I/O operations.

The results summarized in 6.2, 6.3, 6.4, and 6.5 shows that sharing all the available resources among the virtual machines without any reservations or priorities yields unpredictable performance.The following section describes the facilities available in a Linux system, and the hypervisors to define limits and assure virtual machines of their entitlement in terms of CPU, memory, network bandwidth and disk I/O.

Before considering the mechanism to setup dedicated system resources for virtual machines, it is very important to analyze and understand the nature of the workload that the virtual machine is about to run. The knowledge acquired by this analysis would help in two regards:

1. To understand the broader practical workload requirements, which can be used in deciding which virtual machines to provision on the same server. For example, A virtual machine that runs a web server (uses more network and memory resources) and a virtual machine that runs reporting application (more disk I/O) are ideal candidates to be virtualized on the same physical server as they utilize different types of resources. Whereas, running all virtual machines running database workloads, on the same physical machine would not be a good practice as they would contend for the same type of resources.

2. To choose the specifics of what resources to dedicate and what type of resources to share.

Planning the resource policies for the CPU and memory resources always goes hand in hand, as the primary rule of thumb in achieving better performance is to feed the processors with data as fast as possible by keeping the data in memory regions closest to them. To plan the CPU and memory resource allocations, it is important to know their architecture in the physical server. All the modern server class hardware are equipped with multi-core processors and are architected to efficiently scale by distributing system memory near the individual CPUs. This configuration is known as *Non-Uniform Memory Access(NUMA)*. Figure 6.6 shows a block diagram of a hex-core physical server in a 2-node NUMA configuration.

Where, CPU Socket-0 and CPU Socket-1 consists of two physical CPUs, each holding four processing cores. All four processing cores in each CPU-Socket share a common L3 cache memory. The memory (RAM) on NUMA node-0 can be accessed faster by the processing cores contained in

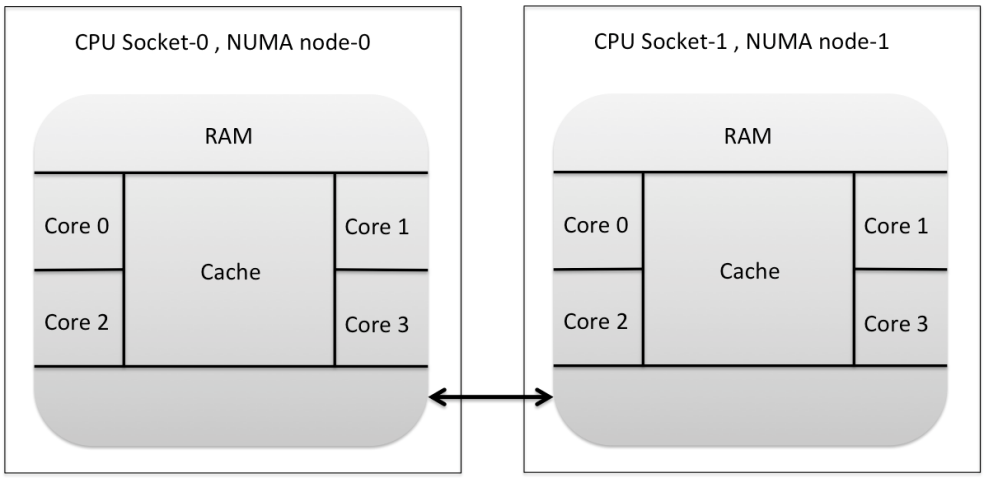


Figure 6.6: Hex-core CPU configuration (2 sockets, 2 node NUMA)

the CPU Socket-0. The memory (RAM) on NUMA node-1 can be accessed faster by the processing cores contained in the CPU Socket-1.

The memory accesses across NUMA nodes will be relatively slower (e.g., processing core in CPU Socket-0 accessing the memory in NUMA node-1), and does not make efficient use of the L3 cache. This level of architectural information about the CPU and the memory hardware can be obtained using the following commands,

lscpu [9] - Used to gather CPU architecture information from sysfs and /proc/cpuinfo

numactl –hardware [42] - Used to display inventory of available nodes on the system.

