12-2013

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

Aravindh Sampathkumar
Clemson University, aravins@g.clemson.edu

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 computing resources, enabling applications to scale with varying demands over time. The core enabler of the cloud computing paradigm is *virtualization*. The concept is not new; virtualization has garnered significant research attention, fostering a race to achieve the lowest possible virtualization overhead. 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 three virtualization approaches.

This thesis evaluates representative implementations of each approach: (i) full-virtualization - *KVM*, (ii) para-virtualization - *Xen*, and (iii) container-based virtualization - *Linux Containers*. First, this thesis discusses the design principles behind the chosen virtualization solutions. Second, this thesis evaluates the solutions based on the overhead they impose to virtualize system resources. Finally, this thesis assesses the benefits and limitations of each solution 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 system architects in selecting the best virtualization platform for a given application.

Dedication

This work is dedicated to my parents, my sister, and many friends whose love, belief, and support enables me to pursue my dreams.

Acknowledgments

I would like to gratefully and sincerely thank Dr. Jason O. Hallstrom, for his advice, encouragement, and support in guiding my research and writing of this thesis. I would also like to thank Dr. Sebastien Goasguen, for his guidance and instilling the idea that has taken form though this research work.

I would like to acknowledge my thesis committee, Dr. Amy Apon, and Dr. Brian A. Malloy for their support, expertise and time to better my work.

This research was supported by the U.S. National Science Foundation under awards CNS-0745846, and CNS-1126344.

I would also like to thank all my friends at Dependable Systems Research Group for making my graduate school life enjoyable through the fellowship they provided.

Finally, a special thanks to my family and friends for making me who I am today.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The adoption of the cloud computing paradigm has changed the perception of 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 [36].

The technology backbone behind cloud computing is *virtualization*. Virtualization is the process of creating multiple isolated operating environments that share the same physical server resources. Virtualization addresses the problem of under-utilization of hardware resources observed in the dedicated server model by running multiple isolated applications simultaneously on a physical machine, also referred to as the "host". The isolated operating environments are referred to as virtual machines (VMs) [89], or containers [64]. Each virtual machine or container provides an abstracted hardware interface to its applications and utilizes computational resources as needed (or available) from the host.

Virtualization gained widespread adoption because of the many benefits it offers. First, virtual machines can be statefully migrated from one physical machine to another in the event of a hardware failure [13]. This capability can also be used to dynamically reposition virtual machines in such a way that virtual machines that frequently work together are virtualized on the same physical server, improving their performance. Virtual machines can also be automatically migrated to achieve a pre-defined goal, such as power efficiency or load balancing [43]. Second, virtual machines can be administered more flexibly than physical servers, as they are usually represented as files, making

them easier to clone, snapshot, and migrate. Virtual machines can also be dynamically started or stopped without affecting other virtual machines on the host. Finally, virtualization enables an additional layer of control between the applications and the hardware, providing more options to effectively monitor and manage hardware resources.

Virtualization can be implemented in multiple ways, such as creating virtual devices that simulate the required hardware, which are in turn mapped to physical resources on the host, or by restricting the physical resources available to each virtual machine using resource management policies provided by the host. The strengths and weaknesses exhibited by the virtualization platforms that implement these strategies vary based on their design objectives and implementation. These variations makes it important to understand the operation of each virtualization platform before choosing a platform to virtualize a given application infrastructure.

1.1 Motivation

The Intelligent River® is a large-scale environmental monitoring system being built by researchers at Clemson University [98]. The effort aims to deploy a large and highly distributed wireless sensor network in the Savannah River Basin. The back-end infrastructure uses a real-time distributed middleware system to receive, analyze, and visualize the sensor observation streams. The time sensitivity of the observation streams, along with the anticipated scale of the Savannah deployment demands that the middleware system be flexible, fault-tolerant, and scalable. A key factor in architecting such a dynamic system is virtualization.

Given the availability of multiple open source virtualization platforms, such as KVM [92], Xen [29], and Linux Containers [64], the choice of virtualization platform is not obvious. We must identify the most effective platform to virtualize the Intelligent River® middleware system. This important choice requires a detailed comparative analysis of the available virtualization platforms. Prior work on the analysis of virtualization platforms for High Performance Computing (HPC) applications [101] claims that KVM performs better than Xen based on the HPCC benchmarks [59]. On the other hand, other prior work claims that OpenVZ [79], a virtualization platform based on Linux Containers performs the best, while Xen performed significantly better than KVM, clearly contradicting prior claims. These (and other) contradictions in prior work indicate that comparing virtualization platforms based only on standard benchmarks is not sufficient to arrive at a definitive

optimality decision. Several other factors must be considered. This motivated us to perform : (i) an operational analysis of the virtualization platforms, (ii) a quantitative analysis of virtualization overhead, (iii) a workload-specific benchmark analysis, and (iv) a qualitative analysis of operational flexibility.

1.2 Contributions

This thesis builds on prior work comparing open source virtualization platforms, but performs a more detailed study by comparing them beyond the standard benchmarks. The contributions of this thesis include (i) a detailed discussion of the operation of KVM, Xen, and Linux Containers, (ii) a quantitative comparison of the virtualization platforms based on the overhead they impose in virtualizing CPU, memory, network bandwidth, and disk access, (iii) a workload-specific benchmark analysis covering pybench, phpbench, nginx, Apache benchmark, pgbench, and others, (iv) a comparative analysis of performance specific to Intelligent River® middleware components, including RabbitMQ, MongoDB, and Apache Jena TDB, (v) a qualitative discussion of the operational flexibility offered by the virtualization platforms, and (vi) a discussion of strategies that can be adopted to enforce resource entitlement with respect to CPU, memory, network bandwidth, and disk access.

Chapter 2

Related Work

Virtualization was pioneered by IBM in the 1960's while developing CP/CMS (Control Program/Conversational Monitor System). CP/CMS has continued to evolve and is still in use in the form of z/VM [38], a hypervisor for IBM System Z (mainframes). The adoption of virtualization by x86-based systems triggered multiple open-source efforts leading to a diverse ecosystem of virtualization platforms. The major open-source virtualization implementations for x86-based systems are Xen, KVM, and Linux Containers, the focus of this thesis. These virtualization platforms are extensively used within several cloud computing technologies, including Amazon Elastic Compute Cloud [39], Apache CloudStack [27], OpenStack [81], Heroku [35], and OpenShift [44]. The widespread use and availability of source code for these platforms prompted a number of comparative studies. Younge et al. [101] compare Xen, KVM, and VirtualBox [78] for their viability in high performance computing environments, using virtual resources from the FutureGrid project [100], a testbed for cloud computing experiments. This study presents a feature comparison between KVM, Xen, and VirtualBox from an HPC perspective. The results show that KVM performs significantly better than Xen under the HPCC [59] and SPEC [93] benchmarks. Che et al. [12] introduce the operational principles of Xen, KVM, and OpenVZ [79], the predecessor of Linux Containers. This study compares and analyzes Xen, KVM, and OpenVZ based on macro-performance measures comprising standard system benchmarks. The results of this benchmark-oriented study show that OpenVZ performs exceptionally well on most of the benchmarks. The results also show that Xen outperforms KVM on almost all benchmarks, contradicting prior results. Xavier et al. [99] evaluate container-based virtualization solutions − Linux Containers, OpenVZ, and VServer [31] − in

a high performance computing context. The results show that Linux Containers outperform the other container-based solutions in terms of execution performance. The results also highlight the shortcomings in isolation provided by Linux Containers. An older study by Neri et al. [86] studies the difference in behavior among VServer, Xen, UML [18], and VMware [45] with respect to scalability and resource isolation. The results highlight the serious performance limitations of the Xen hypervisor under certain operational circumstances. Although prior work compares the virtualization platforms considered in this thesis, the prior results are inconsistent and contradictory, owing to variations in scope, evaluation criteria, and hardware. Also, all the results from prior work are based only on synthetic micro-benchmarks that do not necessarily correlate with the performance of practical applications. This thesis evaluates the performance of representative virtualization platforms for various categories of practical workloads, including a web server (nginx, Apache benchmark), a database server (MySQL, pgbench, SQLite), and runtime environments (Python, PHP, and Java).

Among the many new opportunities made possible by virtualization, Software Defined Networking (SDN) has attracted significant research attention. Rothenberg et al. [15] compare OpenVZ and Linux Containers from the perspective of implementing software-defined routing. The results highlight the performance benefits of Linux Containers in large-scale virtual network emulation using OpenVswitch [91]. Mesos [26] is a cluster platform that provides fine-grained resource sharing among multiple frameworks, including Hadoop [25] and MPI [14]. Mesos utilizes Linux Containers to isolate and limit the CPU, memory, network, and I/O resources, and serves as another practical example of the resource isolation techniques discussed in this thesis. Bardac et al. [3] showcase the scalability of Linux Containers by deploying a solution for testing large-scale peer-to-peer systems. To ensure fair comparison, the default configuration of Linux Containers was used in this thesis. However, the light-weight design of Linux Containers leaves room for interesting possibilities to deploy applications to containers. CoreOS [90], takes a minimalist approach to deploying applications by running only a trimmed down Linux kernel, along with systemd within each container, thereby eliminating redundant operating system functions, and decreasing virtualization overhead. Docker [40], an open source container management tool, also leverages the operational flexibility of Linux Containers discussed in this thesis.

Resource affinity, isolation, and control are discussed in detail in this thesis. Ahuja et al. [1] provide an example of exploiting CPU isolation facilities to improve overall network throughput by pinning application and protocol processing to the same physical core. The affinity analysis

discussed in this thesis will enable this work to be applicable to Linux Containers. The increase in performance of KVM shown by Hao et al. [30] serves as an example for the discussion of cgroups and CPU affinity in this thesis.

Chapter 3

Background

This chapter provides the necessary background required to understand the results of the comparative studies presented in subsequent chapters. A description of the design and operation of the representative virtualization platforms is presented, followed by a discussion of a simplified version of the Intelligent River® middleware architecture.

3.1 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 satisfy such a demand. The Linux community supports multiple mature virtualization platforms, leaving the choice to end-users. This chapter provides a technical overview of the principles behind the operation of three representative virtualization solutions: Kernel-based Virtual Machines (KVM), Xen, and Linux Containers.

3.1.1 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. It is responsible for

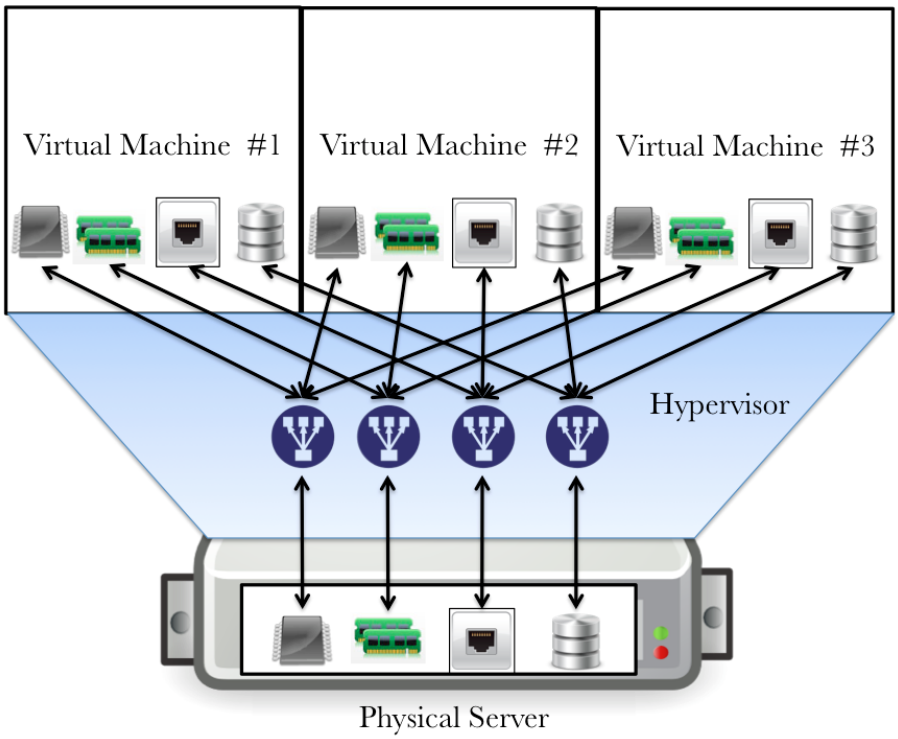


Figure 3.1: A Full-Virtualization System

managing the memory available to the guests, scheduling the processes, managing the network connections to and from the guests, managing 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, the majority of the work is offloaded to the Linux 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) [65]. 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 were running on their own hardware. KVM uses an independent tool known as QEMU [68] to create virtual devices for the virtual machines. Hence, the total solution is commonly referred to 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 scheduling and device management activi-

ties. 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 running directly on the host. The virtual machines contain a set of virtual devices created using QEMU and run an unmodified guest operating system like Linux or Microsoft Windows.

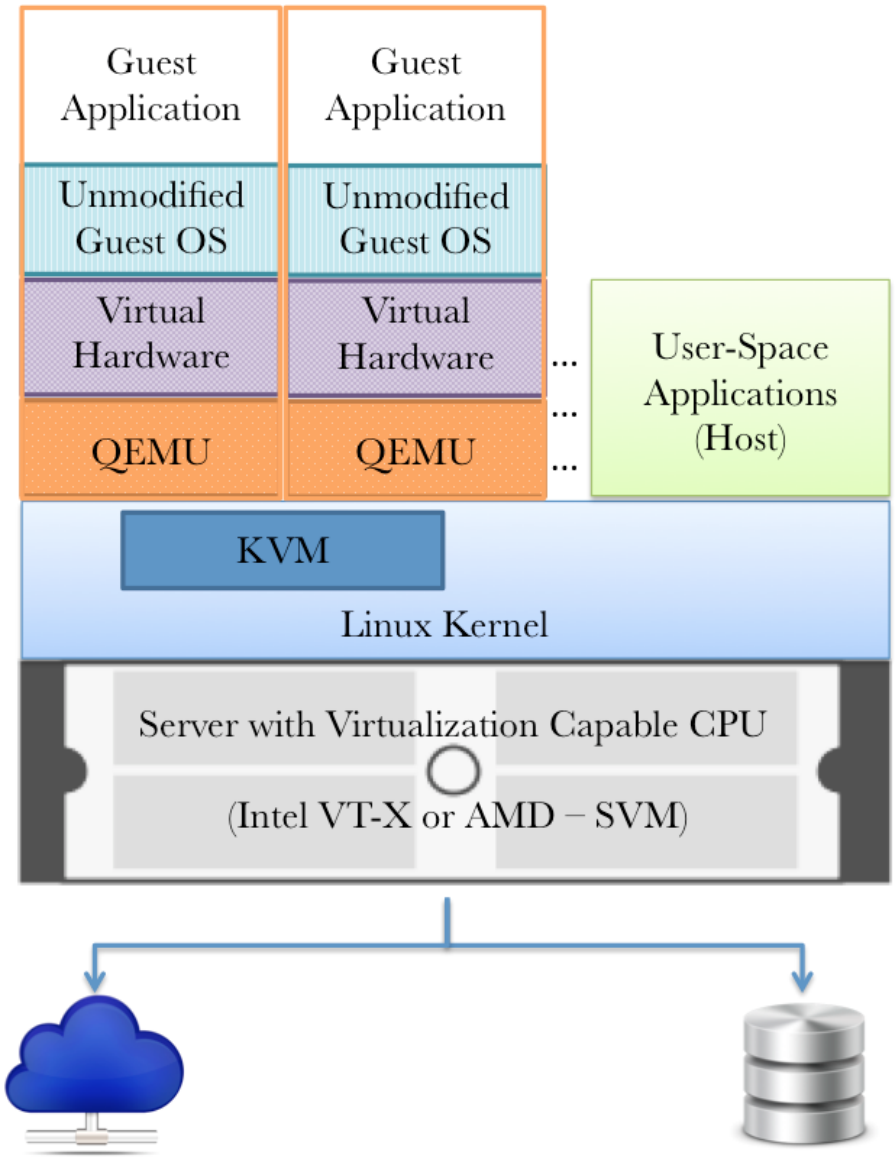


Figure 3.2: KVM - Architecture

In practice, KVM and the Linux kernel treat the virtual machines as regular Linux processes 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 [46] and AMD-SVM [2] are the

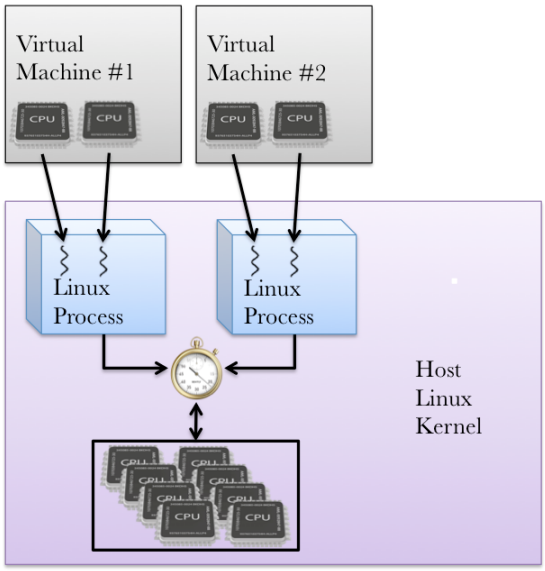


Figure 3.3: KVM - Virtual CPU Management

virtualization extensions provided by Intel and AMD, respectively, which are the most common CPUs used in x86-based servers. KVM relies on these facilities to isolate the instructions 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 improves 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 chapters.

