Performance Optimization for Latency-Sensitive Workloads in Virtual Systems

by

Matthew Welch

A Thesis Presented in Partial Fulfillment
of the Requirement for the Degree
Master of Science

Approved November 2015 by the
Graduate Supervisory Committee:

Violet Syrotiuk, Chair
Gil Speyer
Carole-Jean Wu

ARIZONA STATE UNIVERSITY

December 2015

ABSTRACT

Network workloads have been increasingly migrating to virtual environments. Most of the time, there is *NO TUNING* utilized in these systems.

This is a bad idea and I'm gonna tell you why.

dedication goes here

ACKNOWLEDGEMENTS

Insert acknwledgements here.

Be nice to those that have helped you along the way.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Chapter 1

INTRODUCTION

Introduction goes here. Make it good.

Chapter 2

RELATED WORK

Virtualization has been an important part of computing as soon as more than one user wanted to share the expensive mainframe computer. This section will describe some of the work in the field of virtualization that has brought us to the current state of the art. We will also discuss some of the subsystems in Linux and processor hardware that enable virtualization.

2.1 History of Virtualization

Virtualization has been an important component in computing since the invention of virtual memory to support multiprogramming on mainframes in the 60's. Without going into too much detail:

1. Virtualization in early mainframes (IBM 360)
2. Early commercial virtualization
3. Entrance of Virtualization in x86
4. Virtualization as growth driver for datacenters and enterprise networks
5. Containers
6. Desktop virtualization
7. NFV

"The Linux Foundation Announces Project to Advance Real-Time Linux" [1]

1. the Linux Foundation is the body that guides and sponsors the direction of Linux.

2. Realtime Linux has been struggling to find funding until recently [lwn.net reference here]

3. This project is important because realtime Linux will also improve virtualization determinism.

4. this will ensure that the realtime branch of Linux is supported going forward.

2.2 Virtual Machine Architecture

As we've seen, Virtualization has been around since the early days of computing in many forms. Virtual machines and containers, specifically Docker, are currently very popular for numerous use cases. In the following sections, we will discuss the architecture of these types of virtualization to illustrate the resource needs and potential improvements for each.

2.2.1 Virtual Machines

The architectures available in virtual environments are as widely varied as the types of hardware they seek to emulate. Many of the concepts we use regarding virtual machines originated in a seminal 1974 paper by Popek and Goldberg [2]. Although virtual machines had already been implemented on "third-generation computer systems" such as the IBM 360/67, Popek and Goldberg sought to establish formal requirements for and prove whether other systems of that era were capable of supporting a "virtual machine monitor" [2]. At the time of that writing, their analysis focused on the possibility of a Virtual Machine Monitor (VMM), but the term hypervisor has largely come to replace VMM as the name of a software system that allocates, monitors, and

controls virtual machine resources as an intermediary between the hardware and the virtual machine's operating system. What follows in this section is an illustration of some important concepts in hypervisor and virtual machine architecture and the work that led to them.

2.2.2 Hypervisors

Hypervisors come in a few flavors, depending on the where the hypervisor is running in relation to the hardware and the guest operating system. In a Type 1 hypervisor, the hypervisor is running directly on the CPU or "bare metal". That is, the hypervisor code is not being hosted, translated, or controlled by another system or piece of software, but running as a native process on the CPU. Type 2 hypervisors, however, require a host operating system to provide some system services including memory and filesystem management.

The ideal difference between these two types of hypervisors is that the Type 1 hypervisor does not require an operating system to run, whereas the Type 2 does. The truth is that, although Type 1 hypervisors do not require an operating system to *run*, they *do* require an operating system for *control* [3]. It would appear that Type 1 systems seem to have an efficiency advantage over Type 2 since they tend to use microkernels instead of the macrokernels that are usually required to host Type 2 hypervisors. According to [3], however, the difference between them has little to do with performance, robustness, or other qualitative factors, but, rather, relates back to observations about their differences made in [2] and the analysis of these differences by [4]. Robin's analysis related to the potential for the Pentium processor to support a secure HVM [4], so it was important for them to draw conclusions about the capablities and suitability of various hypervisors. The vendors of hypervisor solutions also have an interest in perpetuating the distinction between the two types of hypervisors, but

the need to differentiate between these products has become less important over time, so we will not go into any more detail herein.

Although it's arguable whether there is a performance difference between Type 1 and Type 2 hypervisors, the market for virtualization is full of both. VMware, one of the more popular enterprise virtualization vendors, has products covering both architectures including ESXi, their Type 1 enterprise hypervisor product that is responsible for running large-scale virtualized systems all around the world[5]. They also provide Type 2 hypervisors such as VMware Workstation Player that runs on systems from desktops to small-scale servers that don't need the higher levels of orchestration and performance offered by ESXi.

In addition to numerous other available hypervisors, the Kernel Virtual Machine (KVM) is the de facto hypervisor in Linux that has been included as a module in the Linux kernel since February 2007 when it was included with Linux kernel version 2.6.20 [6]. The KVM module allows the host operating system to provide the low-level hardware access to the guest that has been enabled by hardware virtualization extensions such as Intel's VT-x [7].

