RC25482 (AUS1407-001) July 21, 2014
Computer Science

IBM Research Report

An Updated Performance Comparison of Virtual Machines and Linux Containers

Wes Felter, Alexandre Ferreira, Ram Rajamony, Juan Rubio
IBM Research Division
Austin Research Laboratory
11501 Burnet Road
Austin, TX 78758
USA

An Updated Performance Comparison of Virtual Machines and Linux Containers

Wes Felter, Alexandre Ferreira, Ram Rajamony, Juan Rubio
IBM Research, Austin, TX
{wmf, apferrei, rajamony, rubioj}@us.ibm.com

***Abstract*—Cloud computing makes extensive use of virtual machines (VMs) because they permit workloads to be isolated from one another and for the resource usage to be somewhat controlled. However, the extra levels of abstraction involved in virtualization reduce workload performance, which is passed on to customers as worse price/performance. Newer advances in container-based virtualization simplifies the deployment of applications while continuing to permit control of the resources allocated to different applications.**

In this paper, we explore the performance of traditional virtual machine deployments, and contrast them with the use of Linux containers. We use a suite of workloads that stress CPU, memory, storage, and networking resources. We use KVM as a representative hypervisor and Docker as a container manager. Our results show that containers result in equal or better performance than VMs in almost all cases. Both VMs and containers require tuning to support I/O-intensive applications. We also discuss the implications of our performance results for future cloud architectures.

I. Introduction

Virtual machines are used extensively in cloud computing. In particular, the state-of-the-art in Infrastructure as a Service (IaaS) is largely synonymous with virtual machines. Cloud platforms like Amazon EC2 make VMs available to customers and also run services like databases inside VMs. Many Platform as a Servive (PaaS) and Software as a Service (SaaS) providers are built on IaaS with all their workloads running inside VMs. Since virtually all cloud workloads are currently running in VMs, VM performance is a crucial component of overall cloud performance. Once a hypervisor has added overhead, no higher layer can remove it. Such overheads then become a pervasive tax on cloud workload performance. There have been many studies showing how VM execution compares to native execution [30, 33] and such studies have been a motivating factor in generally improving the quality of VM technology [25, 31].

Container-based virtualization presents an interesting alternative to virtual machines in the cloud [46]. Virtual Private Server providers, which may be viewed as a precursor to cloud computing, have used containers for over a decade but many of them switched to VMs to provide more consistent performance. Although the concepts underlying containers such as namespaces are well understood [34], container technology languished until the desire for rapid deployment led PaaS providers to adopt and standardize it, leading to a renaissance in the use of containers to provide isolation and resource control. Linux is the preferred operating system for the cloud due to its zero price, large ecosystem, good hardware support, good performance, and reliability. The kernel namespaces feature needed to implement containers in Linux has only become mature in the last few years since it was first discussed [17]. Within the last two years, Docker [45] has emerged as a standard runtime, image format, and build system for Linux containers.

This paper looks at two different ways of achieving resource control today, viz., containers and virtual machines and compares the performance of a set of workloads in both environments to that of natively executing the workload on hardware. In addition to a set of benchmarks that stress different aspects such as compute, memory bandwidth, memory latency, network bandwidth, and I/O bandwidth, we also explore the performance of two real applications, viz., Redis and MySQL on the different environments.

Our goal is to isolate and understand the overhead introduced by virtual machines (specifically KVM) and containers (specifically Docker) relative to non-virtualized Linux. We expect other hypervisors such as Xen, VMware ESX, and Microsoft Hyper-V to provide similar performance to KVM given that they use the same hardware acceleration features. Likewise, other container tools should have equal performance to Docker when they use the same mechanisms. We do not evaluate the case of containers running inside VMs or VMs running inside containers because we consider such double virtualization to be redundant (at least from a performance perspective). The fact that Linux can host both VMs and containers creates the opportunity for an apples-to-apples comparison between the two technologies with fewer confounding variables than many previous comparisons.

We make the following contributions:

- We provide an up-to-date comparison of native, container, and virtual machine environments using recent hardware and software across a cross-section of interesting benchmarks and workloads that are relevant to the cloud.
- We identify the primary performance impact of current virtualization options for HPC and server workloads.
- We elaborate on a number of non-obvious practical issues that affect virtualization performance.
- We show that containers are viable even at the scale of an entire server with minimal performance impact.

The rest of the paper is organized as follows. Section II describes Docker and KVM, providing necessary background to understanding the remainder of the paper. Section III describes and evaluates different workloads on the three environments. We review related work in Section IV, and finally, Section V concludes the paper.

II. Background

A. Motivation and requirements for cloud virtualization

Unix traditionally does not strongly implement the *principle of least privilege*, viz., “Every program and every user of the system should operate using the least set of privileges necessary to complete the job.” and the *least common mechanism* principle, viz., “Every shared mechanism ... represents a potential information path between users and must be designed with great care to be sure it does not unintentionally compromise security.” [42]. Most objects in Unix, including the filesystem, processes, and the network stack are globally visible to all users.

A problem caused by Unix’s shared global filesystem is the lack of *configuration isolation*. Multiple applications can have conflicting requirements for system-wide configuration settings. Shared library dependencies can be especially problematic since modern applications use many libraries and often different applications require different versions of the same library. When installing multiple applications on one operating system the cost of system administration can exceed the cost of the software itself.

These weaknesses in common server operating systems have led administrators and developers to simplify deployment by installing each application on a separate OS copy, either on a dedicated server or in a virtual machine. Such isolation reverses the status quo compared to a shared server with explicit action required for sharing any code, data, or configuration between applications.

Irrespective of the environment, customers want to get the performance they are paying for. Unlike enterprise consolidation scenarios where the infrastructure and workload are owned by the same company, in IaaS and PaaS there is an arms-length relationship between the provider and the customer. This makes it difficult to resolve performance anomalies, so *aaS providers usually provision fixed units of capacity (CPU cores and RAM) with no oversubscription. A virtualization system needs to enforce such *resource isolation* to be suitable for cloud infrastructure use.

B. KVM

Kernel Virtual Machine (KVM) [25] is a feature of Linux that allows Linux to act as a type 1 hypervisor [36], running an unmodified guest operating system (OS) inside a Linux process. KVM uses hardware virtualization features in recent processors to reduce complexity and overhead; for example, Intel VT-x hardware eliminates the need for complex ring compression schemes that were pioneered by earlier hypervisors like Xen [9] and VMware [8]. KVM supports both emulated I/O devices through QEMU [16] and paravirtual I/O devices using virtio [40]. The combination of hardware acceleration and paravirtual I/O is designed to reduce virtualization overhead to very low levels [31]. KVM supports live migration, allowing physical servers or even whole data centers to be evacuated for maintenance without disrupting the guest OS [14]. KVM is also easy to use via management tools such as libvirt [18].

Because a VM has a static number of virtual CPUs (vCPUs) and a fixed amount of RAM, its resource consumption is naturally bounded. A vCPU cannot use more than one real CPU worth of cycles and each page of vRAM maps to at most one page of physical RAM (plus the nested page table). KVM can resize VMs while running by “hotplugging” and “ballooning” vCPUs and vRAM, although this requires support from the guest operating system and is rarely used in the cloud.

