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

Research Division
Almaden – Austin – Beijing – Cambridge – Dublin - Haifa – India – Melbourne - T.J. Watson – Tokyo - Zurich

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

Isolation and resource control for cloud applications has traditionally been achieve through the use of virtual machines. Deploying applications in a VM results in reduced performance due to the extra levels of abstraction. In a cloud environment, this results in loss efficiency for the infrastructure. Newer advances in container-based virtualization simplifies the deployment of applications while isolating them from one another.

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 the CPU, memory, storage and networking resources.

Our results show that containers result in equal or better performance than VM 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 architecture.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords Virtualization, Performance, Cloud Computing

1. Introduction

Isolation and resource control are two crucial requirements when co-locating different workloads in a cloud computing environment. Isolation refers to the requirement that the execution of one workload cannot affect the execution of a different workload on the same system. Resource control refers to the ability to constrain a workload to a specific set of resources.

In the cloud setting, while performance isolation is desirable, it is often secondary to functional and security isolation wherein one workload cannot learn anything or affect the correctness of another workload. Memory capacity and compute capacity are two common levers in resource control through which a workload is constrained to consume no more than a certain amount of memory capacity and execute on no more than a certain number of cores.

Isolation and resource control have traditionally been achieved through the use of virtual machines. By executing each workload inside its own virtual machine (VM) and by imposing resource constraints on the VMs themselves (both in terms of memory capacity and where the VM can execute), both isolation and resource control are achieved. However, this comes at a performance cost. There have been many studies showing how VM execution compares to native execution [31, 35] and such studies have been a motivating factor in generally improving the quality of VM technology [28, 32].

Virtual machines are used extensively in cloud computing; today the concept of infrastructure as a service (IaaS) is largely synonymous with VMs. For example, Amazon EC2 makes VMs available to customers and it also runs services like databases inside VMs. Many PaaS and SaaS providers are built on IaaS which implies that they run all their workloads in 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 a higher layer cannot remove it, so such overhead becomes a pervasive tax on cloud performance.

Container-based virtualization presents an interesting alternative to virtual machines in the cloud [47]. Although the concepts underlying containers such as namespaces are very mature [36], only recently have containers been adopted and standardized in mainstream operating systems, 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 year, Docker [46] has emerged as a standard runtime, image format, and build system for Linux containers.

This paper looks at two different ways of achieving isolation and 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 work makes the following contributions:

- We provide an up-to-date comparison of native, container, and virtual machine environments using 2013-era hardware and software across a cross-section of interesting benchmarks and workloads that are relevant to the cloud.
- We discovered a number of non-obvious practical issues that affect virtualization performance.
- We identify the primary performance impact of current virtualization options for HPC and server workloads.
- 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 2 describes Docker and KVM, providing necessary background to understanding the remainder of the paper. Section 3 describes and evaluates different workloads on the three environments. We review related work in Section 4, and finally, Section 5 concludes the paper.

2. Background

2.1 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 an separate copy of the OS either on a dedicated server or in a virtual machine. Such isolation reverses the status quo compared to a shared server so that any code, data, or configuration sharing between applications must be explicitly configured.

In the cloud, 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.

Likewise, the cloud must take security seriously because it runs multiple tenants on shared hardware.

2.2 KVM

Kernel Virtual Machine (KVM) [28] is a feature of Linux that allows Linux to act as a type 1 hypervisor [37], 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 [10] and VMware [8]. KVM supports both emulated I/O devices through QEMU [16] and paravirtual I/O devices using virtio [41]. The combination of hardware acceleration and paravirtual I/O is designed to reduce virtualization overhead to very low levels [32]. 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 OS and is rarely used in the cloud.

Because each VM is a process, all normal Linux resource management facilities like 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 notifica-

tion 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. 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.

2.3 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 [21], 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 [23]. 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 over-allocate when running in a resource-constrained container.

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 [6].

Several management tools are available for Linux containers, including LXC [49], systemd-nspawn [29], lmctfy [50], Warden [9], and Docker [46]. (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 [40] 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.

3. Evaluation

In this paper we seek to isolate and understand the overhead introduced by virtual machines (specifically KVM) and containers (specifically Docker) relative to non-virtualized Linux. 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 attempt such a comparison in this paper.

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). To measure overhead we have configured our benchmarks to saturate the resources of the system under test. 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. We use microbenchmarks to individually measure CPU, memory, network, and storage overhead. We also measure two real server applications: Redis and MySQL.

All of these 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. Where relevant, we engineer the workload/environment so transparent huge pages can be used.

