# Virtualization Techniques Compared: Performance, Resource, and Power Usage Overheads in Clouds

Selome Kostentinos Tesfatsion
Department of Computing Science
Umeå University, Sweden
selome@cs.umu.se

Cristian Klein
Department of Computing Science
Umeå University, Sweden
cklein@cs.umu.se

Johan Tordsson
Department of Computing Science
Umeå University, Sweden
tordsson@cs.umu.se

## ABSTRACT

Virtualization solutions based on hypervisors or containers are enabling technologies for scalable, flexible, and cost-effective resource sharing. As the fundamental limitations of each technology are yet to be understood, they need to be regularly reevaluated to better understand the trade-off provided by latest technological advances. This paper presents an in-depth quantitative analysis of virtualization overheads in these two groups of systems and their gaps relative to native environments based on a diverse set of workloads that stress CPU, memory, storage, and networking resources. KVM and XEN are used to represent hypervisor-based virtualization, and LXC and Docker for container-based platforms. The systems were evaluated with respect to several cloud resource management dimensions including performance, isolation, resource usage, energy efficiency, start-up time, and density. Our study is useful both to practitioners to understand the current state of the technology in order to make the right decision in the selection, operation and/or design of platforms and to scholars to illustrate how these technologies evolved over time.

**ACM Reference Format:**
Selome Kostentinos Tesfatsion, Cristian Klein, and Johan Tordsson. 2018. Virtualization Techniques Compared: Performance, Resource, and Power Usage Overheads in Clouds. In *ICPE '18: ACM/SPEC International Conference on Performance Engineering, April 9–13, 2018, Berlin, Germany.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3184407.3184414

## 1 INTRODUCTION

Virtualization is a fundamental technology in cloud computing. The motivations for adopting virtualization include increased flexibility, dynamic resource allocation, and improved resource utilization. Virtualization provides the ability to pack applications into fewer physical servers and thereby reduce the power consumption of both physical servers and their cooling systems. Consequently the paradigm has become attractive, leading to the emergence of different solutions over the years. These solutions can be broadly categorized into *hypervisor (H)-based* and *Operating system (OS)-based* virtualization methods.

*H-based systems* are the traditional virtualization systems supported by many cloud computing platforms. For example, Rackspace and Amazon Web Services (AWS) use the XEN Hypervisor [28], which has gained tremendous popularity because of its early open source inclusion in the Linux kernel, and is one of the most mature virtualization solutions available [29]. The Kernel-based Virtual Machine (KVM) [1], a relatively new open source H-based system, has gained momentum and popularity in recent years [37]. It has found its way into more recently established clouds such as those operated by AT&T, HP, Comcast, and Orange [29]. KVM has become a natural choice for Linux VMs because it is included in the upstream Linux kernel. It is also a de facto standard for the open source cloud management platform OpenStack [15].

Hypervisor-free OS-based virtualization systems are widely used by successful cloud providers such as Google to manage their clusters, and have attracted considerable interest because they offer new possibilities for easy provisioning and fast deployment environment. Google has stated that it launches over 2 billion containers a week across all of its data centers [5]. Several OS-based systems have been released, including Linux Container (LXC) [19], Docker [16], BSD Jails [39], and Windows Containers [7]. Docker is the most widely used, whereas LXC, which is included in most Linux distributions, is used by cluster management frameworks such as Mesos [6] and YARN [8] to achieve stronger resource isolation among applications.

Significant efforts have been made to characterize the effects of virtualization on application performance [46, 49, 50]. However, less effort has been invested in the various additional resource overheads that virtualization imposes. For example, a Virtual Machine (VM) process may use other virtualization components (e.g. Dom0 in the case of the XEN, the hypervisor) to handle requests, which in turn generate additional resource overhead and application performance penalties. Moreover, there is a lack of detailed quantitative studies comparing H-based and OS-based platforms and their gaps relative to native environments across multiple resources under a diverse set of workload types. Co-located applications can cause interference problems due to the absence of strong performance and fault isolation, which is an important but under-explored concern for public multi-tenant clouds. In addition, the energy efficiency of different virtualization methods has not been well analyzed. Finally, many earlier works have focused on specific application areas [34, 49] and overlooked opportunities for optimization exposed by individual technologies, leading to under- or over-stated results.

Consequently, a comprehensive, detailed, and up-to-date comparative analysis of H- and OS-based virtualization solutions is needed to allow data center operators make the best possible decisions relating to issues such as resource allocation, admission control, and migration, to accurately bill cloud customers and to improve the overall performance and energy efficiency of their

ACM ISBN 978-1-4503-5095-2/18/04…$15.00
https://doi.org/10.1145/3184407.3184414

infrastructures. In addition, a better understanding of the key factors that contribute to overheads can be used to guide efforts to improve existing systems. For example, efforts such as CoreOS have significantly reduced the slow (in minutes) VM booting time to as fast as few seconds (10s). Container systems are also advancing, for instance in security, e.g. the latest 1.8 version [25] of Kubernetes provides enhanced security support.

To this end, this paper presents the results of a comprehensive investigation into the performance, power, and resource usages overhead of four virtualization technologies widely used in modern data centers (LXC, Docker, KVM, and XEN) running four different workload types (CPU–intensive, memory-intensive, network-intensive, and disk-intensive) under varying intensities. We also compare the virtualization platforms with respect to performance isolation, resource over-commitment, start-up time, and density. To the best of our knowledge, this is the first such comprehensive comparative study to be reported. The empirical results presented in this work are useful for practitioners to understand the current state of the technology so as to make a proper decision in choosing the best technology for a given situation, in making relevant trade-offs, and possibly designing systems that address the limitations. It is also useful for academia to illustrate how these technologies evolved over time and to call upon further research to uncover the underlying causes for each of these platform in the areas where they under-perform. To facilitate comparison, we developed a methodology for virtualization that automates the testing process.

In particular, our contributions are:

(1) A methodology to quantify the resource and power usage overheads of virtualized systems. We present methods for automatic and synchronized monitoring of the utilization of virtual instances, the device driver domain, the virtualization engine/hypervisor, and the physical machine along with server power usages for a diverse set of workloads.
(2) A comprehensive comparison of the selected virtualization techniques in terms of performance (throughput, and latency), resource, and power usages overheads along with analysis of the impact of co-location, scheduling techniques, resource over-commitment, start-up latency and density.
(3) Evaluation results of each platform from several dimensions demonstrates that there is no single technology that outperforms in all cases. This fact provides useful insights in how to choose the best technology for a specific scenario, how to make trade-offs in optimizing systems, or on what to do differently when designing platforms. The results also reveals that part of the limitations of each platform are technological obstacles that did/can improve over time.

