Containers and Virtual Machines at Scale: A Comparative Study

Prateek Sharma[1], Lucas Chaufournier[1], Prashant Shenoy[1], Y.C. Tay[2]
{prateeks, lucasch, shenoy}@cs.umass.edu, dcstayyc@nus.edu.sg
University of Massachusetts Amherst, USA[1], National University of Singapore, Singapore[2]

ABSTRACT

Virtualization is used in data center and cloud environments to decouple applications from the hardware they run on. Hardware virtualization and operating system level virtualization are two prominent technologies that enable this. Containers, which use OS virtualization, have recently surged in interest and deployment. In this paper, we study the differences between the two virtualization technologies. We compare containers and virtual machines in large data center environments along the dimensions of performance, manageability and software development.

We evaluate the performance differences caused by the different virtualization technologies in data center environments where multiple applications are running on the same servers (multi-tenancy). Our results show that co-located applications can cause performance interference, and the degree of interference is higher in the case of containers for certain types of workloads. We also evaluate differences in the management frameworks which control deployment and orchestration of containers and VMs. We show how the different capabilities exposed by the two virtualization technologies can affect the management and development of applications. Lastly, we evaluate novel approaches which combine hardware and OS virtualization.

CCS Concepts

•General and reference → Empirical studies; *Evaluation;* **•Computer systems organization → Cloud computing; •Software and its engineering →** *Operating systems;* Cloud computing;

1. INTRODUCTION

Modern enterprises increasingly rely on IT applications for their business needs. Today's enterprise IT applications are hosted in data centers—servers and storage that provide compute, storage and network resources to these applications. Modern data centers are increasingly virtualized where by applications are hosted on one or more virtual machines that are then mapped onto physical servers in the data center.

Virtualization provides a number of benefits. It enables a flexible allocation of physical resources to virtualized applications where the mapping of virtual to physical resources as well as the amount of resources to each application can be varied dynamically to adjust to changing application workloads. Furthermore, virtualization enables multi-tenancy, which allows multiple instances of virtualized applications ("tenants") to share a physical server. Multi-tenancy allows data centers to consolidate and pack applications into a smaller set of servers and reduce operating costs. Virtualization also simplifies replication and scaling of applications.

There are two types of server virtualization technologies that are common in data center environments—hardware-level virtualization and operating system level virtualization. Hardware level virtualization involves running a hypervisor which virtualizes the server's resources across multiple virtual machines. Each hardware virtual machine (VM) runs its own operating system and applications. By contrast, operating system virtualization virtualizes resources at the OS level. OS-level virtualization encapsulates standard OS processes and their dependencies to create "containers", which are collectively managed by the underlying OS kernel. Examples of hardware virtualization include Xen [26], KVM [40], and VMware ESX [22]. Operating system virtualization is used by Linux containers (LXC [7]), Ubuntu LXD [17], Docker [2], BSD Jails [38], Solaris Zones [28] and Windows Containers [24].

Both types of virtualization technologies also have management frameworks that enable VMs and applications to be deployed and managed at data center scale. Examples of VM management frameworks include commercial offerings like vCenter [23] and open source frameworks like OpenStack [8], CloudStack [12]. Kubernetes [5] and Docker Swarm [13] are recent container management frameworks.

While hardware virtualization has been the predominant virtualization technology for deploying, packaging, and managing applications; containers (which use operating system virtualization) are increasingly filling that role due to the popularity of systems like Docker [2]. Containers promise low-overhead virtualization and improved performance when compared to VMs. Despite the surge of interest in containers in enterprise environments, there is a distinct lack of performance comparison studies which quantify and compare the performance benefits of containers and VMs. Previous research [25, 31] has compared the two technologies for single server environments, and our work builds on past work by examining performance in the presence of interference and also focuses on multi-server deployments that are common in cluster and data center environments.

Given these trends, in this paper we ask the following questions:

1. When deploying applications in a data center environment, what are the advantages and disadvantages of each virtualization platform with regards to application performance, manageability and deployment at scale?

Middleware'16, December 12 - 16, 2016, Trento, Italy
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4300-8/16/12...$15.00
DOI: http://dx.doi.org/10.1145/2988336.2988337

2. Under what scenarios is one technology more suitable than the other?

To answer these questions, we conduct a detailed comparison of hardware and OS virtualization. While some of our results and observations are specific to the idiosyncrasies of the platforms we chose for our experimental evaluation, our goal is to derive general results that are broadly applicable to the two types of virtualization technologies. We choose open source platforms for our evaluation—Linux containers (LXC) and KVM (a Linux-based type-2 hypervisor) , and our method involves comparing four configurations that are common in data center environments: bare-metal, containers, virtual machines, and containers inside VMs.

Our comparative study asks these specific questions:

1. How do the two virtualization approaches compare from a resource isolation and overcommitment perspective?

2. How does each approach compare from the perspective of deploying many applications in VMs/containers at scale?

3. How does each virtualization approach compare with respect to the application lifecycle and developer interaction?

4. Can approaches which combine these two technologies (containers inside VMs and lightweight VMs) enable the best of both technologies to be reached?

Our results show that co-located applications can cause performance interference, and the degree of interference is higher in the case of containers for certain types of workloads (Section 4). We also evaluate differences in the management frameworks which control deployment and orchestration of containers and VMs (Section 5). We show how the different capabilities exposed by the two virtualization technologies can affect the management and development of applications (Section 6). Lastly, we evaluate novel approaches which combine hardware and OS virtualization (Section 7).

2. BACKGROUND

In this section we provide some background on the two types of virtualization technologies that we study in this paper.

2.1 Hardware Virtualization

Hardware virtualization involves virtualizing the hardware on a server and creating virtual machines that provide the abstraction of a physical machine. Hardware virtualization involves running a hypervisor, also referred to as a virtual machine monitor (VMM), on the bare metal server. The hypervisor emulates virtual hardware such as the CPU, memory, I/O, and network devices for each virtual machine. Each VM then runs an independent operating system and applications on top of that OS. The hypervisor is also responsible for multiplexing the underlying physical resources across the resident VMs.

Modern hypervisors support multiple strategies for resource allocation and sharing of physical resources. Physical resources may be strictly partitioned (dedicated) to each VM, or shared in a best effort manner. The hypervisor is also responsible for isolation. Isolation among VMs is provided by trapping privileged hardware access by guest operating systems and performing those operations in the hypervisor on behalf of the guest OS. Examples of hardware virtualization platforms include VMware ESXi, Linux KVM and Xen.

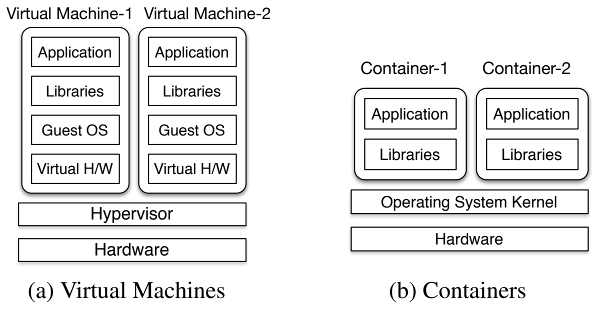


Figure 1: Hardware and operating system virtualization.

2.2 Operating System Virtualization

Operating system virtualization involves virtualizing the OS kernel rather than the physical hardware (Figure 1). OS-level virtual machines are referred to as containers. Each container encapsulates a group of processes that are isolated from other containers or processes in the system. The OS kernel is responsible for implementing the container abstraction. It allocates CPU shares, memory and network I/O to each container and can also provide file system isolation.

Similar to hardware virtualization, different allocation strategies may be supported such as dedicated, shared and best effort. Containers provide lightweight virtualization since they do not run their own OS kernels, but instead rely on the underlying kernel for OS services. In some cases, the underlying OS kernel may emulate a different OS kernel version to processes within a container. This is a feature often used to support backward OS compatibility or emulating different OS APIs such as in LX branded zones [37] on Solaris and in running linux applications on windows [10].