Along with KVM in Linux, there is a user-space component known as the Quick EMUlator or QEMU[8]. QEMU was designed to be an architecture-agnostic emulator and virtualization helper, capable of running software written for one architecture on other architectures [8]. QEMU can acheive relatively good performance using binary translation, but, when paired with KVM or the Xen hypervisor, it can acheive near-native performance by executing guest instructions directly on the host processor [8].

Combined, the KVM/QEMU hypervisor runs natively on Linux with kernel modules and userspace tools, but additional layers such as libvirt [9] can be utilized to simplify VM creation, monitoring, and orchestration. In this work, we have chosen

to eschew any higher-level libraries such as libvirt or VMware orchestration in favor of running as efficiently as possibly without additional software layers of abstraction slowing things down. The hope is for maximal performance so minimal abstraction layers were utilized in the system configuration.

2.2.3 CPU Virtualization

In addition to the variations in hypervisors that may or may not have an impact on performance, the type of virtualizion used to provide devices to the guest operating system can have a significant impact on performance. In the following sections, we will review some important types of hardware emulation comprising virtio, paravirtualization, and hardware assisted virtualization.

One of the early challenges in the virtualization of the x86 architecture was handling privileged instructions. The x86 processor was originally designed with 4 "rings", numbered from 0 to 3, representing decreasing privilege levels as the rings increase [x86 ring architecture reference here]. Operating systems utilizing the x86 instruction set execute user code in ring 3 and privileged instructions (kernel code) in ring 0. Virtual machines on x86 execute their instructions as a user-space process in ring 3 so the host OS can maintain control of the system. The machanism by which the processor executes privileged instructions on behalf of the guest has been the topic of considerable effort in the advancement of virtualization which has spawned multiple techniques for handling these instructions.

One of the earliest methods, known as binary translation, involved trapping privileged guest instructions in the hypervisor and translating them into "sequences of instructions that have the intended effect on the virtual hardware" [10]. The binary translation technique, developed by VMware, was one of the most efficient methods in early hypervisors, but is now considered to have high overhead due to the additional

time required to perform the code translation. This same translation process, however, allows a hypervisor using this technique to run guest operating systems for virtually any processor on any other instruction set, provided that an efficient translation can be achieved. This flexibility makes binary translation one of the most versatile, if not performant, methods of virtualization avaialble which makes it well suited for virtualization of very old instruction sets or hardware.

Paravirtualization is another technique for handling guest privileged instructions. It involves cooperation between the guest and host operating systems to improve efficiency. The guest OS must be modified to replace privileged instructions "with hypercalls that communicate directly with the virtualization layer hypervisor." [10]. VMware has incorporated this method in their vmxnet series of network drivers to accelerate network workloads. A more widely known example of paravirtualization is one of the first open-source hypervisors, known as the Xen hypervisor. At the risk of oversimplification, the Xen project is, essentially, a modified Linux host that communicates directly with the guest kernel. The guest kernels and modules must also be modified to facilitate this communication which places an additional burden on hardware vendors to provide not only open source drivers but also paravirtualized open source drivers for their hardware. Although the Xen hypervisor was originally developed with the intent of being hardware agnostic [11], the modifications required to the guest operating system mean that only vendors wishing to participate in the open-source community will provide paravirtualized drivers. The community itself is free to develop these drivers, but this adds a barrier to adoption for most new hardware products. Despite the additional effort required, Xen maintained their lead as a very popular open-source hypervisor for many years. Paravirtulaizaion has its fans, however, and much work has been done previously comparing Xen to containers and other hypervisors such as KVM [12]–[17].

The third popular virtualization method we will discuss is what appears to be the de-facto standard as virtualization matures. Hardware Assisted virtualization, which is heavily promoted by CPU vendors such as Intel and AMD, started out as an effort to accelerate instruction translation, but early generations of hardware had difficulty keeping up with the binary translation preferred by VMware [10]. The silicon vendors, however have been hard at work improving their VT performance and adding hooks to enable virtualization of their hardware. Since operating systems for x86 were already common when hardware assisted virtualization was introduced, the architecture needed subtle modifications to help enable virtualization without breaking existing software. This was accomplished with the introduction of yet another protection ring that would run below the kernel's ring 0. The hypervisor would now run in ring -1, below the operating system kernel, and trap privileged instructions when they were executed by the guest. [10]. For each virtual machine, a new structure is defined is maintained in the host kernel memory. Logically similar to a process control block, these structures, Known as a Virtual Machine Control Structure (VMCS-IA) or VM Control Block, (VMCB-AMD), maintain the state of the virtual machine and are updated when the guest needs to perform a "vm-exit" to allow the host to execute privileged instructions.[10] This vm-exit process has been a significant source of latency for guest privileged instructions, but advancements in virtualization-aware driver models like SR-IOV [reference here??] have improved both latency and the frequency of these exits.

Although a deep discussion could be had regarding the hardware systems and mechanisms of processor and platform virtualization, that discussion is slightly outside the scope of this document. Suffice to say, however, that the CPU vendors have significant interest in producing higher core-count CPUs and supporting virtualization. We limit the discussion of these processors even further to x86 architecture, but it

should be noted that ARM and other vendors are actively working to enable virtualization with similar methods. Additionally, important components of virtualization such as SR-IOV are managed by the PCI-SIG and are not the exclusive domain of x86. To that end, both Intel and AMD have independently developed processor extensions to enable virtualization [7].

