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

---

A Thesis
Presented to
the Graduate School of
Clemson University

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

---

by
Aravindh Sampath Kumar
December 2013

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

Abstract

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

This thesis evaluates representative implementations of each approach: (i) full-virtualization - *KVM*, (ii) para-virtualization - *Xen*, and (iii) container-based virtualization - *Linux Containers*. First, this thesis discusses the design principles behind the chosen virtualization solutions. Second, this thesis evaluates them based on the overhead they impose to virtualize system resources. Finally, this thesis assesses the benefits and limitations of each based on their operational flexibility, and resource entitlement and isolation facilities.

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

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The adoption of cloud computing paradigm has changed the 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 [32].

The technology backbone behind the idea of cloud computing is *virtualization*. Virtualization is the process of creating multiple isolated operating environments for the applications, that share the physical resources of a server. Virtualization addresses the problem of under-utilization of hardware resources in the dedicated server model by running multiple isolated applications simultaneously on a physical server, also referred to as "host". Such an isolated operating environment created for the applications are referred to as virtual machines (VM) [76] or containers [52]. Each of the virtual machines or containers provides an abstracted hardware interface to their applications, and utilizes as much computational resources as they need (or are available) from the host.

Virtualization gained a wide-spread adoption because of multiple benefits it has to offer. First, virtual machines can be state-fully migrated from one physical machine to another in the event of a hardware failure [14]. This capability can also be used to dynamically re-position virtual machines in such a way that the virtual machines that frequently work together are virtualized on the same physical server, improving the performance of both the virtual machines. The virtual machines can also be automatically migrated to achieve a certain pre-defined goal such as power efficiency, and load balancing [72]. 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 the other virtual machines on the host. Third, virtualization enables an additional layer of control between the applications and the hardware, providing more options to effectively monitor and manage the resources.

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

1.1 Motivation

The Intelligent River® is a large scale environmental monitoring platform being built by researchers at Clemson University [85]. The Intelligent river® project aims to deploy a large distributed wireless sensor network in the Savannah river basin and uses a real-time distributed middleware system to receive, analyze and visualize the observation streams. The volume and unpredictable nature of the observations streams, along with the increasing scale of sensor deployments demand the middleware system to be flexible, fault-tolerant, and scalable. A key factor in architecting such a dynamic system is virtualization. But, given the availability of multiple types of open source virtualization platforms such as KVM [80], Xen [27], and Linux Containers [52], we needed to identify the most suitable virtualization platform to virtualize the components of the Intelligent River® middleware system. To make this important choice, we needed detailed analysis and comparison of the mentioned virtualization platforms. Prior research work on analysis of virtualization platforms for High Performance Computing (HPC) applications [88] claims that KVM performed better than Xen based on the HPCC benchmarks [48]. On the other hand, another research work claims that OpenVZ [65], a virtualization platform based on Linux Containers performed the best, while Xen performed significantly better than KVM, clearly contradicting the prior claims. These contradictions indicate that, comparing the virtualization platforms only based on standard benchmarks is not sufficient to arrive at an answer and that several other results must be factored in making the

decision. This motivated us to perform : (i) a detailed study of operation of each of the virtualization platform, (ii) a quantitative analysis of virtualization overhead, (iii) analyze workload specific benchmarks, and (iv) a qualitative analysis of operational flexibility to identify the virtualization platform that will be best suited for the Intelligent River Middleware system.

1.2 Contributions

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

Chapter 2

Related Work

The core ideas of virtualization having stood the test of time have also prompted a variety of comparative studies by the academia as well as the industries. Andrew J. Younge et al. [88] compared Xen and KVM with virtual resources from the FutureGrid project [87] and claims, KVM performed significantly better than Xen, under the standard HPCC and SPEC benchmarks. Another benchmark oriented synthetic study [12] claims OpenVZ (the predecessor of Linux Containers) performed exceptionally well on almost all benchmarks. But, also claims that Xen performs significantly better than KVM contradicting the results of other publishers. A recent study by Miguel G. Xavier et al. [86] and an earlier study [77], again corroborates the fact that Linux Containers performs significantly better than alternative virtualization platforms, Linux-VServer, OpenVZ, and Xen based on a variety of benchmarks including LINPACK, STREM, IOZONE, NETPIPE, and NPB. They also highlighted the shortcomings in isolation, and sharing associated with Linux Containers which have already been addressed in the recent releases. A relatively old research work by Vincent Neri et al. [71] threw light on serious performance limitations on Xen under certain operational circumstances and also motivated the need for microbenchmarks.

Among the many new opportunities that were opened up by the relatively new and lightweight Linux Containers, network virtualization or Software Defined Networking (SDN) attracted lots of research attention. Studies, [16], and [75] discussed the performance benefits of large scale virtual network emulation by implementing open Vswitch [79] on Linux Containers based platforms. Mesos [25] , A clustered platform that provides fine-grained resource sharing among multiple frameworks like Hadoop [24] and MPI [15] utilizes Linux Containers to isolate and limit the CPU,

Memory, Network, and I/O resources. Also, Mircea Bardac et al. [3] showcased the scalability of Linux Containers by deploying a solution for testing large scale Peer-to-Peer systems.