Many OS virtualization techniques exist including Solaris Zones, BSD-jails and Linux LXC. The recent emergence of Docker, a container platform similar to LXC but with a layered filesystem and added software engineering benefits, has renewed interest in container-based virtualization for data centers and the cloud. Linux containers in particular employ two key features:

Cgroups. Control groups [6] are a kernel mechanism for controlling the resource allocation to process groups. Cgroups exist for each major resource type: CPU, memory, network, block-IO, and devices. The resource allocation for each of these can be controlled individually, allowing the complete resource limits for a process or a process group to be specified.

Namespaces. A namespace provides an abstraction for a kernel resource that makes it appear to the container that it has its own private, isolated instance of the resource. In Linux, there are namespaces for isolating: process IDs, user IDs, file system mount points, networking interfaces, IPC, and host names [16].

2.3 Virtualized Data Centers

While hardware and operating system level virtualization operates at the granularity of a single server, data centers are comprised of large clusters of servers, each of which is virtualized. Consequently, data centers must rely on management frameworks that enable virtualized resources of a cluster of servers to be managed efficiently.

Such management frameworks simplify the placement and mapping of VMs onto physical machines, enable VMs to be moved from one machine to another (for load balancing) or allow for VMs to be resized (to adjust to dynamic workloads). Frameworks also support service orchestration, configuration management and automation of cluster management tasks. Examples of popular man-

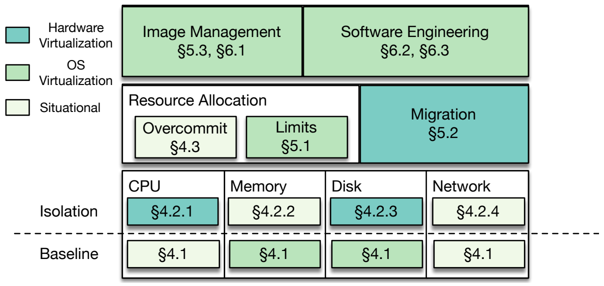


Figure 2: Evaluation map of virtualization platform performance. Shaded areas represent where a platform's capabilities outperforms the other.

agement frameworks for hardware virtualization include OpenStack [8] & VMware vCenter [23] while for OS-level virtualization there exist platforms such as Kubernetes [5] and Docker Swarm [13].

3. PROBLEM STATEMENT

The goal of our work is to conduct a comparative study of hardware and OS-level virtualization from the perspective of a data center. Some qualitative differences between the two are apparent.

OS-level virtualization is lightweight in nature and the emergence of platforms like Docker have brought numerous advantages from an application development and deployment standpoint. VMs are considered to be more heavyweight but provide more robust isolation across untrusted co-resident VMs. Furthermore, while both hardware and OS-level virtualization have been around for decades, the same is not true for their management frameworks.

Management frameworks for hardware virtualization such as vCenter and OpenStack have been around for longer and have acquired more functionality over the years. In contrast, OS-level management frameworks such as Kubernetes are newer and less mature but are evolving rapidly.

From a data center perspective, it is interesting to study what kinds of scenarios are more suitable for hardware virtualization or OS-level virtualization. In particular, our evaluation is guided by the following research questions:

- What are the trade-offs between virtualization platforms on a single server with regards to application performance, resource allocation and resource isolation?
- What are the trade-offs of the two techniques when allocating resources from a cluster perspective?
- What are the benefits from the perspective of deployment and application development process?
- Can the two virtualization techniques be combined to provide high performance and ease of deployment/development?

A summary of our evaluation of the virtualization platforms can be found in Figure 2.

4. SINGLE MACHINE PERFORMANCE

In this section, we compare the single-machine performance of containers and VMs. Our focus is to highlight the performance of different workload types under various deployment scenarios. Prior work on containers and VM performance [25, 31] has focused on comparing the performance of both of these platforms in isolation—the host is only running one instance of the application. Instead, we consider the performance of applications as they are deployed in data center and cloud environments. The two primary characteristics of these environments are multi-tenancy and overcommitment. Multi-tenancy arises when multiple applications are deployed on shared hardware resources. Data centers and cloud platforms may also *overcommit* their hardware resources by running applications with resource requirements that exceed available capacity. Both multi-tenancy and overcommitment are used to increase consolidation and reduce the operating costs in clouds and data centers. Therefore, for our performance comparison of containers and VMs, we also focus on multi-tenancy and overcommitment scenarios, in addition to the study of the virtualization overheads when the applications are running in isolation.

In all our experiments, we use KVM [40] (a type-2 hypervisor based on Linux) for running VMs, and LXC [7] for running containers. This allows us to use the same Linux kernel and reduces the differences in the software stacks when comparing the two platforms, and tease out the differences between OS and hardware virtualization. Since virtual machine performance can be affected by hardware and hypervisor features, we restrict our evaluation to using hardware virtualization features that are present in standard default KVM installations. Wherever applicable, we will point to additional hypervisor and hardware features that have shown to reduce virtualization overheads in specific scenarios.

Methodology. We configured both containers and VMs in such a way that they are comparable environments and are allocated the same amount of CPU and memory resources. We configured each LXC container to use two cores, each pinned to a single core on the host CPU. We set a hard limit of 4 GB of memory and used bridged networking for public IP addresses. We configured each KVM VM to use 2 cores, 4GB of memory and a 50GB hard disk image. We configured the VMs to use virtIO for both network and disk I/O and used a bridged networking interface with TAP for network connectivity. The guest operating system for the VMs is Ubuntu 14.04.3 with a 3.19 Linux kernel. The LXC containers also use the same Ubuntu 14.04.3 userspace libraries (since they are containers, the kernel is shared with the host).

Setup. The hardware platform for all our experiments is a Dell PowerEdge R210 II server with a 4 core 3.40GHz E3-1240 v2 Intel Xeon CPU, 16GB memory, and a 1 TB 7200 RPM disk. We disabled hyperthreading to reduce the effects of hardware scheduling and improve the stability of our results. The host ran on Ubuntu 14.04.3 (64 bit) with a 3.19 Linux Kernel. For virtualization we used LXC version 1.0.7 and QEMU with KVM version 2.0.0.

Workloads. We use these workloads which stress different resources (CPU, memory, disk, network):

Filebench. We use the customizable file system benchmark filebench v.1.4.91 with its randomrw workload to test file IO performance. The randomrw workload allocates a 5Gb file and then spawns two threads to work on the file, one for reads and one for writes. We use the default 8KB IO size.

Kernel-compile. We use the Linux kernel compile benchmark to test the CPU performance by measuring the runtime of compiling Linux-4.2.2 with the default configuration and multiple threads (equal to the number of available cores).

SpecJBB. SpecJBB2005 is a popular CPU and memory intensive benchmark that emulates a three tier web application stack and exercises the underlying system supporting it.

RUBiS. RUBiS is a multi-tier web application that emulates the popular auction site eBay. We run RUBiS version 1.4.3 with three guests: one with the Apache and PHP frontend, one with the RUBiS backend MySQL database and one with the RUBiS client and

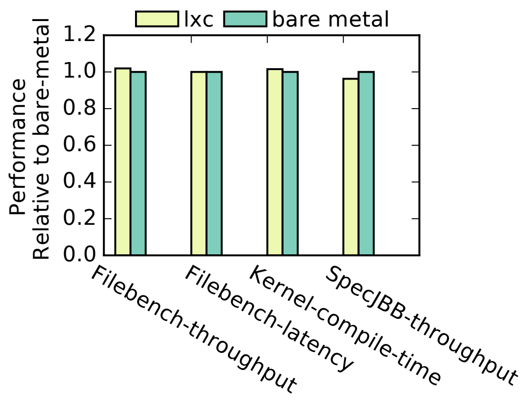


Figure 3: LXC performance relative to bare metal is within 2%.

workload generator.

YCSB. YCSB is a workload generator developed by Yahoo to test different key value stores used in the cloud. YCSB provides statistics on the performance of load, insert, update and read operations. We use YCSB version 0.4.0 with Redis version 3.0.5 key value store. We use a YCSB workload which contains 50% reads and 50% writes.