2.2.4 Memory Virtualization

Virtualization of the CPU was the first challenge in the development of secure virtualization, but, in Von Neumann processor architecture, the memory unit is equally important. Memory has become an increasingly significant source of latency in processors as the CPU core has improved its performance faster than the improvement of memory so improvements in this area are doubly beneficial for virtual machines. Virtualization uses memory structures similar to those developed for virtual memory in early computing systems. Virtual memory was created to allow multiprogramming and process address spaces that are larger than the available physical memory. Modern CPUs implement virtual memory with the aid of a memory management unit (MMU) and translation lookaside buffer (TLB) to manage page tables and accelerate page lookups. The host system must carefully control access to memory which holds the system state along with program data. Fairly recent developments in x86 processors have allowed hypervisors to maintain guest memory mappings with shadow page tables, known as Extended Page Tables (EPT) in Intel processors and Nested Page Tables (NPT) for AMD [reference??]. A hypervisor may use TLB hardware to map guest memory pages onto the physical memory similarly to a native process, reducing the overhead of guest OS memory access. Along with virtualization of guest memory, another significant improvement to virtual machine performance can be had with the utilization of hugepage memory [18]. Hugepages, referred to as superpages in

Romer, can simply be described as a memory page that is a power-of-two multiple of a standard 4096 byte (4k) memory page. Romer et. al. demonstrated a significant improvement in performance when using hugepages for memory-hungry applications [18]. When system memory sizes were very small, virtual memory and swapping smaller pages was found to be more efficient than larger pages. As memory sizes have grown to hundreds of gigabytes per server and applications can consume multiple gigabytes each, 4k memory pages no longer seem like such an obvious fit. Hugepages seek to reduce the frequency of TLB lookups and page table walks by simply using larger pages of memory. Romer showed that they could reduce TLB overhead by as much as 99% using *superpage promotion* which is a system of aggregating smaller pages together as they are used [18]. Current implementations of hugepages in Linux, however, offer a set of large memory pages available to the kernel for memory allocation [reference here]. Unlike Romer's superpages, hugepages must be allocated at boot, rather than being coalesced dynamically, but the performance of the Linux hugepages are similar except for increased memory consumption due to unused portions of hugepages. For virtual machines that are potentially using multiple levels of memory page walk, thereby increasing the latency of those operations, any reduction in the frequency of lookups will be beneficial. Hugepages are not standard practice, however, so we have chosen to include hugepages as an important mechanism for improving the memory performance of our virtual machines. One popular use of hugepages in accelerating network workloads is through the Data Plane Development Kit, first introduced by Intel and now an open source project [19]. NOTE: I would really like to use DPDK, but don't think I have time so maybe I should just drop it here :(

2.2.5 I/O Virtualization

After achieving performant virtualization solutions for compute and memory, the last component in achieving full system virtualization was the virtualization of devices and I/O (IOV) [history reference?]. Along with compute and storage, processing of I/O is one of the most critical components of a server's workload due simply to the fact that I/O can be a large source of latency for remote transactions. Efficiently utilizing I/O allows virtual machines to perform nearly any task possible in a native system, particularly the processing of network packets and workloads. Similar to the "virtualization penalty" that occurs with nested page table lookups and binary translation, latency inherent in the processing of network packets with multiple levels of handoffs should be avoided when possible. Virtualization is essentially software emulation of hardware devices so the natural first attempt at device virtualization is to emulate a device in the kernel, providing a software device to the guest OS that is based on a common physical device. This method is common in some workstation virtualization products such as VMware workstation [20]. Instead of emulating a software device in the kernel, it is also possible to emulate the device in user-space. This is the method used by QEMU which provides both device emulation and a platform-agnostic hypervisor. In Linux operating systems, QEMU is often combined with the KVM modules. In this configuration, QEMU provids user-space device emulation for simple devices such as mouse and keyboard and KVM provides virtualization of the physical hardware. Userspace emulation has the advantage of removing the responsibility from the kernel, thereby minimizing the potential attack surface of the kernel. As mentioned earlier, paravirtualized devices are another variation on this theme of emulation where the paravirtualized guest driver communicates with the host paravirtualized devices. In addition to Xen's paravirtualized driver model, the

KVM virtio library is the basis for paravirtualized devices in Linux [virtio1]. The virtio library takes inspiration from paravirtualization and userspace emulation and uses qemu to implement the device emulation in userspace so host drivers are not necessary [virtio1].