Resource affinity, Isolation and Control are discussed in detail in this thesis. Vishal Ahuja et al. [1] provides an example of exploiting CPU isolation facilities to improve overall network throughput by pinning application, and protocol processing to the same processor core. The discussions in this thesis on providing resource affinity will enable Vishal Ahuja et al.'s work to be applicable to Linux Containers. Another interesting attempt by Zhaoling Guo and Qinfen Hao [28] made use of cgroups in combination with KVM to enable CPU affinity to achieve improved performance.

CoreOS [78], in its very early stages and being actively developed at the time of writing of this thesis is a very light weight operating system (Linux Kernel+systemd) that is built to run in containerized environments attempts to lower the overhead by eliminating redundant operating system functions. Docker [34] is an open source engine that leverages to automate the deployment of applications as lightweight, portable and self-sufficient containers.

Chapter 3

Background

This chapter provides the necessary background required to understand the results of the comparative studies presented in the subsequent chapters. A description of the design and operation of the representative virtualization platforms, KVM, Linux Containers, and Xen are presented followed by a discussion of simplified version of Intelligent River® middleware system 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 this problem. The linux community supports multiple mature virtualization platforms, leaving the choice to the end-users. This chapter provides a technical over the principles behind the operation of three representative virtualization solutions: Kernel-based Virtual Machine(KVM), Xen, and Linux Containers.

3.1.1 Kernel based Virtual Machines (KVM)

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

A hypervisor is a critical part of a stable operating environment as it is responsible for

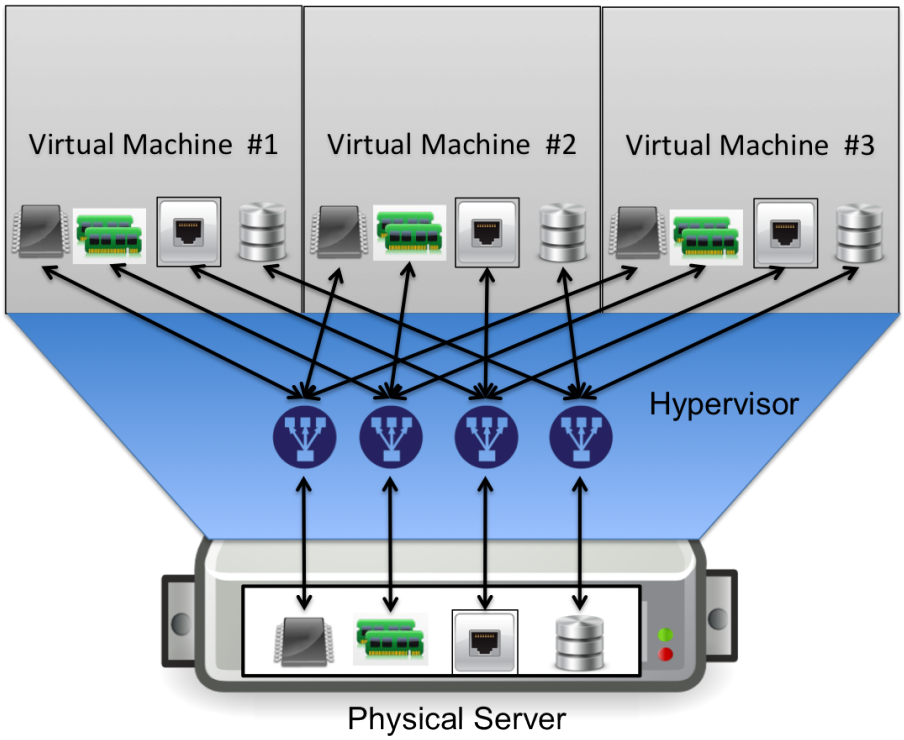


Figure 3.1: A Full-virtualization System

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

In the KVM based approach to virtualization, majority of the work is offloaded to the Linux 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) [53]. KVM itself is only part of the virtualization solution. It turns the Linux kernel into a Virtual Machine Monitor (VMM) (i.e, hypervisor), which enables several virtual machines to operate simultaneously, as if they are running on their own hardware. The emulated virtual devices and the virtual machine itself are created by an independent tool known as QEMU [56]. Hence the total solution is commonly referred as QEMU-KVM. KVM is packaged as a lightweight kernel module which implements the virtual machines as regular Linux processes, and therefore leverages the Linux kernel on the host for all the scheduling and device management

activities. Figure 3.2 shows the architecture of a server virtualized using QEMU-KVM. A server with a virtualization capable processor, disk storage, and network interfaces runs a standard linux operating system that contains KVM. Virtual machines co-exist with the user-space applications that may be running directly on the host. The virtual machines contain a set of virtual devices created using an user-space tool called QEMU, and run an unmodified guest operating system like Linux, Microsoft Windows.