Because each VM is a process, all normal Linux resource management facilities such as scheduling and cgroups (described in more detail later) apply to VMs. This simplifies implementation and administration of the hypervisor but complicates resource management inside the guest OS. Operating systems generally assume that CPUs are always running and memory has relatively fixed access time, but under KVM vCPUs can be descheduled without notification and virtual RAM can be swapped out, causing performance anomalies that can be hard to debug. VMs also have two levels of allocation and scheduling: one in the hypervisor and one in the guest OS. Many cloud providers eliminate these problems by not overcommitting resources, pinning each vCPU to a physical CPU, and locking all virtual RAM into real RAM. (Unfortunately, OpenStack has not yet enabled vCPU pinning, leading to uneven performance compared to proprietary public clouds.) This essentially eliminates scheduling in the hypervisor. Such fixed resource allocation also simplifies billing.

VMs naturally provide a certain level of isolation and security because of their narrow interface; the only way a VM can communicate with the outside world is through a limited number of hypercalls or emulated devices, both of which are controlled by the hypervisor. This is not a panacea, since a few hypervisor privilege escalation vulnerabilities have been discovered that could allow a guest OS to “break out” of its VM “sandbox”.

While VMs excel at isolation, they add overhead when sharing data between guests or between the guest and hypervisor. Usually such sharing requires fairly expensive marshaling and hypercalls. In the cloud, VMs generally access storage through emulated block devices backed by image files; creating, updating, and deploying such disk images can be time-consuming and collections of disk images with mostly-duplicate contents can waste storage space.

C. Linux containers

Rather than running a full OS on virtual hardware, container-based virtualization modifies an existing OS to provide extra isolation. Generally this involves adding a container ID to every process and adding new access control checks to every system call. Thus containers can be viewed as another level of access control in addition to the user and group permission system. In practice, Linux uses a more complex implementation described below.

Linux containers are a concept built on the *kernel namespaces* feature, originally motivated by difficulties in dealing with high performance computing clusters [17]. This feature, accessed by the `clone()` system call, allows creating separate instances of previously-global namespaces. Linux implements filesystem, PID, network, user, IPC, and hostname namespaces. For example, each filesystem namespace has its own root directory and mount table, similar to `chroot()` but more powerful.

Namespaces can be used in many different ways, but the most common approach is to create an isolated container that has no visibility or access to objects outside the container. Processes running inside the container appear to be running on a normal Linux system although they are sharing the underlying kernel with processes located in other namespaces. Containers can nest hierarchically [19], although this capability has not been much explored.

Unlike a VM which runs a full operating system, a container can contain as little as a single process. A container that behaves like a full OS and runs `init`, `inetd`, `sshd`, `syslogd`, `cron`, etc. is called a *system container* while one that only runs an application is called an *application container*. Both types are useful in different circumstances. Since an application container does not waste RAM on redundant management processes it generally consumes less RAM than an equivalent system container or VM. Application containers generally do not have separate IP addresses, which can be an advantage in environments of address scarcity.

If total isolation is not desired, it is easy to share some resources among containers. For example, bind mounts allow a directory to appear in multiple containers, possibly in different locations. This is implemented efficiently in the Linux VFS layer. Communication between containers or between a container and the host (which is really just a parent namespace) is as efficient as normal Linux IPC.

The Linux *control groups* (cgroups) subsystem is used to group processes and manage their aggregate resource consumption. It is commonly used to limit the memory and CPU consumption of containers. A container can be resized by simply changing the limits of its corresponding cgroup. Cgroups also provide a reliable way of terminating all processes inside a container. Because a containerized Linux system only has one kernel and the kernel has full visibility into the containers there is only one level of resource allocation and scheduling.

An unsolved aspect of container resource management is the fact that processes running inside a container are not aware of their resource limits [27]. For example, a process can see all the CPUs in the system even if it is only allowed to run on a subset of them; the same applies to memory. If an application attempts to automatically tune itself by allocating resources based on the total system resources available it may overallocate when running in a resource-constrained container. As containers mature, this limitation is likely to be addressed.

Securing containers tends to be simpler than managing Unix permissions because the container cannot access what it cannot see and thus the potential for accidentally over-broad permissions is greatly reduced. When using user namespaces, the root user inside the container is not treated as root outside the container, adding additional security. The primary type of security vulnerability in containers is system calls that are not namespace-aware and thus can introduce accidental leakage between containers. Because the Linux system call API set is huge, the process of auditing every system call for namespace-related bugs is still ongoing. Such bugs can be mitigated (at the cost of potential application incompatibility) by whitelisting system calls using seccomp [5].

Several management tools are available for Linux containers, including LXC [48], systemd-nspawn [35], lmctfy [49], Warden [2], and Docker [45]. (Some people refer to Linux containers as "LXC", but this causes confusion because LXC is only one of many tools to manage containers). Due to its feature set and ease of use, Docker has rapidly become the standard management tool and image format for containers. A key feature of Docker not present in most other container tools is layered filesystem images, usually powered by AUFS (Another UnionFS) [12]. AUFS provides a layered stack of filesystems and allows reuse of these layers between containers reducing space usage and simplifying filesystem management. A single OS image can be used as a basis for many containers while allowing each container to have its own overlay of modified files (e.g., app binaries and configuration files). In many cases, Docker container images require less disk space and I/O than equivalent VM disk images. This leads to faster deployment in the cloud since images usually have to be copied over the network to local disk before the VM or container can start.

Although this paper focuses on steady-state performance, other measurements [39] have shown than containers can start much faster than VMs (less than 1 second compared to 11 seconds on our hardware) because unlike VMs, containers do not need to boot another copy of the operating system. In theory CRIU [1] can perform live migration of containers, but it may be faster to kill a container and start a new one.

III. Evaluation

Performance has numerous aspects. We focus on the issue of overhead compared to native, non-virtualized execution because it reduces the resources available for productive work. Thus we investigate scenarios where one or more hardware resources are fully utilized and we measure workload metrics like throughput and latency to determine the overhead of virtualization.

All of our tests were performed on an IBM System x3650 M4 server with two 2.4-3.0 GHz Intel Sandy Bridge-EP Xeon E5-2665 processors for a total of 16 cores (plus HyperThreading) and 256 GB of RAM. The two processors/sockets are connected by QPI links making this a non-uniform memory access (NUMA) system. This is a mainstream server configuration that is very similar to those used by popular cloud providers. We used Ubuntu 13.10 (Saucy) 64-bit with Linux kernel 3.11.0, Docker 1.0, QEMU 1.5.0, and libvirt 1.1.1. For consistency, all Docker containers used an Ubuntu 13.10 base image and all VMs used the Ubuntu 13.10 cloud image.

Power management was disabled for the tests by using the performance cpufreq governor. Docker containers were not restricted by cgroups so they could consume the full resources of the system under test. Likewise, VMs were configured with 32 vCPUs and adequate RAM to hold the benchmark's working set. In some tests we explore the difference between stock KVM (similar to a default OpenStack configuration) and a highly-tuned KVM configuration (similar to public clouds like EC2). We use microbenchmarks to individually measure CPU, memory, network, and storage overhead. We also measure two real server applications: Redis and MySQL.