While device emulation can provide important flexibility and hardware independence, it brings up the recurring theme of software managing hardware functions which is often less efficient than utilizing dedicated, specialized hardware to perform the function. The alternative is to avoid any device emulation and allow the guest OS to access hardware directly as if it belonged to the guest rather than the host system. Since it is conceptually similar to virtualization of the MMU for memory, virtualizing the DMA transactions of modern I/O devices requires an I/O MMU (IOMMU) to allow communication between the guest and I/O devices. In x86 architecture, this feature is known as AMD-Vi or Intel VT-d, but both utilize the same concept of an IOMMU to avoid vm-exits when processing I/O. This allows the hypervisor to unbind a hardware device from its kernel and "pass through" or "direct assign" the device to the guest OS [20]. Direct assignment of hardware to guests also comes with the high cost of dedicating network interfaces or other important devices to a guest, but these devices provide significant performance improvements over paravirtualized or emulated guest devices, thereby enabling applications that could not previously be virtualized due to low-latency constraints [20]. If a host system has a large number of virtual machines that need high-performing I/O, it can be difficult to fit enough peripheral cards into the host chassis to passthrough devices to the virtual machines, thus providing each VM with dedicated hardware. A solution to this apparent conflict of interests can be found in "PCI-SIG Single Root I/O Virtualization" (SR-IOV) [**˙pcisig˙1**]. SR-IOV is a general method by which the host system can configure a single hardware device to create and control multiple additional *virtual functions* of

the device. These virtual functions can be directly assigned to and accessed by the guest, similar to passthrough, without removing the main physical function from the host. The host's physical function represents the original hardware and is responsible for the management functions of the peripherial. An illustrative example is the Intel 82599 10 gigabit network card, used later in this study. Each physical network interface (physical function or pf) in these network cards has 64 Rx/Tx queue pairs that are normally used as send and receive buffers to maintain multiple simultaneous flows. The individual queue pairs may be "broken out" of the main pool to create a virtual function that is, essentially, a new network device sharing the same physical interface as the physical function. These new devices are assigned unique MAC addresses and IP addresses to differentiate them on the network so that an outside observer cannot tell them apart from a physical device. The advantage with SR-IOV is that one interface can be multiplexed into multiple independent interfaces that can each reach near-native performance levels [**˙nasa˙1**]. As we will see later, this method has great potential in systems supporting large numbers of virtual machines or containers with only limited host physical resources.

2.2.6 Hypervisor Options

2.3 Container Architecture

At a high level, containers can be thought of as another level of access control beyond the traditional user and group permission systems. While those systems provide resource access control, containers can provide these resources with finer granularity, thus allowing only those resources and privileges that are required by the process [12]. In a sense, containers are finally applying bounds to program execution that should

logically have been included in process control from the very start so it is encouraging to see increased interest in their performance and security. [opinion, conclusions??]

2.3.1 Operating System Virtualiztion

Also known as *operating system virtualization*, containers are a construct of the operating system that serves to provide limited context, and resources to a process executing on the host system. The subsystems that enable a containerized process, including chroot, cgroups, and namespaces, have been gradually added to the Linux kernel over time.

The current state of Linux containers has been stable since the introduction of [TODO chroot, then cgroups, then namespaces] [reference]. Much like the limitations imposed on apps in mobile devices [reference needed], all processes on a server or desktop system should have a view of the system that is limited in context and scope - they should only have access to the resources that they require (maybe with some margin for error). One of the most important duties of the kernel is to control access to the system's resources including CPU, memory, and I/O. In a sense, containers represent a desired level of control by the kernel of any arbitrary process. Without containers or a similar resource limitation, the kernel is not fulfilling its whole purpose [TODO: elaborate here].

The earliest popular concept in containers was the idea of limiting the filesystem scope of an executing process, which is embodied in the chroot system call and user-binary. The chroot call has its origins in BSD Jails and Solaris Zones [ref?]. Next on the container scene was the addition of cgroups to the Linux kernel. Contributed by Google [reference], cgroups serves to limit the resources available to a process in a hierarchy, enabling child processes to inherit their parent's cgroup assignment. Namespaces, a recent addition to the kernel [ref?], are the mechanism by which the

scope or view of the process may be limited and controlled. Linux capabilities are the mechanism by which the monolithic permissions granted to the root user for full system administration are broken down into more granular permissions.

Although it may be a controversial statement, it is the author's opinion that it is somewhat incorrect to think of containers as *virtualization* per se. Without arguing the definition of virtualization, a container does little to actually virtualize anything, but is more of a system for isolating and limiting the resource consumption of processes while providing dependencies necessary for them to run. The only actual virtualization that is happening in a container is the masking of resources and, perhaps, virtualization of network or other I/O resources to enable controlled sharing with other containers and processes in the system.

2.3.2 Container Systems

While Docker has recently popularized this form of sandboxing, related concepts have been used in earlier operating systems. FreeBSD introduced the Jails concept in 1999 as a process isolation technique [Zones1]. The Solaris-variant of the Unix operating system added the concept of Zones (Solaris Containers) in 2004 which was supposed to be an upgrade of the BSD Jails concept [Zones1]. Other containers have come before (Linux VServer, OpenVZ, FreeBSD Jails, Solaris Zones) and even LXC more recently.

"LXC is nothing else than OpenVZ redesigned to be able to be merged into the mainline kernel. Therefore, OpenVZ will eventually sunset, to be fully replaced by LXC." [http://blog.docker.com/2013/08/containers-docker-how-secure-are-they/ Comparison with Other Containerization Systems]

Docker even used to be based completely on LXC, calling lxc-* operations in the background.

The Linux container infrastructure has evolved some as have the many other containerization systems to the point that there was need to unify the standards on a single container format, (name??).