4.1 Baseline Performance

We first measure the virtualization overhead when only a single application is running on a physical host. This allows us to observe and measure the performance overhead imposed by the virtualization layer. We run the same workload, and configure the containers and the VMs to use the same amount of CPU and memory resources. We shall show the performance of CPU, memory, and I/O intensive workloads.

Because of virtualizing at the OS layer, running inside a container does not add any noticeable overhead compared to running the same application on the bare-metal OS. As alluded to in Section 2, running an application inside a container involves two differences when compared to running it as a conventional OS process (or a group of processes). The first is that containers need resource accounting to enforce resource limits. This resource accounting is also done for the various operations that the kernel performs on behalf of the application (handling system calls, caching directory entries, etc), and adds only a minimal overhead. The other aspect of containers is isolation, which is provided in Linux by namespaces for processes, users, etc. Namespaces provide a virtualized view of some kernel resources like processes, users, mount-points etc, and the number of extra kernel-mode operations involved is again limited. We can thus think of containers in this case as extensions of the `u-limit` and `r-limit` [14] functionality. Our experiments (Figure 3) did not yield any noticeable difference between bare-metal and LXC performance, and for ease of exposition, we assume that the bare-metal performance of applications is the same as with LXC.

CPU. Figure 4a shows the difference in performance for CPU intensive workloads. The performance difference when running on VMs vs. LXCs is under 3% (LXC fares slightly better). Thus, the hardware virtualization overhead for CPU intensive workloads is small, which is in part due to virtualization support in the CPU (VMX instructions and two dimensional paging) to reduce the number of traps to the hypervisor in case of privileged instructions.

Memory. Figure 4b shows the performance of Redis in-memory key-value store under the YCSB benchmark. For the load, read, and update operations, the VM latency is around 10% higher as compared to LXC.

Disk. For testing disk I/O performance, we use the filebench randomrw workload. Because all guest I/O must go through the hypervisor when using virtIO, the VM I/O performance is expected to be worse than LXC. Figure 4c shows the throughput and latency for the filebench benchmark. The disk throughput and latency for VMs are 80% worse for the randomrw test. The randomrw filebench test issues lots of small reads and writes, and each one of them has to be handled by a single hypervisor thread. I/O workloads running inside VMs that are more amenable to caching and buffering show better performance, and we chose the randomrw workload as the worst-case workload for virtIO.

Network. We use the RUBiS benchmark described earlier to measure network performance of guests. For RUBiS, we do not see a noticeable difference in the performance between the two virtualization techniques (Figure 4d).

Summary of baseline results: *The performance overhead of hardware virtualization is low when the application does not have to go through the hypervisor, as is the case of CPU and memory operations. Throughput and latency of I/O intensive applications can suffer even with paravirtualized I/O.*

4.2 Performance Isolation

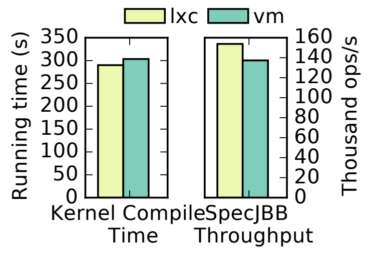
So far, we have shown the performance overheads of virtualization when only a single application is running on the physical host. However, multiple applications of different types and belonging to different users are often co-located on the same physical host to increase consolidation. In this subsection, we measure the performance interference due to co-located applications running inside VMs and containers.

When measuring this "noisy neighbor" effect, we are interested in seeing how one application affects the performance of another. We shall compare the performance of applications when co-located with a variety of neighbors versus their stand-alone performance. In all our experiments, the VMs and containers are configured with the same amount of CPU and memory resources. Since the application performance depends on the co-located applications, we compare the application performance for a diverse range of co-located applications:

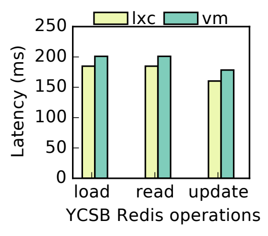
Competing. These co-located applications are contending for the same resource. For example, if our target application is CPU-intensive, then the other co-located applications are also CPU-intensive. This tests how well the platform layer (OS or the hypervisor) is able to partition and multiplex the resources among multiple claimants.

Orthogonal. In this scenario, the co-located applications seek different resources. For example, if one application is CPU-intensive, the other one is network intensive. This scenario is likely when the scheduling and placement middleware prefers to co-locate non-competing applications on the same host to minimize resource contention and improve consolidation.

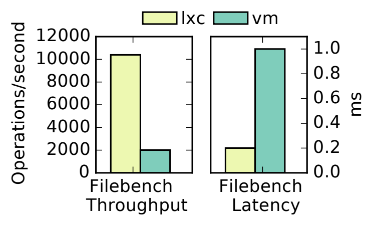
Adversarial. In this scenario, the other co-located application is a misbehaving, adversarial application which tries to cause the other application to be starved of resources. In contrast to the earlier configuration, these adversarial applications may not represent realistic workloads, but may arise in practice if multiple users are allowed to run their applications on shared hardware, and these applications present a vector for a denial of resource attack.



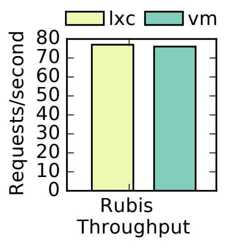
(a) CPU intensive



(b) Memory intensive



(c) Disk intensive



(d) Network intensive

Figure 4: Performance overhead of KVM is negligible for our CPU and memory intensive workloads, but high in case of I/O intensive applications. Unlike CPU and memory operations, I/O operations go through the hypervisor—contributing to their high overhead.

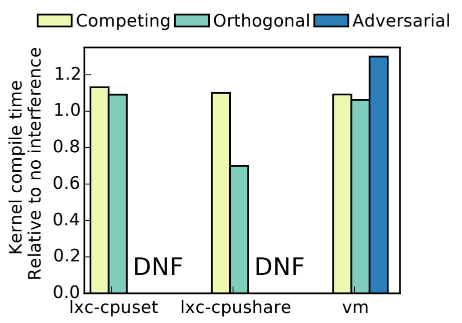


Figure 5: CPU interference is higher for LXC even with CPU-sets, especially since adversarial workloads can cause co-located applications to be starved of resources and not finish execution (DNF: did not finish).

4.2.1 CPU Isolation

To measure the performance interference for CPU bound workloads, we compare the running time of the kernel compile application when it is co-located with other applications relative to its stand-alone performance without any interference. We use another instance of the kernel compile workload to induce CPU contention, SpecJBB as the orthogonal workload , and a fork-bomb as the adversarial workload scenario. The fork bomb is a simple script that overloads the process table by continually forking processes in an infinite loop. Figure 5 shows the kernel compile performance relative to the no interference case for LXC and VMs. In the case of LXC, there are two ways of allocating CPU resources. The LXC containers can either be assigned to CPU-cores (`cpu-sets`), or the containers can be multiplexed across all CPU cores in a fair-share manner by the Linux CPU scheduler (`cpu-shares`).

The same amount of CPU resources were allocated in both the CPU-shares and CPU-sets cases—50% CPU and 2 out of 4 cores respectively. Despite this, running containers with CPU-shares results in a greater amount of interference, of up to 60% higher when compared to the baseline case of stand-alone no-interference performance.

From Figure 5, we see that when co-located with the adversarial fork-bomb workload, the LXC containers are starved of resources and do not finish in any reasonable amount of time, while the VM manages to finish with a 30% performance degradation. While the fork-bomb test may be an extreme adversarial workload, we note that the leaky container abstraction which does not provide careful accounting and control of every physical *and* kernel operation can cause many more such cases to arise.

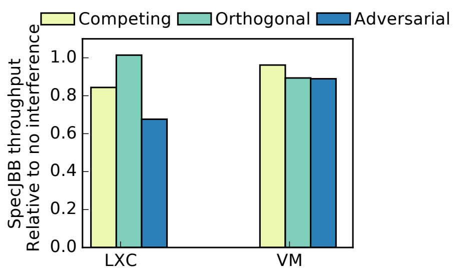


Figure 6: Performance interference is limited for memory intensive workloads, although LXC suffers more with the adversarial (malloc-bomb) workload.