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 [66], and Huge Pages [63] to optimize the memory allocated to the virtual machines. NUMA support is discussed in chapter 6.

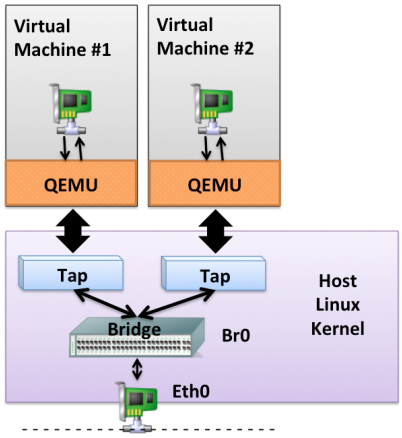
3. **Network Interfaces**:

   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 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.

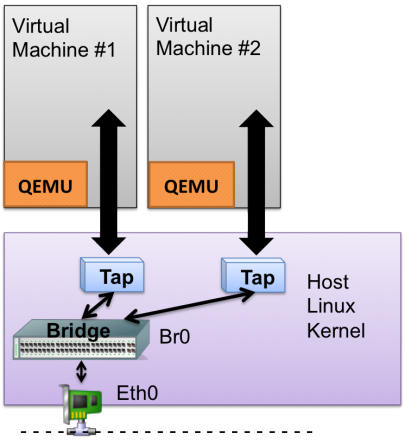
   Figure 3.4b shows a networking configuration where the guest operating system running in the virtual machine is "virtualization aware" and cooperates with the host by bypassing the device emulation provided by QEMU. This is known as *para-virtualized networking*. Para-virtualized networking trades compatibility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

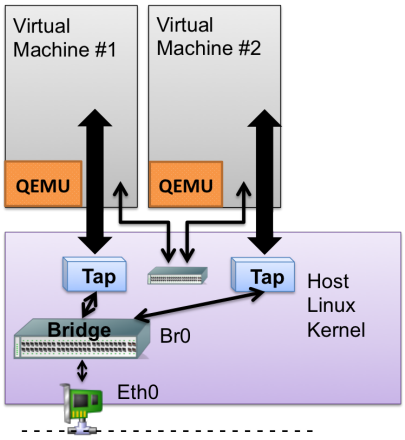
   To support the virtualization of network-intensive applications, physical network interfaces



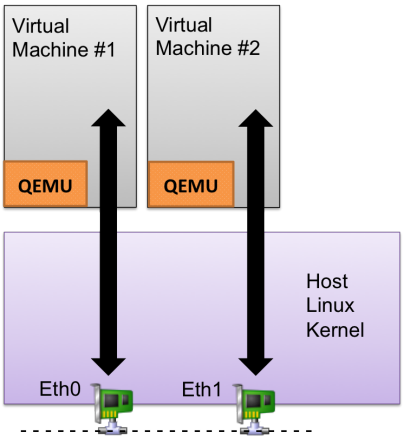
(a) Emulated Networking Devices



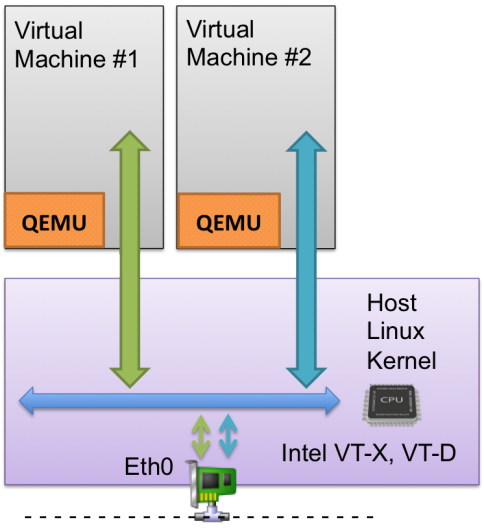
(b) Para-Virtualized Networking



(c) Public and Private Networking



(d) Direct Device Assignment



(e) Direct Device Assignment with SR-IOV

Figure 3.4: Common Networking Options for KVM

attached to the host can be directly assigned to the virtual machines, 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, applicable when individual hardware devices are available to be dedicated to the virtual machines. The key to achieving this direct device access is the hardware assistance provided through Intel VT-D [48], and 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) [47]. 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.

3.1.2 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 inability to provide superior resource isolation and effective scalability [16]. 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. 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 [87], Solaris Zones [77], Chroot [60], and most importantly, OpenVZ [79] - 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 filesystem 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 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, but must use the host's kernel instead of its own. For example, a host server running RedHat Enterprise Linux [34] may run containers that run Debian [17], 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.

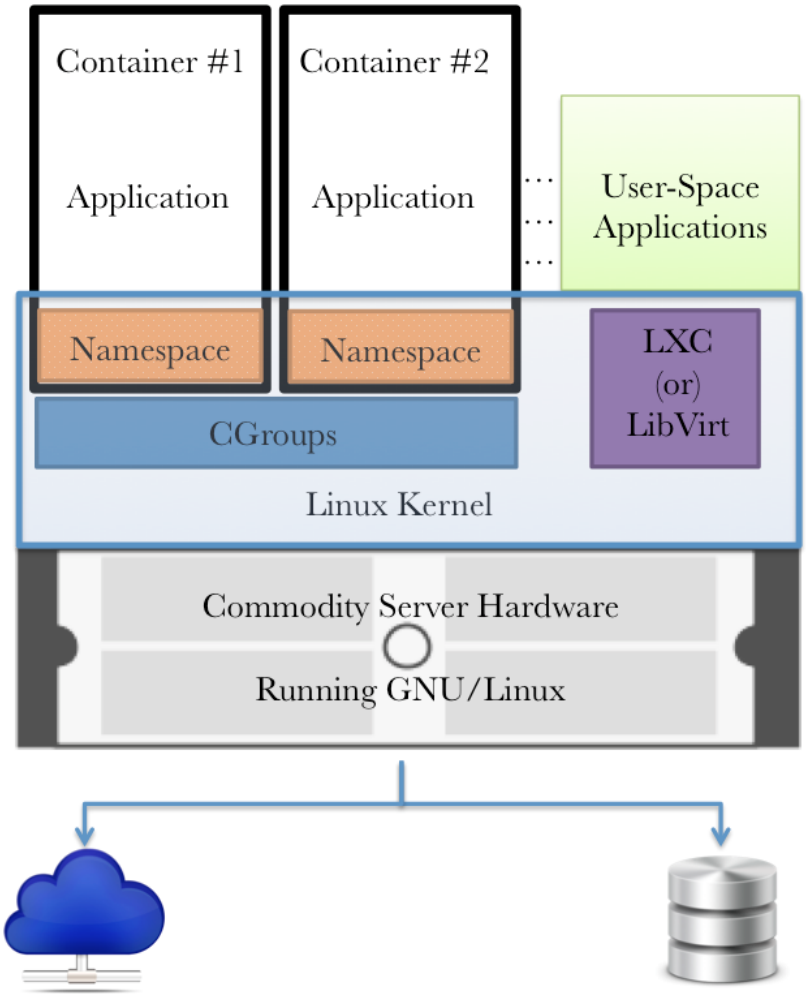


Figure 3.5: Linux Containers - Architecture

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 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 coexist 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 [51], [52], [53], [49], [54], and [50]. Namespaces "wrap" a global system resource on the host and make 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 the mount points 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 provide a secure alternative. They also improve portability, as each container can retain its filesystem tree, regardless 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 names.

- **IPC namespaces :**

  IPC namespaces isolate the System V IPC objects [69] and POSIX message queues [67] 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 numeric PIDs, one in the container's namespace, and the other outside the

namespace (i.e., in the host kernel). 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 into 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 has two user and group IDs, one inside the container's namespace, and the other outside the namespace. This enables a user to possess a 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 privilege greatly improves the security of container-based virtualization solutions. It should be noted that this feature is only available in Linux kernel versions 3.8+.

Four new namespaces are being developed for future inclusion in 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 [32].

- **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) [61] [88] [33] are another key feature of the Linux kernel, used to allocate resources such as CPU, memory, network bandwidth, and 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 CPU access for containers
Cpuset	Used to confine containers to system subsets of CPUs and memory (memory nodes)
Cpuacct	Used to track 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 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.

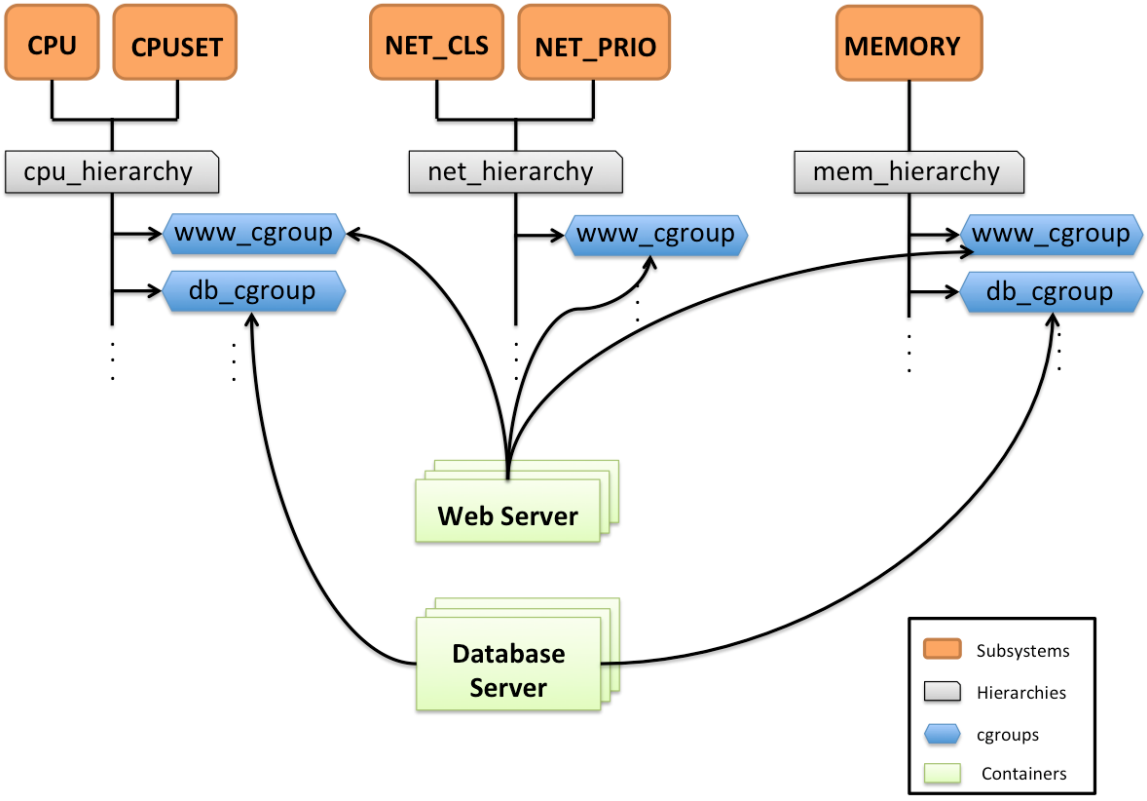


Figure 3.6: cgroups Example Configuration

Figure 3.6 shows an example configuration of cgroups for a system running multiple web and database servers within containers. The objective is to limit the CPU, network bandwidth, and memory available to each container based on whether it runs a web server or a database server. First, a hierarchy is created for the type(s) of subsystem(s) required. For example, mem_hierarchy is created with an association to the memory subsystem. Second, custom cgroups are created under the hierarchy based on the anticipated workload categories. For example, www_cgroup is created to define policies for all the containers that run a web server. The kernel automatically populates the cgroups directory with the required settings. Third, limits are set on the created cgroup. For example, the amount of memory available to all containers associated with a 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 approach 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

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

3.1.3 Xen

Xen [83] 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 [97] [102], a technique where 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 [97], [4], [80].

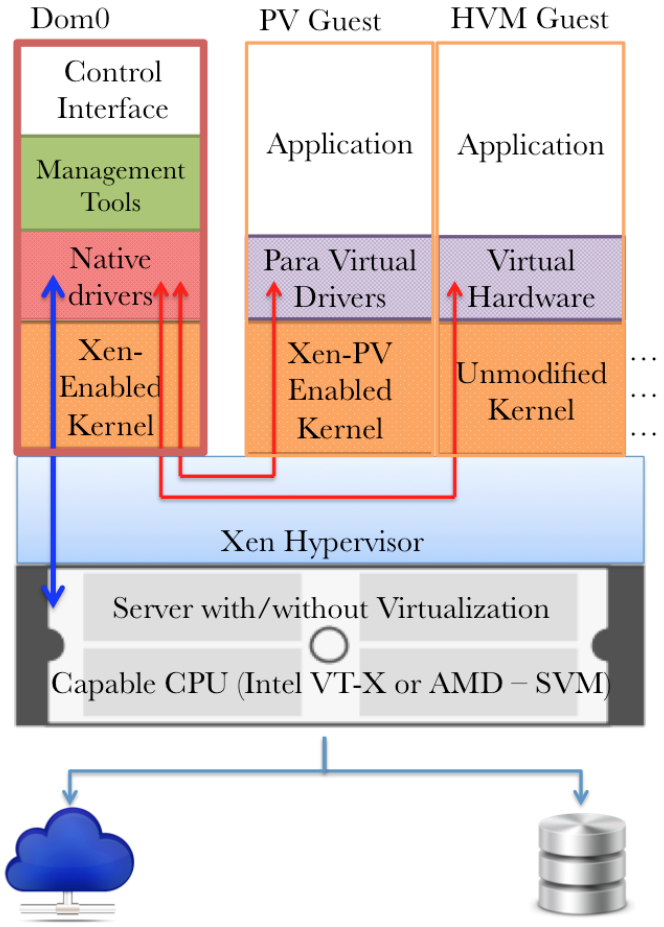


Figure 3.7: Xen - Architecture

The architecture of a system virtualized using Xen is shown in Figure 3.7. The Xen hypervisor runs directly from the bootloader. The hypervisor is 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 administrators 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 Controller [82]. 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 a 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 solutions.

3.2 Intelligent River® Middleware

Intelligent River® is an ongoing interdisciplinary research initiative at Clemson University that aims to deploy a highly distributed ecological sensor network in the Savannah River Basin. Specially designed sensor nodes measure various ecological factors and transmit observations to a real-time Intelligent River® Middleware system. The middleware system is responsible for (i) performing semantic analysis of the real-time observation stream, (ii) reliably persisting the observations, and (iii) publishing the processed observations in multiple formats for further analysis and visualization. The comparative study presented in this thesis was motivated by a need to identify the most suitable virtualization platform to virtualize the constituent middleware components.

Here we briefly introduce the architectural model of the Intelligent River® middleware system and describe the operational characteristics of its components to set the context for the comparative studies presented in subsequent chapters.