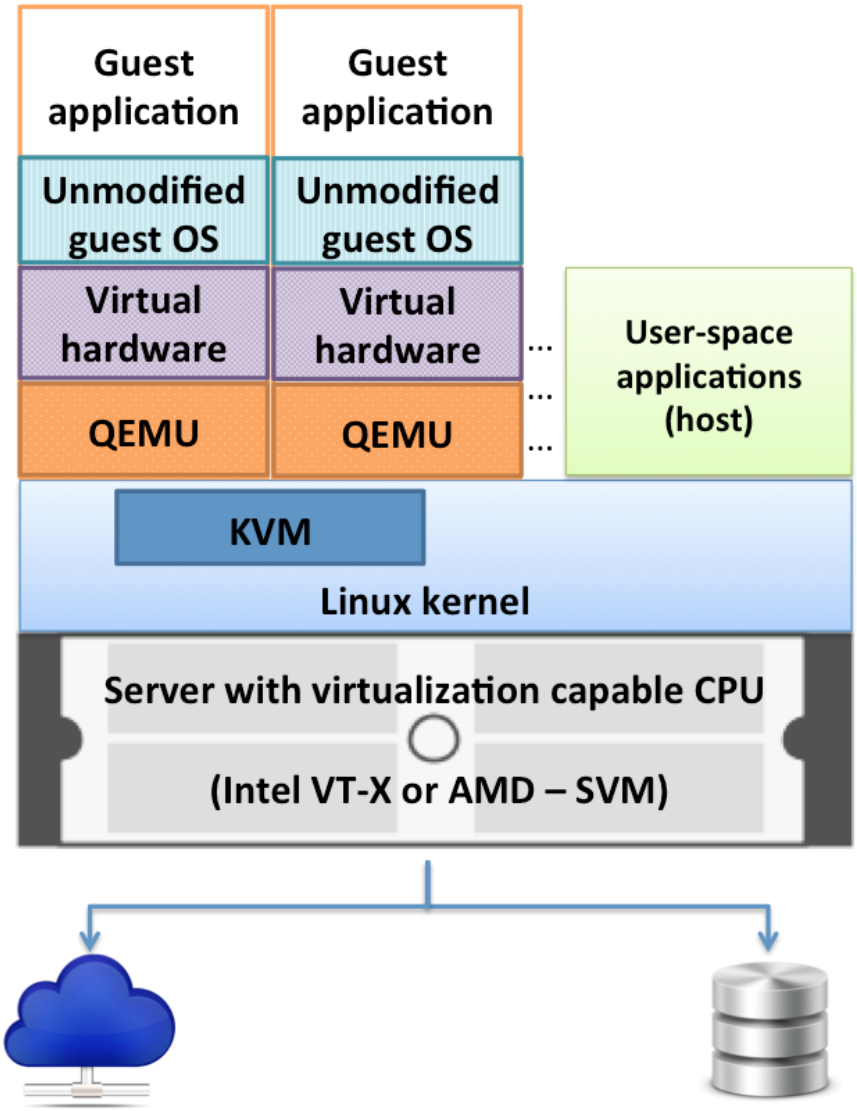


Figure 3.2: KVM - Architecture

In practice, KVM and the Linux kernel treat the virtual machines as regular Linux processes

on the host and perform the mapping of virtual devices to real devices in each of the following categories.

1. **CPU**:

   KVM requires the CPU to be virtualization aware. Intel VT-X [35] and AMD-SVM [2] are

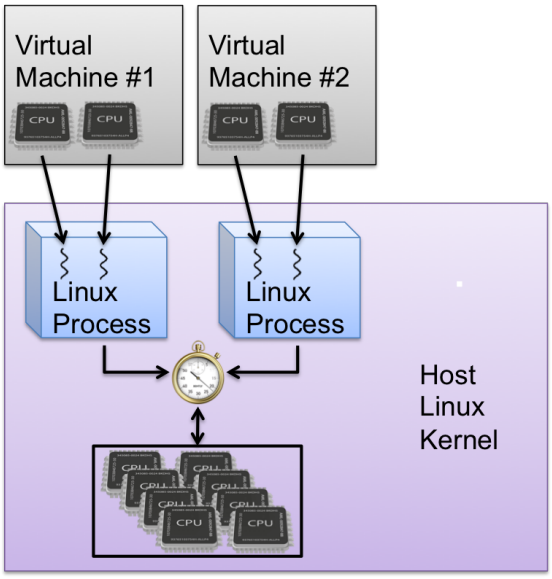


Figure 3.3: KVM - Virtual CPU Management

   the virtualization extensions provided by Intel and AMD, respectively, which are the most common CPUs used in the x86 servers. KVM relies on these facilities to isolate the instructions generated by the guest operating systems from those generated by the host itself. Every virtual CPU associated with a virtual machine is created as a thread belonging to the virtual machine's process on the host as shown in Figure 3.3. Hence, enabling multiple virtual CPUs improve the virtual machine's performance by utilizing the multi-threading facilities on the host. The virtual machine's CPU requests are scheduled by the host kernel using the regular CPU scheduling policies. Improving CPU performance on the guest and ensuring fair CPU entitlement are discussed in later sections.