## 2 BACKGROUND

This section introduces the core concepts of H-based and OS-based virtualization platforms, and provides brief overviews of the four state-of-the-art platforms considered in our evaluation.

### 2.1 H-based platforms

In H-based systems, a hypervisor or a virtual machine monitor is used to emulate the underlying physical hardware by creating virtual hardware. As such, the virtualization occurs at the hardware level. The hypervisor manages the execution of virtual machines (VMs) and the underlying physical hardware. Because hypervisors isolate the VMs from the host system, the platform is OS-agnostic in the sense that multiple instances of many different OSes may share the virtualized hardware resources. Hypervisors are generally classified into two categories: Type-1 (native or bare-metal) hypervisors that operate directly on the host hardware (e.g. XEN), and Type-2 (hosted) hypervisors that operate on top of the host's operating system. However, the distinction between the two types of hypervisors is not always clear. For example, KVM has characteristics of both types [24]. In this paper, we focus on two H-based systems that are widely used in production systems: XEN and KVM.

*2.1.1 XEN.* XEN is well known for its paravirtualization (PV) implementation. In PV, the interface presented to the guest OS differs slightly from the underlying hardware and the kernel of the guest OS is modified specifically to run on the hypervisor. As guests are aware that they are running on a hypervisor, no hardware emulation is needed and overhead is reduced.

To achieve virtualization, XEN relies on special privileged VMs called Domain-0 (Dom0). The Dom0 VM provides access to the management and control interface of the hypervisor itself and manages other unprivileged VMs (DomU). Each DomU VM runs a simple device driver that communicates with Dom0 to access the real hardware devices.

*2.1.2 KVM.* KVM is an open source solution that allows VMs to run with unmodified guest OS. Guest VMs need not be aware that they are running in a virtualized environment. KVM is implemented as a loadable kernel module, reducing the hypervisor size significantly by reusing many Linux kernel facilities such as the memory manager and scheduler. From the host's perspective, every VM is implemented, scheduled, and managed as a regular Linux process. QEMU is used to provide emulation for devices such as the BIOS, PCI bus, USB bus, disk controllers and network cards. KVM can also be used with a standard paravirtualized framework, *VirtIO*, to increase I/O performance for network and block devices. Recently introduced processors include hardware-assisted virtualization features (such as Intel-VT and AMD-V) that KVM uses to reduce complexity and overhead.

### 2.2 OS-based platforms

OS-based platforms, which are also known as container-based systems, virtualize resources at the OS level. They do not achieve virtualization in the same sense as VMs, but can be used for many of the same reasons one would use VMs [34]. The OS kernel is shared among containers, with no extra OS installed in each container. The containers, which are created by encapsulating the standard OS processes and their dependencies, are collectively managed by the underlying OS kernel. More specifically, the container engine (CE) performs the same duties as the hypervisor in a traditional virtualization, managing containers and images while leveraging the underlying OS kernel for core resource management and allocation. Because containers are sandboxed environments running on the kernel, they take up fewer resources than traditional VMs, making them a light weight alternative to H-based virtualization.

OS-based platforms can further be classified as either system containers or application containers. System containers allow multiple processes to be run in a single container, as can be done in a VM. They are designed to provide a complete runtime environment but with a more lightweight design. OS-based platforms that use system containers include LXC, OpenVZ [21], and Linux VServer [17]. Application containers, on the other hand, are designed to run a single process, and are a lightweight alternative for deploying applications based on distributed microservices. Platforms that use application containers include Docker and Rocket [22]. The OS-based platforms examined in this paper are LXC and Docker.

*2.2.1 LXC.* LXC is an OS virtualization method for running multiple isolated containers on a host using a single Linux kernel. Each container has its own process and network space, which provides the illusion of a VM but without the overhead of having a separate kernel. Resource management is performed by the kernel *namespace* and *CGroup*. CGroup is used to partition and restrict resources amongst different process groups, and the kernel namespace is used to isolate resources from the host and any other containers.

*2.2.2 Docker.* Docker is an open-source platform for container deployment and management. In Docker, containers are built on top of decoupled images in a multi-layer filesystem model, usually powered by AUFS (Another Union File System). A single OS image can be used as a basis for many containers while each container can have its own overlay of modified files. The host OS launches a Docker daemon, which is responsible for managing Docker images as well as creating and monitoring containers. Docker introduces a new level of granularity in virtualization in terms of fast deployment and easy management of systems. Like LXC, Docker uses CGroups and namespaces for resource allocation and isolation.

## 3 BENCHMARKS, METRICS, AND MEASUREMENT METHOD

### 3.1 Benchmarks and performance metrics

We chose a representative set of benchmarks to address different resource consumption characteristics across applications. The benchmarks chosen for each resource type have relatively little impact on other resources, and are described below.

*3.1.1 CPU-Sysbench.* To investigate the effect of virtualization overhead on CPU resources, we used the CPU-bound benchmark from the SysBench [14] package to compute prime numbers from 1 to N (user-specified). The application was loaded with a maximum of 4,000,000 requests. The number of threads (and hence target CPU utilization) was changed during the course of the experiments to evaluate the behaviour of each platform under different workload intensities. The benchmarking tool reports the 95th percentile latency as well as the total number of requests and elapsed time, which are used to calculate the average throughput.

*3.1.2 Memory-STREAM.* Memory performance was evaluated using STREAM [2], a synthetic benchmark program that measures sustainable memory bandwidth for four vector operations (Copy, Scale, Add, and Triad). STREAM is configured to use datasets much larger than (more than 4X the size of) the available cache memory in the physical machine to ensure that only the time to access RAM is measured and not the cache access speed.

*3.1.3 Disk I/O-FIO.* Disk I/O performance was investigated using the Flexible I/O (FIO) [10] benchmark, which measures the file system's read and write performance. It spawns a number of processes that perform particular kinds of disk I/O operations specified by the user. We used sequential and random read/writes with different file sizes, using the default block size of 4 KiB. The value of the ioengine parameter was set to *libaio*, meaning that the benchmark used a Linux-native asynchronous I/O library. Direct I/O mode was used to disallow prefetching and writing behind the filesystem's buffer cache. The benchmark measured disk I/O operations per second (IOPS), bandwidth (KB/s), and the 95th percentile latency.