3.1 CPU—Linpack

Linpack solves a dense system of linear equations using an algorithm that carries out LU factorization with partial pivoting [20]. 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. The topology of our system under test is complex, combining NUMA, shared caches, and simultaneous multithreading. By default, KVM does not expose this NUMA and cache topology information to VMs, so the guest OS believes it is running on a 32-socket system with one core per socket. This is a double-edged sword; abstracting the hardware can improve portability but it also eliminates some opportunities for optimization. [13]

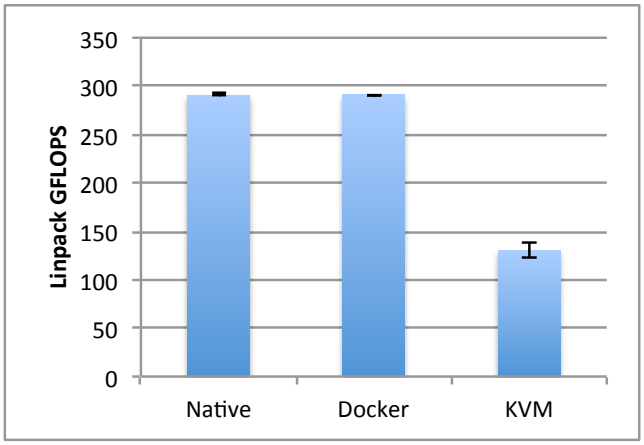

Figure 1. Linpack performance on two sockets (16 cores). Each data point is the arithmetic mean obtained from ten runs. Error bars indicate the standard deviation obtained over all runs.

Figure 1 shows the performance of Linpack on Linux, Docker, and KVM. A Linpack execution spends the bulk of its time performing mathematical floating point operations. By basing the code on an optimized linear algebra library, the execution gives rise to fairly regular memory accesses and mainly stresses the floating point capability of the core. As we point out before, the math library is highly adaptive and uses system-provided information to tune itself to the

architecture upon which it is running. Performance is almost identical on both Linux and Docker–this is not surprising given how little OS involvement there is during the execution. However, the KVM performance is markedly worse, showing the costs of abstracting/hiding system information from the execution. By being unable to detect the exact nature of the system, the execution employs a more general algorithm with consequent performance penalties.

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. CPU-bound programs that don't attempt such tuning will likely have equal but equally poor performance across native, Docker, and KVM.

3.2 Memory bandwidth—Stream

The STREAM benchmark is a simple synthetic benchmark program that measures sustainable memory bandwidth when performing simple operations on vectors [20]. 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. 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 1.

Table 1. 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

Our working set for Stream is about 36 GB; the results are not sensitive to working set size as long as it is substantially larger than the 2x20 MB L3 caches. We execute the benchmark in two configurations – the first where the computation is completely contained within one NUMA node (comprising one processor and its attached memory), and the second where the computation uses both nodes (all cores across both processors) and the memory is interleaved in a round-robin manner across both processors. All three execution environments (viz., Linux, Docker, and VM) for both configurations use transparent huge pages. The program is written to use huge pages via `mmap` and `madvise` and the results of the first iteration are discarded.

Figure 2 shows the performance of Stream across Linux, Docker, and KVM when the workload is constrained to execute on one NUMA node. 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.

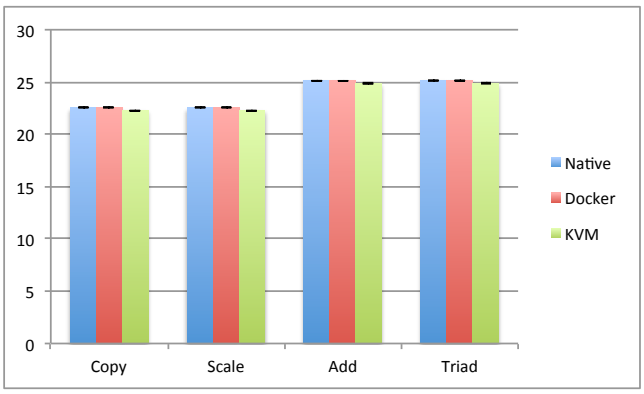

Figure 2. Stream performance in GB/s on one socket (eight cores). Each data point is the arithmetic mean obtained from ten runs. Error bars indicate the standard deviation obtained over all runs.

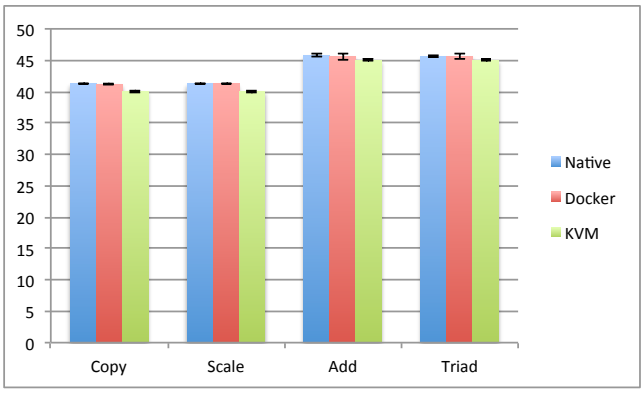

Figure 3. Stream performance (GB/s) on both sockets (all 16 cores). Each data point is the arithmetic mean obtained from ten runs. Error bars indicate the standard deviation obtained over all runs.