2. **Memory**:

   KVM inherits the memory management facilities of the Linux kernel. The memory of a virtual machine is an abstraction over the virtual memory of the standard Linux process on the host. This memory can be swapped, shared with other processes, merged, and otherwise managed

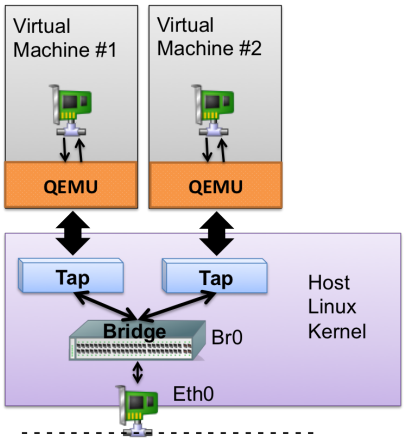
by the Linux kernel. Thus, the total memory associated with all the virtual machines on a host can be greater than the physical memory available on the host. This feature is known as *memory over-commitment*. Though memory over-commitment increases overall memory utilization, it creates performance problems when all the virtual machines try to utilize their memory share at the same time, leading to swapping on the host. Since KVM offloads memory management to the Linux kernel, it enjoys the support of NUMA (Non-Uniform Memory Access) awareness [54], and Huge Pages [51] to optimize the memory allocated to the virtual machines. NUMA support and Huge Pages will be discussed in the later sections.

3. **Network Interfaces**:

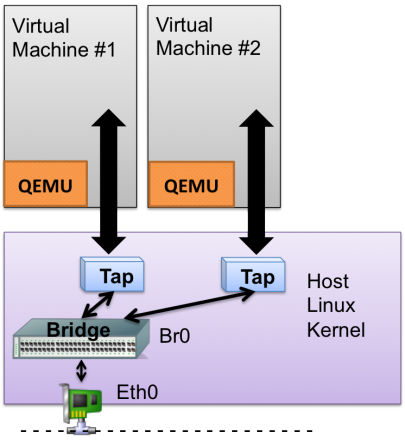
   Networking in a virtualized infrastructure enables an additional layer of convenience and control over conventional networking practices. Virtual machines can be networked to the host, among each other, or even participate in the same network segment as the host. Several configurations are possible, trading-off device compatibility and performance. Figure 3.4 shows the most common virtual networking options used in a KVM based infrastructure. Figure 3.4a shows a virtual networking configuration where unmodified guest operating systems use their native drivers to interact with their virtual network devices. Virtual network devices are connected to the Tap devices [8] on the host kernel. Tap interfaces are software-only interfaces existing only in the kernel; they relay ethernet frames to and from the Linux bridge. This setup trades performance to achieve superior device compatibility.

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

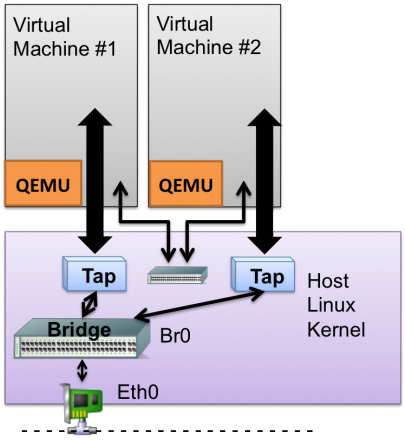
   Figure 3.4c shows a combination of both public and private networking. Two points are important to note. First, a virtual machine may have multiple virtual networking devices. Second, virtual machines can be privately networked using a Linux bridge that is not backed by a physical ethernet device. This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



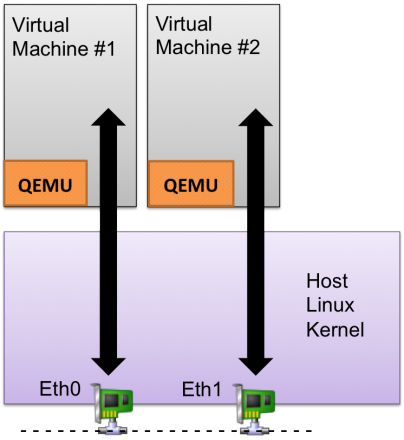
(a) Emulated Networking devices



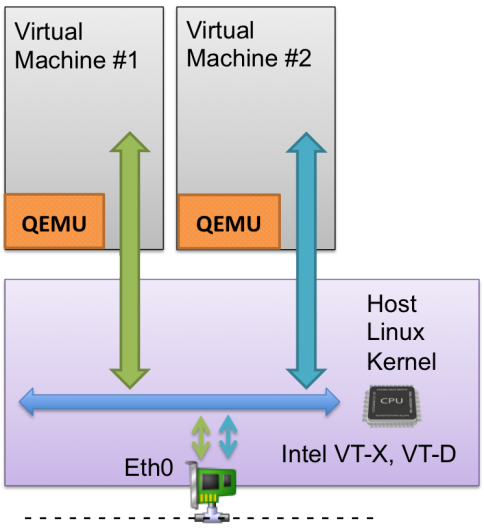
(b) Para-Virtualized Networking



(c) Public and Private Networking



(d) Direct Device Assignment



(e) Direct Device Assignment with SR-IOV

Figure 3.4: Common Networking Options for KVM

plication server could then communicate with privately networked database server.