A. CPU—PXZ

Compression is a frequently used component of cloud workloads. PXZ [11] is a parallel lossless data compression utility that uses the LZMA algorithm. We use PXZ 4.999.9beta (build 20130528) to compress enwik9 [29], a 1 GB Wikipedia dump that is often used for compression benchmarking. To focus on compression rather than I/O we use 32 threads, the input file is cached in RAM, and the output is piped to /dev/null. We use compression level 2.

Table I shows the throughput of PXZ under different configurations. As expected, native and Docker performance are very similar while KVM is 22% slower. We note that

TABLE I. RESULTS FOR PXZ, LINPACK, STREAM, AND RANDOMACCESS. EACH DATA POINT IS THE ARITHMETIC MEAN OF TEN RUNS. DEPARTURE FROM NATIVE EXECUTION IS SHOW WITHIN PARENTHESES "()". THE STANDARD DEVIATION IS SHOWN WITHIN SQUARE BRACKETS "[]".

Workload		Native	Docker	KVM-untuned	KVM-tuned
PXZ (MB/s)		76.2 [±0.93]	73.5 (-4%) [±0.64]	59.2 (-22%) [±1.88]	62.2 (-18%) [±1.33]
Linpack (GFLOPS)		290.8 [±1.13]	290.9 (-0%) [±0.98]	241.3 (-17%) [±1.18]	284.2 (-2%) [±1.45]
RandomAccess (GUPS)		0.0126 [±0.00029]	0.0124 (-2%) [±0.00044]	0.0125 (-1%) [±0.00032]	Tuned run not warranted
Stream (GB/s)	Add	45.8 [±0.21]	45.6 (-0%) [±0.55]	45.0 (-2%) [±0.19]	
	Copy	41.3 [±0.06]	41.2 (-0%) [±0.08]	40.1 (-3%) [±0.21]	
	Scale	41.2 [±0.08]	41.2 (-0%) [±0.06]	40.0 (-3%) [±0.15]	
	Triad	45.6 [±0.12]	45.6 (-0%) [±0.49]	45.0 (-1%) [±0.20]	

tuning KVM by vCPU pinning and exposing cache topology makes little difference to the performance. While further experimentation is required to pinpoint the source of KVM overhead, we suspect it is caused by the extra TLB pressure of nested paging. PXZ may benefit from using large pages.

B. HPC—Linpack

Linpack solves a dense system of linear equations using an algorithm that carries out LU factorization with partial pivoting [21]. The vast majority of compute operations are spent in double-precision floating point multiplication of a scalar with a vector and adding the results to another vector. The benchmark is typically based on a linear algebra library that is heavily optimized for the specific machine architecture at hand. We use an optimized Linpack binary (version 11.1.2.005)[3] based on the Intel Math Kernel Library (MKL). The Intel MKL is highly adaptive and optimizes itself based on both the available floating point resources (e.g., what form of multimedia operations are available), as well as the cache topology of the system. By default, KVM does not expose topology information to VMs, so the guest OS believes it is running on a uniform 32-socket system with one core per socket.

Table I shows the performance of Linpack on Linux, Docker, and KVM. Performance is almost identical on both Linux and Docker–this is not surprising given how little OS involvement there is during the execution. However, untuned KVM performance is markedly worse, showing the costs of abstracting/hiding hardware details from a workload that can take advantage of it. By being unable to detect the exact nature of the system, the execution employs a more general algorithm with consequent performance penalties. Tuning KVM to pin vCPUs to their corresponding CPUs and expose the underlying cache topology increases performance nearly to par with native.

We expect such behavior to be the norm for other similarly tuned, adaptive executions, unless the system topology is faithfully carried forth into the virtualized environment.

C. Memory bandwidth—Stream

The STREAM benchmark is a simple synthetic benchmark program that measures sustainable memory bandwidth when performing simple operations on vectors [21]. Performance is dominated by the memory bandwidth of the system with the working set engineered to be significantly larger than the caches. The main determinants of performance are the bandwidth to main memory, and to a much lesser extent, the cost of handling TLB misses (which we further reduce using large pages). The memory access pattern is regular and the hardware prefetchers typically latch on to the access pattern and prefetch data before it is needed. Performance is therefore gated by memory *bandwidth* and not *latency*. The benchmark has four components: COPY, SCALE, ADD and TRIAD that are described in Table II.

TABLE II. STREAM COMPONENTS

Name	Kernel	Bytes per iteration	FLOPS per iteration
COPY	$a[i] = b[i]$	16	0
SCALE	$a[i] = q * b[i]$	16	1
ADD	$a[i] = b[i] + c[i]$	24	1
TRIAD	$a[i] = b[i] + q * c[i]$	24	2

Table I shows the performance of Stream across the three execution environments. All four components of Stream perform regular memory accesses where once a page table entry is installed in the TLB, all data within the page is accessed before moving on to the next page. Hardware TLB prefetching also works very well for this workload. As a consequence, performance on Linux, Docker, and KVM is almost identical, with the median data exhibiting a difference of only 1.4% across the three execution environments.

D. Random Memory Access—RandomAccess

The Stream benchmark stresses the memory subsystem in a *regular* manner, permitting hardware prefetchers to bring in data from memory *before* it is used in computation. In contrast, the RandomAccess benchmark [21] is specially designed to stress random memory performance. The benchmark initializes a large section of memory as its working set, that is orders of magnitude larger than the reach of the caches or the TLB. Random 8-byte words in this memory section are read, modified (through a simple XOR operation) and written back. The random locations are generated by using a linear feedback shift register requiring no memory operations. As a result, there is no dependency between successive operations permitting multiple independent operations to be in flight through the system. RandomAccess typifies the behavior of workloads with large working sets and minimal computation such as those with in-memory hash tables and in-memory databases.

As with Stream, RandomAccess uses large pages to reduce TLB miss overhead. Because of its random memory access pattern and a working set that is larger than the TLB reach, RandomAccess significantly exercises the hardware page table walker that handles TLB misses. As Table I shows, On our two-socket system, this has the same overheads for both virtualized and non-virtualized environments.

E. Network bandwidth—nuttcp

We used the nuttcp tool [7] to measure network bandwidth between the system under test and an identical machine connected using a direct 10 Gbps Ethernet link between two Mellanox ConnectX-2 EN NICs. We applied standard network tuning for 10 Gbps networking such as enabling TCP window scaling and increasing socket buffer sizes. As shown in Figure 1, Docker attaches all containers on the host to a bridge and connects the bridge to the network via NAT. In our KVM configuration we use virtio and vhost to minimize virtualization overhead.