While analyzing the workloads with respect to CPU and memory entitlements, answering the following questions will greatly help.

1. Is the application is multi-threaded ?
2. If yes, are multiple threads doing independent CPU-intensive tasks or are dependent on each other and shared memory resources ?

By default, the Linux scheduler tries to keep all the CPUs busy by moving tasks from overloaded CPUs to idle CPUs. This may be detrimental to the performance of the VMs on NUMA based hardware, as a guest cannot take advantage of its accumulated state in the processor, including the instructions and data in the local cache [72]. If the applications are single-threaded, the CPU reservations can be straight forward; “pinning” or assigning an “affinity” to the CPU to a physical CPU core would greatly benefit the application running on the virtual machine. CPU pinning or

affinity refers to assigning a specific process or virtual machine to a particular CPU core, such that the virtual machine would always be scheduled only on that particular CPU core. In a carefully planned system, if all the virtual machines are limited to appropriate set(s) of CPU cores, the virtual machines can run on their dedicated cores with least interruptions leading to best practical performance. As an aside, access to an additional CPU core that may be shared across the system would help in alleviating interruptions by offloading the threads performing operating system maintenance activities.

If the workload is multi-threaded, and the threads are performing mostly independent CPU-intensive tasks, the ideal policy would be to distribute them across as many processing cores as possible. This will improve parallelism and hence improve performance. On the other hand, pinning such virtual machines running multiple CPU intensive threads, to single processing core would result in sub-optimal performance. If the workload is multi-threaded, and the threads are performing memory intensive tasks with shared resources, the ideal policy would be to make sure they are scheduled on the same NUMA node. So that, they can utilize multiple processing cores, yet, share the local cache memory efficiently. CPU pinning and NUMA assignment as discussed above can be implemented using the following set of commands.

- **Taskset** [53] - To retrieve the CPU affinity of a running process, given its PID, or to launch a new command with a given CPU affinity.
  Example, `taskset -c 0,2,4 kvmtest1`
  would pin the kvmtest1 process to the cores, 0, 2, and 4.

- **Numactl** [42] - To operate on a higher abstraction than taskset, and can be used to confine a process to a NUMA node. This helps to prevent virtual machines from being scheduled across NUMA pools.
  Example, `numactl --cpunodebind=0 --membind=0 kvmtest1`
  would bind the guest kvmtest1 to the first CPU Socket and also restrict memory use to the associated NUMA pool.

Though CPU pinning and NUMA node assignment helps greatly towards limiting virtual machines onto confined resource pools, it may be impractical in dense servers, where the number of virtual machines are much more than the number of processing cores. In such conditions, a granular resource management mechanism, cgroups(control groups) (discussed earlier) fits nicely. cgroups are a linux

kernel feature, that allows limiting and accounting of resources at a granular level. cgroups enables us to assign relative CPU shares to the virtual machines, indicating their priorities in accessing CPU resources, while still obeying the policies set by taskset and numactl.

The easiest method to achieve network isolation among the virtual machines, is to dedicate individual network interface hardware on the host to virtual machines. This method is often justified by the relatively low cost of network interface cards. In situations where the network bandwidth needs to be shared among the virtual machines, cgroups can be used to classify the virtual machines into groups, and dynamically assign priorities to them.

Implementing entitlement policies for disk I/O operations are relatively simpler, as in most cases a physical server would contain several hardware adapters for the locally attached storage, or individual block devices that are dedicated for the virtual machines. The disk I/O operations that a virtual machine performs can be rate-limited specifically for reads or write operations by individual processes using cgroup hierarchies. The "blkio" subsystem of cgroups, enable throttling and accounting of I/O operations across the linux I/O scheduler. On a performance note, the virtual machines tend to perform relatively better with the deadline I/O scheduler with the "nocache" option than the default CFQ I/O scheduler due to the avoidance of double caching.