The Linux subsystems have become sophisitcated enough that Docker or LXC are no longer necessary to create containers, since administrators can create their containers with native Linux tools such as SystemD or just scripts that stitch together the correct cgroups, chroot, and namespace allocation. The de-facto standard, however seems to be Docker, based on the amount of media coverage and hype surrounding the young startup. Docker is popular for good reason: in addition to providing a robust tool set for isolating processes, the Docker team has been extremely open about their development and responsive to issues submitted by users on their github account: [https://github.com/docker/docker/]. It is for the reasons of it's ubiquity and ease of use that Docker is the container format selected for comparison herein.

https://docs.docker.com/introduction/understanding-docker/ (architecture, good pictures too)

We will consider Docker within this work as representative of container systems. Docker is largely responsible for the current wave of "hype/excitement/momentum" around containers so seems to be the de facto container in the mind of most people. Recently, several industrial players interested in containers banded together to standardize on the container format and container runtime [list members, reference for unification]. Docker has stated that the standardized format (libc?) and runtime will be adopted which will aid in proliferation and sharing of containers.

Characteristics of Docker/containers

Docker is a "Lightweight container virtualization platform with workflows and tooling that help your manage and deploy your applications."

2.3.3 Resource Scope

Linux chroot: Introduced in Version 7 Unix in 1979, one of the earliest Linux systems to enable some kind of Operating System virtualization is the chroot system call. It was later expanded to become FreeBSD Jails and, later, Solaris Zones. This tool allows a user to change the root directory of a process and its children to a specified directory such that the process subsequently views that directory as its root directory, "/" . This is useful for providing partial filesystem isolation to processes thus controlling libraries and binaries that are available to the process. Although this should work well for a normal process, it will not prevent a process from accessing arbitrary inodes or files if it tries to so it can be fairly easily circumvented. It simply makes this process more difficult, thereby "hardening" the target, but was not intended to fully sandbox a process or restrict its filesystem system calls [chroot1]. The Linux capability required for chroot is CAP_SYS_ROOT.

Some of the techniques common in virtualization such as protecting certain instructions, have been extended into containers as the protection of system capabilities such as cap_sys_nice and cap_sys_chroot. Essentially, if a container needs this capability, it must be explicitly assigned and cannot be "trapped" in the host kernel as is common in virtualization.

Linux namespaces: Namespaces and cgroups have been included only for a short time since version 3.8 of the Linux kernel.

2.3.4 Resource Control

Resource Control (cpu/mem scope too): Linux cgroups

"Control Groups provide a mechanism for aggregating/partitioning sets of tasks, and all their future children, into hierarchical groups with specialized behaviour." [https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt]

Cgroups are organized into hierarchies with the root being the default cgroup for all processes. Child processes can be organized into sub-trees that subdivide the resources of its parent process. Each process must belong to one and only one cgroup.

Linux capabilities

Subdivision of the capabilities of the root user

Implications & differences between VMs and containers

since containers need not emulate a second kernel or set of virtual hardware

Network

Maximum theoretical speed of Ethernet (copper or fiber) is only about 95% of the line rate. This is due only to the overhead of Ethernet frames (1518 bytes + 8 byte preamble and 12 byte (96 ns) interframe gap (IFG)), Ethernet header (14 bytes), 4 byte Frame Check Sequence (FCS), 20 byte IP header, 20 byte TCP header, 1460 MSS for TCP. Overall, this amounts to 78 bytes of overhead at various layers to transmit only 1460 bytes of data or 94.9% efficient in the best case.

"https://docs.docker.com/articles/networking/#How-Docker-networks-a-container"

Chapter 3

EXPERIMENTAL DESIGN

As we've seen in the previous sections, the performance comparisons that have been made between containers and virtual machines have, for the most part, not tuned their systems for the comparison. Indeed, due to the numerous potential parameters and the resulting massive number of configuration combinations, the sheer number of comparisons to be made among the effects of various tuning parameters are myriad. To avoid this potential explosion in the scope of the comparison, we will compare performance between only two system configurations for each type of virtual environment. The two configurations include the standard Linux kernel, version 3.18.20, and the same kernel with preempt-rt patches applied [preempt-rt patch reference here] and some important tuning parameters for the kernel and environment.

Potential section headings include: Design, Architecture, Preempt-RT, Kernel Tuning, Benchmark Analysis, Hardware Utilized, Experimental Setup and Hardware.

3.1 Experimental Setup

3.1.1 Hardware

SuperMicro board, DualCPU, memory, network card

Expand on hardware details and BIOS settings including hyperthreading (+reasoning)

The PCIe network interface cards used were Intel 82599ES with two 10 Gbps optical interface interfaces per card. 10 Gigabit network interfaces used because almost any virtualization system, be it container or virtual machine, can push 1 Gbps of traffic per

core. With a 10 Gbps interface, however, the work required to push that bandwidth through the interface means that both the virtual machines and containers should see some performance differences

3.1.2 Networking and Topology

There are many varieties of network interfaces avaialable in virtual environments. These include both hardware and software options with varying degrees of versatility and performance. Most common among software options, virtual bridges are operated by the host kernel with a non-trivial CPU involvement. Software bridges are also commonly used by virtual machines and often combined with the virtio library [**˙virtio˙1**]. Bridges are the *de facto* layer 2 networking used by Docker. The Docker daemon also uses the common Linux firewall IPTables for layer 3 connectivity, routing, and NAT. This implies that bridging or any networking paradigm that requires host kernel involvement cannot scale to a large number of guests because the work required by the host to monitor and handle bridge traffic and layer 3 decisions for multiple guests can be significant.