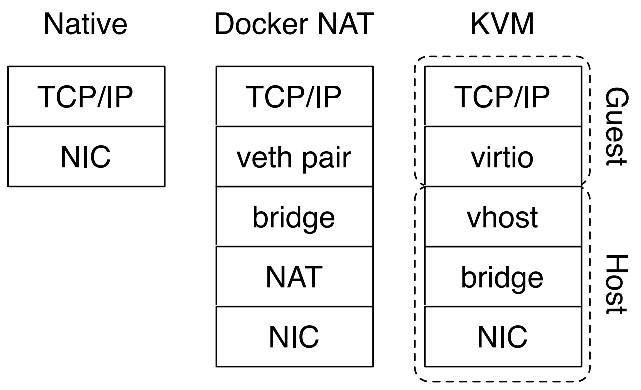

Fig. 1. Network configurations

We used nuttcp to measure the goodput of a unidirectional bulk data transfer over a single TCP connection with standard 1500-byte MTU. In the client-to-server case the system under test (SUT) acts as the transmitter and in the server-to-client case the SUT acts as the receiver; it is necessary to measure both directions since TCP has different code paths for send and receive. All three configurations reach 9.3 Gbps in both the transmit and receive direction, very close to the theoretical limit of 9.41 Gbps due to packet headers. Due to segmentation offload, bulk data transfer is very efficient even given the extra layers created by different forms of virtualization. The bottleneck in this test is the NIC, leaving other resources mostly idle. In such an I/O-bound scenario, we determine overhead by measuring the amount of CPU cycles required to transmit and receive data. Figure 2 shows system-wide CPU utilization for this test, measured using `perf stat -a`. Docker's use of bridging and NAT noticeably increases the transmit path length; vhost-net is fairly efficient at transmitting but has high overhead on the receive side. Containers that do not use NAT have identical performance to native Linux. In real network-intensive workloads, we expect such CPU overhead to reduce overall performance.

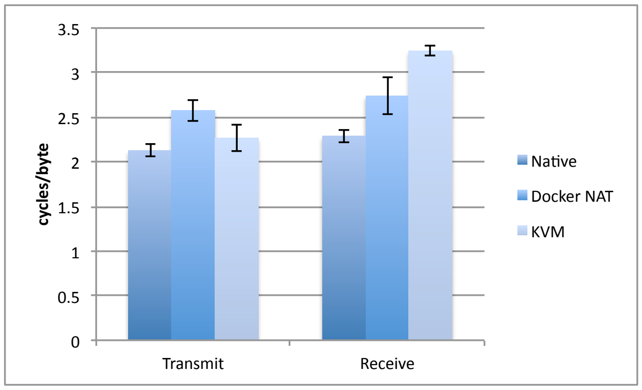

Fig. 2. TCP bulk transfer efficiency (CPU cycles/byte)

Historically, Xen and KVM have struggled to provide line-rate networking due to circuitous I/O paths that sent every packet through userspace. This has led to considerable research on complex network acceleration technologies like polling drivers or hypervisor bypass. Our results show that vhost, which allows the VM to communicate directly with the host kernel, solves the network throughput problem in a straightforward way. With more NICs, we expect this server could drive over 40 Gbps of network traffic without using any exotic techniques.

F. Network latency—netperf

We used the netperf request-response benchmark to measure round-trip network latency using similar configurations as the nuttcp tests in the previous section. In this case the system under test was running the netperf server (netserver) and the other machine ran the netperf client. The client sends a 100-byte request, the server sends a 200-byte response, and the client waits for the response before sending another request. Thus only one transaction is in flight at a time.

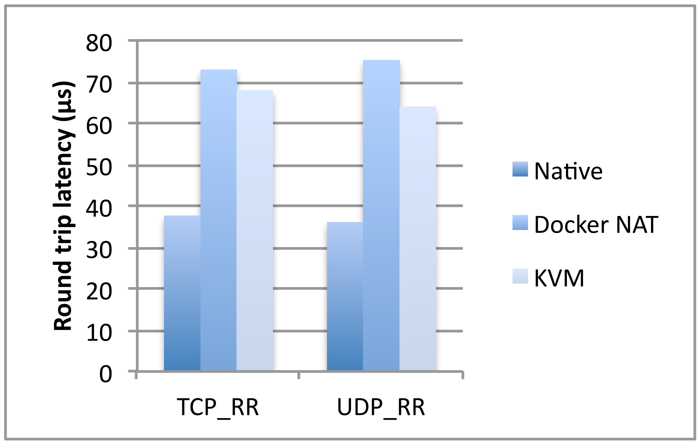

Fig. 3. Network round-trip latency (μs).

Figure 3 shows the measured transaction latency for both TCP and UDP variants of the benchmark. NAT, as used in Docker, doubles latency in this test. KVM adds 30μs of overhead to each transaction compared to the non-virtualized network stack, an increase of 80%. TCP and UDP have very similar latency because in both cases a transaction consists of a single packet in each direction. Unlike as in the throughput test, virtualization overhead cannot be amortized in this case.

G. Block I/O—`fio`

SAN-like block storage is commonly used in the cloud to provide high performance and strong consistency. To test the overhead of virtualizing block storage, we attached a 20 TB IBM FlashSystem 840 flash SSD to our test server using two 8 Gbps Fibre Channel links to a QLogic ISP2532-based dual-port HBA with `dm_multipath` used to combine the two links. We created an `ext4` filesystem on it using default settings. In the native case the filesystem was mounted normally while in the Docker test it was mapped into the container using the `-v` option (avoiding AUFS overhead). In the VM case the block device was mapped into the VM using `virtio` and mounted inside the VM. These configurations are depicted in Figure 4. We used `fio` [13] 2.0.8 with the `libaio` backend in O_DIRECT mode to run several tests against a 16 GB file stored on the SSD. USing O_DIRECT allows accesses to bypass the OS caches.

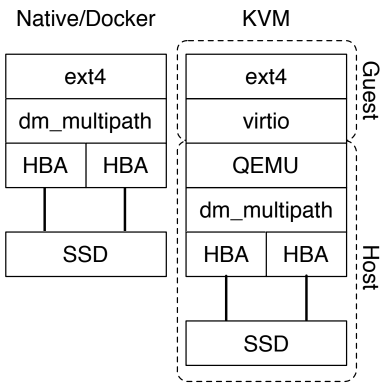

Fig. 4. Storage configurations used for `fio`

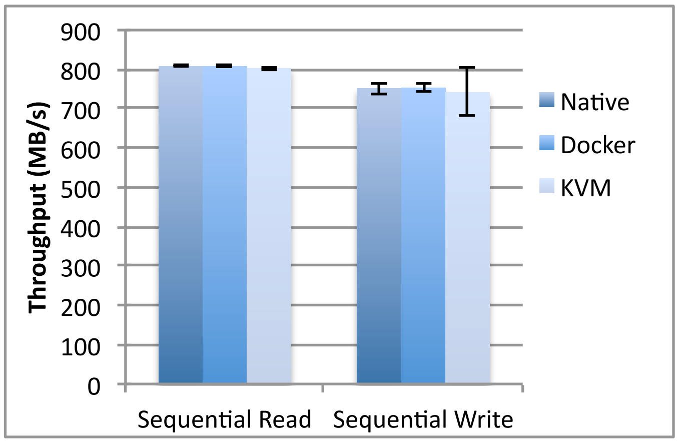

Fig. 5. Sequential I/O throughput (MB/s).

Figure 5 shows sequential read and write performance averaged over 60 seconds using a typical 1 MB I/O size. Docker and KVM introduce negligible overhead in this case, although KVM has roughly four times the performance variance as the other cases. Similar to the network, the Fibre Channel HBA appears to be the bottleneck in this test.