Figure 3 shows the performance of Stream across the three execution environments when the workload executes on both processors. We constrain memory allocation to be interleaved with allocated hardware pages being round-robined across the two memory pools. In this configuration we expect half of all memory accesses to be remote, making the QPI links a bottleneck. Performance improves, although not by a factor of two from the single-socket case. Hardware constraints such as the number of outstanding load misses and the bandwidth of the inter-processor QPI link generally cause this behavior. As with the single-socket case, perfor-

mance is again almost identical across the three execution environments.

3.3 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 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.

As with Stream, we engineer the benchmark to use `mmap` and `madvise` so that the working set is backed by huge pages. By virtue 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. This has different performance overheads depending on whether the execution is taking place in a virtualized or non-virtualized environment.

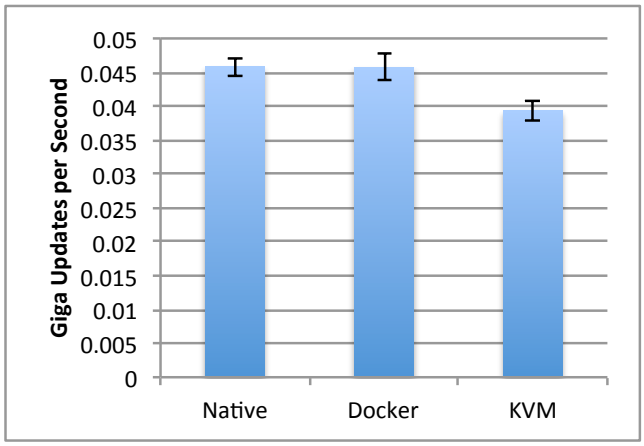

Figure 4. RandomAccess performance on a single socket (8 cores). Each data point is the arithmetic mean obtained from ten runs. Error bars indicate the standard deviation obtained over all runs.

Figure 4 shows the performance of RandomAccess on one socket. Each access typically gives rise to a miss in both the core caches and in the TLB. Since the number of outstanding misses from a core is not sufficient to hide these latencies, performance is determined by the latency of the hardware page table walks and main memory load latency. This explains why both Linux and Docker environments perform similarly. On the other hand, the KVM environment suffers from two hardware page table walks (one for the guest and one for the host system) in addition to the main memory latency. This gives rise to the lower performance with KVM.

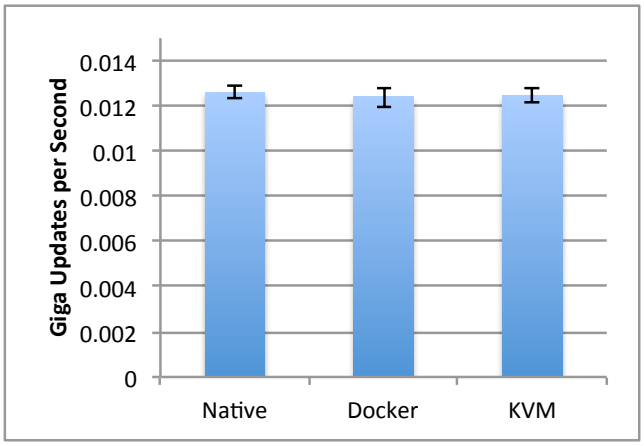

Figure 5. RandomAccess performance on both sockets (all 16 cores). Each data point is the arithmetic mean obtained from ten runs. Error bars indicate the standard deviation obtained over all runs.

Figure 5 shows the performance of RandomAccess on both sockets. On two sockets, the memory is interleaved in a round-robin manner across both processors. Two aspects of RandomAccess' performance require explanation. First, performance deteriorates when going from one socket to two sockets *even for native execution*. This is due to the nature of the workload – RandomAccess places excessive stress upon the memory subsystem wherein only eight bytes out of every cache line transferred from main memory is actually useful. For each thread participating in the execution, half the working set will be resident in memory attached to the other processor. Consequently, the inter-processor bus acts as a choke point and is the main performance constraint. For the two-socket case as well, the TLB cannot contain the working set despite using transparent huge pages. However, the performance overhead of the extra page table walk in the VM execution environment is hidden by the performance constraint of the inter-processor bus. As a result, the two-socket performance with KVM mirrors that of Docker and Linux.

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.

3.4 Network bandwidth—nuttcp

We used the nuttcp [4] tool 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. The system under test (SUT) ran the nuttcp client and the other ma-

chine ran the nuttcp server. As shown in Figure 6, 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. The server did not use any kind of virtualization in any of the tests.