*3.1.4 Network I/O-netperf.* The impact of virtualization on network performance was measured using the netperf [38] benchmark. We used the TCP STREAM (TCP_STREAM) unidirectional bulk data transfer option for throughput and the TCP request-response (TCP_RR) mode to test round-trip latency. Netperf has two components: netserver and netperf. Netserver is run on one machine, and waits for netperf client connections from another machine. The client connects to the server, does a short handshake, and then sends data to the netserver. To reduce performance variation due to network congestion or other issues, we configured both the server and the client to run in the same network. Table 1 summarizes the workloads, metrics, and load intensities used for evaluating various resource management dimensions. Refer Table 6 under Section 4.4 for additional benchmarks used for resource isolation tests.

**Table 1: Benchmarks, performance metrics, and workload intensities used.**

| Resource | Metric | Benchmark | Workload intensity | | | |
|---|---|---|---|---|---|---|
| CPU | Throughput (reqs/s) | Sysbench | CPU(%) | 1, 30, 60, 90, 99 | | |
| | Latency (ms) | | | | | |
| Memory | Throughput (MB/s) | Stream | | | | |
| Disk | Throughput (KB/s-sequential) | Fio | Disk I/O-R/W (KB/s) | Seq | Rand | Rand-mix |
| | Throughput (IOPS-random) | | | | | |
| | Latency (us) | | | 5, 20, 30 | 30 | 50R/50W |
| Network | Throughput (Mb/s) | netperf | Network I/O (KB) | 0.5, 1, 4, 8, 16 | | |
| | Latency (s) | | | | | |

### 3.2 Virtualization overhead metrics

Virtualization overhead was quantified in terms of performance, power, and resource usages overhead. The performance overhead is the performance loss relative to the baseline scenario (i.e. execution in the native environment), and is defined as:

$$Perf_{ovh} = \frac{|Perf_{virt} - Perf_{native}|}{Perf_{native}}, \tag{1}$$

where $Perf_{ovh}$ is the performance overhead, computed by dividing the performance under virtualization ($Perf_{virt}$) by that in the native ($Perf_{native}$) environment.

The power usage overhead is the extra power usage relative to running in the native environment. It is defined as:

$$Power_{ovh} = \frac{|Power_{virt} - Power_{native}|}{Power_{native}}. \tag{2}$$

For resource utilization overhead, we use a more fine-grained approach that takes into account the resource usage of different components involved in virtualization (guests, Dom0, and/or hypervisor/container engine). The resource overhead is defined as the extra resources used by the virtualization system relative to the resources used by the guest alone. More precisely, it is defined as:

$$U_{j-ovh} = \frac{|\sum U^i_{j-guest} - U_{j-virt}|}{\sum U^i_{j-guest}}, \tag{3}$$

where $\sum U^i_{j-guest}$ is the summation of the *j* resource (*j*=CPU, memory bandwidth, disk or network I/O) utilization of all instances running on a server, $U_{j-virt}$ is the overall *j* resource usage of the $virt$ (LXC,Docker,KVM, or XEN) virtualization system including usages for guests, Dom0, and/or hypervisor/container engine.

## 3.3 Measurement methods and tools

To understand the utilization overhead of multiple resources, it is important to monitor resource usage at all levels of virtualization. There are several tools that measure resource usage, but none of them can simultaneously measure every resource of interest in this work. We therefore extended a set of previously developed scripts [30] for analyzing the resource utilization of the XEN platform using one benchmark. The extended scripts can be used to analyze resource usage based on several benchmarks for each platform considered in this work (XEN, KVM, LXC, Docker, native). The scripts incorporate various tools (shown in Table 2) for different resource types, and performs automatic and synchronized measurements.

The scripts implement the following stepwise procedure for each virtualized platform:

(1) It logs into the system under test (SUT), i.e. the native or virtualized environment, and starts running a benchmark.
(2) It waits for a user-specified period X (we used a 5s warm-up interval) for the benchmark to stabilize before starting monitoring. We sampled monitoring data once per second.
(3) After running the benchmark and monitoring resource usage at multiple virtualization levels (VM/host, Dom0, H/CE/PM) for a user-specified period Y, the script triggers the termination of the benchmark and monitoring tools.
(4) The script then waits for a user-specified interval, Z, before running the next benchmark to increase the reliability of the results. The length of this interval depends on the benchmark.
(5) The test is completed after repeating the above steps for a user-specified number of times (4 repeats were used in this work).
(6) The script then summarizes the measured information into one file per VM/container and PM, and compiles these files into a single log file containing results for all used machines. Finally the results are averaged over the duration of the run.

We collected utilization data using standard tools available for the Linux OS platform (top, free, mpstat, vmstat, and ifconfig) and specialized tools provided with the tested virtualization platforms for monitoring instances: xentop (XEN), virt-top (KVM), ctop and lxc-info (LXC), and Docker stats (Docker).

**Monitoring overhead**: We evaluated the overhead of our chosen resource monitoring tools, including resource and power usages, by running them alone for the four virtualization platforms. Monitoring was a light weight process in all cases (CPU and memory usage by less than 1%), and hence the results are not shown here. This reflects that virtualization overheads are essentially unaffected by the resource monitoring systems used in the benchmarks.

**Table 2: Measurement tools used for the virtualization systems under study.**

| | | VM | | | | DOM0 | | | | H/PM | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| XEN | | CPU | Mem | I/O | Net | CPU | Mem | I/O | Net | CPU | Mem | I/O | Net |
| | xentop | Y | | Y | Y | Y | | Y | Y | | | | |
| | free | | Y* | | | | Y | | | | | | |
| KVM | | VM | | | | | | | | H/PM | | | |
| | | CPU | | Mem | | I/O | | Net | | CPU | Mem | I/O | Net |
| | virt-top | Y | | | | Y | | Y | | | | | |
| | free | | | Y* | | | | | | | Y | | |
| LXC | | Container | | | | | | | | CE/PM | | | |
| | | CPU | | Mem | | I/O | | Net | | CPU | Mem | I/O | Net |
| | ctop | Y | | Y | | Y | | | | | | | |
| | lxc-info | | | | | | | Y | | | | | |
| | free | | | | | | | | | | Y | | |
| Docker | | Container | | | | | | | | CE/PM | | | |
| | | CPU | | Mem | | I/O | | Net | | CPU | Mem | I/O | Net |
| | Docker stats | Y | | Y | | Y | | Y | | | | | |
| | free | | | | | | | | | | Y | | |
| All | | VM/Container | | | | | | | | H/CE/PM | | | |
| | | | | | | | | | | CPU | Mem | I/O | Net |
| | mpstat | | | | | | | | | Y | | | |
| | vmstat | | | | | | | | | | | Y | |
| | ifconfig | | | | | | | | | | | | Y |
| Power | snmp | PM | | | | | | | | | | | |

*-need to run inside the VM.