Figure 6 shows the performance of random read, write and mixed (70% read, 30% write) workloads using a 4 kB block size and concurrency of 128, which we experimentally determined provides maximum performance for this particular SSD. As we would expect, Docker introduces no overhead

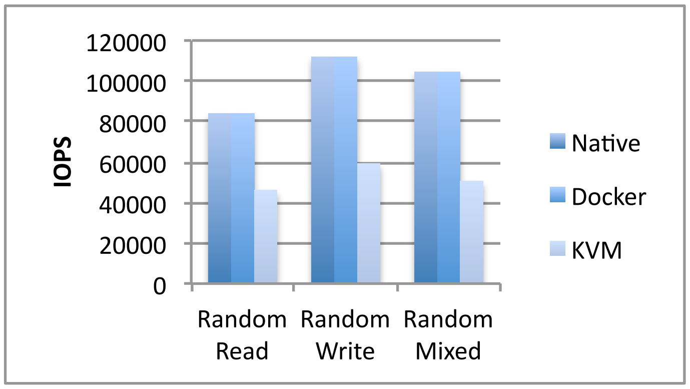

Fig. 6. Random I/O throughput (IOPS).

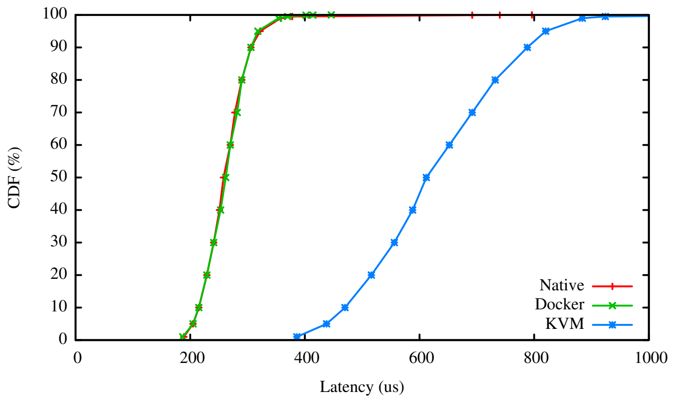

Fig. 7. Random read latency CDF, concurrency 16 (μs). The Native and Docker lines are almost superimposed atop one another.

compared to Linux, but KVM delivers only half as many IOPS because each I/O operation must go through QEMU. While the VM's absolute performance is still quite high, it uses more CPU cycles per I/O operation, leaving less CPU available for application work. Figure 7 shows that KVM increases read latency by 2-3x, a crucial metric for some real workloads.

We also note that this hardware configuration ought to be able to exceed 1.5 GB/s in sequential I/O and 350,000 IOPS in random I/O, so even the native case has significant unrealized potential that we did not manage to exploit while the hardware was on loan to us.

H. *Redis*

Memory-based key-value storage is commonly used in the cloud for caching, storing session information, and as a convenient way to maintain hot unstructured data sets. Operations tend to be simple in nature, and require a network round-trip between the client and the server. This usage model makes the application generally sensitive to network latency. The challenge is compounded given the large number of concurrent clients, each sending very small network packets to a number of servers. As a result, the server spends a sizable amount of time in the networking stack.

There are several such servers, for our evaluations, we selected Redis [43] due to its high performance, rich API and widespread use among PaaS providers (e.g., Amazon Elasticache, Google Compute Engine). We obtained Redis 2.8.13 from the main GitHub repository, and built it in our Ubuntu 13.10 platform. The resulting binaries are then used in each of the deployment modes: native, Docker and KVM. To improve performance, and given that Redis is a single-threaded application, we affinitize the container or VM to a core close to the network interface. The test consists of a number of clients issuing requests to the server. A mix of 50% read and 50% writes was used. Each client maintains a persistent TCP connection to the server and can pipeline up to 10 concurrent requests over that connection. Thus the total number of requests in flight is 10 times the number of clients. Keys were 10 characters long, and values were generated to average 50 bytes. This dataset shape is representative of production Redis users as described by Steinberg et. al. [47]. For each run, the dataset is cleared and then a deterministic sequence of operations is issued, resulting in the gradual creation of 150 million keys. Memory consumption of the Redis server peaks at 11 GB during execution.

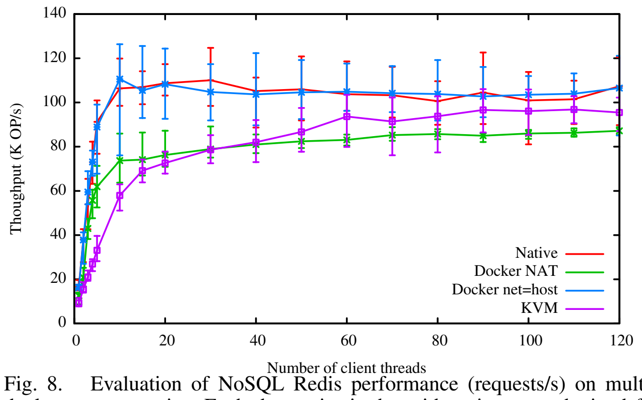

Fig. 8. Evaluation of NoSQL Redis performance (requests/s) on multiple deployment scenarios. Each data point is the arithmetic mean obtained from 10 runs.

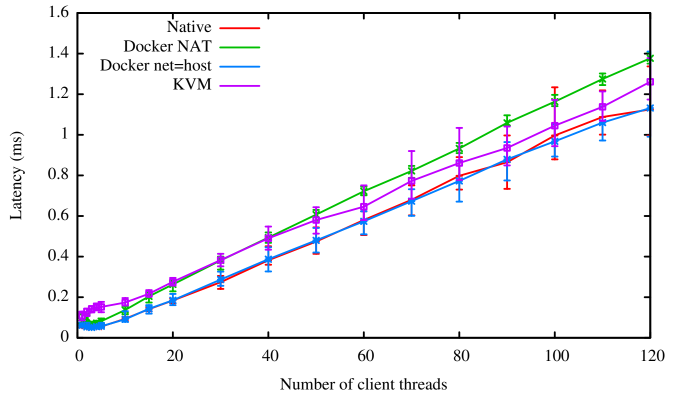

Fig. 9. Average latency (in ms) of operations on different Redis deployments. Each data point is the arithmetic mean obtained from 10 runs.

Figure 8 shows the throughput (in requests per second) with respect to the number of client connections for the different deployment models. Figure 9 shows the corresponding average latencies (in μs) for each of the experiments. In the native deployment, the networking subsystem is quite sufficient to handle the load. So, as we scale the number of client connections, the main factor that limits the throughput of a Redis server is saturation of the CPU – note that Redis is a single-threaded, event-based application. In our platform, that occurs quickly at around 110 k request per second. Adding more clients results in requests being queued and an increase in the average latency.

A Redis server puts a lot of pressure in the networking and memory subsystems. When using Docker with the host networking stack, we can see that both throughput and latency are virtually the same as the native case. The story is quite different when using Docker with NAT enabled as shown in Figure 1. In this case the latency introduced grows with the number of packets received over the network. Whereas it is comparable to native with 4 concurrent connections ($51\mu s$ or 1.05x that of native), it quickly grows once the number of connections increases (to over 1.11x with 100 connections). Additionally, NAT consumes CPU cycles, thus preventing the Redis deployment from reaching the peak performance seen by deployments with native networking stacks.