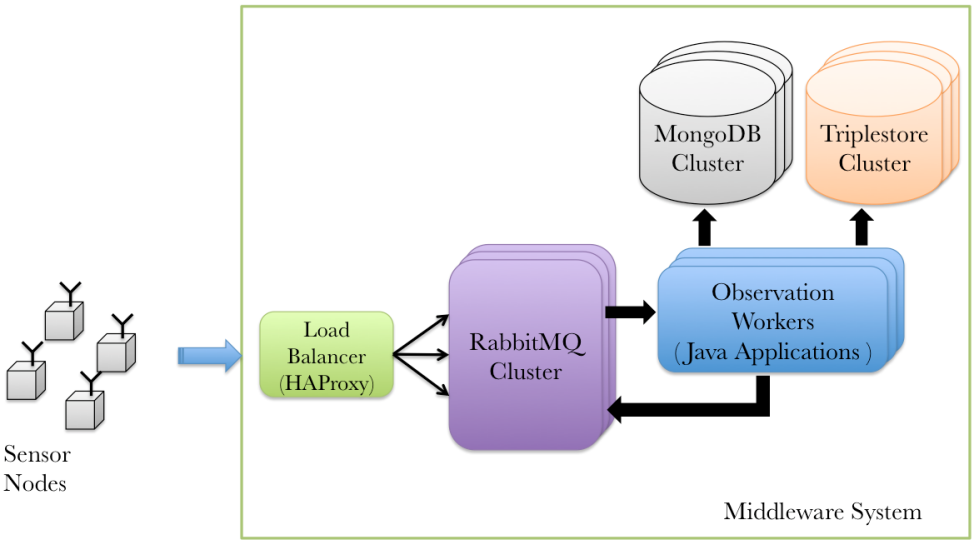


Figure 3.8: Simplified Architectural Model of the Intelligent River® Middleware System

Figure 3.8 shows a simplified version of the Intelligent River® Middleware architecture. *Sensor nodes* are custom-designed hardware devices known as *MoteStacks* [98]. The *Load balancer* is an open source TCP load-balancing application, HAProxy [94] running on a physical server. The *RabbitMQ cluster* comprises of multiple instances of an open source message broker, RabbitMQ [71]. The *observation workers* are custom developed Java applications, each deployed on a physical server. The *MongoDB cluster* comprises of multiple instances of MongoDB [41], setup in a sharded configuration [42]. The *Triplestore cluster* comprises of multiple instances of Apache Jena TDB [28].

Sensor nodes gather data from commodity sensor probes and transmit the collected observations to the load balancer through multiple networks, both wired and wireless. The load balancer relays the observations to any of the servers in the RabbitMQ cluster, which queues the observations to be fetched by observation workers. Worker applications fetch the queued observations from the message queues in the RabbitMQ cluster and perform (i) semantic analysis on the observations, (ii) persist both processed and unprocessed observations to MongoDB, (iii) persist the processed observations to a triplestore, and (iv) publish the processed observations back to RabbitMQ for further analysis and visualization. The architecture of the middleware system is based on a hybrid of

message queuing, client-server, and publish/subscribe patterns. Such an architecture yields a system with components that are loosely coupled, enabling them to scale independently based on demand. If the middleware system receives observations at a rate higher than the existing set of observation worker applications can process, additional instances can be commissioned on new servers to fetch and process observations off the queues, improving overall system throughput. Similarly, the MongoDB cluster is setup using a sharded configuration, so that new servers, each running MongoDB, can be added as shards to the existing cluster based on demand. The triplestore cluster is also designed so that new servers, each running Apache Jena TDB, can be added to the cluster on an as-needed basis.

Given the loosely coupled configuration and the dynamically changing demands of the middleware system, it serves as an ideal candidate to leverage the benefits of a virtualized infrastructure.

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 that 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. A key 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 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 access to CPUs. Second, we measure the overhead associated with memory access bandwidth available to applications running within a virtual machine. 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. Fifth, we measure the performance of each of the virtualization platforms using the standard system benchmarks. Finally, we measure the performance of the Intelligent River® middleware components under each of the virtualization platforms.

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 a fair comparison, we preserve the default operating system and hypervisor configurations. Modification of any default settings required to evaluate specific scenarios noted. 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, virtualization 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, 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 [21].

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 [58].

4. Generation of 500 RSA private keys using openssl [24].

The objective of this test suite is to measure the execution time of the sample application and its individual tasks as described above. The tests were 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.

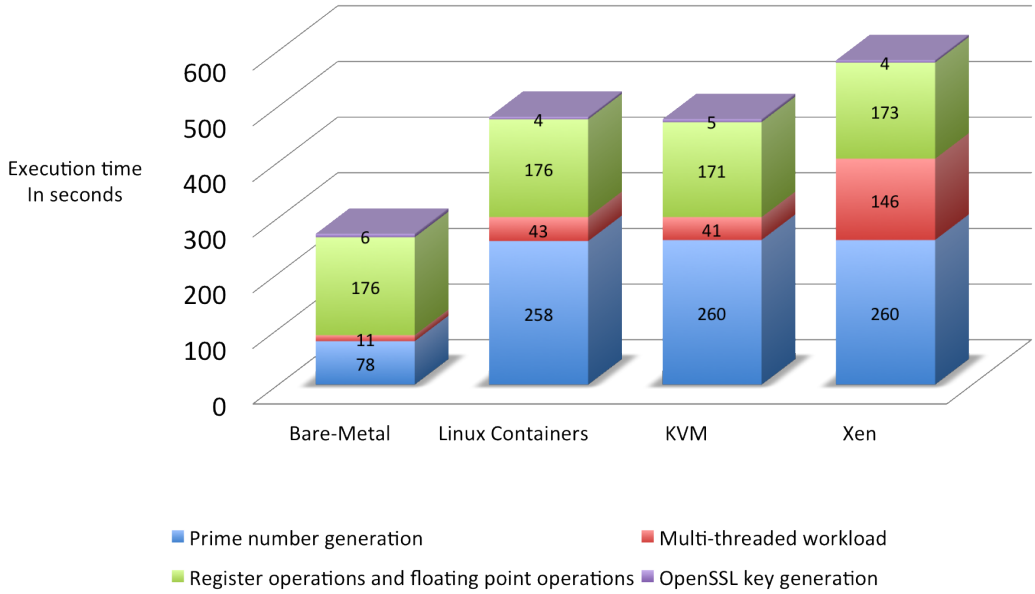


Figure 4.1: CPU Virtualization Overhead - 1 Virtual CPU

A virtual machine was created on each of the three physical servers, each with one virtual CPU, 300 GB of disk storage and 7680 MB of memory. Figure 4.1 shows the overhead introduced 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 CPU access. In particular, Xen took longer to complete the task involving multiple threads competing for a set of mutexes, indicating the poor relative performance of the CPU scheduler in the Xen hypervisor.

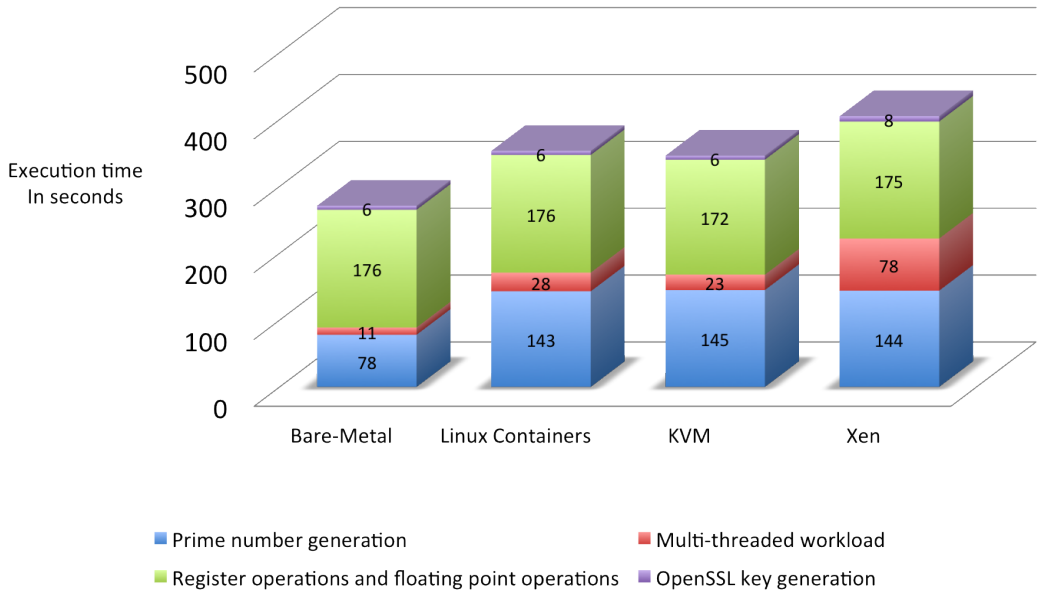


Figure 4.2: CPU Virtualization Overhead - 2 Virtual CPUs

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 as the previous test. Note 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 the 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.

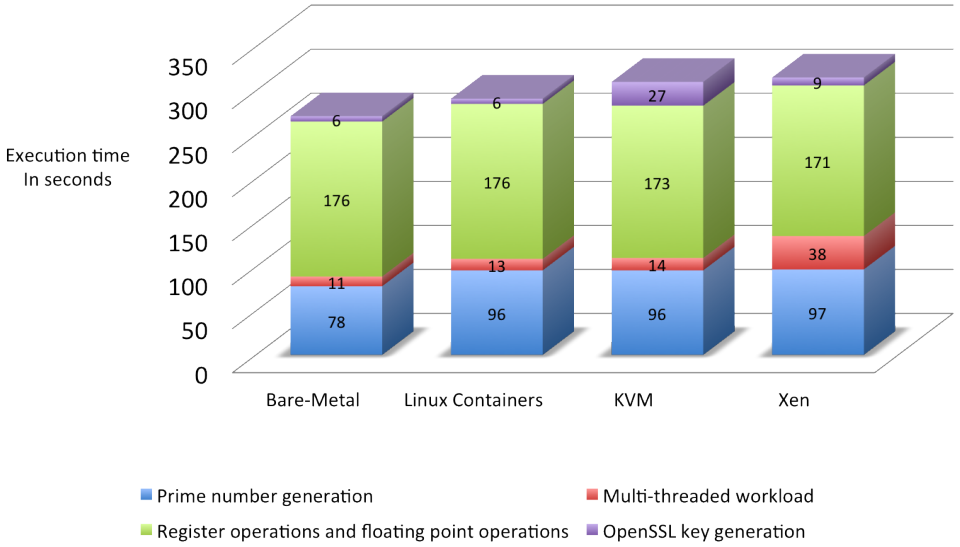


Figure 4.3: CPU Virtualization Overhead - 4 Virtual CPUs

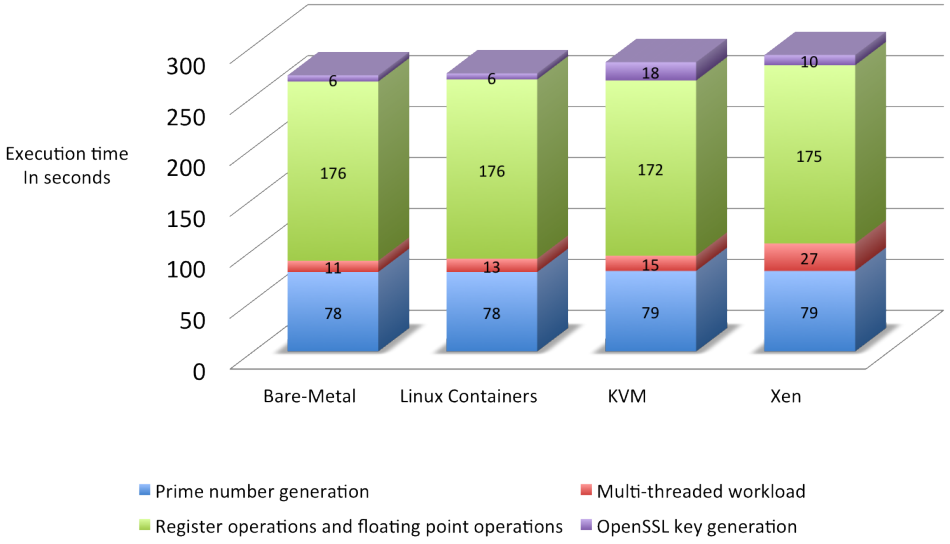


Figure 4.4: CPU Virtualization Overhead - 8 Virtual CPUs

To further examine the overhead introduced when virtualizing the CPUs, we ran the test again, with the virtual machines configured 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 were identical to the bare-metal host, with the only exception being an added hypervisor. The difference in execution times among the virtual machines representing Linux containers, KVM, and Xen and

the bare-metal host can be correlated with overall CPU overhead. Both Linux Containers, and KVM exhibit the least overhead, followed by Xen.

Based on these results, we conclude that Linux Containers and KVM perform the best for both single-threaded and multi-threaded workloads, exhibiting the least overhead compared to the bare-metal performance. Though Xen performed identically to the others for single-threaded workloads, it exhibited relatively poor performance when 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 [37] to allocate two arrays of a given size, and then copies data from one to the other. 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 available to the virtual machine created on each of the three virtualized servers, compared to the

bandwidth observed on an independent bare-metal host.

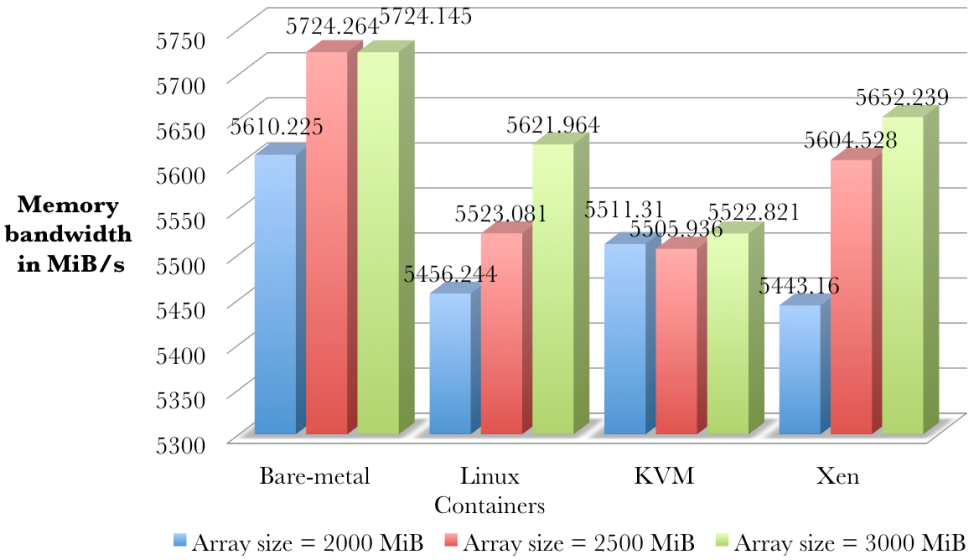


Figure 4.5: Memory Virt. Overhead - (Virtual Machine Memory = Host Memory = 8 GB)

A virtual machine was created on each of the three physical servers 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 the X-axis representing the virtualization platforms, and the 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 platforms exhibits a different overhead pattern. 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" [75] causing an overhead in low utilization scenarios.

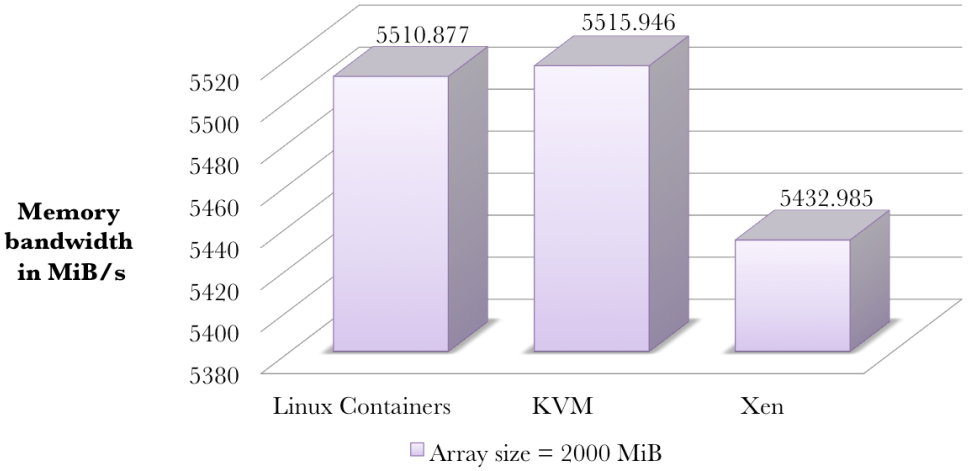


Figure 4.6: Memory Virt. Overhead (Virtual Machine Memory(5 GB) <Host Memory(8 GB))

To further examine 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.

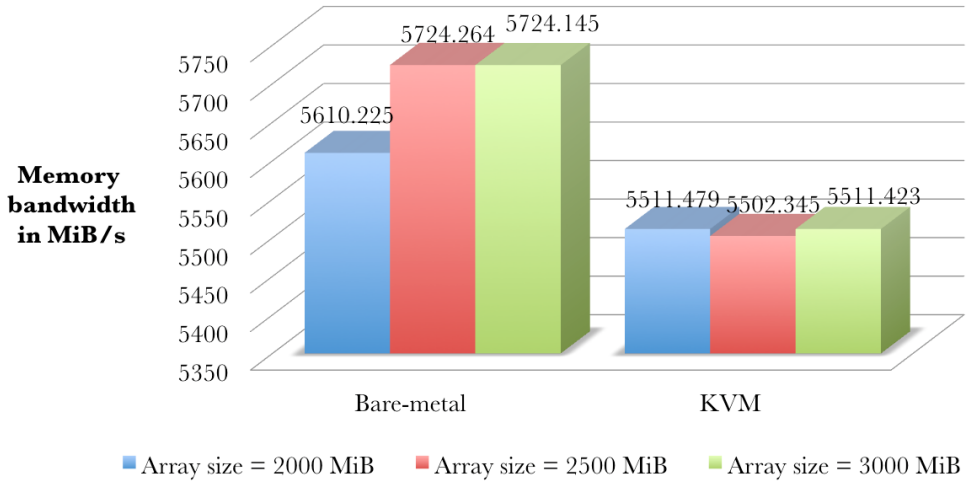


Figure 4.7: Memory Virt. Overhead - Virtual Machine Memory(10 GB) >Host Memory(10 GB)

KVM allows virtual machines to run their own kernel, yet manages the virtual machine memory from the host kernel. As a result, it provides the ability to leverage memory over com-

mitment facilities. 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 behavior 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, shown in Figure 4.7, confirm our claim that KVM performs the same, even in an over-committed environment − until all virtual machines attempt to use their share of memory at the same time.