Result: *Interference for CPU-bound workloads is mitigated by hypervisors because of separate CPU schedulers in the guest operating systems. The shared host OS kernel can cause performance disruption in the case of containers, and can potentially lead to denial of service.*

4.2.2 Memory Isolation

To measure performance interference for memory based workloads, we compare SpecJBB throughput against its baseline performance. For the competing case we ran an additional instance of SpecJBB, and used kernel compile as an orthogonal workload. To illustrate an adversarial workload we used a malloc bomb, in an infinite loop, that incrementally allocates memory until it runs out of space.

Figure 6 shows the memory interference result and shows that memory isolation provided by containers is sufficient for most uses. Both the competing and orthogonal workloads for VMs and LXC are well within a reasonable range of their baseline performance. In the adversarial case however, it appears that the VM outperforms LXC. LXC sees a performance decrease of 32% where as the VM only suffers a performance decrease of 11%.

4.2.3 Disk Isolation

For disk interference, we compare filebench's baseline performance against its performance while running alongside the three types of interference workloads. We chose the following workloads as neighbors: a second instance of filebench for the competing case, kernel compile for orthogonal and an instance of Bonnie++, a benchmark that runs lots of small reads and writes, for the adversarial case. The containers were configured with equal block-IO cgroup weights to ensure equal I/O bandwidth allocation. Figure 7 shows the disk interference result.

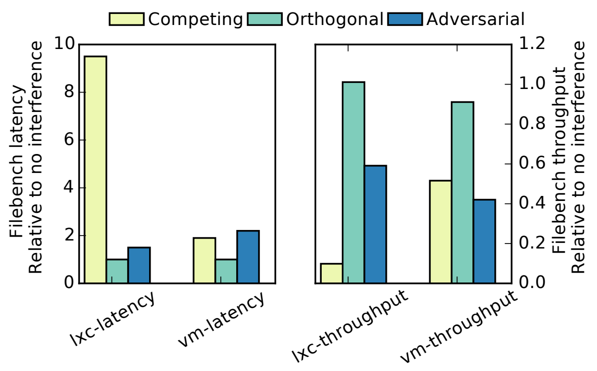


Figure 7: Disk performance interference is high for both containers and virtual machines.

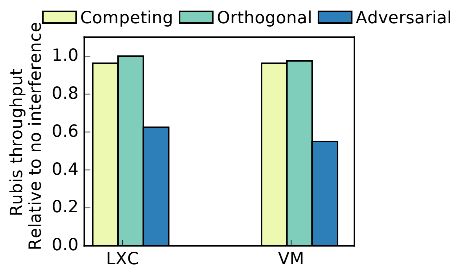


Figure 8: Network performance interference when running RUBiS is similar for both containers and virtual machines.

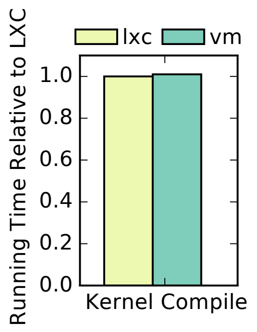
For LXC, the latency increases 8 times. For VMs, the latency increase is only 2x. This reduction can be attributed to the fact that the disk I/O performance for VMs is quite bad even in the isolated case (Figure 4c), and raw disk bandwidth is still available for other VMs to use. However, a reduction of 8x in the case of containers is still significant, and points to the lack of disk I/O isolation.

Result: *Sharing the host OS block layer components like the I/O scheduler increases the performance interference for disk bound workloads in containers. VMs may offer more isolation and shield better against noisy neighbors, but the effectiveness of hypervisors in mitigating I/O interference is reduced because of shared I/O paths for guest I/O.*

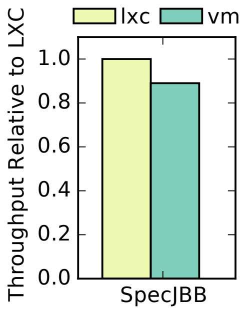
4.2.4 Network Isolation

To measure performance interference for network intensive workloads, we compare RUBiS throughput against its baseline performance while running co-located with the three types of interfering workloads. To measure competing interference we run RUBiS alongside the YCSB benchmark while we use SpecJBB to measure orthogonal interference. To measure adversarial interference performance we run a guest that is the receiver of a UDP bomb. The guest runs a UDP server while being flooded with small UDP packets in an attempt to overload the shared network interface. Figure 8 shows the network interference results. For each type of workload, there is no significant difference in interference.

Summary of interference results: *Shared hypervisors or operating system components can reduce performance isolation, especially with competing and adversarial workloads in the case of containers.*



(a) CPU intensive



(b) Memory intensive

Figure 9: Kernel compile and SpecJBB performance of VMs when the CPU is overcommitted by a factor of 1.5.

4.3 Overcommitment

To improve packing efficiency, it is often necessary to overcommit the physical resources. Containers use overcommitment mechanisms built into the operating system itself, such as CPU overcommitment by multiplexing and memory overcommit by relying on the virtual memory subsystem's swapping and paging. Hypervisors provide overcommitment by multiplexing the virtual hardware devices onto the actual physical hardware. The overcommitment provided by hypervisors may be of reduced effectiveness because the guest operating system may be in the dark about the resources that have been taken from the VM. For example, the hypervisor might preempt a vCPU of a VM at the wrong time when it is holding locks. Such lock holder and lock waiter preemptions can degrade performance for multi-threaded applications [45]. We investigate how application performance behaves in resource overcommitment scenarios.

Figure 9 compares the relative performance of LXC and VMs in an overcommitment scenario where both the CPU and memory have been oversubscribed by a factor of 1.5. CPU overcommitment is handled in both hypervisors and operating systems by multiplexing multiple vCPUs and processes onto CPU cores, and for the CPU intensive kernel compile workload, VM performance is within 1% of LXC performance (Figure 9a).

Memory overcommitment in hypervisors is more challenging, because virtual machines are allocated a fixed memory size upon their creation, and overcommitting memory involves "stealing" pages from the VM via approaches like host-swapping or ballooning. For the more memory intensive SpecJBB workload (Figure 9b), the VM performs about 10% worse compared to LXC. Memory overcommitment for VMs can also be improved by approaches like page deduplication [50, 27, 47] which reduce the effective memory footprint of VMs by sharing pages, or by using approaches like transcendent memory [44], which can provide more efficient in-memory compressed storage of evicted pages.

Result: *Hypervisors handle CPU overcommitment more gracefully than memory overcommitment, because vCPUs can be multiplexed onto CPU cores dynamically.*

5. CLUSTER MANAGEMENT

In this section, we illustrate the differences in managing containers and VMs at scale. Cluster operators seek to satisfy the resource requirements of all applications and increase consolidation to reduce the operating costs. In the rest of this section, we shall see how the different characteristics of containers and VMs affect the

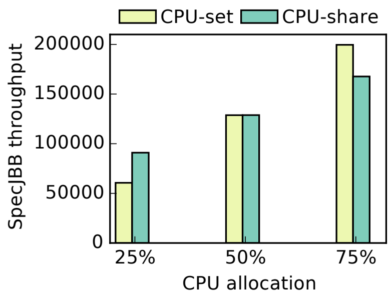
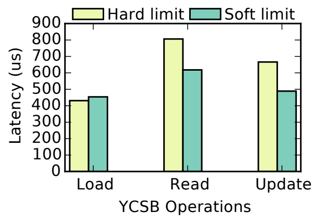
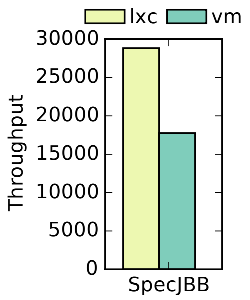


Figure 10: CPU-shares vs. CPU-sets allocation can have a significant impact on application performance, even though the same amount of CPU resources are allocated in each instance.