3.1.2.1 Software Networking

Dockers default networking paradigm is to create a bridge and pair of virtual Ethernet interfaces. Both interfaces are bound to the bridge, one remains in the hosts namespace and the other is placed in the container namespace as its primary network interface. Docker also uses IPTables and Network Address Translation to allow the container to communicate with the outside world. IPTables is also going to have an impact on the bridge performance, but we chose to use it since it represents the *de facto* networking configuration of Docker. Another reason to study the bridged networking paradigm is that bridges are very important in networking in general

and useful in systems where containers may be linked together or their functions pipelined. Additionally, the popular software switch package, Open vSwitch (OVS) may be configured to use virtual bridging as its default network topology.

[insert bridged topology diagram here]

3.1.2.2 Hardware networking

In order to obtain near-native performance for both virtual machines and containers, a physical interface may be passed from the host into the domain of the virtual machine or into the namespace of the container. As we have seen in 2.2.5, this is known as passthrough for virtual machines and is supported by CPU virtualization extensions [7].. In the version of Docker used herein, 1.8.1, direct physical assignment is not enabled by default. Instead, a fairly simple script was written [prepare_network_stack.sh], with inspiration borrowed from Jermoe Petazzoni [reference], and this stack overflow pot [reference] to assign a physical interface to the container namespace.

Another network interface that may be provided to a virtual environment is that of a virtual function. Weve seen these described in 2.2.5 as a multiplexed interface into the actual physical connection of the device. These interfaces run at line rates, but have fewer receive and transmit queues than the original interface. This implies that they can achieve native performance for small workloads that do not saturate their buffers. These interfaces have significantly improved performance over the bridged interfaces since the host kernel need not be involved in processing these packets which happens in the card itself instead. For the most part, virtual functions may be handled by the operating system or guest OS as if they were any other physical device and passthrough or direct assignment is used to attach them to VMs and containers, respectively.

3.1.3 Software Environment

The operating system used in this study needed to have a balance of stability, advanced features, and performance so a Long Term Support LTS version of Ubuntu, 14.04.3 was chosen. The Linux 3.18.20 kernel was selected for both the host and guest OS due to a set of factors affecting usability and performance. Choosing which Linux kernel version to use for this study was complicated by factors that were not known until system configuration. It was already understood prior to configuration that Docker was not supported in kernel versions prior to 3.10 [reference?] which helped to narrow down the possibilities. One of the side-goals of this study was to use as much standard and open software as possible so that the kernel and system configurations could be easily reproduced by other investigators. This goal had a substantial influence on the choice of kernel due to the performance impact of docker's storage driver. By default, docker currently uses the AUFS filesystem for its back-end storage driver. This module allows the classical union filesystem approach of masking important host files and providing private copies of system files to each container. Other options for this driver include devicemapper, zfs, btrfs, and overlayfs. The AUFS driver has been deprecated in the Linux kernel so it is not available upstream which disqualifies it from any forward-looking uses. The devicemapper driver, while ubiquitous, shows significant performance degradation and usability issues disqualifying it as well. Both zfs and btrfs have stability issues and are not yet standard filesystem drivers so they were also disqualified. The remaining choice was overlayfs which was recently upstreamed into the 3.18 kernel and seems to be the choice for docker storage driver going forward until zfs and btrfs become standard [reference here re: overlayfs RedHat study]. Further limitations, while not as impactful as the choice of the 3.18 kernel include the availability of preempt-rt patches for the kernel.

3.1.3.1 Kernel Configuration

One of the steps involved in tuning the RT variant of our kernel was to apply a specific set of kernel configuration choices during the kernel build process. The kernel shipped with Ubuntu 14.04.3 was version 3.19.0-25-generic. That kernels default configuration (.config) was used as the basis for the configuration and building of the test kernel version 3.18.20. This default kernel configuration omits even simple omtimizations such as disabling CPU frequency scaling, and enabling a pre-emptible kernel (CONFIG_PREEMPT), but the default kernel configuration is probably the most common on servers that haven't done significant performance tuning and optimizations so it seemed appropriate. The "performance" kernel used was the 3.18.20-rt18 kernel which is the 3.18.20 kernel with the 3.18.20-rt18 patches, dated 2015-08-11, from [https://www.kernel.org/pub/linux/kernel/projects/rt/3.18/older/] applied. The initial kernel configuration used was that of the 3.18.20 kernel. Only a few important parameters of the kernel were modified to tune the systems for performance with the preempt-rt patch. Kernel CONFIG modifications are described in Table XXYY. : [Insert CONFIG_ table here?]