4.4 Network Overhead

KVM, Xen, and Linux Containers provide mechanisms to transparently virtualize network interfaces on the host, enabling the hosted virtual machines to build independent network stacks over their virtual network interfaces. KVM uses QEMU to emulate virtual network devices over the physical network interfaces. Xen enables virtual machines to use para-virtual drivers to access the network interfaces on the host. Linux Containers use network namespaces to virtualize container network interfaces. 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 a virtual machine can be : (i) network address translated (NAT), where the host proxies communications from the virtual machine, or (ii) bridged, where the virtual machine is allowed to participate in the same LAN segment as the host. To measure the overhead associated with virtualizing networking interfaces, we measured the network bandwidth available to a virtual machine in both NAT and bridged configurations and compared the results with the bandwidth available on an independent bare-metal host. To ensure fair comparison, we set up an Iperf [23] server on a machine outside the test network and measured the network bandwidth by running an iperf client on each of the virtual machines. The results are shown in Figure 4.8, where the X-axis represents the virtualization platforms, and the Y-axis represents the observerved network bandwidth in MBytes/sec.

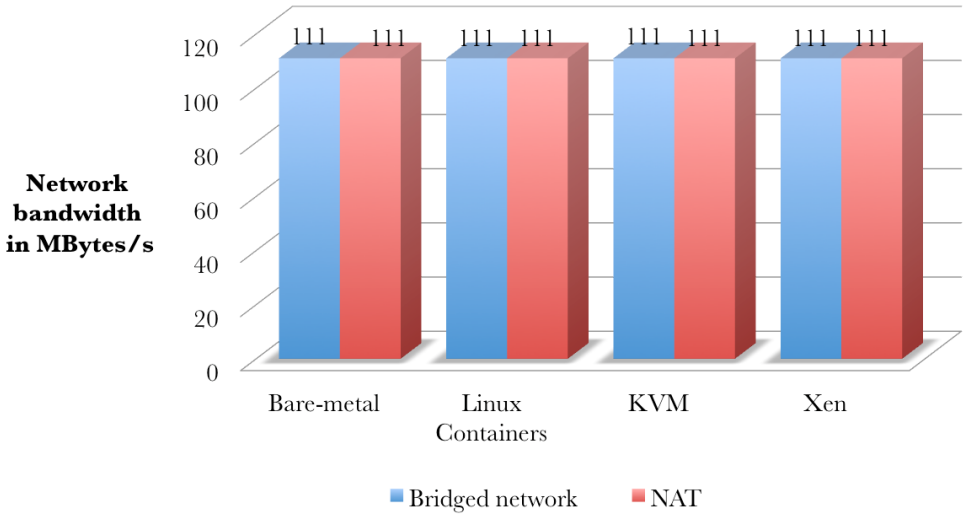


Figure 4.8: Network Virtualization Overhead - Network Bandwidth

It is evident from the results that there is no observable overhead introduced when virtualizing 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. Xen does not expose the disk devices to the virtual machines. When the system boots, dom0 (the privileged domain) 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" 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 [84] and virtio-scsi [85]. 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 [72] entirely bypassing the QEMU block driver [56]. 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 introduced when virtualizing disk I/O, we created a sample application (w4) that uses sysbench [58], ioping [55], and dd [62] 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. The tests were performed on virtual machines created on each of the three virtualized servers, as well as 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.

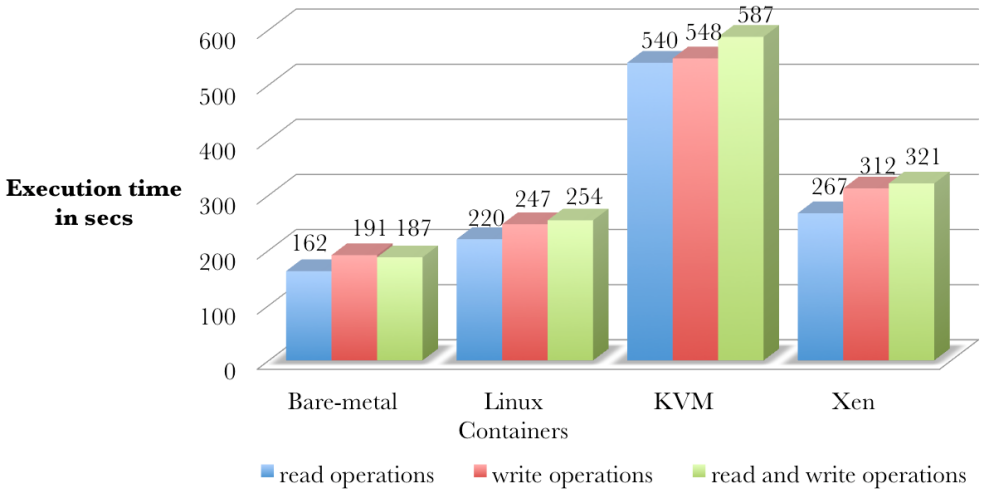


Figure 4.9: Virtualization Overhead - Sequential File I/O

Figure 4.9 summarizes the execution time for performing 10,000 each of sequential reads, sequential writes, and mixed sequential reads and sequential writes. It is evident from the results that KVM exhibits significant overhead in virtualizing sequential I/O operations. Linux containers

exhibit the least overhead, as containers do 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 10,000 each of random reads, random writes, and mixed random reads and random writes. Though KVM exhibits improved performance for random I/O operations, as compared to sequential I/O operations, it is still the worst performer among the solutions. Once again, Linux Containers exhibits the least overhead in performing random I/O operations.

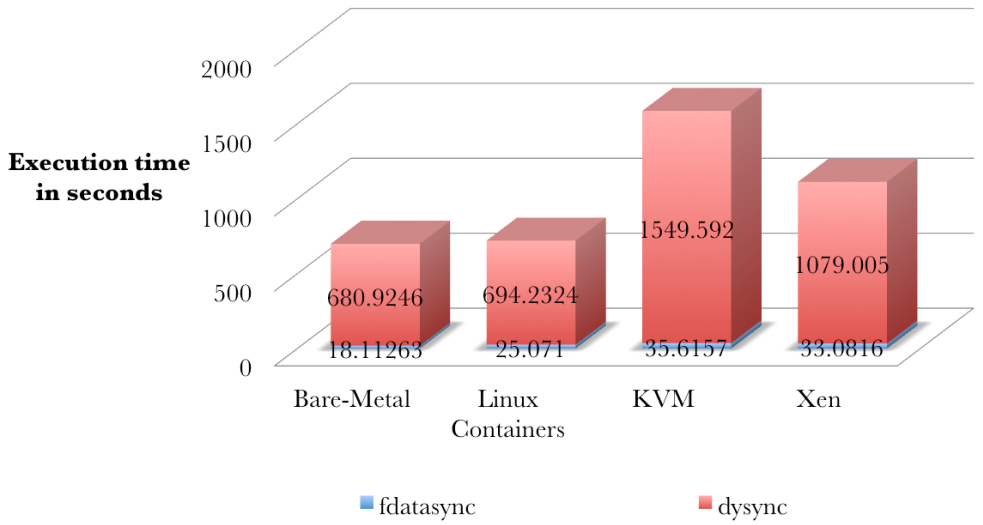


Figure 4.11: Virtualization Overhead - Disk Copy Performance

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 the fdatasync flag (physically write output file data before finishing) and the dsync flag (use synchronized I/O for data). The results summarized in Figure 4.11 conform to our earlier results, with KVM exhibiting the highest overhead, and Linux Containers exhibiting the least. Based on these results, we conclude that Linux Containers perform the best with respect to virtualizing disk I/O operations.

4.6 Performance Comparison - System Benchmarks

To gauge the performance of each of the representative virtualization platforms, we analyzed the overhead they impose in order to virtualize CPU, memory, network access, and disk access. However, further evaluation based on standard system benchmarks is useful in two regards: (i) the performance of individual subsystems can be observed with little influence from other subsystems, and (ii) the results can be used to compare to prior work. This section compares the performance of each of the virtualization platforms using the LINPACK [22], STREAM [73], and IOZONE [74] benchmarks.

LINPACK is a software library that makes use of basic linear algebra subprograms to perform basic vector and matrix operations. It reflects the floating point computing performance of the CPU. The benchmark was run on a virtual machine created with 8 virtual CPUs on each of the three physical servers, and also on an independent bare-metal system used as a base-line.

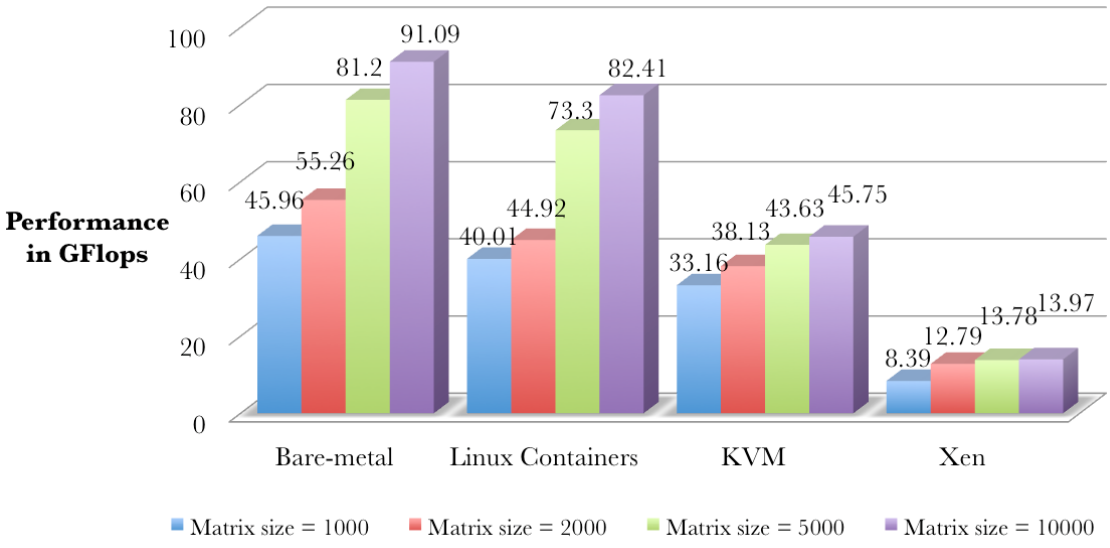


Figure 4.12: Computing Performance using LINPACK

Figure 4.12 summarizes the floating point performance of the virtualization platforms. The X-axis represents the virtualization platforms, and the Y-axis represents the achieved floating point performance in GFlops for different matrix sizes. It is evident from the results that Linux Containers provide near-native floating point computing performance, while Xen exhibits significantly poorer floating point computing performance. KVM performs relatively close to Linux Containers for lower matrix sizes, but the performance degrades with increased matrix sizes.

The STREAM benchmark is a simple, synthetic benchmark designed to measure sustainable memory bandwidth (in MB/s). The benchmark was run on virtual machines created with 8 virtual CPUs, and 7196 MB of memory. The benchmark was run using the "Triad" vector kernel.

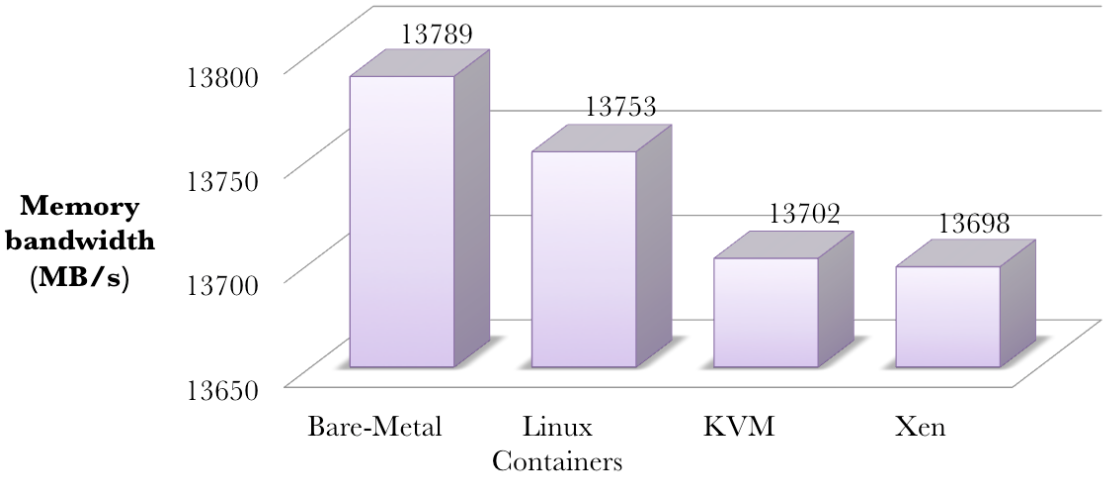
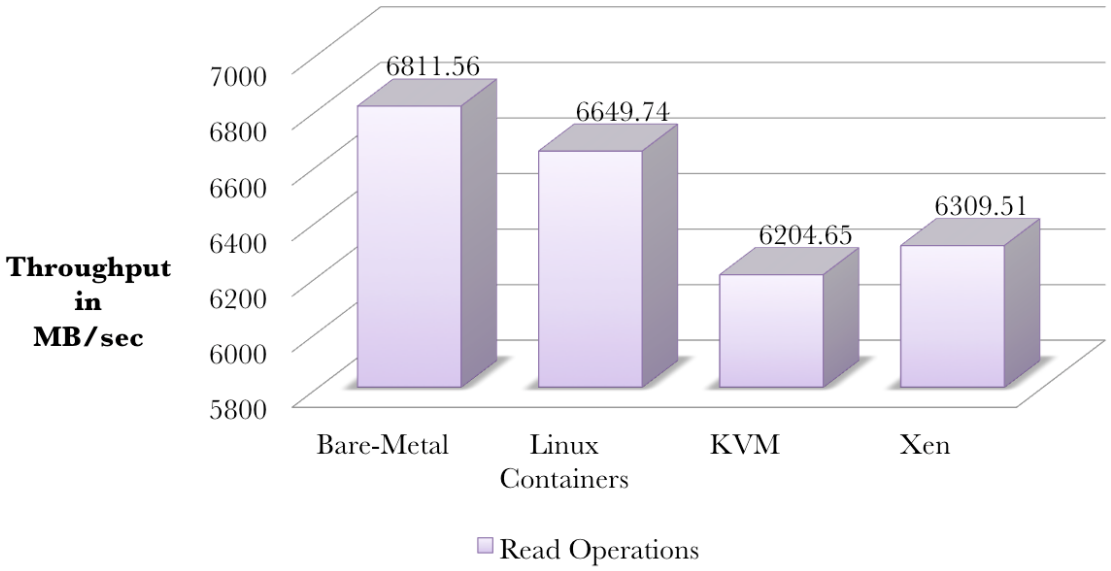


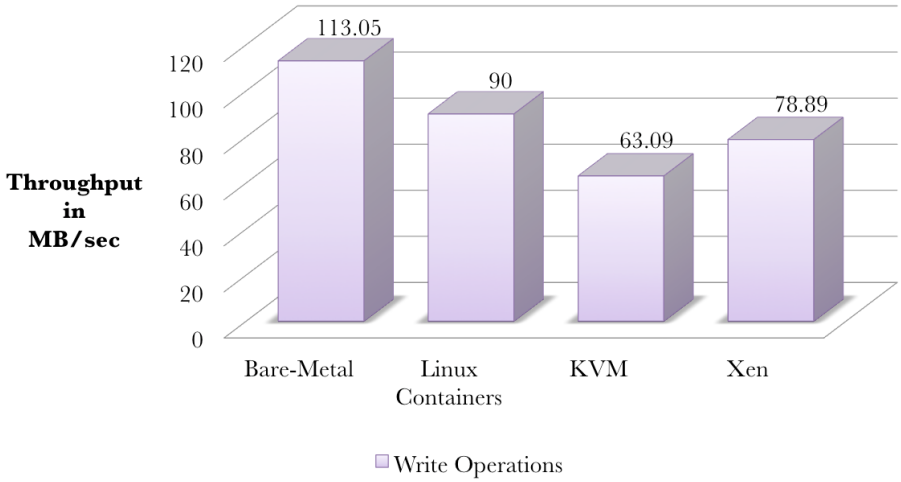
Figure 4.13: Memory Performance using STREAM

Figure 4.13 summarizes the memory access performance of the virtualization platforms. The X-axis represents the virtualization platforms, and the Y-axis represents the observed memory bandwidth (in MB/s). The results corroborate our claim from the prior overhead analysis that Linux Containers provide relatively superior memory performance. KVM and Xen share similar performance.