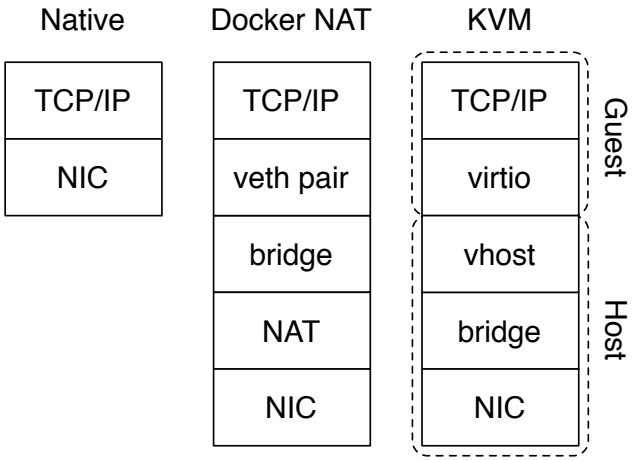

Figure 6. Network configurations

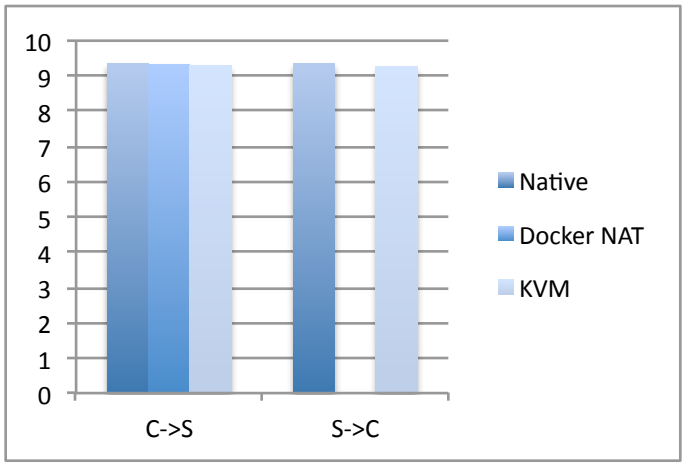

Figure 7. TCP throughput (Gb/s). *(missing bar will be added later)*

Figure 7 shows 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 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 [34]. Due to segmentation offload, bulk data transfer is very efficient even given the extra layers created by different forms of virtualization.

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.

3.5 Network latency—netperf

We used the netperf request-response (RR) 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.

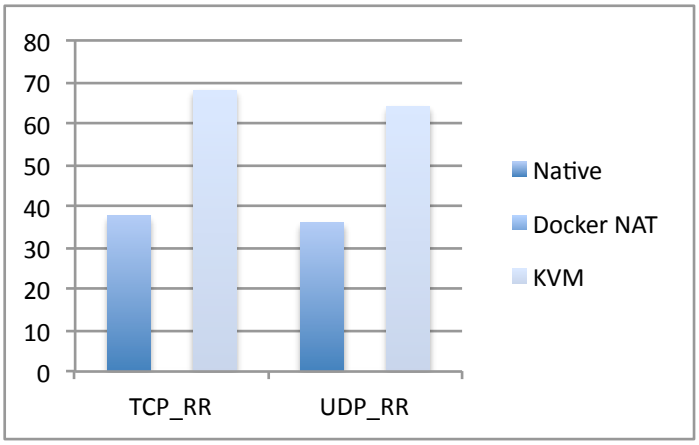

Figure 8. Network round-trip latency (μs). *(missing bars will be added later)*

Figure 8 shows the measured transaction latency for both TCP and UDP variants of the benchmark. *Docker results are not finished at submission time.* 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 in the throughput test, virtualization overhead cannot be amortized in this case.

3.6 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 [2] 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, in the Docker test it was mapped into the container using the `-v` option (avoiding AUFS overhead), and 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 9. We used `fio` [27] 2.0.8 with the `libaio` backend in O_DIRECT mode to run several tests against a 16 GB file stored on the SSD. Because O_DIRECT bypasses OS caches the size of the test file does not matter.

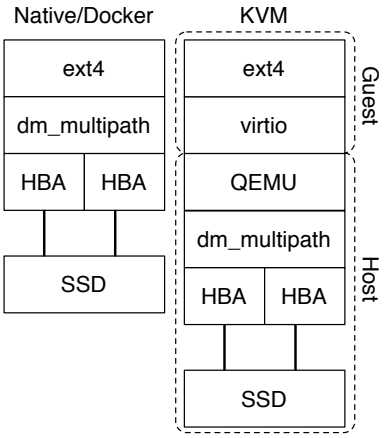

Figure 9. Storage configurations used for `fio`

Figure 10 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.

Figure 11 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 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 12 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.