(a) Performance with hard vs. soft limits when CPU and memory are overcommitted by a factor of 1.5.



(b) Performance of VMs vs. soft-limited containers at overcommitment factor of 2.

Figure 11: Soft limits in containers improve overcommitment by allowing under utilized resources to be used by applications that need them the most.

different options for managing and provisioning resources in a cluster. For comparison purposes, we use VMware vCenter [23] and OpenStack [8] as representative VM management platforms, and Kubernetes [5] as a representative container orchestration system.

5.1 Resource Allocation

Data center and cloud operators use cluster management frameworks to fulfill the resource requirements of applications. We focus on the physical resource requirements of applications, and note that handling application-SLA based requirements is generally outside the scope of management frameworks. Cluster management software rely on the resource provisioning "knobs" that the underlying platforms (hypervisors and operating systems) provide for managing the resource allocation of VMs and containers. The hypervisor controls the allocation of the virtual CPU, memory and I/O devices for each VM. Since VMs can be thought of as sharing the "raw hardware", the resource allocation is also of that granularity. For example, VMs may be allocated a fixed number of virtual CPUs, memory, and I/O devices and bandwidth. For CPU and I/O bandwidth, fair-share or other policies for resource multiplexing may be implemented by the hypervisor.

With containers, resource allocation involves more dimensions, since the resource control knobs provided by the operating system are larger in number. In addition to physical resource allocation like CPU and memory, other kernel resources for controlling CPU scheduling, swapping, etc can also be specified for containers (Table 1). Thus, provisioning for containers involves allocation of physical *and* operating system resources. Not limiting a certain resource may lead to its excessive consumption, and cause performance interference, as we have seen in Section 4.2. As we have also seen, containers have multiple resource allocation options for the same resource (cpu-sets and cpu-shares), and their selection can have a non-negligible impact on performance, as shown in Figure 10. We see that the SpecJBB throughput differs by up to 40% when the container is allocated $\frac{1}{4}$th of cpu cores using cpu-sets, when compared to the equivalent allocation of 25% with cpu-shares.

In addition to the increased number of dimensions required to specify resource requirements, container resource management faces cross-platform challenges. With VMs, abstraction layers like libVirt [15] allow management frameworks like OpenStack to run VMs on multiple hypervisors. Policies for controlling and provisioning VMs can be applied across hypervisors. However, in the case of containers, the resource control interfaces are heavily operating system dependent, and running containers across operating systems may prove to be a challenge. Efforts like the Open Container Initiative and CNCF (Cloud Native Computing Foundation) have been launched to develop a standards for container formats, resource and lifecycle control. This will allow frameworks like Kubernetes to specify resource allocation and control containers of different types (Docker, rkt, etc) across different operating systems.

Soft and hard limits. A fundamental difference in resource allocation with containers is the prevalence of soft limits on resources. Soft limits enable applications to use resources *beyond* their allocated limit if those resources are under-utilized. In the case of virtual machines, resource limits are generally *hard*—the VM cannot utilize more resources than its allocation even if these resources are idle on the host. Dynamically increasing resource allocation to VMs is fundamentally a hard problem. VMs are allocated virtual hardware (CPUs, memory, I/O devices) before boot-up. Adding virtual devices during guest execution requires the guest operating system to support some form of *device hotplug*. Because of the rare nature of hotplugging CPU and memory in the physical environment, hotplug support for operating systems is limited, and therefore not a feasible option in most cases. Virtualization specific interfaces such as transcendent memory [44] can be used to dynamically increase effective VM memory, but current VM management frameworks do not support it as it requires co-operation between the VMs and the hypervisor, which is often hard to guarantee in mulit-tenant environments.

By allowing applications to use under-utilized resources, soft limits may enable more efficient resource utilization. Soft limits are particularly effective in overcommitted scenarios. Figure 11a shows scenarios where the CPU and memory for the containers have been overcommitted by a factor of 1.5. In this scenario, the YCSB latency is about 25% lower for read and update operations if the containers are soft-limited. Similarly, Figure 11b shows results from a scenario where the resources were overcommitted by a factor of two. The containers were again soft-limited, and we compare against VMs which have hard limits. We again see a big improvement with soft limits, as the SpecJBB throughput is 40% higher with the soft-limited containers compared to the VMs.

Result: *Soft-limits enable containers to run with better performance on overcommitted hosts. Soft-limits are inherently difficult with VMs because their resource allocation is fixed during guest operating system boot-up.*

	KVM	LXC/Docker
CPU	VCPU count	CPU-set/CPU-shares, cpu-period, cpu-quota,
Memory	Virtual RAM size	Memory soft/hard limit, kernel memory, overcommitment options, shared-memory size, swap size, swappiness
I/O	virtIO, SR-IOV	Blkio read/write weights, priorities
Security Policy	None	Privilege levels, Capabilities(kernel modules, nice, resource limits, setuid)
Volumes	Virtual disks	File-system paths
Environment vars	N/A	Entry scripts

Table 1: Configuration options available for LXC and KVM. Containers have more options available.

Application	Container memory size	VM size
Kernel Compile	0.42	4
YCSB	4	4
SpecJBB	1.7	4
Filebench	2.2	4

Table 2: Memory sizes (in Gigabytes) which have to be migrated for various applications. The mapped memory which needs to be migrated is significantly smaller for containers.

5.2 Migration

Live migration is an important mechanism to transfer the application state from one host to another, and is used in data centers for load balancing, consolidation and fault-tolerance. Live-migration for VMs works by periodically copying memory pages from the source to the destination host, and management platforms trigger migrations based on migration policies which take into account the availability of resources on the source and destination hosts. The duration of live migration depends on the application characteristics (the page dirty rate) as well as the memory footprint of the application.

Virtual machine live migration is mature and widely used in data centers, and frameworks like vCenter have sophisticated policies for automatically moving VMs to balance load. Unlike VM migration, container migration requires process migration techniques and is not as reliable a mechanism. Container migration is harder to implement in practice because of the large amount of operating system state associated with a process (process control block, file table, sockets, etc) which must be captured and saved along with the memory pages. As a result, projects such as CRIU [11] (Checkpoint Restart In Userspace) have been attempting to provide live-migration for containers, but the functionality is limited to a small set of applications which use the supported subset of OS services. Container migration is hence not mature (yet), and is not supported by management frameworks. Instead of migration, killing and restarting stateless containers is a viable option for consolidation of containers. Furthermore, container migration depends on the availability of many additional libraries and kernel features, which may not be available on all the hosts. These dependencies may limit the number of potential destination hosts for container migration.

Migrating VMs involves the transfer of both the application state and the guest operating system state (including slab and file-system page caches), which can lead to increased migration times compared to migrating the application state alone. We compare the memory sizes of various applications when deployed in a containers and KVM VMs in Table 2. In both the cases, the containers and VMs are configured with the same memory hard-limit. Except for the YCSB workload, which uses the in-memory Redis key-value store, the application memory footprint with containers is about 50-90% smaller as compared to the equivalent VMs.

Result: *The memory footprint of containers is smaller than VMs, leading to potentially smaller migration times. But container migration mechanisms are not currently mature enough, and this lack of maturity along with the larger number of dependencies may limit their functionality.*

5.3 Deployment

Launching applications is the key feature provided by management frameworks, and the ability to launch these applications at low latency is an important requirement. This is especially the case for containers whose use-cases often call for rapid deployment. Launching applications requires the management platform to support and implement many policies for provisioning hardware resources and deploying the VMs/containers. A significant amount of effort has gone into provisioning policies and mechanisms for VMs, and these policies may or may not translate to containers.

Management and orchestration frameworks must also provide policies for application placement, which involves assigning applications to physical hosts with sufficient resources available. Virtual machine placement for consolidation has received significant attention [51]. For containers, which require a larger number of dimensions to specify resource limits, existing placement and consolidation policies may need to be modified and refined. Management platforms also enforce co-location (affinity) constraints which ensure that applications can be "bundled" and placed on the same physical host. In Kubernetes, this is accomplished by using pods, which are groups of containers and also function as the unit of deployment. As we have shown earlier, containers suffer from larger performance interference. Because of this concern, container placement might need to be optimized to choose the right set of neighbors for each application.