Similarly, when running in KVM, Redis appears to be network-bound. KVM adds approximately $83\mu s$ of latency to every transaction. We see that the VM has lower throughput

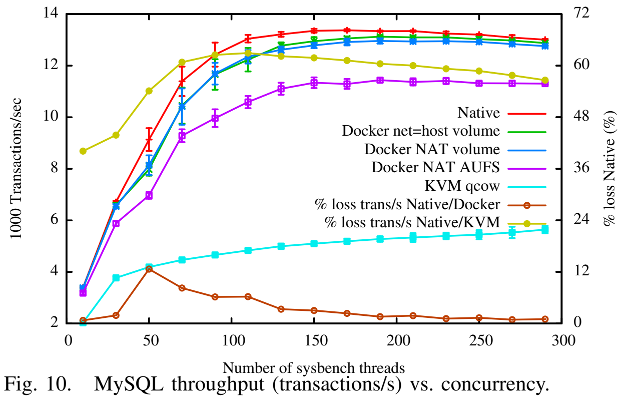

Fig. 10. MySQL throughput (transactions/s) vs. concurrency.

at low concurrency but asymptotically approaches native performance as concurrency increases. Beyond 100 connections, the throughput of both deployments are practically identical. This is can be explained by Little's Law; because network latency is higher under KVM, Redis needs more concurrency to fully utilize the system. This might be a problem depending on the concurrency level expected from the end user in a cloud scenario.

I. *MySQL*

MySQL is a popular relational database that is widely used in the cloud and typically stresses memory, IPC, filesystem, and networking subsystems. We ran the SysBench [6] oltp benchmark against a single instance of MySQL 5.5.37. MySQL was configured to use InnoDB as the backend store and a 3GB cache was enabled; this cache is sufficient to cache all reads during the benchmark runs. The oltp benchmark uses a database preloaded with 2 million records and executes a fixed set of read/write transactions choosing between five SELECT queries, two UPDATE queries, a DELETE query and an INSERT. The measurements provided by SysBench are statistics of transaction latency and throughput in transactions per seconds. The number of clients was varied until saturation and ten runs were used to produce each data point. Five different configurations were measured: MySQL running normally on Linux (native), MySQL under Docker using host networking and a volume (Docker net=host volume), using a volume but normal Docker networking (Docker NAT volume), storing the database within the container filesystem (Docker NAT AUFS) and MySQL running under KVM; these different configurations are summarized in Table III. Although MySQL accesses the fileystem, our configuration has enough cache that it performed almost no actual disk I/O, making different KVM storage options moot.

TABLE III. MYSQL CONFIGURATIONS

Configuration	Network (Figure 1)	Storage
Native	Native	Native
Docker net=host Volume	Native	Native
Docker NAT Volume	NAT	Native
Docker NAT AUFS	NAT	AUFS
KVM	vhost-net	virtio + qcow

Figure 10 shows transaction throughput as a function of the number of users simulated by SysBench. The right Y axis shows the loss in throughput when compared to native.

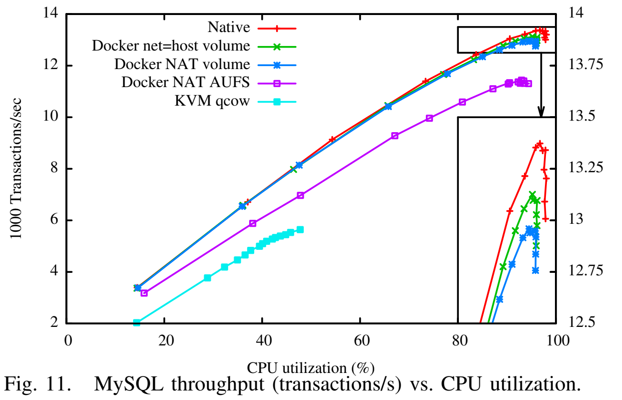

Fig. 11. MySQL throughput (transactions/s) vs. CPU utilization.

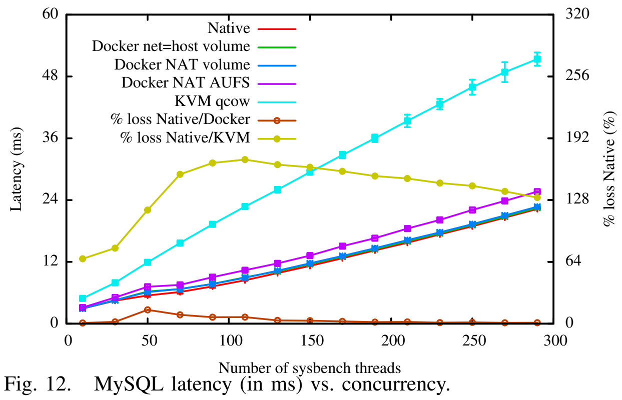

Fig. 12. MySQL latency (in ms) vs. concurrency.

The general shape of this curve is what we would expect: throughput increases with load until the machine is saturated, then levels off with a little loss due to contention when overloaded. Docker has similar performance to native, with the difference asymptotically approaching 2% at higher concurrency. KVM has much higher overhead, higher than 40% in all measured cases. AUFS introduces significant overhead which is not surprising since I/O is going through several layers, as seen by comparing the *Docker NAT volume* and *Docker NAT AUFS* results. The AUFS overhead demonstrates the difference between virtualizing above the filesystem, as Docker does, and virtualizing below the filesystem in the block layer, as KVM does. We tested different KVM storage protocols and found that they make no difference in performance for an in-cache workload like this test. NAT also introduces a little overhead but this workload is not network-intensive. KVM shows a interesting result where saturation was achieved in

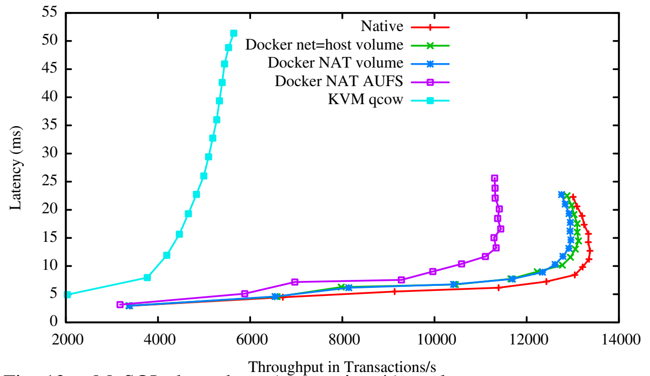

Fig. 13. MySQL throughput (transactions/s) vs. latency.

the network but not in the CPUs (Figure 11). The benchmark generates a lot of small packets, so even though the network bandwidth is small, the network stack is not able to sustain the number of packets per second needed. Since the benchmark uses synchronous requests, a increase in latency also reduces throughput at a given concurrency level.

Figure 12 shows latency as a function of the number of users simulated by SysBench. As expected the latency increases with load, but interestingly Docker increases the latency faster for moderate levels of load which explains the lower throughput at low concurrency levels. The expanded portion of the graph shows that native Linux is able to achieve higher peak CPU utilization and Docker is not able to achieve that same level, a difference of around 1.5%. This result shows that Docker has a small but measurable impact.

Figure 11 plots throughput against CPU utilization. Comparing Figures 10 through Figure 12, we note that the lower throughput for the same concurrency for Docker and Docker with NAT does not create an equivalent increase in CPU consumption. The difference in throughput is minimal when the same amount of CPU is used. The latency otherwise is not the same with Docker being substantially higher for lower values of concurrency, we credit this behavior to mutex contention. Mutex contention also prevents MySQL from fully utilizing the CPU in all cases, but it is more pronounced in the Docker case since the transactions take longer. Figure 11 shows clearly that in the case of VM the limitation is not CPU but network but the overhead of KVM is apparent even at lower numbers of clients.