3.7 Redis

Memory-based key-value storage is commonly used in the cloud for caching, session storage 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 chal-

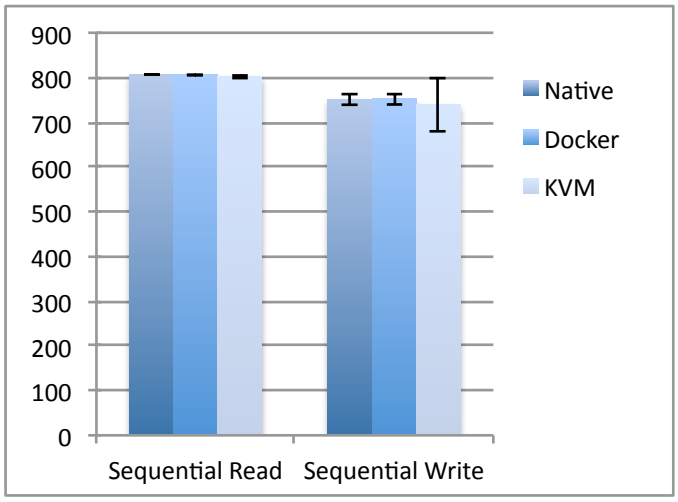

Figure 10. Sequential I/O throughput (MB/s).

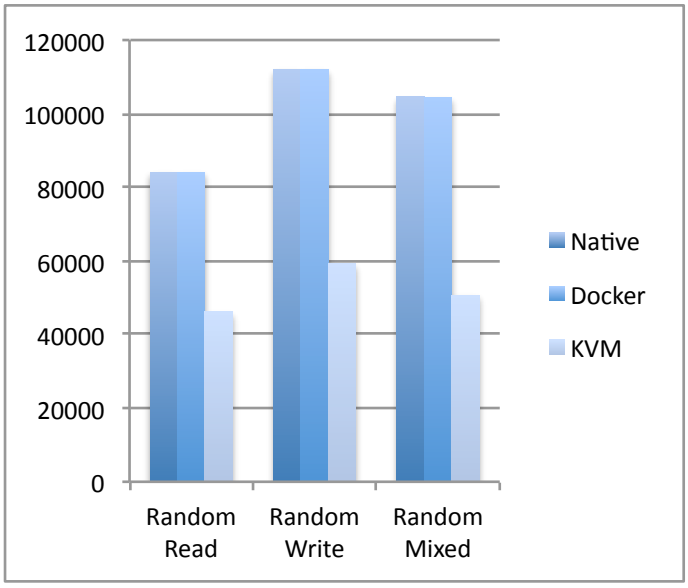

Figure 11. Random I/O throughput (IOPS).

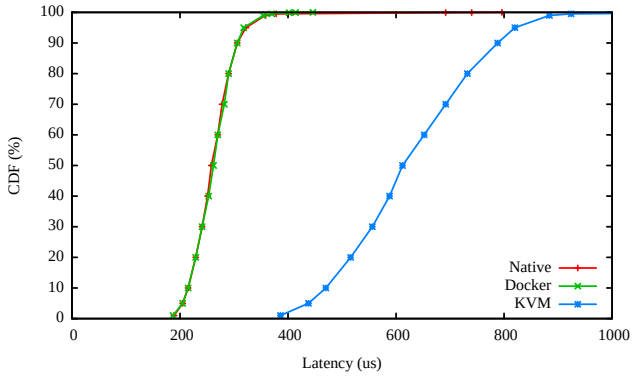

Figure 12. Random read latency CDF, concurrency 16 (μs).

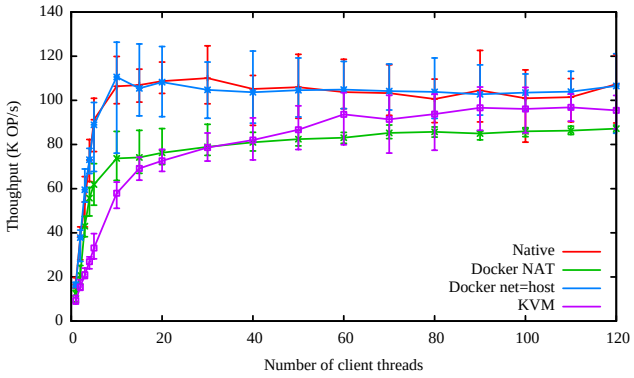

Figure 13. Evaluation of NoSQL Redis performance (requests/s) on multiple deployment scenarios. Each data point is the arithmetic mean obtained from 10 runs.