By running multiple instances of an application, virtualization enables easy horizontal scaling. The number of replicas of a container or a VM can be specified to the management frameworks. Additionally, Kubernetes also monitors for failed replicas and restarts failed replicas automatically. Quickly launching application replicas to meet workload demand is useful to handle load spikes etc. In the unoptimized case, booting up virtual machines can take tens of seconds. By contrast, container start times are well under a second [25]. An optimization to reduce the start-up latency of VMs is to employ fast VM cloning [41]. A similar functionality is provided by vCenter linked-clones.

Multi-tenancy An important aspect of virtualization is the ability to share a cluster among multiple users. Due to hardware virtualization's strong resource isolation, multi-tenancy is common in virtual machine environments. Because the isolation provided by containers is weaker, multi-tenancy is considered too risky especially for Linux containers. In cases where the isolation provided by the OS is strong enough (as in the case of Solaris), containers have been used for multi-tenancy in production environments [4].

The multi-tenant support for container management platforms like Kubernetes is under development but because of the security risks of sharing machines between untrusted users, policies for security-aware container placement may need to be developed. Unlike VMs which are "secure by default", containers require several security configuration options (Table 1) to be specified for safe execution. In addition to satisfying resource constraints, management frameworks also need to verify and enforce these security constraints.
Summary. We have shown how the unique characteristics of VMs and containers lead to the different management capabilities of their respective management frameworks. As container technology matures, some of these are bound to change, and hence we have focused on the fundamental differences. There are opportunities and challenges for resource allocation in containers as a result of their richer resource allocation requirements; lack of live-migration, and multi-tenancy constraints due to security and performance isolation concerns.

6. END-TO-END DEPLOYMENT

Performance is not the sole appeal of containers. Systems like Docker have surged in popularity by exploiting both the low deployment overheads and the improvements in the software development life cycle that containers enable. In this section, we look at how containers and VMs differ in the end-to-end deployment of applications—from the developer's desktop/laptop to the production cloud or data center.

6.1 Constructing Images

Disk images encapsulate the code, libraries, data, and the environment required to run an application. Containers and VMs have different approaches to the image construction problem. In the case of VMs, constructing an image is really constructing a virtual disk which has the operating system, application libraries, application code and configuration, etc. The traditional way to build a virtual machine is to build from scratch which involves allocating blocks for a disk image and installing and configuring the operating system. Libraries and other dependencies of the application are then installed via the operating system's package management toolchain.

Constructing VM images can be automated with systems like Vagrant [21], Chef, Puppet, etc., wherein the VM images are created by specifying configuration scripts and recipes to automate operating system configuration, application installation and service management. Additionally, cloud operators also provide virtual appliances—pre-built and pre-configured images for running specific applications (such as MySQL).

Containers implement a different approach due to their lightweight nature. Because containers can use the host file system, they do not need virtual disks (which is a block-level abstraction layer). Thus, a container "image" is simply a collection of files that an application depends on, and includes libraries and other dependencies of the application. In contrast to VMs, no operating system kernel is present in the container image, since containers use the host OS kernel. Similar to virtual appliances, pre-built container images for popular applications exist. Users can download these images and build off of them.

Constructing container images can also be automated in systems like Docker [2] via dockerfiles. Container images can be built from existing ones in a deterministic and repeatable manner by specifying the commands and scripts to build them in the dockerfile. In contrast to VMs where creating images is handled by third party tools like Vagrant, dockerfiles are an integral part of Docker, and enable tighter integration between containers and their provenance information.

Application	Vagrant	Docker
MySQL	236.2	129
Nodejs	303.8	49

Table 3: Time (in seconds) to build an image using Vagrant (for VMs) and Docker.

Due to differences in the contents and the sizes of constructed images, the time to construct them are different for containers and VMs. Table 3 shows the time to build images of popular applications when the build process is specified via Vagrant and Docker. Building both container and VM images involves downloading the base images (containing the bare operating system) and then installing the required software packages. The total time for creating the VM images is about 2× that of creating the equivalent container image. This increase can be attributed to the extra time spent in downloading and configuring the operating system that is required for virtual machines.

Application	VM	Docker	Docker Incremental
MySQL	1.68GB	0.37GB	112KB
Nodejs	2.05GB	0.66GB	72KB

Table 4: Resulting image sizes when deploying various applications. For Docker, we also measure the incremental size when launching multiple containers belonging to the same image.

The differences in creating images discussed above influence another important concern: image size. The resulting image sizes are also smaller for containers, because they do not require an operating system installation and additional overhead caused by the guest OS file system, etc. This difference in image sizes is shown in Table 4, which compares VM and container images for different applications. The smaller container image sizes (by up to 3x) allows for faster deployment and lower storage overhead.
Results: *Container images are faster to create and are smaller, because the operating system and redundant libraries need not be included.*

6.2 Version Control

A novel feature that containers enable is the ability to version control images. Just like version control for software, being able to commit and track lineage of images is useful, and is a key feature in Docker's design. In the case of Docker, this versioning is performed by using copy-on-write functionality (like AuFS) in the host file system. Storing images in a copy-on-write file system allows an image to be composed of multiple *layers*, with each layer being immutable. The base layer comprises of the base operating system, and modifications to existing images create additional layers. In this way, multiple container images can share the same physical files. Updates to files results in creation of new layers.

Virtual machine disks can also use layered storage in the form of copy-on-write virtual disk formats (such as qcow2, FVD [49], etc.). Unlike layered container images, virtual disks employ block level copy-on-write instead of file-level. This finer granularity of versioning creates a semantic decoupling between the user and the virtual disks—it is harder to correlate changes in VM configurations with changes in the the virtual disks. In the case of Docker, layers also store their ancestor information and what commands were used to build the layer. This allows Docker to have a semantically rich image versioning tree.

Workload	Docker	VM
Dist Upgrade	470	391
Kernel install	292	303

Table 5: Running time (in seconds) of operations in Docker and VMs. Docker image layers cause increase in runtime for write intensive applications.

While layering enables versioning of images and allows images to be easily composed, there are performance implications because of the copy-on-write implications. Writes to a file in a layer causes a new copy and a new layer to be created. We examine the performance overhead induced by the copy on write layers in Table 5, which shows the running time of write-heavy workloads. Docker's layered storage architecture contributes results in an almost 40% slowdown compared to VMs. This slow-down is almost entirely attributable to the AuFS [20] copy-on-write performance, and using other file systems with more optimized copy-on-write functionality, like ZFS, BtrFS, and OverlayFS can help bring the file-write overhead down.

Immutable copy-on-write layers also make cloning images extremely fast, and this can be useful to deploy multiple containers from the same image for rapid cloning and scaling. Starting multiple containers from a single image is a lightweight operation, and takes less than one second for most images. This is attributable to the reduced state size which is associated with a container which needs to be copied instead of copying an entire image. The incremental image size for a new Docker container is shown in Table 4. To launch a new container, only 100KB of extra storage space is required, compared to more than 3 GB for VMs.

Result: *Although copy-on-write layers used for storing container images aid in reducing build times and enable versioning, they may reduce application performance, especially for disk I/O intensive applications, because of the overhead of copy-on-write.*

6.3 Continuous Delivery/Integration

Faster image construction, image versioning, and rapid cloning have changed software engineering practices. Encapsulating all software dependencies within an image provides a standardized and convenient way of sharing and deploying software. Both VM and container images are used for packaging application software, but as we have discussed, the smaller image footprint and the semantically rich image versioning has made container images (especially Docker images) an increasingly popular way to package and deploy applications. The metaphor of OS containers being "shipping containers" has also been used widely—instead of custom build and deployment scripts for each applications, container (and VM) images provide standardized formats and tool-chains.