Table:

CONFIG_PARAM1 — effect1

CONFIG_PARAM2 — effect2

3.1.3.2 Kernel Boot Parameters

Another important mechanism for tuning a kernel is to modify the parameters passed to the kernel at boot time. [Insert table here? Describe some of the important parameters]

One of the most important tuning parameters in a virtual machine host relates to the level of oversubscription for each physical CPU. Systems with low performance requirements may stack multiple guest vCPUs (virtual CPUs) on each physical CPU. This stacking can lead to multiple vCPUs from unrelated virtual machines sharing the pCPU or just host processes running on the pCPU assigned to the guest. Either of these scenarios can have a large impact on guest latency. If a pCPU is running another task when a guest vCPU is not in the current context, the additional latency required to change context to the guest and resumt the guest OS operation can be a significant source of latency. These resource conflicts can be all but eliminated by "unplugging" pCPUs assigned to a guest from the host scheduler with the isolcpus=X kernel boot parameters. Assigning a pCPU to the isolcpus list, like isolcpus=2, will remove that pCPU from the kernel scheduler's CPU pool, preventing the kernel scheduler from running any processes there. The isolated CPUs can still be used to run user processes with Linux utilities like taskset or numactl, but will remain quiescent until that happens. Scalability of a system with isolated CPUs is reduced, but in an era of server processors with up to 18 cores [Intel Xeon reference here, Xeons may be moving toward something like many-core Knights Landing], the impact of dedicating individual pCPUs to latency-sensitive processes is reduced.

Hugepages for reduction in TLB misses [Huge TLB paper reference] - Again, huge memory availability leads to not needing to share memory.

3.1.3.3 QEMU and hypervisor parameters

QEMU is responsible for device emulation so this is where it's important to set up the virtual machine to be as performant as possible.

3.1.3.4 Docker parameters

The Docker environment is logically separated into building with 'docker build', back-end bmanagement performed by the Docker daemon, and the 'docker run' command that actually instantiates containers from images and assigns the appropriate cgroups and namespacing to the containers. The Docker daemon essentially sets up the software environment for the containers, specifically their filesystems, libraries, and avaialble binaries. Along with the files available to the container, the daemon also manages cgroups and namespaces to properly control the container's resoruces and limit its scope. The docker run command is responsible for requesting resources and privileges for the container as it is instantiated.

The image building process of Docker is controlled by the 'docker build' command line tool. This tool takes recipe files known as Dockerfiles [**˙dockerfile˙1**] and constructs the image based on instructions therein. A Dockerfile usually specifies the base image that should be used for the build along with setting environment variables and installing necessary packages and binaries to the container's filesystem. It is consumed by the 'docker build' tool and the image is constructed as the composite overlay of each step in the build process [**˙docker)build˙ref**]. All of the containers used herein were based on the official Debian image, 'debian:wheezy', which is based on the latest release of Debian 7: Wheezy. The image was pulled from the Docker.io repository on September 1, 2015 with image ID bbe78c1a5a53. The debian image was chosen since it has a very small starting size of only 85 MB and no additional tools provided by larger base images were necessary. In retrospect, this decision creates a small discrepancy in the comparison of these containers to the virtual machines used herein since the VMs were all using Ubuntu 14.04.3. Since the userpace tools of the distro are not the subject of this investigation, it was decided that the base container image

of Debian would not cause any issues since the kernel function inside the containers was of interest and that versino is common among all of the environments used. The image created for benchmarking was called netbench and created with the Dockerfile.netbench [reference here??]. It only pulled the debian base image, applied some labels and installed netperf. Later that same image was used with iperf by mounting a host volume containing the iperf directory which was then copied into the contaienr's filesystem.

The daemon has a number of parameters that may be modified to tune parameters such as available cgroup cpu sets, bridged networking, and the storage driver used for container filesystems. In this study, the docker daemon was launched with mostly standard parameters but for the following exceptions. As we've discussed in **??**, we chose to use the overlayfs storage driver, requiring the docker daemon to be launched with –storage-driver=overlay. Additionally, Docker's default bridge, docker0, was configured to draw on a pool of IP addresses from the subnet 192.178.42.240/28 for it's bridged virtual ethernet interfaces.

The Docker run command provides a substantial level of flexibility and power to the user in controlling the resources available to containers. It can be used to give the container complete root privileges, essentially defeating the container's isolation, but it can also be used to give very specific capabilities with the There are so many options that the docker run command can perform that the discussino is outside the scope of this document. Our modifications herein are as follows. All containers were run with non-persistent filesystems by using the –rm option in the docker run line. This had the effect of removing the container's filesystem overlay after it exits, preventing any filesystem modifications between runs. Indeed, this is one of the advantages of conatiners in obtaining consistent performance since the container filesystem is reloaded from the base image at every launch. Containers were also run with the

flags -i -t or -it, which requested that an interactive tty be allocated to the container while it's running. This has the effect of a small additional overhead for the terminal process, but this is consistent with the default operation of Docker. Containers are all launched with a specified MAC address for their bridge interface to remove the need to refresh ARP tables with new, randomized MAc addresses.

3.2 Experimental Procedure

Start Here.

Chapter 4

RESULTS

Chapter 5

CONCLUSIONS & FUTURE WORK

Conclusions text here

5.1 Future Work

5.1.1 DPDK

DPDK is, like, super awesome and should be used by everyone!