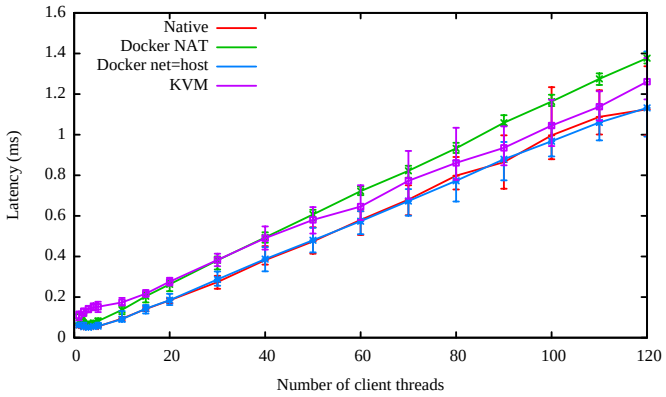

Figure 14. Average latency (in ms) of operations on different Redis deployments. Each data point is the arithmetic mean obtained from 10 runs.

lenge 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 [44] due to its high performance, rich API and widespread use among PaaS providers (e.g., Amazon Elasticache, Google App Engine). We use Redis 2.8.13 obtained directly from the main GitHub repository, which we build 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. That dataset shape is representative of production Redis users as described here [48]. 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. During the runs, the memory consumption of the Redis server peaks at 11 GB.

Figure 13 shows the throughput (in requests per second) with respect to the number of client connections for the different deployment models. Figure 14 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 – remember 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.

When using Docker with the host networking stack, 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 6. 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.

When running in KVM, Redis appears to be network-bound. KVM adds approximately $83\mu s$ of latency to every transaction due to the previously-discussed overhead of entering and exiting the VM. We see that the VM has lower throughput 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.

3.8 MySQL

We chose to evaluate MySQL because it is a polular relational database, it is widely used in the cloud, and it stresses memory, IPC, network and filesystem subsystems. We ran the SysBench [7] 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 eexecutes 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; the specific network and configuration is show on the table 2.

Table 2. MySQL configurations

Configuration	Network (Figure 6)	Storage (Figure 15)
Native	Native	Native
Docker net=host Volume	Native	Native
Docker NAT Volume	Docker NAT	Native
Docker NAT AUFS	Docker NAT	Docker
KVM	KVM vhost	KVM qcow

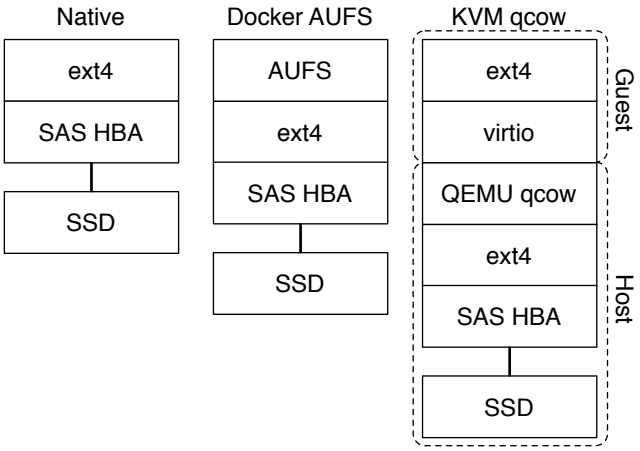

Figure 15. Storage configurations used for MySQL

Figure 16 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. 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 Docker NAT volume with Docker NAT AUFS. NAT also introduces a little overhead but this isn't a network-intensive workload for Docker. KVM shows a interesting result where saturation was achieved in

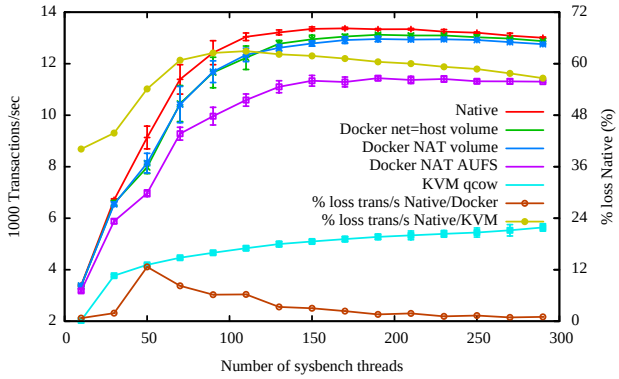

Figure 16. MySQL throughput (transactions/s) vs. concurrency.

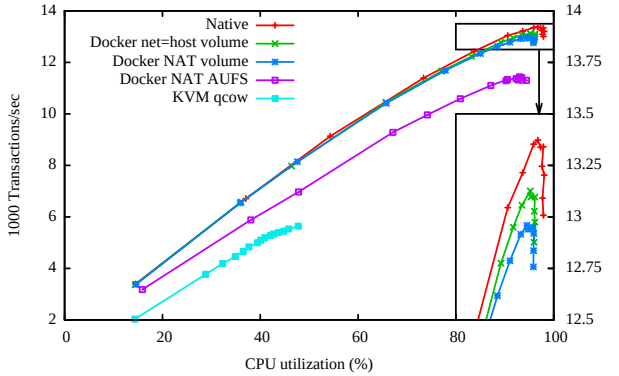

Figure 17. MySQL throughput (transactions/s) vs. CPU utilization.