Another novel use-case of container images is to link container image versions to the application software versions which they correspond to. For example, Docker images can be automatically built whenever changes to a source code repository are committed. This allows easier and smoother continuous integration, since the changes in code base are automatically reflected in the application images. An important aspect of the distributed systems development process is updating deployed services to reflect code changes, add features, etc. Rolling updates of deployed containers is a feature which is exposed by Kubernetes.

7. MIXING CONTAINERS AND VMS

Thus far we have seen that containers and VMs have performance and manageability trade-offs. It is interesting to consider if it is possible to combine the best characteristics of each platform into a single architecture. We shall study the viability of two approaches which mix OS-level and hardware virtualization: nested containers in VMs, and lightweight VMs.

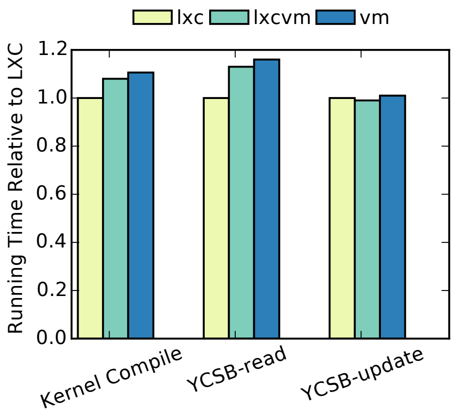


Figure 12: Relative performance of VMs and nested containers (LXCVM) for Kernel compile and YCSB workloads when both CPU and memory are overcommitted by 1.5 times. Containers inside VMs improve the running times of these workloads by up to 5%.

7.1 Containers inside VMs

In the nested container architecture, each container is encapsulated in a VM, and applications run inside the container. One or more containers may be encapsulated inside a VM. This nesting has several advantages. Applications can benefit from the security and performance isolation provided by the VM, and still take advantage of the provisioning and deployment aspects of containers. This approach is quite popular, and is used to run containers in public clouds where isolation and security are important concerns. Container services in public clouds, such as Amazon AWS Elastic Container Service [1] and Google Cloud Platform's Container Engine [3] run user containers inside virtual machine instances which are re-purposed to also support container deployment stacks such as Docker. From a cloud and data center operator's point of view, an advantage of this approach is that the existing hardware virtualization based architecture can be used for resource provisioning and management.

In addition to public cloud settings, nested containers in VMs are also useful on the desktop if the host operating system does not natively support containers, but can run virtual machines. Such a setup is the default way of running Docker on operating systems like Mac-OS (which cannot run Linux binaries, and uses VirtualBox VMs to run Docker containers). This allows users and developers to run containers in their existing environment.

Nested containers in VMs provide another, perhaps surprising performance benefit. Since only containers from a single user may be allowed to run in a VM, neighboring containers within a VM can now be trusted. In addition to reducing the security risk, this also opens the door for using *soft* resource limits. As noted in Section 5.1, soft container resource limits allow containers to use under-utilized resources, thereby increasing the application performance. In Figure 12, we compare the performance of applications running on LXC, VMs, and nested LXC containers inside a VM. In the case of nested containers, the container resource limits (both CPU and memory) are soft-limited, and we run multiple containers inside a VM. With VMs and LXCs, we only run a single application in each container/VM. We see that the running time of kernel-compile in nested containers (LXCVM) is about 2% lower than compared to VMs, and the YCSB read latency is lower by 5% compared to VMs. Thus, running larger VMs and running soft-limited containers inside them *improves* performance slightly when compared to running applications in separate virtual machine silos. **Result:** *Nesting containers in VMs can provide VM-like isolation for applications, and enables the use of soft resource limits which provide slightly improved performance compared to VMs.*

7.2 Lightweight VMs

We have seen how nested containers combine VM-level isolation and the deployment and provisioning aspects of containers. There is another approach which attempts to provide similar functionality, but does not rely on nesting. Instead, lightweight VM approaches like Clear Linux [36] and VMWare's Project Bonneville [29] seek to provide extremely low overhead hardware virtualization. Lightweight VMs run customized guest operating systems which are optimized for quick boot-up, and are characterized by their low footprint and extensive use of para-virtualized interfaces. Lightweight VMs therefore provide reduced footprint and deployment times compared to traditional hardware VMs. We will use the open source Clear Linux system as a representative example of such a lightweight VM.

Lightweight VMs seek to address two of the major drawbacks of traditional VMs: footprint, and host transparency. The VM footprint is reduced by removing redundant functionality provided by the hypervisor, such as bootloaders, emulation for legacy devices such as floppy drives, etc. Along with optimizations for fast kernel boot, this allows the VM to boot in under one second, compared to tens of seconds required to boot traditional VMs. We measured the launch time of Clear Linux Lightweight VMs to be under 0.8 seconds, compared to 0.3 seconds for the equivalent Docker container. While containers and lightweight VMs can be started up faster than traditional VMs, traditional VMs can also be quickly restored from existing snapshots using lazy restore [53], or can be cloned from existing VMs [41]. Thus, instead of relying on a cold boot, fast restore and cloning techniques can be applied to traditional VMs to achieve the same effect.

Containers can access files on the host file system directly, and executables, libraries and data required for application operation does not have to be first transferred to a virtual disk as is the case for traditional VMs. As noted earlier, virtual disks decouple the host and guest operation and pose an inconvenience. An important feature of lightweight VMs like Clear Linux is that the VMs can directly access host file system data. This eliminates the time consuming step of creating bespoke virtual disk images for each application. Instead, VMs can share and access files on the host file system without going through the virtual disk abstraction. This is enabled by new technologies in Linux like Direct-Access (DAX), which allows true zero copy into the VM's user address space and bypasses the page cache completely. In addition to removing the need for dedicated block-level virtual disks, zero-copy and page cache bypass reduces the memory footprint of these VMs by eliminating double caching [47]. To translate guest file system operations for the host file system, Clear Linux uses the 9P file system interface [9].

This combination of lightweight hardware virtualization and direct host file system access enables new ways of combining hardware and operating system virtualization. For example, Clear Linux can run existing Docker containers and run them as lightweight VMs, thus providing security and performance isolation, and making VMs behave like containers as far as deployment goes. **Result:** *Lightweight VMs with host filesystem sharing reduces virtual machine overhead and footprint, providing container-like deployment with the isolation properties of VMs.*

8. RELATED WORK

Both hardware and operating system virtualization have a long and storied history in computing. More recently, hypervisors such as Xen [26], VMware ESX [50], and KVM [40] have been used to provide virtualized data centers. Operating system virtualization in UNIX has been originally implemented in FreeBSD using Jails [38] and in Solaris using Zones [28]. The recent surge of interest in containers has been partly because of the maturation of cgroups and namespaces in Linux, and partly because of Docker [2].

Both hardware and operating system virtualization technologies have been growing at a rapid pace, and research work evaluating the performance aspects of these platforms provides an empirical basis for comparing their performance. A performance study of container and virtual machine performance in Linux is performed in [31, 25], which evaluates LXC, Docker, and KVM across a wide spectrum of benchmarks. While conventional wisdom and folklore point to the lightweight nature of containers compared to VMs, recent work [25] has investigated memory footprint of VMs with page-level memory deduplication [47, 27] and shown that the effective memory footprint of VMs may not be as large as widely claimed. Containers have been pitched as a viable alternative to VMs in domains where strict isolation is not paramount such as high performance computing [46, 52] and big data processing [32]. Docker storage performance is investigated in [33].

The high performance overheads of hardware virtualization in certain situations have been well studied [35], and many approaches have been proposed to reduce the hypervisor overhead [39]. Operating system virtualization has always promised to deliver low overhead isolation, and comparing the two approaches to virtualization is covered in [48, 30]. While most prior work on comparing performance of containers vs. VMs has focused on a straight shoot-out between the two, our work also considers performance interference [43], overcommitment, and different types of resource limits. Isolation in multi-tenants environments is crucial, especially in public clouds which have begun offering container services in addition to conventional VM based servers [1, 3, 4]. In addition to evaluating the performance of the two virtualization technologies, we also show the performance of containers inside VMs and lightweight VMs such as Clear Linux [36].