# 4 EXPERIMENTATION

## 4.1 Hardware setup

All the experiments were performed on physical machines with 32 cores (AMD OpteronTM6272), 12288KB of L3 cache, 56 GB of RAM, 4x500 GB SATA disks, and a 1 GB network adapter. The CPU had two sockets, each socket had 2 NUMA nodes, and each node had 8 cores. One of these machines was used to run the script for starting the benchmarks and monitoring tools. Power management for the SUT host was disabled to reduce the effects of resource scaling and to increase the comparability of the results. The physical server’s power consumption was monitored using HP Intelligent Modular PDUs, which provide per-power-socket power usage data using the Simple Network Management Protocol (SNMP). These PDUs have a resolution of 0.01A (*230V = 2.3W), updated every 0.5s. The server’s idle power consumption was 130W, and its cores operate at 1.7GHz. To analyze the average boot-time latency of the studied techniques, we instrumented the boot process of KVM and XEN VMs using the bootchart [9] tool, used the *systemd-analyze* tool for CoreOs-based VMs and measured instance creation times for LXC and Docker by using the Linux *time* command to determine the duration of the container-start up execution commands.

We used the Ubuntu 14.04 LTS Linux distribution for the native, VM, and container environments. We also used the same kernel version (3.19) for all systems because the use of different kernel versions could introduce experimental noise. Virtualization was achieved using LXC 1.0.9, Docker 1.12.3, QEMU with KVM 2.0.0, and Xen 4.4.2. We used the standard default installations for containers and VMs unless otherwise stated. Each instance was allocated the same amount of memory (10 GB), and CPU allocations were set on a per-instance basis depending on individual needs. The *cpuset.cpus* and *memory.limit* CGroup subsystems were used to limit specific CPUs and the amount of memory per container. We used *virsh’s setvcpus* and *xm vcpu-set* to assign cores to KVM and XEN VMs respectively. The VMs were created with 50GB hard disk images. To measure the overhead imposed by the virtualization layer, we first ran all the workloads on the bare-metal OS in a PM.

We ran the different benchmarks with different intensities using single- and multi-instance (for additional overhead, isolation, and over-commitment) configurations to investigate the impact of virtualization on overall performance (throughput and latency), isolation, resource usage, power consumption, and start-up time.

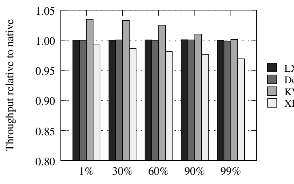
## 4.2 Single-instance virtualization overhead

We evaluated the performance and resource/power usage overhead incurred by using single VMs/containers, and compared them to results obtained in the native environment. This made it possible to determine the overhead imposed by virtualization and the additional overhead imposed by running multiple instances.

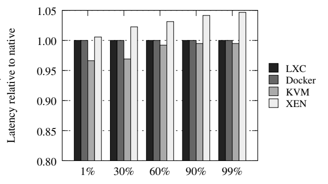
*4.2.1 CPU.* Figure 1 presents the results of the CPU-bound sysbench benchmark analysis for LXC, Docker, KVM and XEN at different levels of CPU load. We normalized the result of each test against the native performance. The results presented in Figures 1a and 1b for throughput and latency, respectively, show that OS-based systems achieved similar performance to the native environment. However, some correlation between performance and input size is apparent for KVM and XEN: their performance decreases slightly as the CPU load increases (a maximum overhead of 4.6% for XEN at full utilization). This aside, the H-based systems do not impose a greater performance penalty than the OS-based platforms when only a single instance is being run in the system.

The resource and power usage associated with the results shown in Figure 1 are presented in Figure 2. We only present CPU utilization data, shown in Figure 2a, because the benchmark had minimal effects on other resources for any instance or virtualization layer. CPU usage overhead was negligible in all cases. The power usage data, shown in Figure 2b also reveal no large or interesting differences between the virtualization platforms. One platform may consume less power for a lightly loaded CPU workload while it may consume higher if the intensity of workload increases (e.g. Docker). All of the tested platforms incur only a small power overhead when compared to the native environment (avg. overhead of 1.5%). Another observation is the lack of energy proportionality (EP) in the system, i.e., the energy consumed does not decrease linearly with load. At lower loads, power usage is primarily dominated by idle power consumption. In the idle scenario, KVM and XEN use 1.46% more power than the native environment (due to issues such as the cost of supporting a complete guest OS). Interestingly, the dynamic power usage (i.e. the power usage after discounting idle power) of the virtualized systems is far greater than would be expected at low utilizations. Figure 2c shows the percentage of the dynamic peak power usage. Although Docker makes the system more EP than the rest, on all the tested systems the average power usage is about 43% of their peak power at very low utilization (1% CPU load).

**Summary**: For the CPU-bound benchmark, the overhead of both H-based and OS-based virtualization platforms are rather insignificant when running a single VM/container. At low workload levels, the systems draw far more power than expected, making them less EP. The insight can be used by system designers to make power-aware workload placement decisions.



**(a) Throughput.**

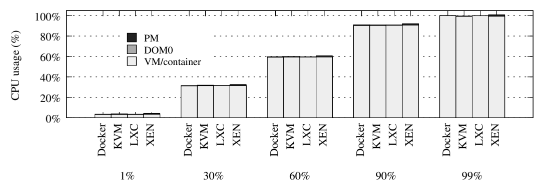


**(b) Latency.**

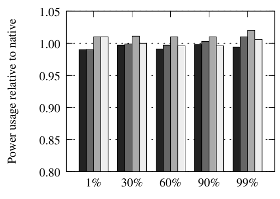
**Figure 1: CPU performance at different load levels.**

*4.2.2 Memory:* The STREAM benchmark supports four vector kernels (Copy, Scale, Add and Triad). We only present results for the Triad operation here because it is most practically relevant to large systems [4].

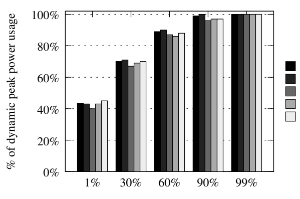
As shown in Table 3, H-based systems had poorer memory throughput than OS-based systems: KVM has a 11% performance overhead, while XEN achieved the worst performance, with an overhead of 22% relative to the native environment. The memory performance of the OS-based platforms was comparable to that of the native environment.