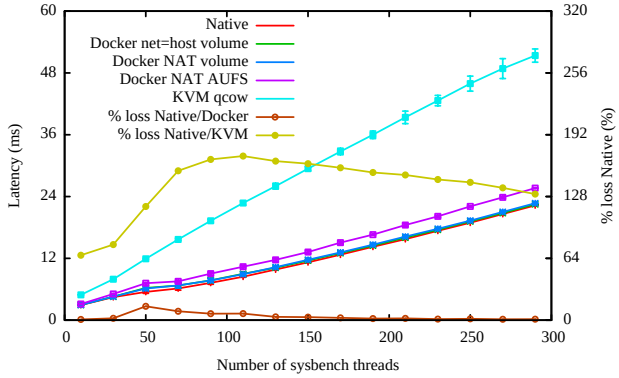

Figure 18. MySQL latency (in ms) vs. concurrency.

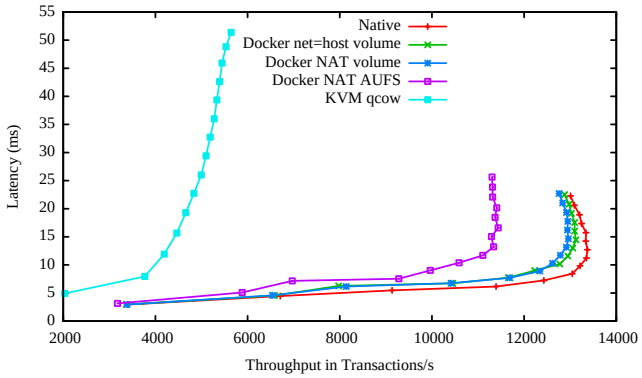

Figure 19. MySQL throughput (transactions/s) vs. latency.

the network but not in the CPUs (Figure 17). 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 18 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 and that 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 17 plots throughput and CPU utilization. Comparing Figure 16, Figure 18 and Figure 17, 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 17 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 19 make it easy to compare the alternatives given a target latency or throughput. One interesting aspect of this curve is the throughput reduction in the native case when more clients are introduced after saturation caused by higher context switching. 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.

4. Related Work

The Multics [33] 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 [43] and a set of security principles [42] that are still used today.

Virtual machines were introduced on IBM mainframes in the 1970s [19] and then reinvented on x86 by VMware [39] in the late 1990s. Xen [15] and KVM [28] 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.

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 [24] 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 [36] 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 [38], a modern implementation of containers.

Linux containers have a long and winding history. The Linux-VServer [47] 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 [5] 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 [30]. 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 [21]. Heroku competitor DotCloud (now known as Docker Inc.) introduced Docker [46] 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. [25, 26, 32]

Past comparison of VMs vs. containers [31, 35, 45, 47, 51] mostly used older software like Xen and out-of-tree container patches.

5. 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 [11, 22] 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.

Acknowledgments

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.

References

[1] Checkpoint/Restore In Userspace. http://criu.org/.

[2] Implementing IBM FlashSystem 840. http://www.redbooks.ibm.com/abstracts/sg248189.html.

[3] Intel Math Kernel Library—LINPACK Download. https://software.intel.com/en-us/articles/intel-math-kernel-library-linpack-download.

[4] The "nuttcp" Network Performance Measurement Tool. http://www.nuttcp.net/.

[5] OpenVZ. http://openvz.org/.

[6] Secure Computing Mode support: "seccomp". http://git.kernel.org/cgit/linux/kernel/git/tglx/history.git/commit/?id=d949d0ec9c601f2b148bed3cdb5f87c052968554.

[7] Sysbench benchmark. https://launchpad.net/sysbench.

[8] Virtualization Overview. http://www.vmware.com/pdf/virtualization.pdf.

[9] Cloud Foundry Warden documentation. http://docs.cloudfoundry.org/concepts/architecture/warden.html.

[10] Xen Project Software Overview. http://wiki.xen.org/wiki/Xen_Overview.

[11] SoftLayer introduces bare metal cloud. http://www.softlayer.com/press/release/96/softlayer-introduces-bare-metal-cloud, Oct 2009.

[12] Advanced multi layered unification filesystem. http://aufs.sourceforge.net, 2014.

[13] Q. Ali, V. Kiriansky, J. Simons, and P. Zaroo. Performance evaluation of HPC benchmarks on VMware's ESXi Server. In M. Alexander et al., editors, *Euro-Par 2011: Parallel Processing Workshops*, volume 7155 of *Lecture Notes in Computer Science*, pages 213–222. Springer Berlin Heidelberg, 2012. . URL http://dx.doi.org/10.1007/978-3-642-29737-3_25.

[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] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. . URL http://doi.acm.org/10.1145/945445.945462.