IOZONE is a filesystem benchmark tool that tests file I/O performance by generating a variety of file operations. The benchmark was run on virtual machines created with 8 virtual CPUs, 7196 MB of memory, and 300 GB of disk storage.



(a) Read Performance



(b) Write Performance

Figure 4.14: Filesystem Performance using IOZONE

Figure 4.14a summarizes the performance of read operations on each of the virtualization platforms. The X-axis represents the virtualization platforms, and the Y-axis represents the observed read throughput (in MB/s). Linux Containers exhibited the highest read throughput, while KVM exhibited the lowest. Overall, the difference between the performance of the bare-metal host and the virtual machines does not look significant, indicating that all the platforms provide good read performance. Figure 4.14b summarizes the performance of write operations on each of the vir-

tualization platforms. The X-axis represents the virtualization platforms, and the Y-axis represents the observed write throughput (in MB/s). Linux Containers exhibited the highest write throughput, while KVM exhibited the lowest.

4.7 Performance Comparison - Workload-Specific Benchmarks

This section evaluates the performance of the representative virtualization platforms with respect to specific workload categories, including web servers, database servers, and program execution environments. To identify the most suitable platform to run a web server, we tested each of the virtualization solutions using the Apache benchmark program running against an nginx web server. The benchmark was run on virtual machines created with 8 virtual CPUs, 7196 MB of memory, and 300 GB of disk storage. This test measures how many requests per second each of the systems can sustain when carrying out 500,000 requests, with upto 100 concurrent requests.

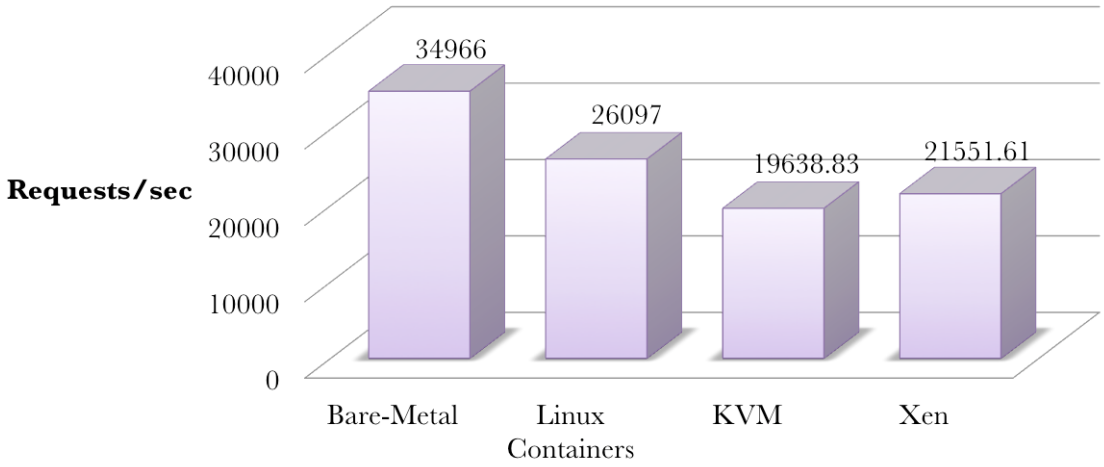


Figure 4.15: Nginx Web Server Performance

It is evident from the results summarized in Figure 4.15 that all of the virtualization platforms yielded low performance when compared with the bare-metal system. Linux Containers provide the highest throughput among the virtualization platforms, closely matched by Xen, while KVM yields the lowest throughput. Based on these results, Linux Containers are the preferred virtualization platform to host web servers.

To identify the most suitable platform to run a database server, we installed PostgreSQL server on each of the virtual machines and tested the instances using the pgbench benchmark pro-

gram. The benchmark was run on virtual machines created with 8 virtual CPUs, 7196 MB of memory, and 300 GB of disk storage. This test runs the same sequence of SQL commands repeatedly in multiple concurrent database sessions, and then calculates the average transaction rate (i.e., transactions per second).

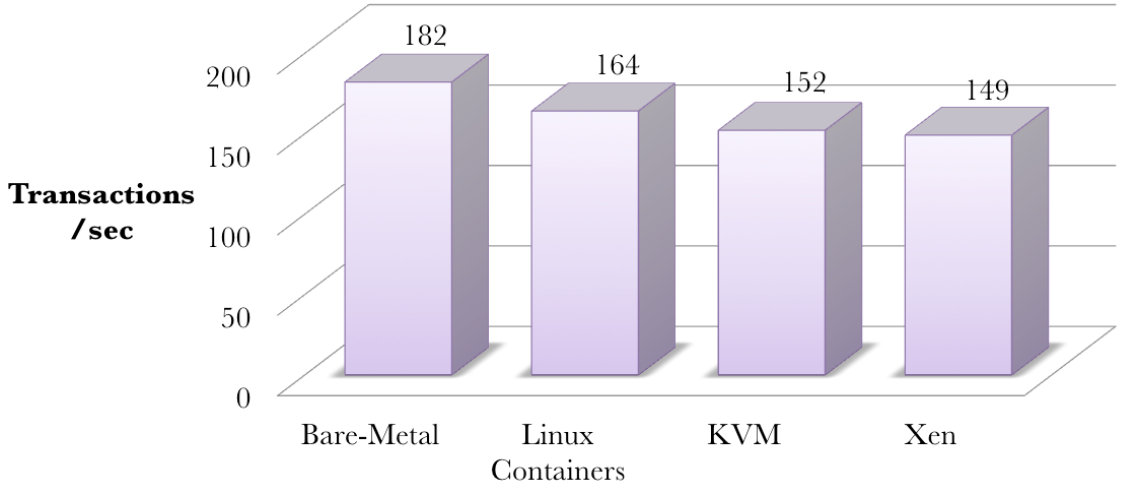


Figure 4.16: PostgreSQL Database Server Performance

It is evident from the results summarized in Figure 4.16 that all of the virtualization platforms yielded performance very close to that provided by the bare-metal system. Linux Containers yielded the highest throughput among the virtualization platforms, closely matched by Xen and KVM.

We chose Python and PHP to represent a custom application workload. To identify the most effective platform to run custom applications developed in Python, we setup the pybench benchmark program on virtual machines created with 8 virtual CPUs, 7196 MB of memory, and 300 GB of disk storage. This test exercises built-in functions and Python language constructs executed within nested for-loops. The result is the total execution time in seconds.

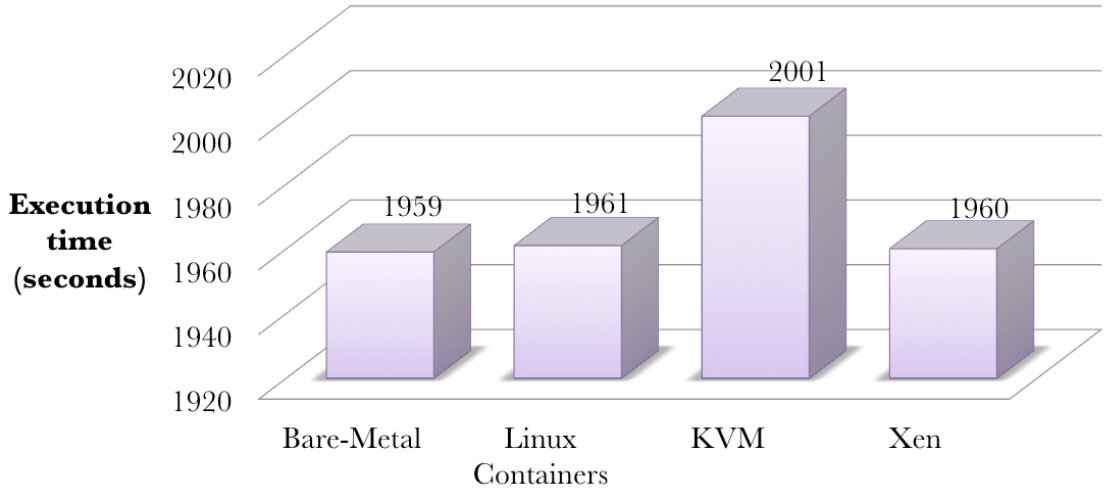


Figure 4.17: Python Application Performance

It is evident from the results summarized in Figure 4.17 that both Linux Containers and Xen yielded performance very close to that provided by the bare-metal system. KVM exhibited relatively low performance.

To identify the most effective platform to run custom applications developed in PHP, we setup the phpbench benchmark program on virtual machines created with 8 virtual CPUs, 7196 MB of memory, and 300 GB of disk storage. This test exercises the PHP interpreter by executing basic PHP language constructs for 1,000,000 iterations. The result is the total execution time in seconds.

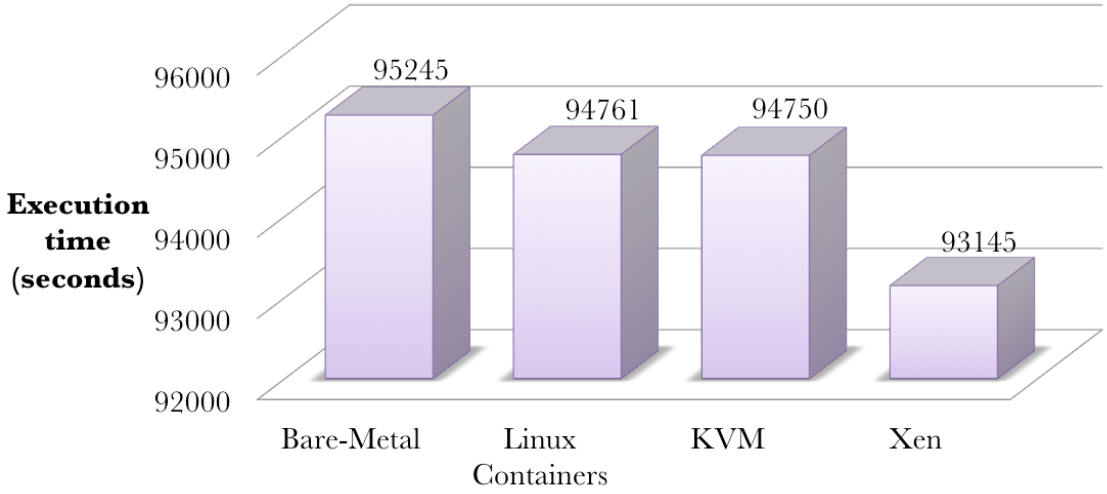


Figure 4.18: PHP Application Performance

It is evident from the results summarized in Figure 4.17 that both Linux Containers and KVM yielded performance very close to that provided by the bare-metal system. Xen exhibited relatively low performance.

4.8 Performance Comparison - Intelligent River® Middleware

A key motivation for this thesis is to identify the most effective virtualization platform to virtualize the Intelligent River® middleware system. Evaluation of the virtualization platforms based on virtualization overhead, their performance on system benchmarks, and workload-specific benchmarks is not sufficient to predict each platform's performance while virtualizing a real-time middleware system. The middleware system described in Chapter 3 is a hybrid of multiple architectural patterns and is designed to scale horizontally (i.e., there can be multiple instances of every component, performing the same task in parallel). We evaluate the virtualization platforms based on the constituent middleware system components, including MongoDB, RabbitMQ, TDB, and custom observation workers.

4.8.1 Virtualization Performance - MongoDB

To evaluate the effectiveness of the platforms in virtualizing MongoDB, we created virtual machines with 8 virtual CPUs, 7196 MB of memory, and 300 GB of disk storage. We next installed MongoDB with the default configuration. We evaluate the performance of the virtual machines running MongoDB, when performing (i) insertion of documents, (ii) querying for a specific document, and (iii) updating of an existing document. To perform this evaluation, we created a test application using Java that performs these tasks. First we observe the time taken to insert 12,000 documents into a collection using 4 independent threads.

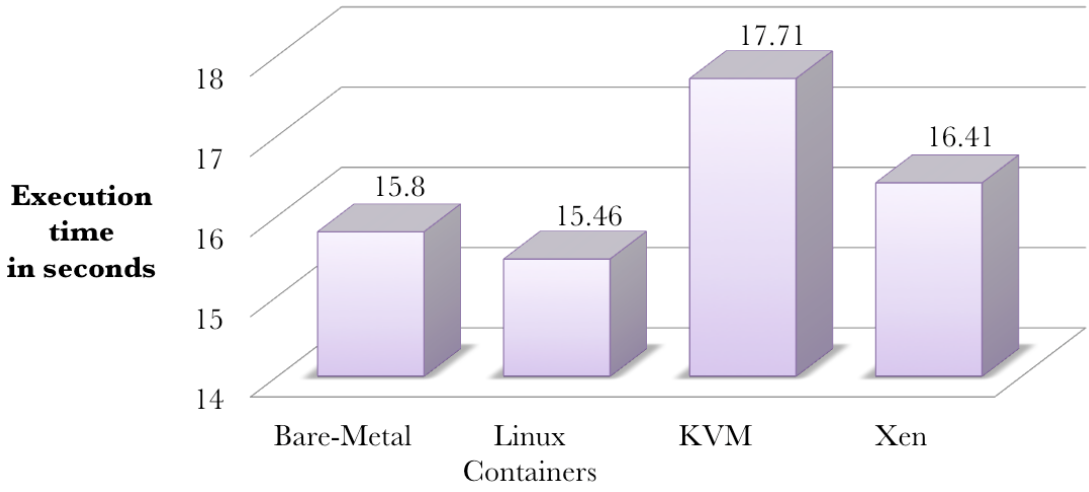


Figure 4.19: MongoDB Insert Performance

Interestingly, the results summarized in Figure 4.19 show that when virtualized in a Linux container, MongoDB yielded slightly better insert throughput than the bare-metal server. The application took marginally longer to insert when virtualized with Xen compared to the bare-metal host. The KVM configuration took the longest time to complete. MongoDB operates on memory mapped files. Hence, the inserts occur in memory and are asynchronously committed to disk. We postulate that, Linux Containers yielded better performance than the bare-metal host due to the execution of fewer memory maintenance activities than the host.

Next, we observed the time taken to execute 12,000 "find" queries using 4 independent threads on each of the virtualized MongoDB servers.

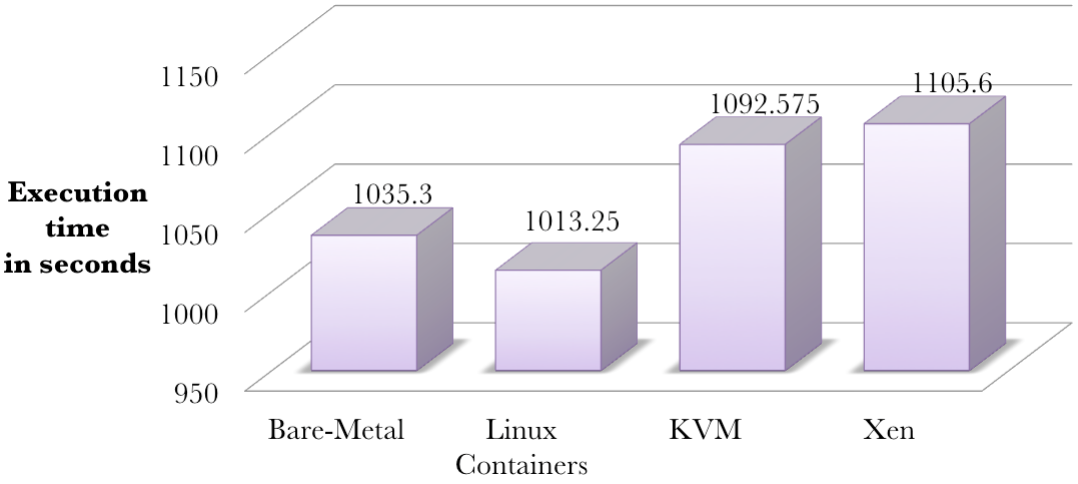


Figure 4.20: MongoDB Find Performance

The results summarized in Figure 4.20 show that MongoDB running within a container outperforms MongoDB running on a bare-metal system when executing 12000 "find" queries. The MongoDB servers virtualized by KVM and Xen took significantly longer to execute the find queries.