Unikernels [42] have been proposed as another virtualization platform that run applications directly on hardware virtual machines without a guest OS. Their value in production environments is still being determined [19].

Both VMs and containers need to be deployed across data centers at large scale, and we have also compared the capabilities of their respective management frameworks. Frameworks such as OpenStack [8], vCenter [23] provide the management functionality for VMs. Kubernetes [5], Docker Swarm [13], Mesos [34], Marathon [18], and many other burgeoning frameworks provide equivalent functionality for containers.

9. CONCLUSIONS

Containers and VMs differ in the virtualization technology they use, and this difference manifests itself in their performance, manageability, and use-cases. Containers promise bare metal performance, but as we have shown, they may suffer from performance interference in multi-tenant scenarios. Containers share the underlying OS kernel, and this contributes to the lack of isolation. Unlike VMs, which have strict resource limits, the containers also allow soft limits, which are helpful in overcommitment scenarios, since they may use underutilized resources allocated to other containers. The lack of isolation and more efficient resource sharing due to soft-limits makes running containers inside VMs a viable architecture.

While containers may offer near bare metal performance and a low footprint, their popularity has also been fueled by their integration into the software development process. In particular, Docker's use of copy-on-write layered file systems and version control has enabled easier continuous delivery and integration and a more agile software development process. Lightweight VMs such as Clear Linux aim to provide the popular container features, but with the isolation properties of VMs, and hybrid virtualization approaches seems to be a promising avenue for research.

Acknowledgements. We thank all the reviewers for their insightful comments, which improved the quality of this paper. This work is supported in part by NSF grants #1422245 and #1229059 and a gift from Cisco. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

10. REFERENCES

[1] Amazon EC2 Container Service. https://aws.amazon.com/ecs, June 2015.

[2] Docker. https://www.docker.com/, June 2015.

[3] Google container engine. https://cloud.google.com/container-engine, June 2015.

[4] Joyent Public Cloud. https://www.joyent.com, June 2015.

[5] Kubernetes. https://kubernetes.io, June 2015.

[6] Linux cgroups. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt, June 2015.

[7] Lxc. https://linuxcontainers.org/, June 2015.

[8] Openstack. https://www.openstack.org, June 2015.

[9] 9p file system interface. http://www.linux-kvm.org/page/9p_virtio, March 2016.

[10] Bash on ubuntu on windows. https://msdn.microsoft.com/en-us/commandline/wsl/about, 2016.

[11] Checkpoint Restore in User Space. https://criu.org/, March 2016.

[12] Cloudstack. https://cloudstack.apache.org/, March 2016.

[13] Docker Swarm. https://www.docker.com/products/docker-swarm, March 2016.

[14] Getting and Setting Linux Resource Limits. http://man7.org/linux/man-pages/man2/setrlimit.2.html, March 2016.

[15] Libvirt Virtualization API. https://libvirt.org, March 2016.

[16] Linux Kernel Namespaces. https://man7.org/linux/man-pages/man7/namespaces.7.html, March 2016.

[17] Lxd. https://linuxcontainers.org/lxd/, January 2016.

[18] Marathon. https://mesosphere.github.io/marathon/, May 2016.

[19] Unikernels are Unfit for Production. https://www.joyent.com/blog/unikernels-are-unfit-for-production, January 2016.

[20] Unioning File Systems. https://lwn.net/Articles/327738/, March 2016.

[21] Vagrant. https://www.vagrantup.com/, March 2016.

[22] VMware ESX hypervisor. https://www.vmware.com/products/vsphere-hypervisor, March 2016.

[23] VMware vCenter. https://www.vmware.com/products/vcenter-server, March 2016.

[24] Windows containers. https://msdn.microsoft.com/virtualization/windowscontainers/containers_welcome, May 2016.

[25] K. Agarwal, B. Jain, and D. E. Porter. Containing the Hype. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, page 8. ACM, 2015.

[26] 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. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[27] S. Barker, T. Wood, P. Shenoy, and R. Sitaraman. An Empirical Study of Memory Sharing in Virtual Machines. In *USENIX Annual Technical Conference*, pages 273–284, 2012.

[28] J. Beck, D. Comay, L. Ozgur, D. Price, T. Andy, G. Andrew, and S. Blaise. Virtualization and Namespace Isolation in the Solaris Operating System (psarc/2002/174). 2006.

[29] B. Corrie. VMware Project Bonneville. http://blogs.vmware.com/cloudnative/introducing-project-bonneville/, March 2016.

[30] R. Dua, A. R. Raja, and D. Kakadia. Virtualization vs Containerization to Support PAAS. In *International Conference on Cloud Engineering*. IEEE, 2014.

[31] 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*, pages 171–172.

[32] M. Gomes Xavier, M. Veiga Neves, F. de Rose, and C. Augusto. A Performance Comparison of Container-based Virtualization Systems for Mapreduce Clusters. In *Euromicro International Conference on Parallel, Distributed and Network-Based Processing*. IEEE, 2014.

[33] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Slacker: Fast Distribution with Lazy Docker Containers. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, pages 181–195. USENIX Association, 2016.

[34] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, volume 11, pages 22–22, 2011.

[35] J. Hwang, S. Zeng, F. Y. Wu, and T. Wood. A Component-based Performance Comparison of Four Hypervisors. In *IFIP/IEEE International Symposium on Integrated Network Management*. IEEE.

[36] Intel. Clear linux. http://clearlinux.org, March 2016.

[37] Joyent. Lx branded zones. https://wiki.smartos.org/display/DOC/LX+Branded+Zones, June 2015.

[38] P.-H. Kamp and R. N. Watson. Jails: Confining the Omnipotent Root. In *Proceedings of the 2nd International SANE Conference*, volume 43, page 116, 2000.

[39] E. Keller, J. Szefer, J. Rexford, and R. B. Lee. NoHype: Virtualized Cloud Infrastructure without the Virtualization. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 350–361. ACM, 2010.

[40] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[41] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. De Lara, M. Brudno, and M. Satyanarayanan. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing. In *EuroSys*, April 2009.

[42] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 461–472, New York, NY, USA, 2013. ACM.

[43] J. N. Matthews, W. Hu, M. Hapuarachchi, T. Deshane, D. Dimatos, G. Hamilton, M. McCabe, and J. Owens. Quantifying the Performance Isolation Properties of Virtualization Systems. In *Workshop on Experimental computer science*, page 6. ACM, 2007.

[44] D. Mishra and P. Kulkarni. Comparative Analysis of Page Cache Provisioning in Virtualized Environments. In *Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2014 IEEE 22nd International Symposium on*, pages 213–222. IEEE, 2014.

[45] J. Ouyang and J. R. Lange. Preemptable Ticket Spinlocks: Improving Consolidated Performance in the Cloud. In *ACM SIGPLAN Notices*, volume 48, pages 191–200. ACM, 2013.

[46] C. Ruiz, E. Jeanvoine, and L. Nussbaum. Performance Evaluation of Containers for HPC. In *Euro-Par: Parallel Processing Workshops*, 2015.

[47] P. Sharma and P. Kulkarni. Singleton: System-wide Page Deduplication in Virtual Environments. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 15–26. ACM, 2012.

[48] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. Container-based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors. In *EuroSys*. ACM, 2007.

[49] C. Tang. Fvd: A High-Performance Virtual Machine Image Format for Cloud. In *USENIX Annual Technical Conference*, 2011.

[50] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, Dec. 2002.

[51] T. Wood, P. J. Shenoy, A. Venkataramani, and M. S. Yousif. Black-box and Gray-box Strategies for Virtual Machine Migration. In *NSDI*, volume 7, pages 17–17, 2007.

[52] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose. Performance Evaluation of Container-based Virtualization for High Performance Computing Environments. In *Euromicro International Conference on Parallel, Distributed and Network-Based Processing*. IEEE, 2013.

[53] I. Zhang, A. Garthwaite, Y. Baskakov, and K. C. Barr. Fast Restore of Checkpointed Memory Using Working Set Estimation. In *VEE*, 2011.