The throughput-latency curves in Figure 13 make it easy to compare the alternatives given a target latency or throughput. One interesting aspect of this curve is the throughput reduction caused by higher context switching in several cases when more clients are introduced after saturation. Since there is more idle time in the Docker case a higher overhead does not impact throughput for the number of clients used in the benchmark.

J. Discussion

We see several general trends in these results. As we expect given their implmentations, containers and VMs impose almost no overhead on CPU and memory usage; they only impact I/O and OS interaction. This overhead comes in the form of extra cycles for each I/O operation, so small I/Os suffer much more than large ones. This overhead increases I/O latency and reduces the CPU cycles available for useful work, limiting throughput. Unfortunately, real applications often cannot batch work into large I/Os.

Docker adds several features such as layered images and NAT that make it easier to use than LXC-style raw containers, but these features come at a performance cost. Thus Docker using default settings may be no faster than KVM. Applications that are filesystem or disk intensive should bypass AUFS by using volumes. NAT overhead can be easily eliminated by using –net=host, but this gives up the benefits of network namespaces. Ultimately we believe that the model of one IP address per container as proposed by the Kubernetes project can provide flexibility and performance.

While KVM can provide very good performance, its configurability is a weakness. Good CPU performance requires careful configuration of large pages, CPU model, vCPU pinning, and cache topology; these features are poorly documented and required trial and error to configure. We advise readers to avoid using qemu-kvm directly and instead use libvirt since it simplifies KVM configuration. Even on the latest version of Ubuntu we were unable to get vhost-scsi to work, so there is still room for improvement. This complexity represents a barrier to entry for any aspiring cloud operator, whether public or private.

IV. Related Work

The Multics [32] project set out to build utility computing infrastructure in the 1960s. Although Multics never saw widespread use and cloud computing couldn't take off until the Internet became widespread, the project produced ideas like the end-to-end argument [41] and a set of security principles [42] that are still relevant today.

Virtual machines were introduced on IBM mainframes in the 1970s [20] and then reinvented on x86 by VMware [38] in the late 1990s. Xen [15] and KVM [25] brought VMs to the open source world in the 2000s. The overhead of virtual machines was initially high but has been steadily reduced over the years due to hardware and software optimizations [25, 31].

Operating system level virtualization also has a long history. In some sense the purpose of an operating system is to virtualize hardware resources so they may be shared, but Unix traditionally provides poor isolation due to global namespaces for the filesystem, processes, and the network. Capability-based OSes [22] provided container-like isolation by virtue of not having any global namespaces to begin with, but they died out commercially in the 1980s. Plan 9 introduced per-process filesystem namespaces [34] and bind mounts that inspired the namespace mechanism that underpins Linux containers.

The Unix chroot() feature has long been used to implement rudimentary "jails" and the BSD jails feature extends the concept. Solaris 10 introduced and heavily promoted Zones [37], a modern implementation of containers, in 2004. Cloud provider Joyent has offered IaaS based on Zones since 2008, although with no effect on the overall cloud market.

Linux containers have a long and winding history. The Linux-VServer [46] project was an initial implementation of "virtual private servers" in 2001 that was never merged into mainstream Linux but was used successfully in PlanetLab. The commercial product Virtuozzo and its open-source version OpenVZ [4] have been used extensively for Web hosting but were also not merged into Linux. Linux finally added native containerization starting in 2007 in the form of kernel namespaces and the LXC userspace tool to manage them.

Platform as a service providers like Heroku introduced the idea of using containers to efficiently and repeatably deploy applications [28]. Rather than viewing a container as a virtual server, Heroku treats it more like a process with extra isolation. The resulting application containers have very little overhead, giving similar isolation as VMs but with resource sharing like normal processes. Google also pervasively adopted application containers in their internal infrastructure [19]. Heroku competitor DotCloud (now known as Docker Inc.) introduced Docker [45] as a standard image format and management system for these application containers.

There has been extensive performance evaluation of hypervisors, but mostly compared to other hypervisors or non-virtualized execution. [23, 24, 31]

Past comparison of VMs vs. containers [30, 33, 44, 46, 50] mostly used older software like Xen and out-of-tree container patches.

V. Conclusions and Future Work

Both VMs and containers are mature technology that have benefited from a decade of incremental hardware and software optimizations. In general, Docker equals or exceeds KVM performance in every case we tested. Our results show that both KVM and Docker introduce negligible overhead for CPU and memory performance (except in extreme cases). For I/O-intensive workloads, both forms of virtualization should be used carefully.

We find that KVM performance has improved considerably since its creation. Workloads that used to be considered very challenging, like line-rate 10 Gbps networking, are now possible using only a single core using 2013-era hardware and software. Even using the fastest available forms of paravirtualization, KVM still adds some overhead to every I/O operation; this overhead ranges from significant when performing small I/Os to negligible when it is amortized over large I/Os. Thus, KVM is less suitable for workloads that are latency-sensitive or have high I/O rates. These overheads significantly impact the server applications we tested.

Although containers themselves have almost no overhead, Docker is not without performance gotchas. Docker volumes have noticeably better performance than files stored in AUFS. Docker's NAT also introduces overhead for workloads with high packet rates. These features represent a tradeoff between ease of management and performance and should be considered on a case-by-case basis.

In some sense the comparison can only get worse for containers because they started with near-zero overhead and VMs have gotten faster over time. If containers are to be widely adopted they must provide advantages other than steady-state performance. We believe the combination of convenience, faster deployment, elasticity, and performance is likely to become compelling in the near future.

Our results can give some guidance about how cloud infrastructure should be built. Conventional wisdom (to the extent such a thing exists in the young cloud ecosystem) says that IaaS is implemented using VMs and PaaS is implemented using containers. We see no technical reason why this must be the case, especially in cases where container-based IaaS can offer better performance or easier deployment. Containers can also eliminate the distinction between IaaS and "bare metal" non-virtualized servers [10, 26] since they offer the control and isolation of VMs with the performance of bare metal. Rather than maintaining different images for virtualized and non-virtualized servers, the same Docker image could be efficiently deployed on anything from a fraction of a core to an entire machine.

We also question the practice of deploying containers inside VMs, since this imposes the performance overheads of VMs while giving no benefit compared to deploying containers directly on non-virtualized Linux. If one must use a VM, running it inside a container can create an extra layer of security since an attacker who can exploit QEMU would still be inside the container.

Although today's typical servers are NUMA, we believe that attempting to exploit NUMA in the cloud may be more effort than it is worth. Limiting each workload to a single socket greatly simplifies performance analysis and tuning. Given that cloud applications are generally designed to scale out and the number of cores per socket increases over time, the unit of scaling should probably be the socket rather than the server. This is also a case against bare metal, since a server running one container per socket may actually be faster than spreading the workload across sockets due to the reduced cross-traffic.

In this paper we created single VMs or containers that consumed a whole server; in the cloud it is more common to divide servers into smaller units. This leads to several additional topics worthy of investigation: performance isolation when multiple workloads run on the same server, live resizing of containers and VMs, tradeoffs between scale-up and scale-out, and tradeoffs between live migration and restarting.