Finally, we observed the time taken to execute 12,000 "set" queries using 4 independent threads on each of the virtualized MongoDB servers.

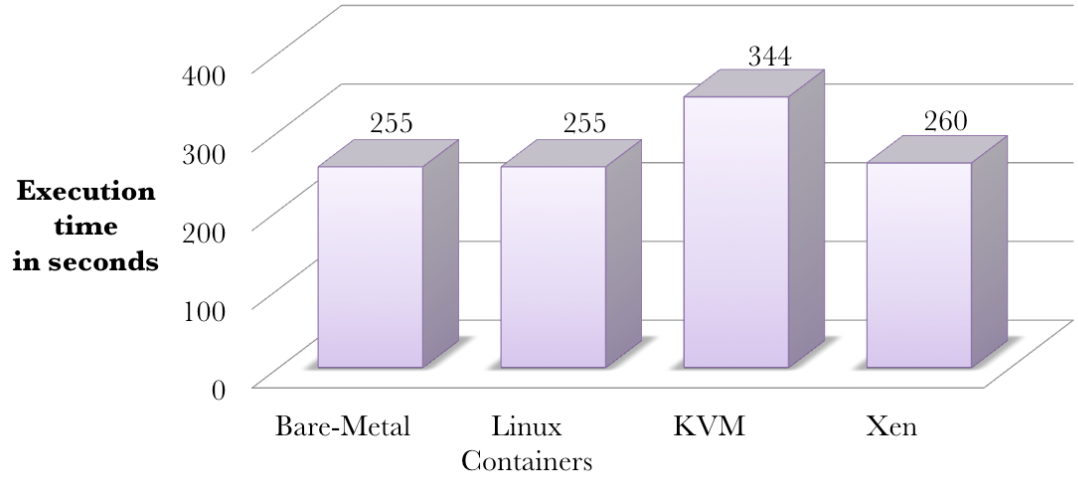


Figure 4.21: MongoDB Set Performance

The results summarized in Figure 4.20 show that MongoDB virtualized in a container yields the same performance as MongoDB running on a bare-metal system when executing 12000 "set"

queries. While the server virtualized with Xen yielded near-native performance, the server virtualized with KVM took significantly longer to execute the set queries.

Based on the results presented above, Linux Containers are the preferred virtualization platform to virtualize MongoDB servers for the Intelligent River® middleware system.

4.8.2 Virtualization Performance - RabbitMQ

RabbitMQ serves as an entry point for observations to flow into the Intelligent River® middleware system. All middleware components communicate with each other through messages delivered via RabbitMQ. Hence, RabbitMQ is a key component of the middleware architecture. To evaluate the performance of RabbitMQ server in a virtualized environment, we observe the achieved publication throughput and the subscription throughput.

To evaluate the performance of the virtualization platforms in virtualizing RabbitMQ server, we created virtual machines with 8 virtual CPUs, 7196 MB of memory, and 300 GB of disk storage. We then installed RabbitMQ with the default configuration. To perform the evaluation, we created a Java application that publishes 50,000 observations using 8 independent threads to each of the virtualized RabbitMQ servers, and also the RabbitMQ server running on the bare-metal system.

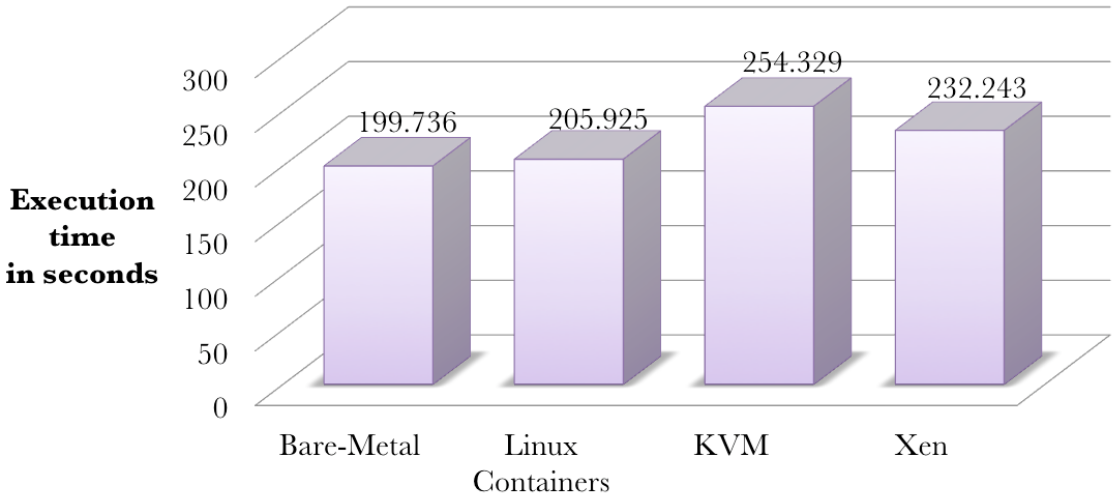


Figure 4.22: RabbitMQ Publish Performance

Figure 4.22 summarizes the time taken by the application to publish 50,000 observations to RabbitMQ. It is evident from the results that Linux Containers provide near native performance, while Xen and KVM exhibit a significant decrease in the publication throughput.

Finally, to evaluate the subscription throughput of the virtualized RabbitMQ servers, we created another Java application to consume the 50,000 messages inserted during the prior evaluation. The Java application uses 8 independent threads to consume the messages.

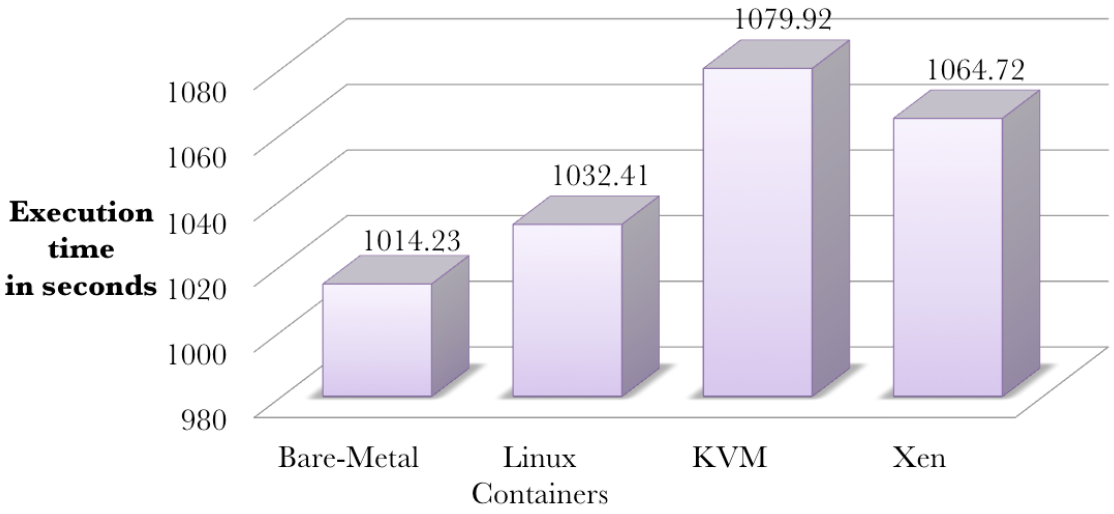


Figure 4.23: RabbitMQ Subscription Performance

Figure 4.23 summarizes the time taken by the application to consume the 50,000 observations. The results once again show that Linux Containers provide near native performance, while Xen and KVM exhibit a significant decrease in the subscription throughput.

Based on the results presented above, we conclude that Linux Containers are the preferred platform to virtualize RabbitMQ servers for the Intelligent River® middleware system.

4.8.3 Virtualization Performance - Triplestore

A triplestore is used by the middleware system to store the semantically processed observations. The two major communication patterns used to interact with the triplestore are: (i) REST API, to store and retrieve data, and (ii) SPARQL, to retrieve and process the semantic data. Since the performance of the SPARQL queries varies based on the complexity of the queries and data, we restrict the scope of this evaluation to the insertion and retrieval of semantic data through the REST API. The triplestore used by the Intelligent River® middleware system is Apache Jena TDB. TDB stores and manages data as regular files. To evaluate the performance of TDB when virtualized using the three virtualization platforms, we created virtual machines with 8 virtual CPUs, 7196 MB

of memory, and 300 GB of disk storage. We then installed TDB with the default configuration. To perform the evaluation, we created a Python script that inserts 1,000 observations.

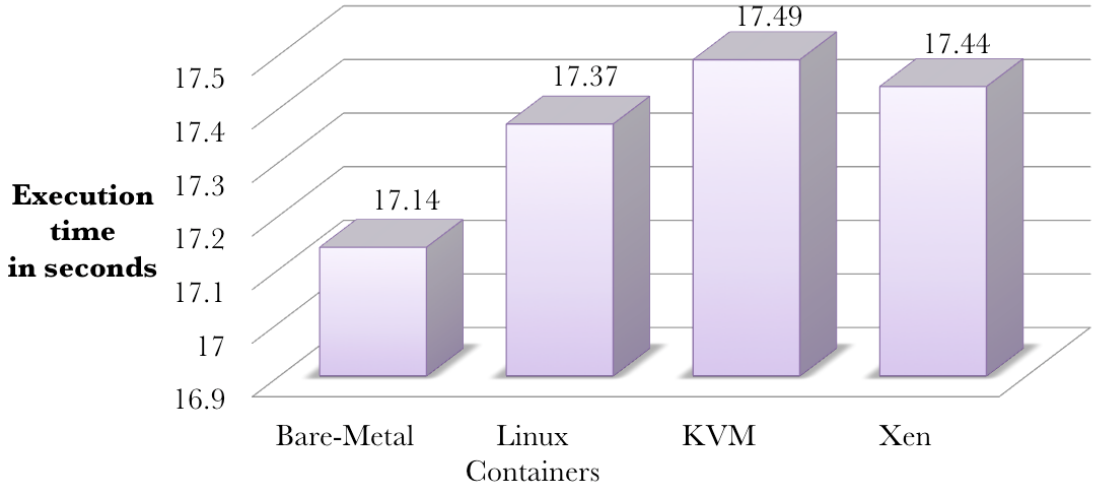


Figure 4.24: Triplestore Performance - Storing Observations

Figure 4.24 summarizes the time taken by the script to insert 1,000 observations into TDB. It is evident from the results that all three virtualization platforms provide near native performance.

Finally, to evaluate the retrieval performance of the virtualized TDB servers, we created another Python script that retrieves the 1,000 observations inserted during the prior test.

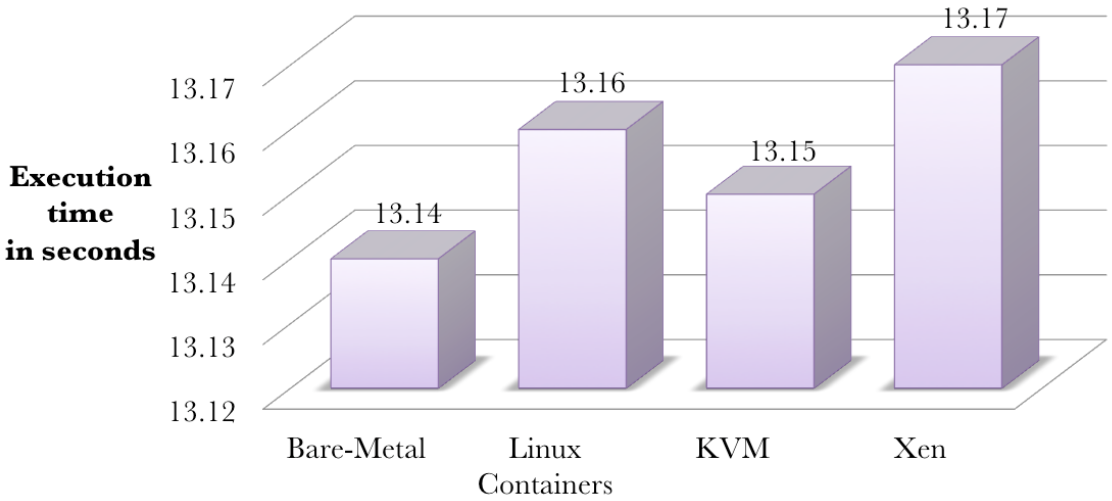


Figure 4.25: Triplestore Performance - Retrieving Observations

Figure 4.25 summarizes the time taken to retrieve the observations. The results once again

show that virtualization does not impact the performance of the TDB server.

Based on the results above, we conclude that the triplestore is not impacted by virtualization for batch insertions and retrievals.

4.8.4 Virtualization Performance - Observation Workers

Observation workers are the core components of the Intelligent River® middleware system. The performance of an observation worker instance is dependent on the performance of all the other components of the middleware system. An instance of the observation worker application fetches the observations from RabbitMQ, inserts the observations into MongoDB, converts the observations into a semantically annotated format, and then inserts the results into the triplestore. The throughput of the observation worker can be correlated to the throughput of the entire middleware system. To evaluate its performance, we simulated an operational scenario by publishing 1,000 test observations to the RabbitMQ servers, and let the observation worker perform its processes. We observed the time taken by one instance of the observation worker to complete the processing of the 1,000 observations.

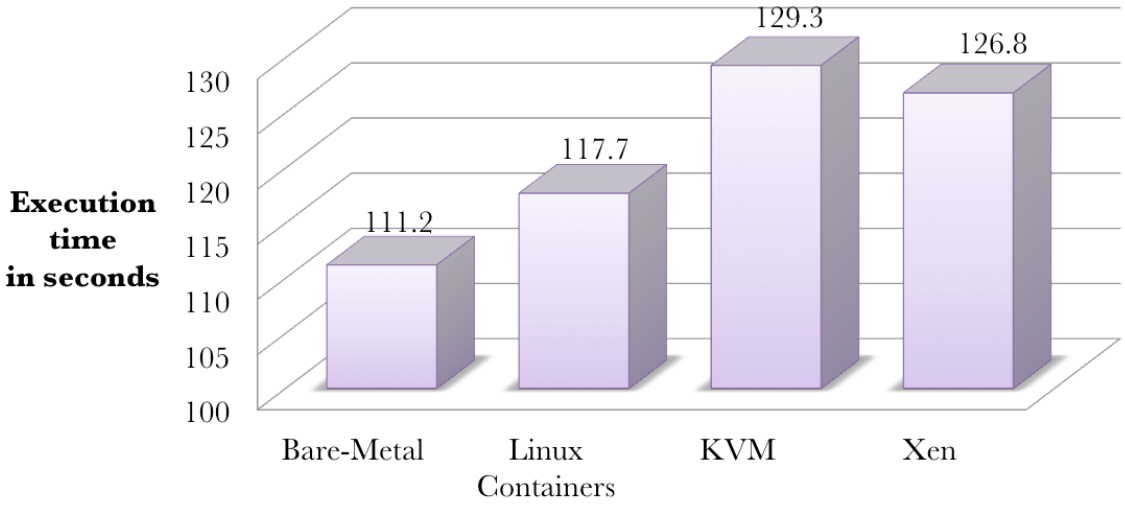


Figure 4.26: Observation Worker Performance

Figure 4.26 summarizes the time taken by an observation worker to complete the processing of the 1,000 observations. It is evident from the results that Linux Containers provide the best performance among the virtualization platforms in virtualizing the observation worker application.

Based on the results above, we conclude that Linux Containers are the best platform choice

to virtualize the Intelligent River® middleware system.

Chapter 5

Operational Flexibility

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 considered carefully 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 common tasks, such as virtual machine provisioning, cloning, booting, and rebooting ?

2. What are the features and operational constraints associated with each platform ?

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. Consider two real-world examples of what enterprises expect out of their virtualization systems. 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 highly dynamic. Being able to dynamically instantiate new virtual machines, bring them down, and scale a virtual machine's resources are important. Second, consider an e-commerce company that prioritizes reliability, availability, and scalability over flexibility in common execution tasks. Such a company needs compliance to industry standards, superior resource isolation, security, and device compatibility. To identify the best suited virtualization platform for enterprises with such varying needs, it is important to evaluate the platforms based on their operational characteristics, design constraints, and management facilities.

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 using an automated installation procedure [96]; (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; and (iv) clone a virtual machine and all its resources to create a new virtual machine.

Virtualization platform	**Time required to provision a virtual machine (seconds)**
KVM	532
Xen	718.5
Linux containers	27.502

Table 5.1: Operational Flexibility - Provisioning a VM and Installing Ubuntu Server

Table 5.1 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. The root filesystem 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 do not create virtual devices for the containers, but only initialize their own namespace in the host

kernel and use cgroups for resource management. The unique design of Linux Containers gives it the advantage of being very fast when creating a new container. Although both KVM and Xen perform similar processes during provisioning and installation, we observed KVM to be faster than Xen.