**(a) CPU usage.**



**(b) Power usage.**



**(c) Energy proportionality.**

**Figure 2: Resource utilization at different level of CPU load.**

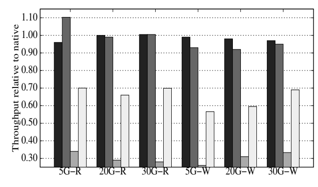
As the main determinant of performance is the bandwidth to the main memory, the benchmark is less dependent on the total amount of memory or other resources [32]. We therefore do not discuss resource usage across the execution environments. Note that the benchmark was run using default and non-tuned memory configurations, allowing the tested systems to use any NUMA node. Performance could be improved by ensuring execution was done using a single NUMA node. A more detailed analysis of the effect of NUMA is left for future work.

**Summary:** H-based virtualization imposes a much higher memory access overhead than OS-based virtualization.

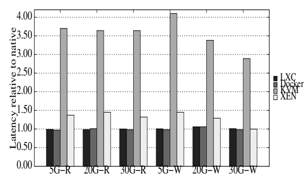
**Table 3: Throughput of Triad operation using STREAM benchmark.**

| | Native | LXC | Docker | KVM | XEN |
|---|---|---|---|---|---|
| (MB/s) | 4384 | 4389 | 4289 | 3882 | 3419 |

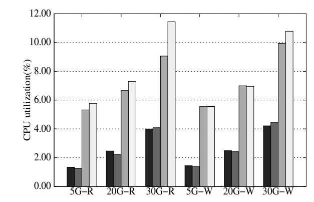
*4.2.3 Disk I/O.* For KVM and XEN, the file system is mounted inside the VMs and both use raw file formats. LXC uses the default directory backing store as file system. We tested Docker with two file stores, *AUFS* and *volume*. We first show the results for AUFS, the default storage driver for managing images and layers and later for volumes that are suited for write-heavy workloads. We measure the performance of Disk I/O when reading and writing files of 5GB to 30GB using sequential and random access modes. Figure 3



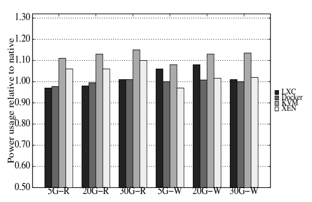
**(a) Throughput.**



**(b) Latency.**



**(c) CPU utilization.**



**(d) Power usage.**

**Figure 3: Performance, CPU and power usage for sequential operations.**

shows the sequential read and write throughput, latency, CPU, and

**Table 4: Average CPU and disk I/O usage for read and write operations.**

| | | VM/container | Hypervisor/PM | Dom0 |
|---|---|---|---|---|
| CPU (%) | LXC | 2.2 | 0.3 | |
| | Docker | 2.8 | 0.2 | |
| | KVM | 6.9 | 0.23 | |
| | XEN | 4.2 | 1.16 | 2.5 |
| Disk I/O (KB/s) | XEN | 73371 | 87932 | 0 |

power usage for the tested systems during the I/O benchmarking experiments. LXC and Docker achieved native-like performance, but KVM and XEN were slower. As shown in Figure 3a, the average throughput overhead of KVM and XEN were 69% and 35% respectively. The latency overhead for disk I/O using KVM was much more severe – 256%, as shown in Figure 3b. This is mostly due to the greater overhead (buffering, copying, and synchronization) involved in pushing filesystem activities from the guest VM via the Dom0/hypervisor to the disk device. Each guest I/O operation must go through the paravirtualized VirtIO drivers, which cannot yet match the performance of OS-based systems.

Figures 3c and 3d show the measured CPU and power usage values. While the CPU usage never rose above 12% during the disk-intensive benchmarks, KVM and XEN exhibited larger increases in CPU usage, and therefore used more power (8% more, on average). As shown in Table 4, the extra CPU overhead imposed by virtualization with XEN (46%) was greater than that for the other tested platforms due to the CPU utilization of Dom0 and the hypervisor. XEN also imposes extra disk I/O overhead on the H/PM; the magnitude of this extra overhead is equal to (for read) or greater than (for write) than that for the VM's I/O. The remaining techniques do not impose extra I/O utilization on the H/PM. Therefore, we do not show any corresponding results for these platforms.

We investigated the KVM disk I/O process in more detail to identify possible pitfalls in its performance, focusing on the impact of cache settings, image formats, and disk controllers. Experiments were performed using different KVM caching policies: no caching, write-through (read-only cache), and the default write-back (read and write cache). The *write-back* technique outperformed the read-only and write-only options by 40% for write operations and/or 20% for read operations. We also changed the disk controller from the *VirtIO* to the *IDE* driver, but found that this caused a performance loss of around 66%. These results indicate that using VirtIO produces a much lower virtualization overhead than regular emulated devices. We also compared the raw file format to the copy-on-write (*QCOW2*) format. While QCOW2 is well known for its snapshot support, the overhead associated with its I/O operations is greater than that for raw storage. In our experiments, it achieved only 80% of the raw-storage performance for write operations.

To quantify the disk I/O performance of using different file stores other than AUFS for Docker, we tested the use of *volumes*, which mounts a file system in the container, and *direct device mapping* that exposes the host directory to the container. The test involved reading and writing 30 GB on an existing file. Table 5 shows the results for the three file stores. AUFS exhibited significant overhead due to the *copy up* operation it performs in its image layering implementation: it copies the entire file even if only a small part of the file is being modified. If the file is being modified for the first time, the entire file is copied up from the underlying image to the container's top writable layer, producing the worst possible write performance (0.552 bytes/s). Subsequent reads and writes are faster, however, because the operations are performed on the file copy that is already available in the container's top layer. The use of Docker volumes confers a noticeable performance increase over AUFS, and direct device mapping offers a smaller increase. Hence, the performance penalty of AUFS must be balanced against its advantages in terms of fast start-up times and efficient use of storage and memory. The magnitude of the penalty depends on the size of the file being manipulated, the number of image layers, and/or the depth of the directory tree.

**Table 5: Docker sequential disk I/O performance with AUFS, volumes, and direct device mapping.**

| Operation | | AUFS | Volume | Direct mapping |
|---|---|---|---|---|
| Write-30G (KB/s) | First | 0.55 | 161793 | 128407 |
| | Second | 84251 | | |
| | Subsequent | 153378 | | |
| Read-30G(KB/s) | | 172493 | 319754 | 184458 |

Figure 4 presents measured IOPS results relative to native for random read, write, and mixed (50% read/50% write) operations on 30 GB files. The OS-based systems achieved similar performance as native, followed by KVM. XEN showed worst performance (40% overhead) on random read operation. However, it performs better than the rest on random write operations, calling for more investigation on possible optimization available in the platform.