5.1.2 Virtual Functions

Virtual functions would be really good for scaling

REFERENCES

[1] L. Foundation, *The linux foundation announces project to advance real-time linux*, http://www.linuxfoundation.org/news-media/announcements/2015/10/linux-foundation-announces-project-advance-real-time-linux, [Online; accessed 5-October-2015], 2015.

[2] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures", *Commun. ACM*, vol. 17, no. 7, pp. 412–421, Jul. 1974, ISSN: 0001-0782. DOI: 10.1145/361011.361073.

[3] A. Liguori, *The myth of type i and type ii hypervisors*, http://blog.codemonkey.ws/2007/10/myth-of-type-i-and-type-ii-hypervisors.html, [Online; accessed 22-September-2015], 2007.

[4] J. S. Robin and C. E. Irvine, "Analysis of the intel pentium's ability to support a secure virtual machine monitor", SSYM'00, pp. 10–10, 2000. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251306.1251316.

[5] VMWare, *Vmware products*, http://www.vmware.com/products/, [Online; accessed 22-September-2015], 2015.

[6] Linux, *Linux 2.6.20 change log*, "http://kernelnewbies.org/Linux_2_6_20", [Online; accessed 22-September-2015], 5 February, 2007.

[7] S. Grinberg and S. Weiss, "Architectural virtualization extensions: A systems perspective", *Computer Science Review*, vol. 6, no. 5-6, pp. 209–224, 2012, ISSN: 1574-0137. DOI: 10.1016/j.cosrev.2012.09.002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1574013712000342.

[8] qemu, *Qemu.org wiki, main page*, http://wiki.qemu.org/Main_Page, [Online; accessed 22-September-2015], 2015.

[9] libvirt, *Libvirt.org main page*, http://libvirt.org, [Online; accessed 22-September-2015], 2015.

[10] VMWare, *Understanding full virtualization, paravirtualization, and hardware assist*, http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf, [Online; accessed 22-September-2015]; Revision 20070911, Item: WP-028-PRD-01-01, 2007.

[11] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization", in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03, Bolton Landing, NY, USA: ACM, 2003, pp. 164–177, ISBN: 1-58113-757-5. DOI: 10.1145/945445.945462.

[12] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers", in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, Mar. 2015, pp. 171–172. DOI: 10.1109/ISPASS.2015.7095802.

[13] A. Younge, R. Henschel, J. Brown, G. von Laszewski, J. Qiu, and G. Fox, "Analysis of virtualization technologies for high performance computing environments", in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, Jul. 2011, pp. 9–16. DOI: 10.1109/CLOUD.2011.29.

[14] G. Wang and T. E. Ng, "The impact of virtualization on network performance of amazon ec2 data center", In proceedings of INFOCOMM10, 2010.

[15] J. Che, Y. Yu, C. Shi, and W. Lin, "A synthetical performance evaluation of openvz, xen and kvm", in *Services Computing Conference (APSCC), 2010 IEEE Asia-Pacific*, Dec. 2010, pp. 587–594. DOI: 10.1109/APSCC.2010.83.

[16] M. J. Scheepers, "Virtualization and containerization of application infrastructure: A comparison", in *21st Twente Student Conference on IT*, Jun. 2014, pp. 1–7. [Online]. Available: http://referaat.cs.utwente.nl/conference/21/paper/7449/virtualization-and-containerization-of-application-infrastructure-a-comparison.pdf.

[17] Z. Wang, X. Zhu, P. Padala, and S. Singhal, "Capacity and performance overhead in dynamic resource allocation to virtual containers", in *Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium on*, May 2007, pp. 149–158. DOI: 10.1109/INM.2007.374779.

[18] T. H. Romer, W. H. Ohlrich, A. R. Karlin, and B. N. Bershad, "Reducing tlb and memory overhead using online superpage promotion", *SIGARCH Comput. Archit. News*, vol. 23, no. 2, pp. 176–187, May 1995, ISSN: 0163-5964. DOI: 10.1145/225830.224419. [Online]. Available: http://doi.acm.org/10.1145/225830.224419.

[19] *Dpdk data plane development kit*, http://dpdk.org, Website accessed 22-September-2015, Sep. 2015.

[20] M. T. Jones, *Linux virtualization and pci passthrough device emulation and hardware i/o virtualization*, http://www.ibm.com/developerworks/linux/library/l-pci-passthrough/l-pci-passthrough-pdf.pdf, Accessed 22-September-2015, Oct. 2009.

[21] K. Adams, "A comparison of software and hardware techniques for x86 virtualization", in *In ASPLOS-XII: Proceedings of the 12th international conference on Architectural*, ACM Press, 2006, pp. 2–13.

[22] N. M. K. Chowdhury and R. Boutaba, "A survey of network virtualization", *Comput. Netw.*, vol. 54, no. 5, pp. 862–876, Apr. 2010, ISSN: 1389-1286. DOI: 10.1016/j.comnet.2009.10.017.

[23] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A centralized "zero-queue" datacenter network", *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 307–318, Aug. 2014, ISSN: 0146-4833. DOI: 10.1145/2740070.2626309.