Creation of the virtual machine and the installation of the operating system is a one-time activity in the lifecycle of a virtual machine, whereas booting and rebooting a virtual machine are common operational actions. We observed the time required for each virtualization platform to complete the guest operating system boot process, and the time required to reboot a running virtual machine.

Virtualization platform	Time required for boot (seconds)	Time required to reboot (seconds)
KVM	5.86	38
Xen	5.34	11
Linux containers	0.053	10

Table 5.2: Operational Flexibility - Booting and Rebooting a VM

The results summarized in table 5.2 show that a Linux container starts up and becomes available for use in a fraction of the time required by KVM and Xen. Since the container does not have a kernel to boot, it simply spawns a group of processes on the host kernel and becomes available very quickly. Similarly, the reboot process is very quick; it must only de-allocate its resources on the host operating system and start afresh.

It is common practice to create a standard virtual machine (often referred to as a golden image) with all the required customizations, and to then clone the golden image whenever a new virtual machine is provisioned. We observed the time required by each of the virtualization platforms to create a new virtual machine by cloning the golden image.

Virtualization platform	Time required to clone a VM from a disk image (seconds)
KVM	35
Xen	215
Linux containers	0.553

Table 5.3: Operational Flexibility - Cloning a VM from disk image

The results shown in Table 5.3 once again favor Linux Containers by a significant margin.

The difference between the results for KVM and Xen are due to the storage format used by KVM. KVM uses copy-on-write (QCOW2) as the default storage mechanism, resulting in a smaller disk footprint, whereas Xen uses a raw disk image by default, resulting in a need to create the entire provisioned disk.

These results again support our earlier claims that Linux Containers trade isolation for operational benefits. They are most suitable in situations demanding dynamic creation and deletion of virtual machines. KVM and Xen are more appropriate for environments that require superior isolation and extended runtimes without reboots. Note that only KVM and Xen can virtualize the applications that require kernel customization, as a Linux container cannot modify the host kernel.

5.2 Operational Features and Constraints

While evaluating the virtualization platforms from an operational perspective, the features and constraints associated with the platforms play an important role. Often times, application infrastructure may demand a set of features from the virtualization platform for effective operation. For example, an application that requires frequent kernel updates will not be a good candidate to be virtualized on a platform that does not allow isolated kernel updates. The following table summarizes a key set of features and constraints associated with each of the virtualization platforms.

Feature/Constraint	**Linux Containers**	**KVM**	**Xen**
Virtual machines running an operating system different than the host	**No**. (A 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 [76])	**Yes**.	**Yes**.
Memory over-commitment	**Yes**. (Backed by virtual memory on the host)	**Yes**. (Backed by virtual memory on the host)	**No**. (Para-virtual virtual machines)
CPU over-commitment	**Yes**. (CPUs are shared by default, controlled by cgroups)	**Yes**. (Limited by the max-vcpus configuration)	**Yes**. (Limited by configuration parameters)

Table 5.4: Operational Flexibility - Features And Constraints

Feature/constraint	**Linux Containers**	**KVM**	**Xen**
Leverage hardware extensions in CPU	**No**.	**Yes**. (Required)	**No** (Para-virtualized virtual machines)
Memory density	**High** (Lowest memory footprint. 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 impact all 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 is part of the supported mainline Linux kernel	**Yes.**	**Yes.**	**Yes**. (Linux with mainline kernel can run as Dom0 and DomU, but the hypervisor is separate)

Table 5.5: Operational Flexibility - Features And Constraints

Chapter 6

Resource Entitlement

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

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 tries to utilize all four CPUs accessible to it, the other virtual machines begin to starve, even though they hold other resources, such as memory, network, and disk I/O. Such situations must be avoided, and individual machines must be guaranteed their minimum entitled share of the resources at all times. This chapter analyzes the default isolation effectiveness provided

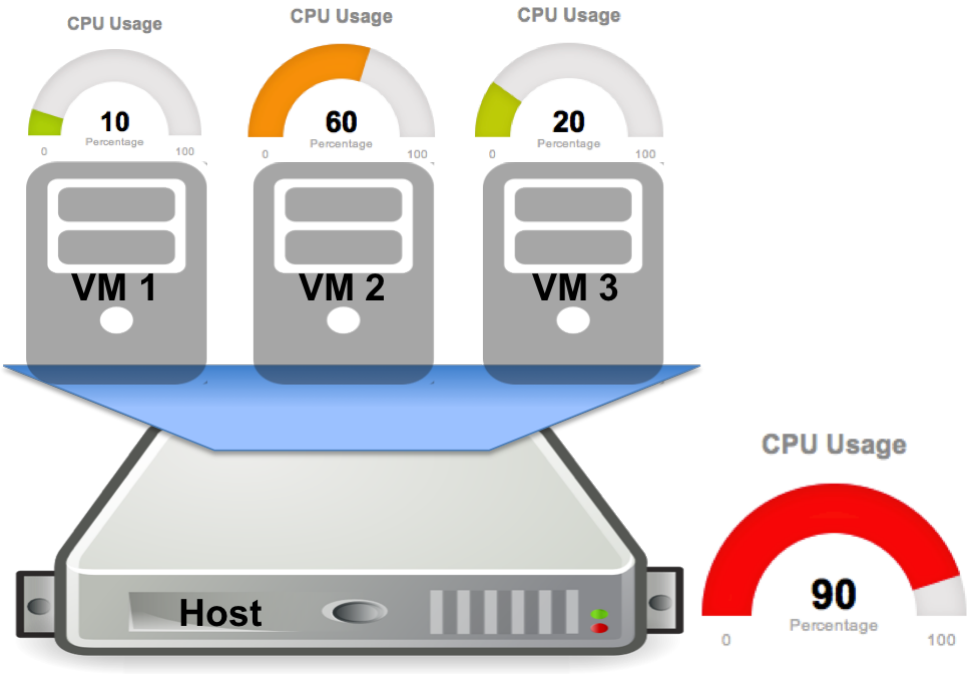
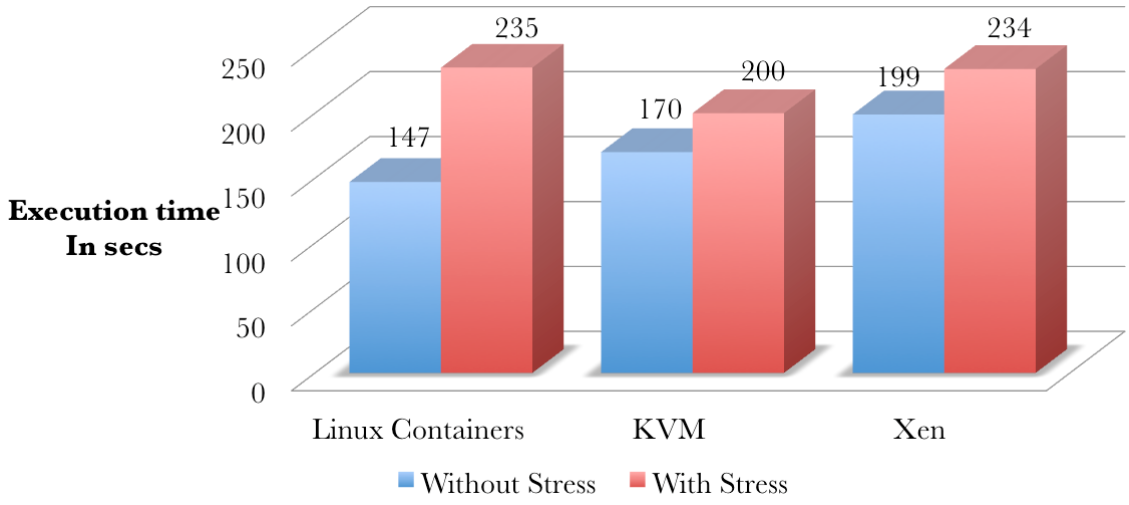


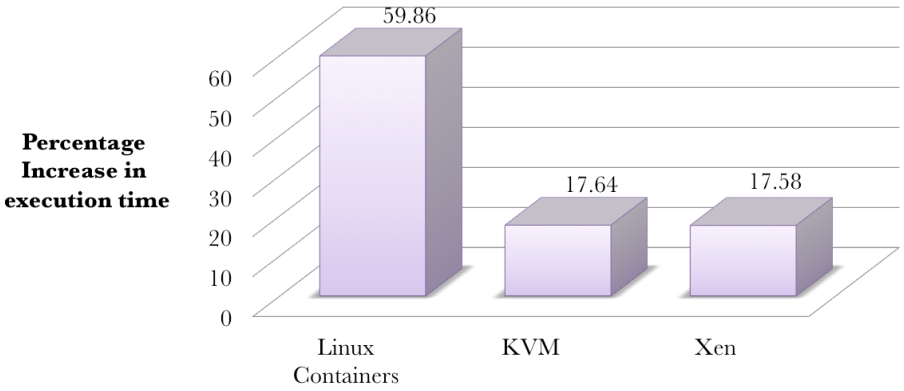
Figure 6.1: Virtualization - CPU Utilization Perspective

by three 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 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 physical servers, sharing all of the host's resources between them, and then measured the change in execution times of the sample applications (w1, w2, w3, and w4) running on one virtual machine when stress was created on the other virtual machine using the *stress* [20] tool.



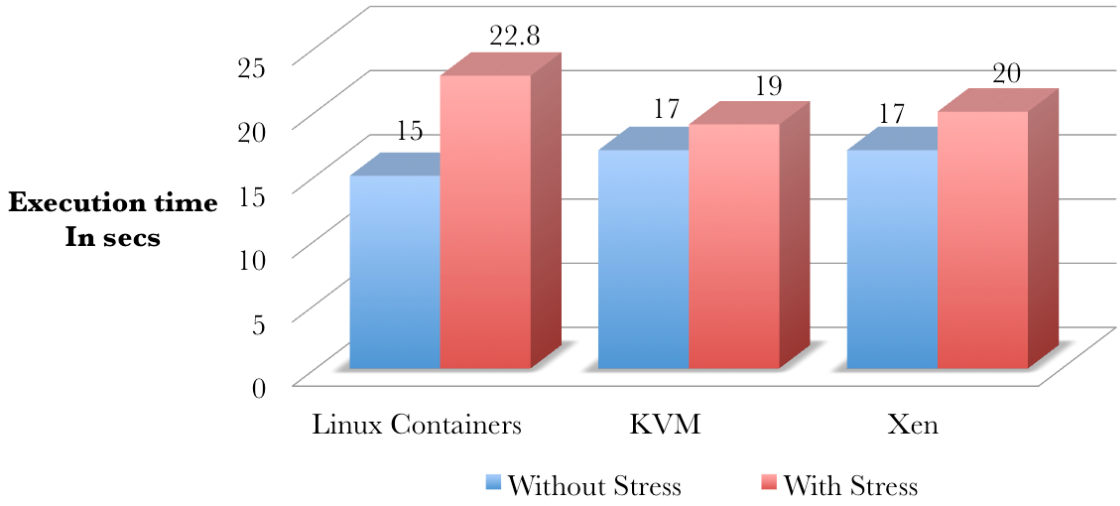
(a) Impact of stress on sample application w1



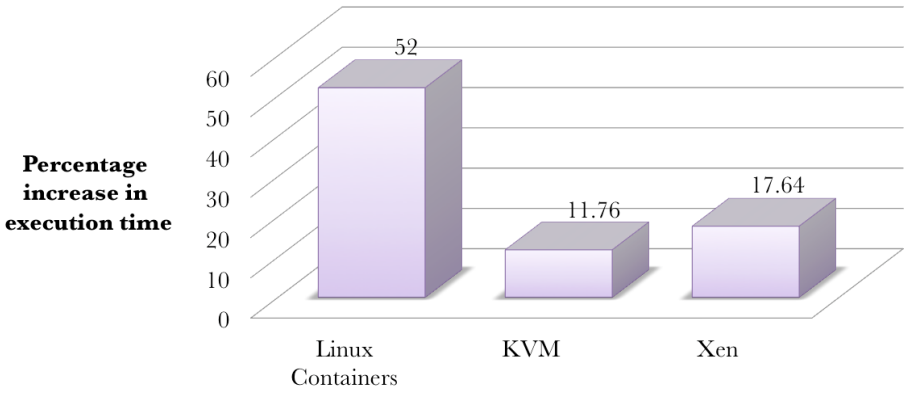
(b) Impact of stress (percentage increase in execution time)

Figure 6.2: CPU Isolation Efficiency

Figure 6.2a summarizes the change in execution time of the sample application w1 (performing CPU-intensive tasks) on each of the virtualization platforms. The X-axis represents the virtualization platforms, and the Y-axis represents execution time, both before and after stress was applied. Based on the results shown in Figure 6.2a, the 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 virtualization platform. In other words, in an effectively isolated environment, the increase in execution time would be 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 respect to CPU.



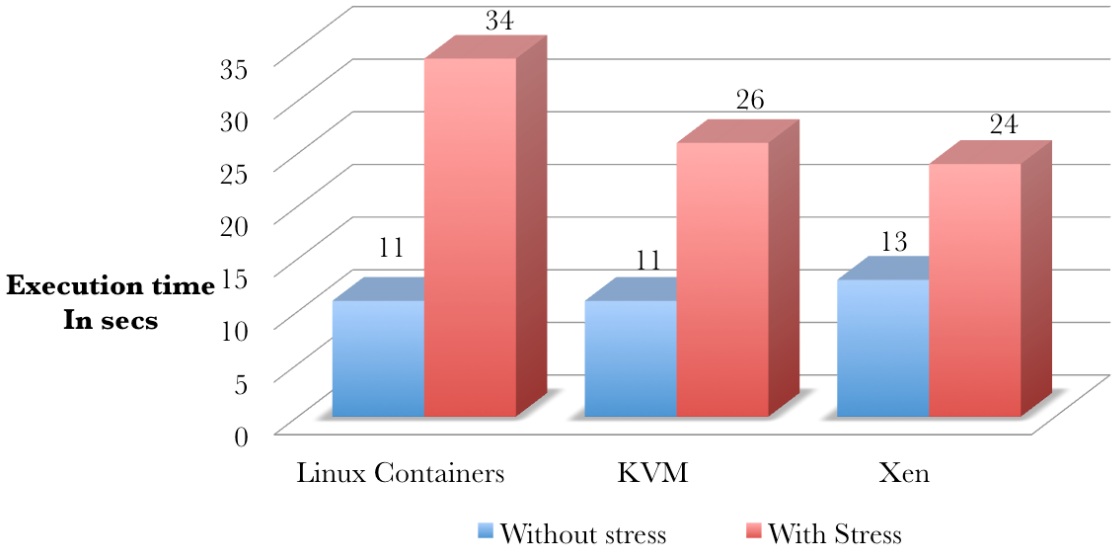
(a) Impact of stress on sample application w2



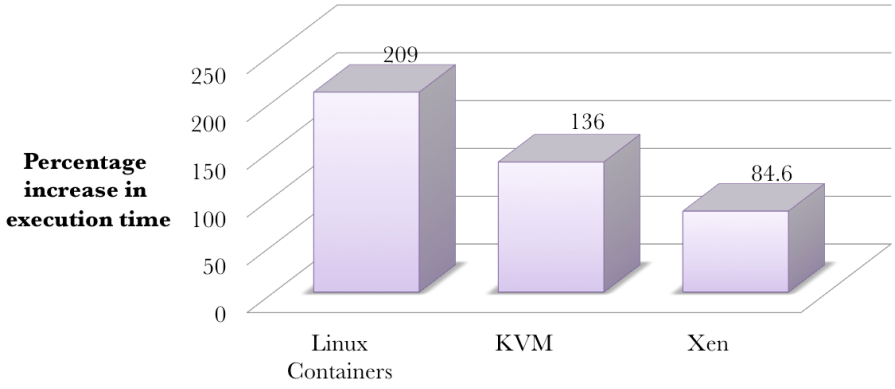
(b) Impact of stress (percentage increase in execution time)

Figure 6.3: Memory Isolation Efficiency

Figure 6.3a summarizes the change in execution time of the sample application w2 (performing memory intensive tasks) on each of the virtualization platforms. The X-axis represents the virtualization platforms, and the Y-axis represents execution time, both before and after stress was applied. Based on the results shown in Figure 6.3a, the percentage increase in execution time 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 virtualization platform. In other words, in an effectively isolated environment, the increase in execution time would be minimal. It is evident from the results that KVM offers superior memory isolation for the virtual machines, while Linux Containers are poorly isolated with respect to memory.