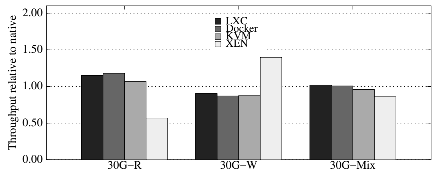
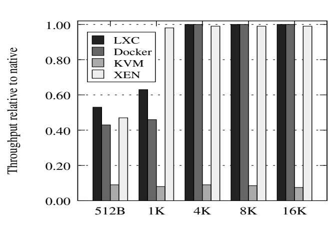


**Figure 4: Disk performance for random operations.**

**Summary**: Virtualization technology should be selected with extra care in case of disk intensive workloads due to variations in performance and resource usage.

*4.2.4 Network.* LXC, KVM and XEN use bridged networks for public IP addresses, whereas Docker uses the default docker0 bridge. Figure 5 shows the throughput and latency for Netperf benchmark. Figure 5a shows the unidirectional bulk transfer throughput for outgoing communication. For packet sizes of 4K bytes and above, LXC, Docker and XEN all achieved the maximum possible throughput and equaled the performance of the native environment, showing that bulk data transfer can be handled efficiently by both H- and OS-based systems. KVM achieved the worst throughput, showing that H-based systems still suffer from a significant performance gap. Smaller packet sizes (512B and 1K) require more CPU resources and power (as shown in Figure 6). Figure 5b shows the results obtained from Netperf request and response (TCP_RR) test. KVM increased the round trip latency and has an overhead of 250% overhead, while the latency overhead for LXC, Docker, and XEN were 7%, 12% and 58%, respectively.

Making use of the host's networking instead of bridged network allows Docker containers to achieve near-native performance. Figure 7 shows the impact of this approach relative to the default network setting. Bridged networking achieves worse performance in terms of both throughput and latency for smaller message sizes, as shown in Figures 7a and 7b, respectively. Although host networking gives direct access to the host network stack and improves performance, it removes the benefits of network namespaces and isolation between the host machine and containers.


(a) TCP bulk transfer throughput.

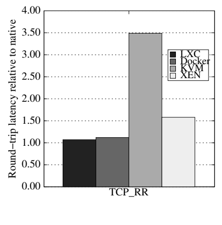
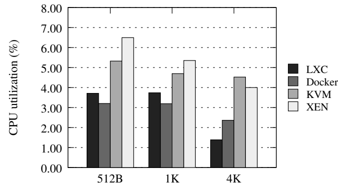

(b) Round trip latency.

Figure 5: TCP bulk transfer throughput and round trip latency.


(a) CPU utilization.

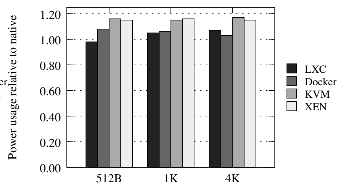
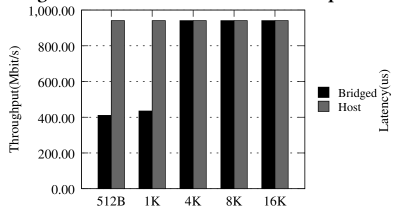

(b) Server power consumption.

Figure 6: CPU utilization and server power usage for TCP bulk transfer.


(a) Throughput.

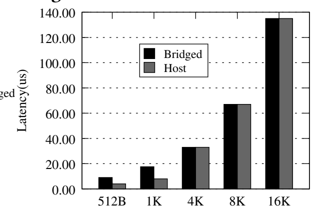

(b) Latency.

Figure 7: Network performance of Docker under two network configurations.

## 4.3 Multi-instance virtualization overhead

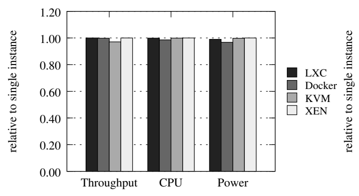
We conducted another set of experiments to study how virtualization overhead for the selected platforms is affected by co-location. The intent here is to quantify the change in overhead and hence not show the observations similar to single instance run. Figure 8 shows the performance, and CPU/power usage for compute-, network-, and disk intensive workloads run on a PM hosting four concurrent instances. In all cases, the load was distributed equally across the instances to match the baseline single instance test.

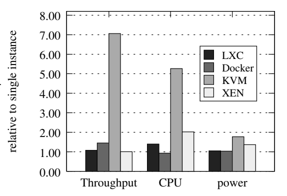
Figure 8a shows the sums of the measured throughout, CPU and power usage for the compute-bound benchmark over four instances. The workload run on each individual instance is equal to 25% of that used in the single instance tests. The results clearly show that co-locating CPU-bound instances (in the absence of resource contention) does not impose any significant overhead on throughput, CPU usage, or power consumption for any platform.

Figure 8b shows the total bandwidth, CPU, and power usage for the network-bound benchmark based on a test in which each instance sends a 512 byte packet (the corresponding single instance test used 2k byte packages). XEN achieved similar bandwidth to the baseline value, whereas LXC, Docker and KVM exhibited 1.1x, 1.45x, and 7x bandwidth improvements, respectively. The use of multiple instances had no appreciable effect on CPU or power usage for Docker. However, LXC, KVM and XEN exhibited CPU usage increases of 40%, 426%,and 100%, respectively, and power usage increases of 4.7%, 79%, and 38%, respectively. The increased CPU usage was attributed to increases in the CPU usage of individual instances and the virtualization layers in order to serve the workload. Figure 9a shows the distribution of CPU usage for both OS- and H-based systems. In general, instance CPU usage increased for all techniques. While the Docker and LXC container engines exhibit minimal increments, the KVM hypervisor and XEN Dom0 incur more overhead when running multiple VM instances (increment of 270% and 111%, respectively, compared to single VMs). This could significantly affect the performance of applications with high CPU requirements in oversubscribed systems.

Figure 8c shows the total disk I/O performance, CPU and power usage for instances reading/writing 5GB of files. Every platform performed better (particularly on read operations) in the multi-instance test but had higher CPU usage. KVM exhibited the largest increase in CPU usage (234% ) for read and write operations, and offered smaller performance gains due to scaling. Figure 9b shows the distribution of CPU usage for read operations on each platform. While the increase in CPU usage for KVM is due to an increase in the CPU usage of the VMs themselves, much of the extra CPU usage under XEN is due to DOM0 and the hypervisor for the co-located guest VMs (224% and 109%, respectively). The throughputs of the individual instances in our multi-instance memory-bandwidth experiment were similar to those for the single instance test. Therefore, results for the memory benchmark are not shown.