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] Daniel P. Berrange. Rfc: Death to the bqdl (big qemu driver lock). `http://www.redhat.com/archives/libvir-list/2012-December/msg00747.html`, November 2013 (Last Accessed).

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

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

[9] Heiko Carstens Cai Qian, Karel Zak. lscpu - display information about cpu architecture. `http://www.dsm.fordham.edu/cgi-bin/man-cgi.pl?topic=lscpu&ampsect=1`, October 2013 (Last Accessed).

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

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

[12] 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.

[13] Citrix. Xencenter. `http://www.xenserver.org/overview-xenserver-open-source-virtualization-download.html`, October 2013 (Last Accessed).

[14] 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.

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

[16] 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.

[17] 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).

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

[19] Linux Kernel documentation. Overview of linux capabilities. http://linux.die.net/man/7/capabilities, November 2013 (Last Accessed).

[20] Ian Smith et al. byte-unixbench, a unix benchmark suite. https://code.google.com/p/byte-unixbench/, November 2013 (Last Accessed).

[21] Jon Dugan et al. iperf - tcp and udp bandwidth performance measurement tool. https://code.google.com/p/iperf/, November 2013 (Last Accessed).

[22] Mark J. Cox et al. Openssl - cryptography and ssl/tls toolkit. http://www.openssl.org/docs/apps/openssl.html, November 2013 (Last Accessed).

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

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

[25] 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.

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

[27] 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).

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

[29] 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).

[30] Andras Horvath. mbw. http://manpages.ubuntu.com/manpages/lucid/man1/mbw.1.html, October 2013 (Last Accessed).

[31] 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).

[32] 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).

[33] 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).

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

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

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

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

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

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

[40] Konstantin Khlebnikov. mbw. https://code.google.com/p/ioping/, October 2013 (Last Accessed).

[41] Andrew Theurer Khoa Huynh and Stefan Hajnoczi. Kvm virtualized i/o performance. ftp://public.dhe.ibm.com/linux/pdfs/KVM_Virtualized_IO_Performance_Paper.pdf, November 2013 (Last Accessed).

[42] Andi Kleen. numactl - control numa policy for processes or shared memory. http://linux.die.net/man/8/numactl, October 2013 (Last Accessed).

[43] Alexey Kopytov. sysbench. http://sysbench.sourceforge.net/, October 2013 (Last Accessed).

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

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

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

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

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

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

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

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

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

[53] Robert M. Love. taskset - retrieve or set a process's cpu affinity. http://linux.die.net/man/1/taskset, October 2013 (Last Accessed).

[54] Linux kernel Open source. Kernel samepage merging. http://www.linux-kvm.org/page/KSM, November 2013 (Last Accessed).

[55] OpenVZ. Checkpoint and restore facility for linux in user space. http://criu.org/Main_Page, October 2013 (Last Accessed).

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

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

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

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

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

[61] qemu kvm. Virtio-blk latency measurements. http://www.linux-kvm.org/page/Virtio/Block/Latency, November 2013 (Last Accessed).

[62] qemu kvm. Virtio-scsi overview. http://wiki.qemu.org/Features/VirtioSCSI, November 2013 (Last Accessed).

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

[64] 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).

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

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

[67] 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.

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

[69] 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.

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

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

[72] Martin Thompson. Processor affinity. http://mechanical-sympathy.blogspot.com/2011/07/processor-affinity-part-1.html, October 2013 (Last Accessed).

[73] Ubuntu. Automatic installation of ubuntu using kickstart. https://help.ubuntu.com/12.04/installation-guide/i386/automatic-install.html, October 2013 (Last Accessed).

[74] 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.

[75] 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.

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

[77] 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.

[78] 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.

[79] lie Deloumeau. Lxc web panel. http://lxc-webpanel.github.io/, October 2013 (Last Accessed).