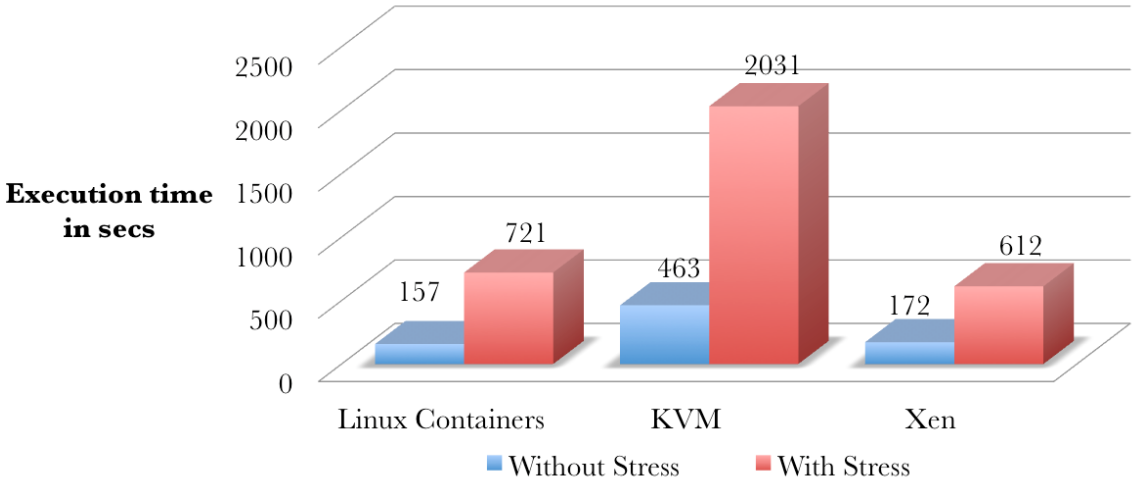
(a) Impact of stress on sample application w3



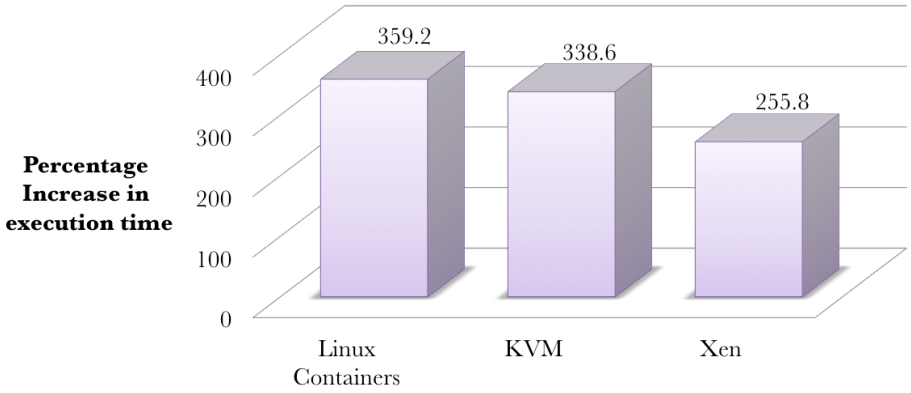
(b) Impact of stress (percentage increase in execution time)

Figure 6.4: Network Isolation Efficiency

Figure 6.4a summarizes the change in execution time of the sample application w2 (performing network I/O operations) on each of the virtualization platforms. The X-axis represents the virtualization platforms, and the Y-axis represents execution time, both before and after stress was applied. Based on the results shown in Figure 6.4a, the percentage increase in execution time 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 an effectively isolated environment, the increase in execution time would be minimal. It is evident from the results that Xen offers superior network isolation for the virtual machines, while Linux Containers are poorly isolated with respect to network bandwidth.



(a) Impact of stress on sample application w4



(b) Impact of stress (percentage increase in execution time)

Figure 6.5: Disk I/O Isolation Efficiency

Figure 6.5a summarizes the change in execution time of the sample application w4 (performing a large number of disk I/O operations) on each of the virtualization platforms. The X-axis represents the virtualization platforms, and the Y-axis represents execution time, both before and after stress was applied. Based on the results shown in Figure 6.5a, the percentage increase in execution time was calculated for each platform and summarized in Figure 6.5b. The percentage increase in execution time is inversely proportional to the efficiency of isolation for disk I/O operations offered by the virtualization platform. In an effectively isolated environment, the increase in execution time would be minimal. It is evident from the results that Xen offers superior isolation for disk I/O operations, while Linux Containers and KVM provide poor isolation with respect to

disk I/O operations.

The results summarized in Figures 6.2, 6.3, 6.4, and 6.5 show that none of the virtualization solutions provide effective resource isolation, as stress applied on one virtual machine severely affected the performance of the application running on the other virtual machine. This lack of isolation would translate into unpredictable performance of applications running in virtualized infrastructure. The rest of this chapter addresses this issue by describing the facilities available to define limits and assure virtual machines of their entitlement in terms of CPU, memory, network bandwidth, and disk I/O. These facilities are applicable to all of the virtualization platforms.

Before considering the mechanisms used to configure dedicated system resources for virtual machines, it is important to analyze and understand the nature of the workloads that the virtual machines will run. This information is helpful in two regards:

1. Prior knowledge of the workloads can be used in deciding which virtual machines to provision on the same server. For example, a virtual machine that runs a web server (i.e, using more network and memory resources), and a virtual machine that runs a reporting application (i.e, with more disk I/O) are both ideal candidates to be virtualized on the same physical server, as they utilize different types of resources. On the other hand, running many virtual machines executing database workloads on the same physical machine would not be a good practice, as they would contend for the same types of resources.

2. The characteristics of the application can be used in deciding which resources to dedicate and which resources to share.

Planning the resource policies for CPU and memory resources go 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 understand their architecture in the physical server. All modern server class hardware is equipped with multi-core processors architected to efficiently scale by distributing system memory near the individual CPUs. This configuration is known as *Non-Uniform Memory Access (NUMA)*.

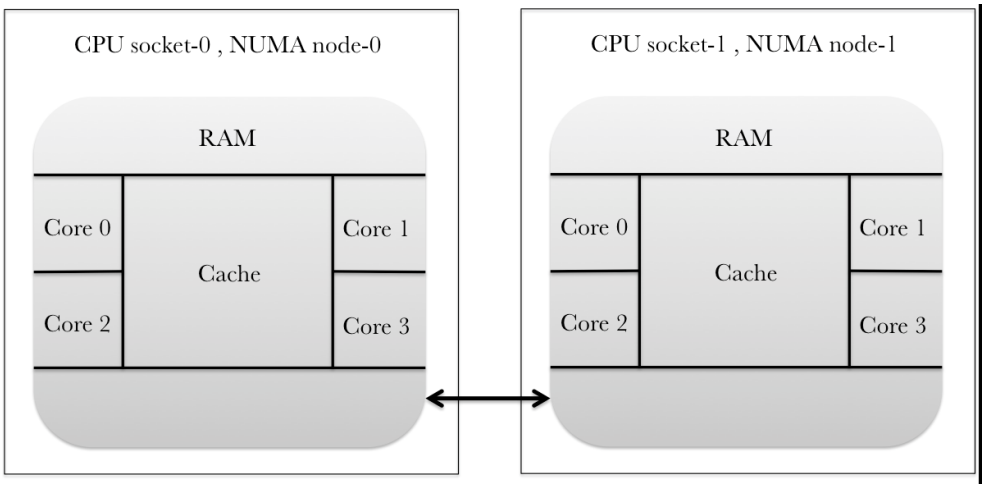


Figure 6.6: Hex-Core CPU Configuration (2 sockets, 2 node NUMA)

Figure 6.6 shows a block diagram of a hex-core physical server in a 2-node NUMA configuration. CPU socket-0 and CPU socket-1 consist 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 on NUMA node-0 can be accessed faster by the processing cores contained in the CPU Socket-0. The memory on NUMA node-1 can be accessed faster by the processing cores contained in CPU socket-1.

Memory access across NUMA nodes will be relatively slow (e.g., a processing core in CPU socket-0 accessing 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** [57] - Used to display an inventory of available nodes on the system

While analyzing the workloads with respect to CPU and memory entitlements, it is important to identify whether the application is multi-threaded. If yes, we must determine whether multiple threads are doing independent CPU-intensive tasks, or are dependent on each other and shared memory resources. By default, the Linux scheduler tries to keep all 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 [95]. If the applications are single-threaded, the CPU reservations can be straightforward; "pinning", or assigning an "affinity" to the CPU, binding it to a physical CPU core will improve the performance of the application running on the virtual machine. CPU pinning (or setting an affinity) refers to assigning a specific process or virtual machine to a particular CPU core, such that the virtual machine is always scheduled (only) on that particular core. In a carefully planned system, if all virtual machines are limited to appropriate set(s) of CPU cores, the virtual machines can run on their dedicated cores with little interruption, leading to improved performance. It is also useful to note that access to an additional CPU core that may be shared across the system would help in alleviating interruptions by offloading 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 would improve parallelism, and hence improve performance. On the other hand, pinning virtual machines running multiple CPU-intensive threads to a 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 can be implemented using the following Linux commands.

- **Taskset** [70] - Used 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`
  pins the kvmtest1 process to the cores, 0, 2, and 4.

- **Numactl** [57] - Used to confine a process to a NUMA node, i.e, multiple cores sharing the same L3 cache. This helps to prevent virtual machines from being scheduled across NUMA pools.
  Example: `numactl --cpunodebind=0 --membind=0 kvmtest1`
  binds the guest kvmtest1 to the first CPU Socket, and also restricts memory use to the associated NUMA pool.

Though CPU pinning and NUMA node assignment limit virtual machines to confined resource

pools, it may be impractical to use these mechanism in situations where the number of virtual machines is much greater than the number of processing cores. In such scenarios, a more granular resource management mechanism, such as cgroups(control groups) (discussed in chapter 3) work well. cgroups are a Linux kernel feature that allows limiting and accounting of resources at a granular level. cgroups enable administrators to assign relative CPU shares to the virtual machines, indicating their priorities when accessing CPU resources, while still obeying the policies set by taskset and numactl.

The most convenient method to achieve network isolation among virtual machines is to dedicate individual network interface hardware on the host to each virtual machine. This method is often justified by the relatively low cost of network interface cards. In situations where network bandwidth needs to be shared among 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 is relatively simple. In most cases, a physical server contains several hardware adapters for the locally attached storage or block devices, each dedicated to a virtual machine. The disk I/O operations that a virtual machine performs can be specifically rate-limited for reads/write operations by individual processes using cgroup hierarchies. The blkio subsystem of cgroups enables throttling and accounting of I/O operations across the Linux I/O scheduler. On a performance note, virtual machines tend to perform better with the deadline I/O scheduler using the "nocache" option, rather than the default CFQ I/O scheduler, due to the avoidance of double caching.

Chapter 7

Conclusion

The pursuit of making server infrastructure more dynamic, efficient, and cost-effective has led to the evolution of virtualization technologies in multiple dimensions. As a result of this evolution, we now have a diverse set of virtualization platforms to choose from, each prioritizing a different subset of the overall virtualization goals. There is a clear need to analyze and understand the focus of each of these platforms, and to evaluate the platform choices based on factors beyond the set of standard benchmarks. We chose KVM, Xen, and Linux Containers to represent full-virtualization, para-virtualization, and container-based virtualization, respectively. We evaluated the platforms based on (i) the overhead they impose to virtualize CPU, memory, network access, and disk access; (ii) their performance on workload-specific benchmarks; (iii) their performance in virtualizing the Intelligent River® middleware components; (iv) the flexibility each offers in performing common operational tasks; and (v) the isolation they provide for applications.

Through a detailed experimental analysis, we found that Linux Containers exhibit the least overhead in virtualizing CPU and disk access, whereas KVM exhibits the least overhead with respect to memory. Xen was found to exhibit significant overhead when used to virtualize multi-threaded aplications. KVM, in its current form, exhibits significant overhead in virtualizing disk access; however, we note that the problem is being addressed with major design changes slated for release in the near future. From an operational standpoint, Linux Containers provide a significant advantage by performing the common tasks of provisioning, booting, and rebooting a virtual machine several times faster than both KVM and Xen. Although Linux Containers fare well on most of our evaluation criteria, they come with the downside of providing poor isolation among the containers. Xen was

found to offer superior isolation among the virtual machines.

There is no single *best* virtualization platform. Based on our results, we conclude that Linux Containers are preferred to virtualize infrastructure that is dynamic by design, and does not require a high degree of resource isolation. It is important to note, however, that Linux Containers cannot be used in situations where the applications need kernel-level customization. For infrastructure that demands superior resource isolation, KVM is preferred over Xen, if most of the applications are memory intensive; whereas Xen is preferred over KVM if the applications involve significant disk access.

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. Tun tap interfaces. `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] 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.

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

[15] 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.

[16] 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).

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

[18] Jeff Dike. User-mode linux. `http://user-mode-linux.sourceforge.net/`, November 2013 (Last Accessed).

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

[20] Linux Kernel documentation. Stress - tool to impose load on and stress test systems. `http://manpages.ubuntu.com/manpages/precise/man1/stress.1.html`, November 2013 (Last Accessed).

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

[22] Jack Dongarra et al. Linpack. `http://www.netlib.org/linpack/`, October 2013 (Last Accessed).

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

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

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

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

[27] The Apache Software Foundation. Apache cloudstack - open source cloud computing. `http://cloudstack.apache.org/`, November 2013 (Last Accessed).

[28] The Apache Software Foundation. Apache jena-tdb. `http://jena.apache.org/documentation/tdb/`, November 2013 (Last Accessed).

[29] The Linux Foundation. The xen project. `http://www.xenproject.org/`, November 2013 (Last Accessed).

[30] 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.

[31] Jacques Glinas. Linux-vserver. http://linux-vserver.org/Welcome_to_Linux-VServer.org, November 2013 (Last Accessed).

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

[33] 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).

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

[35] Heroku. Heroku - cloud application platform. http://www.heroku.com/, November 2013 (Last Accessed).

[36] 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).

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

[38] IBM. z/vm operating system. http://www.vm.ibm.com/, November 2013 (Last Accessed).

[39] Amazon Web Services Inc. Amazon elastic compute cloud. http://aws.amazon.com/ec2/, November 2013 (Last Accessed).

[40] Docker inc. Docker: The linux container engine. https://www.docker.io/, November 2013 (Last Accessed).

[41] MongoDB Inc. Mongodb - an open-source document database. http://www.mongodb.org/, November 2013 (Last Accessed).

[42] MongoDB Inc. Mongodb - sharding concepts. http://docs.mongodb.org/manual/sharding/, November 2013 (Last Accessed).

[43] Red Hat Inc. 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).

[44] Red Hat Inc. Openshift by redhat. http://www.openshift.com/, November 2013 (Last Accessed).

[45] VMware Inc. Vmware virtualization. http://www.vmware.com, November 2013 (Last Accessed).

[46] 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).

[47] 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).

[48] 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).

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

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

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

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

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

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

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

[56] 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).

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

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

[59] Innovative Computing Laboratory. Hpc challenge benchmark. http://icl.cs.utk.edu/hpcc/, November 2013 (Last Accessed).

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

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

[62] Linux. dd - utility to convert and copy a file. http://linux.die.net/man/1/dd, October 2013 (Last Accessed).

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

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

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

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

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

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

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

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

[71] Rabbit Technologies Ltd. Rabbitmq - open source message broker. http://www.rabbitmq.com/, November 2013 (Last Accessed).

[72] Linux Programmer's Manual. aio - posix asynchronous i/o overview. http://man7.org/linux/man-pages/man7/aio.7.html, October 2013 (Last Accessed).

[73] John McCalpin. Stream. http://www.cs.virginia.edu/stream/, October 2013 (Last Accessed).

[74] William Norcott. Iozone - filesystem benchmarking utility. https://www.iozone.org, October 2013 (Last Accessed).

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

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

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

[78] Oracle. Oracle vm virtualbox. https://www.virtualbox.org/, November 2013 (Last Accessed).

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

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

[81] The OpenStack project. Openstack - open source cloud computing software. http://www.openstack.org/, November 2013 (Last Accessed).

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

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

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

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

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

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

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

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

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

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

[92] Open source software. Kernel based virtual machine (kvm). http://www.linux-kvm.org/page/Main_Page, November 2013 (Last Accessed).

[93] SPEC. Standard performance evaluation corporation. http://www.spec.org/, November 2013 (Last Accessed).

[94] Willy Tarreau. Haproxy - the reliable, high performance tcp/http load balancer. http://haproxy.1wt.eu/, November 2013 (Last Accessed).

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

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

[97] 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.

[98] D.L. White, S. Esswein, J.O. Hallstrom, F. Ali, S. Parab, G. Eidson, J. Gemmill, and C. Post. The intelligent river : Implementation of sensor web enablement technologies across three tiers of system architecture: Fabric, middleware, and application. In *Collaborative Technologies and Systems (CTS), 2010 International Symposium on*, pages 340–348, 2010.

[99] 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.

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

[101] 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.

[102] 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.