[16] F. 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. URL http://dl.acm.org/citation.cfm?id=1247360.1247401.

[17] E. W. Biederman. Multiple instances of the global Linux namespaces. In *Proceedings of the 2006 Ottawa Linux Symposium*, 2006. URL https://www.kernel.org/doc/ols/2006/ols2006v1-pages-101-112.pdf.

[18] M. Bolte, M. Sievers, G. Birkenheuer, O. Niehörster, and A. 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. URL http://dl.acm.org/citation.cfm?id=1870926.1871061.

[19] R. J. Creasy. The origin of the VM/370 time-sharing system. *IBM Journal of Research and Development*, 25(5):483–490, Sep 1981. ISSN 0018-8646. .

[20] J. Dongarra and P. Luszczek. Introduction to the HPCChallenge Benchmark Suite. Technical report, ICL Technical Report, 10 2005. ICL-UT-05-01.

[21] Eric Brewer. Robust containers. http://www.slideshare.net/dotCloud/eric-brewer-dockercon-keynote, June 2014.

[22] Ev Kontsevoy. OnMetal: The right way to scale. http://www.rackspace.com/blog/onmetal-the-right-way-to-scale/, June 2014.

[23] Fabio Kung. Memory inside Linux containers. http://fabiokung.com/2014/03/13/memory-inside-linux-containers/, March 2014.

[24] N. Hardy. KeyKOS architecture. *SIGOPS Oper. Syst. Rev.*, 19(4):8–25, Oct. 1985. . URL http://doi.acm.org/10.1145/858336.858337.

[25] N. Huber, M. von Quast, M. Hauck, and S. Kounev. Evaluating and modeling virtualization performance overhead for cloud environments. In *CLOSER*, pages 563–573, 2011.

[26] J. Hwang, S. Zeng, F. 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.

[27] Jens Axboe. Flexible IO Tester. http://git.kernel.dk/?p=fio.git;a=summary.

[28] 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, Ottawa, Ontario, Canada, June 2007. URL http://linux-security.cn/ebooks/ols2007/OLS2007-Proceedings-V1.pdf.

[29] Lennart Poettering and 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.

[30] J. Lindenbaum. Deployment that just works. https://blog.heroku.com/archives/2009/3/3/deployment_that_just_works, Mar 2009.

[31] 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 *Proceedings of the 2007 Workshop on Experimental Computer Science*, ExpCS '07, 2007. . URL http://doi.acm.org/10.1145/1281700.1281706.

[32] R. McDougall and J. Anderson. Virtualization performance: Perspectives and challenges ahead. *SIGOPS Oper. Syst. Rev.*, 44(4):40–56, Dec. 2010. ISSN 0163-5980. . URL http://doi.acm.org/10.1145/1899928.1899933.

[33] E. I. Organick. *The Multics system: an examination of its structure*. MIT Press, 1972.

[34] P. Dykstra. Protocol overhead. http://sd.wareonearth.com/~phil/net/overhead/, Aug 2013.

[35] P. Padala, X. Zhu, Z. Wang, S. Singhal, K. G. Shin, et al. Performance evaluation of virtualization technologies for server consolidation. *HP Labs Technical Report*, 2007. URL http://www.hpl.hp.com/techreports/2007/HPL-2007-59R1.html.

[36] R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. 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. . URL http://doi.acm.org/10.1145/506378.506413.

[37] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974. ISSN 0001-0782. . URL http://doi.acm.org/10.1145/361011.361073.

[38] D. Price and A. Tucker. Solaris Zones: Operating system support for consolidating commercial workloads. In *LISA*, volume 4, pages 241–254, 2004.

[39] M. Rosenblum and T. Garfinkel. Virtual machine monitors: current technology and future trends. *Computer*, 38(5):39–47, May 2005. ISSN 0018-9162. .

[40] B. Russell. KVM and Docker LXC Benchmarking with OpenStack. http://bodenr.blogspot.com/2014/05/kvm-and-docker-lxc-benchmarking-with.html, May 2014.

[41] R. Russell. Virtio: Towards a de-facto standard for virtual I/O devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, July 2008. ISSN 0163-5980. . URL http://doi.acm.org/10.1145/1400097.1400108.

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

[43] 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, Nov. 1984. ISSN 0734-2071. . URL http://doi.acm.org/10.1145/357401.357402.

[44] Salvatore Sanfilippo and others. The Redis Key-Value Data Store. http://redis.io/.

[45] R. Shea and J. 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. URL http://dl.acm.org/citation.cfm?id=2330748.2330775.

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

[47] 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 *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 275–287, 2007. . URL http://doi.acm.org/10.1145/1272996.1273025.

[48] Y. 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.

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

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

[51] M. Xavier, M. Neves, F. Rossi, T. Ferreto, T. Lange, and C. 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. .