**Summary**: H-based systems use more resources in co-located environments, particularly on disk- and network-intensive workloads.


(a) CPU.


(b) Network.

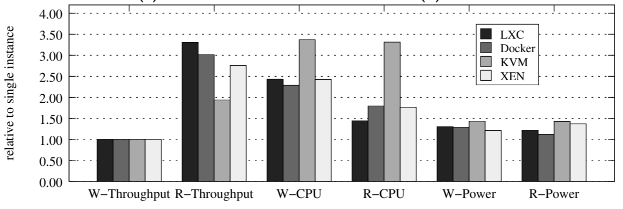
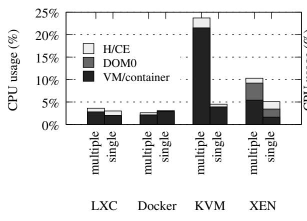

(c) Disk.

Figure 8: Throughput, CPU and power usage of multiple instances for different benchmarks.


(a) Network.

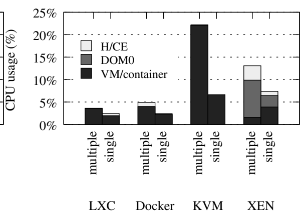

(b) Disk.

Figure 9: CPU usage for network bandwidth- and disk-intensive workloads.

## 4.4 Resource isolation

So far, we have discussed the virtualization overhead in terms of performance, resource usage, and power consumption for physical machines running single and multiple instances for *each* benchmark described in Table 1. Our focus now is to highlight the interference caused by deploying a *diverse* range of co-located applications on shared hardware resources as commonly done in a cloud. To this end, we performed isolation tests using the four environments, with two guests running on the same host machine.

We first ran a baseline application in one guest and used the obtained data to compare to the results obtained when running the same application side-by-side with another application. The second application was chosen to complement the baseline application, compete with it, or have an adversarial impact. The summary of experimentation performed based on Table 6 is shown in Figure 10. The sections below present analysis of the results obtained in terms of CPU, memory, disk, and network resources.

**Table 6: Benchmarks used for isolation tests.**

| | CPU | Memory | Disk | Network | adversarial |
|---|---|---|---|---|---|
| CPU *Sysbench* | lookbusy [18] | Memtester [20] | FIO | Netperf | ForkBomb [12] |
| Memory *Blogbench [11]* | lookbusy | Blogbench | FIO | Netperf | MallocBomb [13] |
| Disk *Fio* | lookbusy | Memtester | FIO | Netperf | Bonnie++ [3] |
| Network *Neperf* | lookbusy | Memtester | FIO | Netperf | TCP SYN |

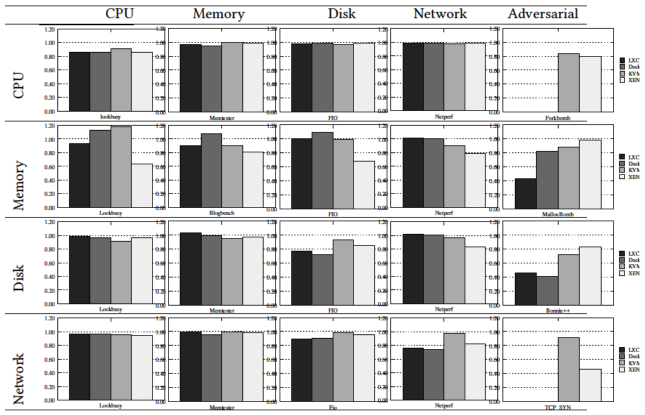


**Figure 10: Performance of applications when co-located with instances running different workload types.**

4.4.1 *CPU isolation.* The top row of Figure 10 shows how the performance of Sysbench was affected by the different co-located applications. All the virtualization systems performed within a reasonable range of their baseline performance for complementary workloads. When co-located with the adversarial fork-bomb workload – a classic test that loops to create new child processes until there are no resources available, KVM and XEN achieved 83% and 80% of their stand-alone performance. On the other hand, LXC and Docker were unable to complete the benchmark: the container that ran Sysbench was starved of resources and thus unable to serve its requests. Although it is possible to prevent the forkbomb effect by using the *pids cgroup* in LXC or the *nproc cgroup* in Docker to limit the number of processes allowed, preventing this type of extreme starvation generally requires careful accounting for, and control over, every physical and kernel operation [43].

In the competing workload experiments, LXC, Docker and XEN achieved 85% of the performance observed in the absence of co-location, but KVM performed better, achieving 91%. It is noteworthy that all of the available physical CPU cores were shared between the guests on all the platforms.

**CPU pinning**: To demonstrate the impact of CPU pinning on CPU interference, we performed an experiment in which each VM/container was assigned to specific physical CPU cores while using the same amount of CPU resources as in the previous case. For this purpose, we used CGroup methods to assign CPU affinity for LXC and Docker, and the vCPU affinity technique for KVM and XEN. Figure 11 shows the performance achieved with and without pinning. For both H- and OS-based systems, the unpinned configuration outperformed the pinned configuration when the VM/container was running in isolation, showing that the default scheduler does work well in the absence of resource competition [48]. However as competition increased (i.e. when the baseline application was co-located with a competing or adversarial VM/container), the pinned configuration outperforms that without pinning (by up to 19%).

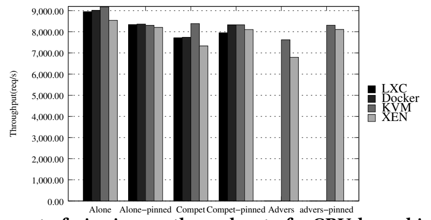
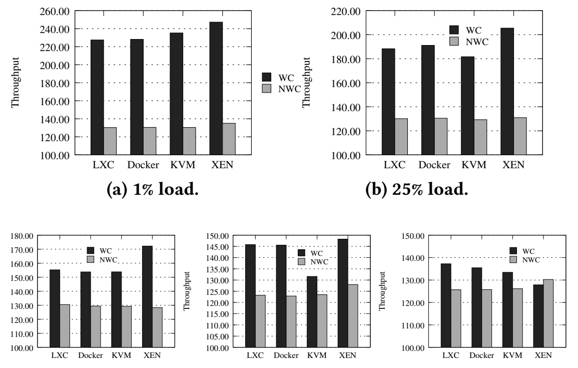


**Figure 11: Impact of pinning on throughput of a CPU-bound instance when co-located with competing and adversarial instances.**

**Scheduling technique:** CPU schedulers can support work-conserving (WC) and/or non work-conserving (NWC) scheduling modes [31], with the former usually being the default choice. In WC mode, CPU resources are allocated in proportion to the number of shares (weights) that VMs have been assigned. For example, given two VMs with equal weights, each VM would be allowed to use at least 50% of CPU and could potentially consume the entire CPU if the other VM is idle. Conversely, NWC mode defines a hard bound or ceiling enforcement on CPU resources, i.e. each instance owns a specific fraction of the CPU. In the above example, each VM would be allocated up to, but no more than, 50% of the CPU resources even if the other VM was completely idle.