To support the virtualization of network intensive applications, physical network interfaces attached to the host can be directly assigned to the virtual machine, bypassing the host kernel and QEMU, as shown in Figure 3.4d. This setup provides "bare-metal" performance by providing direct access to the hardware, and is applicable when individual hardware devices are available to be dedicated to the virtual machines. The key to achieving this direct device access is the hardware assistance provided through Intel VT-D [37], 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) [36]. 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 incompetence to provide superior resource isolation and effective scalability [17]. Containers are the proposed solution. Digging deeper, hypervisors were created to isolate workloads and create virtual operating environments, with individual kernels optimally configured in accordance with workload requirements. But, the key question to be answered is, *Whether it is the responsibility of an operating system to flexibly isolate its own workloads.* If the linux kernel could solve this problem without the overhead and complexity of running several individual kernels, there would not be a need for

hypervisors!

The linux community saw a partial solution to this problem in BSD Jails [73], Solaris Zones [64], Chroot [49], and most importantly, OpenVZ [65] - a fork of the Linux kernel maintained by Parallels. BSD Jails were designed to restrict the visibility of the host's resources from a process. For example, when a process runs inside a jail, its root directory is changed to a sub-directory on the host, thereby limiting the extent of the file system the process can access. Each process in a jail is provided its own directory sub tree, an IP address defined as an alias to the interface on the host, and optionally, a hostname that resolves to the jail's own IP address.

The linux kernel's approach to solving the resource isolation problem is *"Containers"* incorporating the benefits of the above mentioned inspirations and more. The core idea is to isolate only a set of processes and their resources in containers without involving any device emulation, or imposing virtualization requirements on the host hardware. Like virtual machines, several containers can run simultaneously on a single host, but all of them share the host kernel for their operation. Isolated containers run directly on the bare-metal hardware using the device drivers native to the host kernel without any intermediate relays.

Containers expand the scope of BSD jails, providing a granular operating system virtualization platform, where, containers can be isolated from each other, running their own operating system, yet sharing the kernel with the host. Containers are provided their own independent file system and network stack. Every container can run its own distribution of Linux that may be different from the host. For example, a host server running RedHat Enterprise Linux [31] may run containers that run Debian [18], Ubuntu [10], CentOS [11], etc., or even another copy of RedHat Enterprise Linux. This level of abstraction in the containers creates an illusion of running virtual machines, as discussed earlier with KVM.

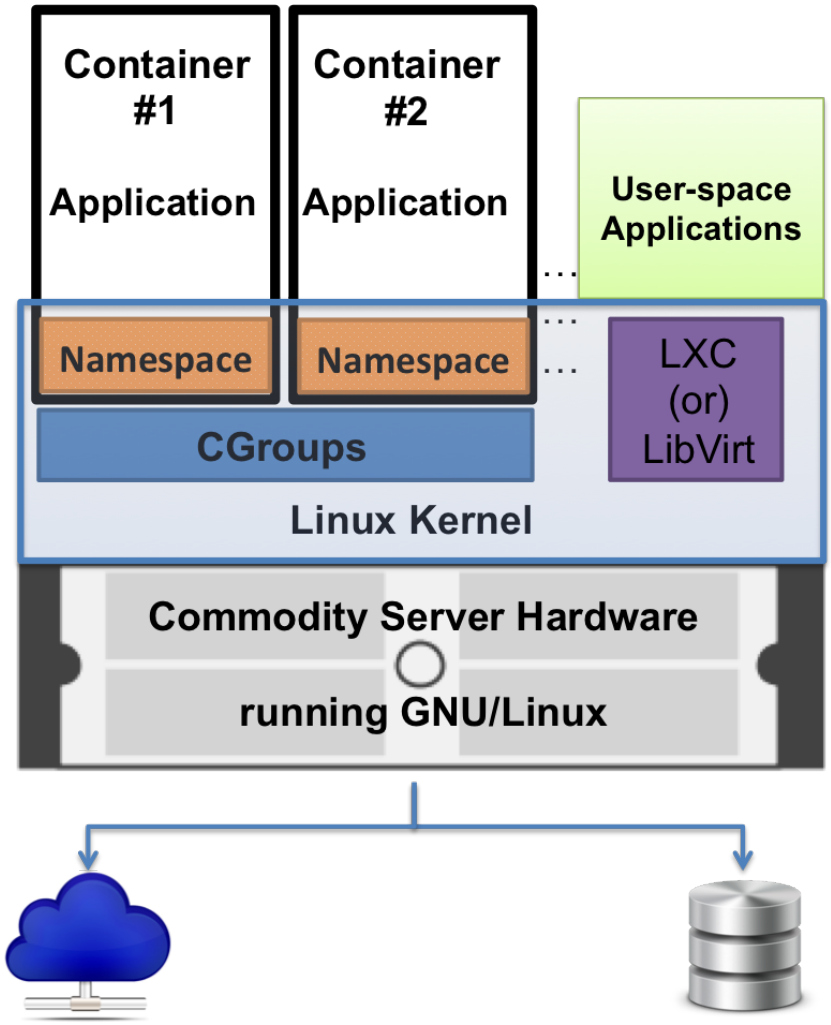


Figure 3.5: Linux Containers - Architecture

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

The following description of namespaces in the Linux kernel is based on [40], [41], [42], [38], [43], and [39]. Namespaces "wrap" a global system resource on the host and makes it appear to the processes within the namespace (containers) as though they have their own isolated instance of the global resource.