Source code

The scripts to run the experiments from this paper are available at https://github.com/thewmf/kvm-docker-comparison.

Acknowledgements

This work was supported in part by the Office of Science, United States Department of Energy under award number DE-SC0007103.

We thank our manager Karthick Rajamani for his comments on drafts of this paper.

Lawyers made us include this

References

[1] Checkpoint/Restore In Userspace. http://criu.org/.
[2] Cloud Foundry Warden documentation. http://docs.cloudfoundry.org/concepts/architecture/warden.html.
[3] Intel Math Kernel Library—LINPACK Download. https://software.intel.com/en-us/articles/intel-math-kernel-library-linpack-download.
[4] OpenVZ. http://openvz.org/.
[5] Secure Computing Mode support: “seccomp”. http://git.kernel.org/cgit/linux/kernel/git/tglx/history.git/commit/?id=d949d0ec9c601f2b148bed3cdb5f87c052968554.
[6] Sysbench benchmark. https://launchpad.net/sysbench.
[7] The “nuttcp” Network Performance Measurement Tool. http://www.nuttcp.net/.
[8] Virtualization Overview. http://www.vmware.com/pdf/virtualization.pdf.
[9] Xen Project Software Overview. http://wiki.xen.org/wiki/Xen_Overview.
[10] SoftLayer introduces bare metal cloud. http://www.softlayer.com/press/release/96/softlayer-introduces-bare-metal-cloud, Oct 2009.
[11] PXZ—parallel LZMA compressor using liblzma. https://jnovy.fedorapeople.org/pxz/, 2012.
[12] Advanced multi layered unification filesystem. http://aufs.sourceforge.net, 2014.
[13] Jens Axboe. Flexible IO Tester. http://git.kernel.dk/?p=fio.git;a=summary.
[14] Ari Balogh. Google Compute Engine is now generally available with expanded OS support, transparent maintenance, and lower prices. http://googledevelopers.blogspot.com/2013/12/google-compute-engine-is-now-generally.html, Dec 2013.
[15] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP ’03, pages 164–177, New York, NY, USA, 2003.
[16] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC ’05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.
[17] E. W. Biederman. Multiple instances of the global Linux namespaces. In *Proceedings of the 2006 Ottawa Linux Symposium*, 2006.
[18] Matthias Bolte, Michael Sievers, Georg Birkenheuer, Oliver Niehörster, and André Brinkmann. Non-intrusive virtualization management using libvirt. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE ’10, pages 574–579. European Design and Automation Association, 2010.
[19] Eric Brewer. Robust containers. http://www.slideshare.net/dotCloud/eric-brewer-dockercon-keynote, June 2014.
[20] R. J. Creasy. The origin of the VM/370 time-sharing system. *IBM Journal of Research and Development*, 25(5):483–490, Sep 1981.
[21] J. Dongarra and P. Luszczek. Introduction to the HPCChallenge Benchmark Suite. Technical report, ICL Technical Report, 10 2005. ICL-UT-05-01.
[22] Norman Hardy. KeyKOS architecture. *SIGOPS Oper. Syst. Rev.*, 19(4):8–25, October 1985.
[23] Nikolaus Huber, Marcel von Quast, Michael Hauck, and Samuel Kounev. Evaluating and modeling virtualization performance overhead for cloud environments. In *CLOSER*, pages 563–573, 2011.
[24] Jinho Hwang, Sai Zeng, F.Y. Wu, and T. Wood. A component-based performance comparison of four hypervisors. In *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*, pages 269–276, May 2013.
[25] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, Ottawa, Ontario, Canada, June 2007.
[26] Ev Kontsevoy. OnMetal: The right way to scale. http://www.rackspace.com/blog/onmetal-the-right-way-to-scale/, June 2014.
[27] Fabio Kung. Memory inside Linux containers. http://fabiokung.com/2014/03/13/memory-inside-linux-containers/, March 2014.
[28] James Lindenbaum. Deployment that just works. https://blog.heroku.com/archives/2009/3/3/deployment_that_just_works, Mar 2009.
[29] Matt Mahoney. Large text compression benchmark. http://mattmahoney.net/dc/textdata.html, 2011.
[30] Jeanna Neefe Matthews, Wenjin Hu, Madhujith Hapuarachchi, Todd Deshane, Demetrios Dimatos, Gary Hamilton, Michael McCabe, and James Owens. Quantifying the performance isolation properties of virtualization systems. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, ExpCS ’07, 2007.
[31] Richard McDougall and Jennifer Anderson. Virtualization performance: Perspectives and challenges ahead. *SIGOPS Oper. Syst. Rev.*, 44(4):40–56, December 2010.
[32] Elliott I. Organick. *The Multics system: an examination of its structure*. MIT Press, 1972.
[33] Pradeep Padala, Xiaoyun Zhu, Zhikui Wang, Sharad Singhal, Kang G Shin, et al. Performance evaluation of virtualization technologies for server consolidation. *HP Labs Technical Report*, 2007.
[34] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The Use of Name Spaces in Plan 9. In *Proceedings of the 5th Workshop on ACM SIGOPS European Workshop: Models and Paradigms for Distributed Systems Structuring*, pages 1–5, 1992.
[35] Lennart Poettering, Kay Sievers, and Thorsten Leemhuis. Control centre: The systemd Linux init system. http://www.h-online.com/open/features/Control-Centre-The-systemd-Linux-init-system-1565543.html, May 2012.
[36] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974.
[37] Daniel Price and Andrew Tucker. Solaris Zones: Operating system support for consolidating commercial workloads. In *LISA*, volume 4, pages 241–254, 2004.
[38] M. Rosenblum and T. Garfinkel. Virtual machine monitors: current technology and future trends. *Computer*, 38(5):39–47, May 2005.
[39] Boden Russell. KVM and Docker LXC Benchmarking with OpenStack. http://bodenr.blogspot.com/2014/05/kvm-and-docker-lxc-benchmarking-with.html, May 2014.
[40] Rusty Russell. Virtio: Towards a de-facto standard for virtual I/O devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, July 2008.
[41] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Computer Systems*, 2(4):277–288, November 1984.

[42] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE*, volume 63, Sep 1975.

[43] Salvatore Sanfilippo et al. The Redis Key-Value Data Store. http://redis.io/.

[44] Ryan Shea and Jiangchuan Liu. Understanding the impact of denial of service attacks on virtual machines. In *Proceedings of the 2012 IEEE 20th International Workshop on Quality of Service*, IWQoS '12, pages 27:1–27:9, 2012.

[45] Solomon Hykes and others. What is Docker? https://www.docker.com/whatisdocker/.

[46] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 275–287, 2007.

[47] Yoav Steinberg. I have 500 million keys but what's in my Redis DB? http://redislabs.com/blog/i-have-500-million-keys-but-whats-in-my-redis-db, Jun 2014.

[48] Stphane Graber and others. LXC—Linux containers. https://linuxcontainers.org/.

[49] Victor Marmol and others. Let me contain that for you: README. https://github.com/google/lmctfy/blob/master/README.md.

[50] M.G. Xavier, M.V. Neves, F.D. Rossi, T.C. Ferreto, T. Lange, and C.AF. De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 233–240, Feb 2013.