Linux implements 6 types of namespaces:

- **Mount namespaces :**

  Mount namespaces enable isolated file system trees to be associated with specific containers (or groups of regular Linux processes). A container can create its own file system setup, and the subsequent *mount()* and *unmount()* system calls issued by the process affect only its mount namespace, instead of the whole system. For example, multiple containers on the same host can issue *mount()* calls to create a mount point "/data", and access them at "/data" simultaneously. They will reside at different locations on the filesystem tree, e.g., "/<containername1>/data", "/<containername2>/data", and so on. Of course, the same setup can be achieved using the *chroot()* command, but, chroot can be escaped with certain capabilities, including CAP_SYS_CHROOT [19]. Mount namespaces provides a secure alternative. Mount namespaces greatly improve the portability of the containers as they can retain their filesystem trees irrespective of the host's environment.

- **UTS namespaces :**

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

- **IPC namespaces :**

  IPC namespaces isolate the System V IPC objects [57] and POSIX message queues [55] associated with individual containers.

- **PID namespaces :**

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

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

- **Network namespaces :**

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

- **User namespaces :**

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

Four new namespaces are being developed for future inclusion into the Linux kernel [7]:-

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

- **Security keys namespace :** This namespace aims to provide an independent security key space for containers to isolate the /proc/keys and /proc/key-users files based on the namespace [29].

- **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) [50] [74] [30] are another key feature of the Linux kernel, used to allocate resources such as CPU, memory, network bandwidth, disk access bandwidth among user-defined groups of processes (containers). cgroups instrument the Linux kernel to limit, prioritize, monitor and isolate system resources. With proper usage of cgroups, hardware and software resources on the host can be efficiently allocated and shared among several containers. cgroups can be dynamically re-configured and made persistent across reboots of the host. Though cgroups are generic and apply to individual processes and process groups, the remainder of this section discusses their relavance to containers.

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

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

Table 3.1: Cgroups - Subsystems

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

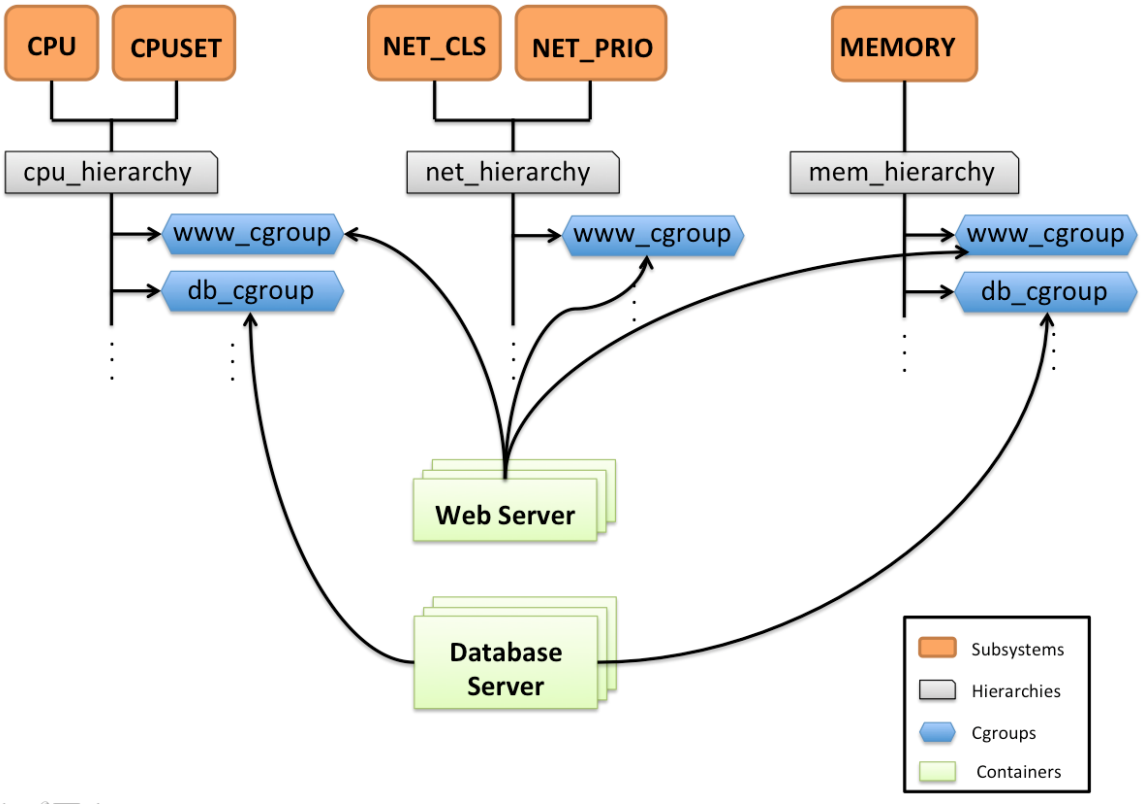


Figure 3.6: Cgroups Example Configuration

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

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

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

3.1.3 Xen

Xen [68] 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 [84] [89], a technique where the guest operating systems are modified to run on the host using an interface that is easier to virtualize. Para-virtualization significantly reduces overhead and improves performance according to [84], [4], [66].

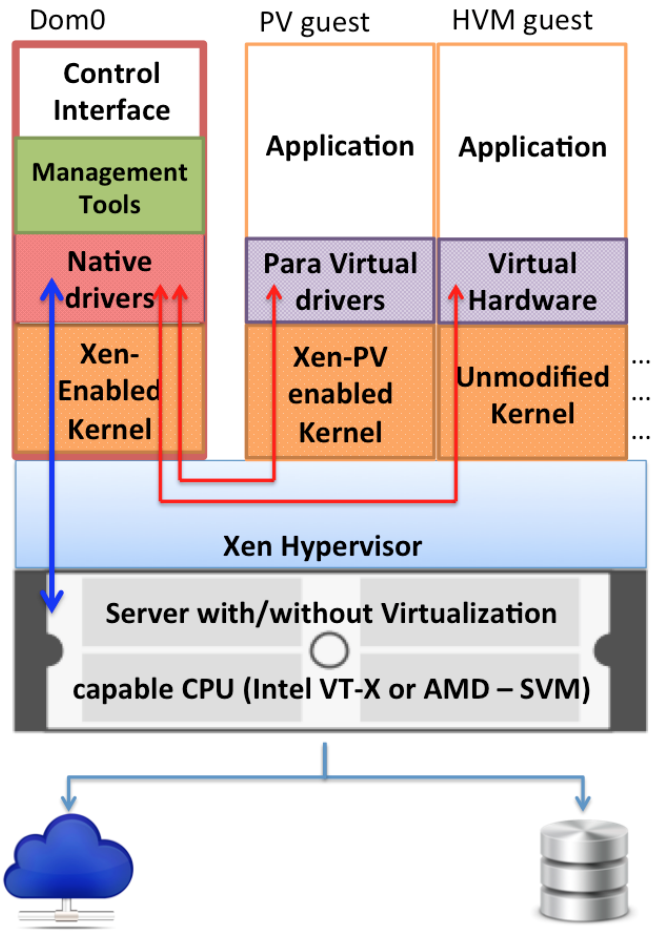


Figure 3.7: Xen - Architecture

The architecture of a system virtualized using Xen is shown in Figure 3.7. The Xen hypervisor is run directly from the bootloader. The hypervisor also referred to as the *Virtual Machine Monitor (VMM)*; it is responsible for managing CPU, memory, and interrupts. The virtual machines, referred to as *domains* or *guests* run on top of the hypervisor. A special domain referred to as *domain0* acts as a controller, containing the drivers for all the devices in the system. domain0 accesses the hardware directly, interacting with the other domains, and acts as the control interface for anyone to control the entire system. This controller domain also contains the set of tools to create, configure, and destroy virtual machines. It runs a modified version of the Linux kernel that can perform the role of the controller [67]. All other domains are totally isolated from the hardware and can use these resources only through the controller; they are referred to as *unprivileged domains (DomU)*. The DomUs can be either para-virtualized (PV) or hardware-assisted (HVM). The para-virtualized guests can run only modified operating systems, as they require Xen-PV-enabled kernel and PV drivers, which make them aware of the hypervisor. As an upside, para-virtualized guests do not require the CPUs to have virtualization extensions, and are usually lightweight compared to the unmodified operating systems. HVM guests can run unmodified operating systems, but require virtualization extensions on the CPU, just like KVM. The HVM guests use QEMU to emulate virtual hardware to provide the unmodified guest operating system. Both para-virtualized guests and hardware-assisted guests can run on a single system at the same time. Recent work on the Xen project also attempts to utilize the para-virtualized drivers on a HVM guest to improve performance, combining the best of both worlds.

3.2 Intelligent River® Middleware

Intelligent River® is an ongoing interdisciplinary research initiative at Clemson University that aims to deploy a 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, (iii) publishing the processed observations in multiple formats for further analysis and visualization. The comparative study presented in this thesis was motivated by the quest to identify the most suitable virtualization platform to virtualize all the components of the mentioned middleware system.

This chapter introduces the architectural model of the Intelligent River® middleware system, and describes the operational characteristics of its individual components so as to set the context for the comparative study presented in subsequent chapters.

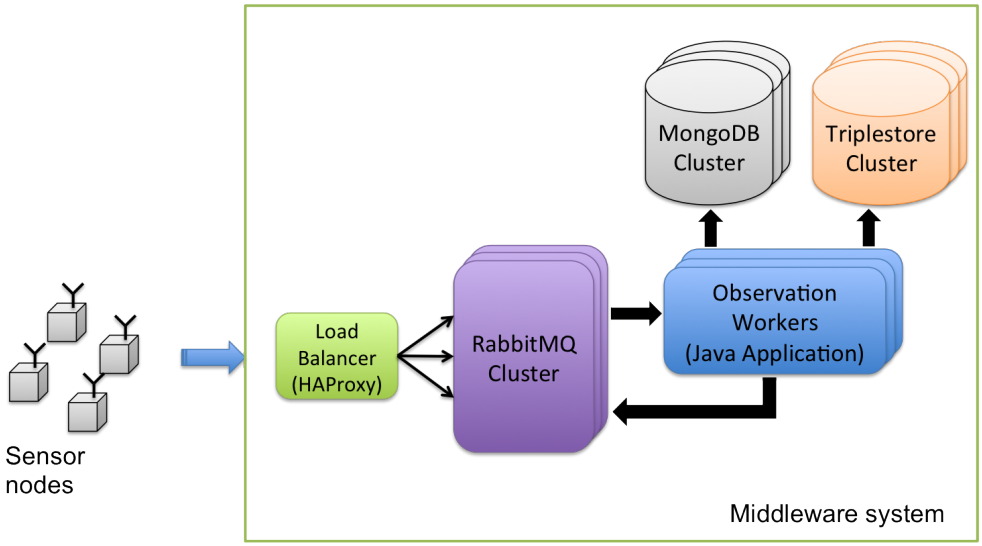


Figure 3.8: Simplified Architectural Model of Intelligent River® Middleware

Figure 3.8 shows a simplified version of the architecture of Intelligent River® Middleware system, where, sensor nodes are custom designed hardware devices known as Motestacks [85], Load balancer is a physical server running an open source TCP load balancing application, HAProxy [81], RabbitMQ cluster is a set of physical servers each running an open source message broker, RabbitMQ [59], Observation workers are a set of physical servers each running the custom developed Java applications, MongoDB cluster is a set of physical servers configured to run MongoDB [60] in a sharded configuration [61], Triplestore cluster is a set of physical servers each running Apache Jena TDB [26].

Sensor nodes or motestacks, gather data from commodity sensor probes and transmit the collected observations to the load balancer through wireless networks. The load balancer then relays the observations to any of the servers in the RabbitMQ cluster, which queues the observations to be fetched by the Observation worker applications. Observation worker applications fetch the queued observations from the message queues in RabbitMQ cluster, and performs (i) semantic analysis on the observations, (ii) persist both processed and unprocessed observations onto MongoDB, (iii) persist the processed observations onto Triplestore, and (iv) publish the processed observation back

to RabbitMQ for further analysis and visualizations. 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, whose components are loosely coupled to each other, 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 could process, additional instances of observation worker applications can be commissioned on new servers to fetch and process observations off the queues, improving the overall system throughput. Similarly, the MongoDB cluster is setup on a sharded configuration, so that new servers each running MongoDB can be added as shards to the existing cluster based on demand. The triple store cluster is also designed in a way 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 makes an ideal candidate to leverage the benefits of running on a virtualized infrastructure.

Chapter 4

Conclusion

The pursuit of making the server infrastructure more dynamic, efficient and cost-effective has led the evolution of virtualization 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 goal of virtualization. There is a clear need to analyze and understand the focus of each of these platforms and evaluate them based on factors beyond a set of standard benchmarks. We chose KVM, Xen, and Linux containers to represent full-virtualization, para-virtualization, and container-based virtualization respectively. We evaluated them based on the overhead they impose to virtualize CPU,memory, network access, and disk access, their performance on workload specific benchmarks, their performance in virtualizing the components that the Intelligent River® Middleware system is built upon, the flexibility each of them offer in performing common operational tasks, and the efficacy of isolation they provide for the applications.

We found that, Linux Containers exhibit the least overhead in virtualizing CPU, and disk access, whereas KVM exhibited the least overhead with respect to memory. Xen was found to exhibit a significant overhead when used to virtualize multi-threaded aplications. KVM in its current form, exhibited a significant overhead in virtualizing disk access, however, it is noted that the problem is being addressed with major design changes slated for release in the immediate future. From an operational standpoint, Linux Containers gained a significant advantage by performing the common tasks of provisioning, booting, and rebooting a virtual machine, several times faster than both KVM and Xen. Even though Linux Containers seem to have fared well on most of our criteria mentioned above, they come with a downside of providing poor isolation between the containers. Xen was

found to offer superior isolation between the virtual machines than KVM and Linux Containers.

As stated earlier in this thesis, 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 high degree of resource isolation. It is important to note that Linux Containers cannot be used in situations where, the applications need kernel-level customization.For infrastructure that demand 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 lot of disk accesses.

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[28] 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.

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

[30] 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).

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

[32] 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).

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

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

[35] 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).

[36] 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).

[37] 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).

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

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

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

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

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

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

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

[45] 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).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[84] 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.

[85] 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.

[86] 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.

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

[88] 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.

[89] 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.

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