Figure 12 shows the performance of the compute-bound Sysbench benchmark when co-located with the background lookbusy instance under WC and NWC modes. XEN uses the *weight* and *cap* functionality of the credit scheduler to specify WC and NWC behaviors, respectively. LXC, Docker, and KVM use the standard Linux control group scheduling interfaces, *CGroup-quota* (WC) and *CGroup-sharing* (NWC) in the Completely Fair Scheduler (CFS). The same CPU resources were allocated in both the WC and NWC cases — 50% of total CPU cycles. There are three notable aspects of Figure 12. First, the WC technique exploits all available CPU cycles to achieve better performance at lower background loads. As the background load increases, its performance decreases (with an average reduction of 44% when increasing from 1% to 100% load), reducing each application's performance isolation. On the other hand, NWC only permits each instance to use resources up to a predefined threshold, resulting in consistent performance. Second, the performance of WC is higher than that of NWC, even at higher loads, on all platforms. This is because it exploits any underutilized CPU resource in the system. This is particularly important for applications that have strict performance requirements or when high system utilization is desired. Third, the XEN WC method achieves higher throughput than the others (except at full load). This presumably occurs because the XEN credit scheduler tends to over-allocate CPU share to guest VMs [31, 51].

**Summary**: H-based systems provide stronger isolation than OS-based systems that use shared host OS kernels, which can potentially lead to denial of service. CPU allocations can be optimized by using CPU pinning and scheduling to manage resource contention.



(a) 1% load. (b) 25% load.

(c) 50% CPU load. (d) 75% load. (e) 100% load.

**Figure 12: Performance impact of using WC and NWC scheduling modes at varying level of background CPU loads.**

4.4.2 *Memory Isolation.* The second row of Figure 10 shows the normalized read performance of the baseline application, Blogbench (high memory- and low disk-intensive application). LXC achieves only 42% of the standalone performance when faced with the adversarial workload, while the other platforms achieve at least 80% of the standalone performance. Though not shown in the table, Docker achieved lower write scores (70% of the standalone value, on average) with a competitive workload. All of the platforms achieved write scores of at least 93% with an adversarial workload except Docker, which achieved only 83%.

4.4.3 *Disk Isolation.* The results of disk isolation tests (shown in the third row of Figure 10) show that LXC and Docker achieved 77% and 72% of the standalone performance when co-located with the competitive workload, but only 45% and 41%, respectively, when co-located with the adversarial Bonnie++ workload (Bonnie++ was set to continuously read and write to the hard disk). XEN achieved consistent performance for these two workload types, with an average of 83% of the standalone value. KVM performed slightly worse when co-located with the adversarial workload, achieving 71% of the stand-alone performance.

4.4.4 *Network Isolation.* To measure adversarial interference, we used the *hping3* tool to create and send a large number of TCP SYN packets to a target system. We configured the co-located guest to serve as the victim of the attack, and the attack emulator was run on another machine in the same local area network. The last row of Figure 10 shows the impact of co-location on the network-bound application's throughput. KVM performed better than the other platforms for all workload types, while XEN performed particularly badly with the adversarial workload, exhibiting 54% degradation compared to standalone. LXC and Docker could not complete the benchmark when co-located with the adversarial workloads, but achieved 76% and 74% of their standalone performance when co-located with competitive workloads. Similar results were obtained for latency, and are hence not shown.

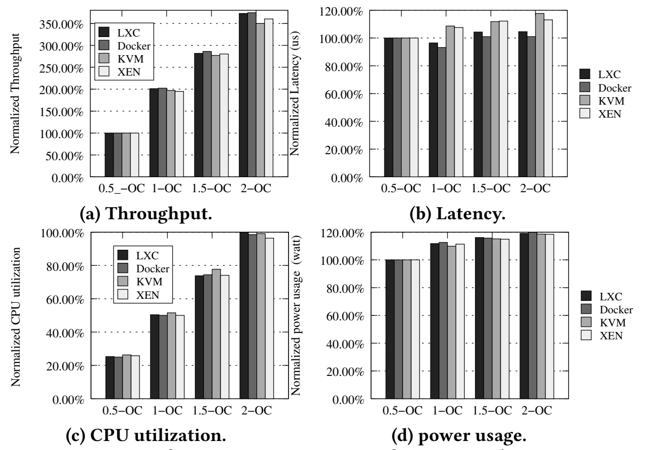
**Summary:** While isolation is better managed by H-based systems, no platform achieves perfect performance isolation. Except for CPU which can properly be isolated by tuning the CPU allocation, other resources such as cache, memory bandwidth, and disk I/O are difficult to isolate. Our findings may be of interest for designers of interference-aware resource allocation systems that aim to predict expected performance by grouping applications based on workload characteristics.

## 4.5 Over-commitment

In cloud data centers, it is often observed that all requested capacity is not fully utilized; utilization can be as low as 20% [33]. This creates an opportunity to employ resource over-commitment—allocating more virtualized resources than are available in the physical infrastructure [47]. In this work, we use instances running at 50% utilization to represent a fairly utilized datacenter.

We analyze and quantify the level of overcommitment for each virtualized system in relation to its impact on performance, CPU usage, and power consumption using the compute-intensive sysbench benchmark. The virtualization platforms accommodate CPU-overcommitment by multiplexing the virtual CPUs onto the actual physical cores. The results obtained are shown in Figure 13. For all environments, the throughput (Figure 13a) increases sub-linearly for higher OC ratios (vCPU to pCPU ratios). LXC and Docker show higher throughput rates at higher OC ratios than KVM and XEN. However, as shown in Figure 13b, unlike throughput, the latency for KVM and XEN starts to increase quickly with the OC ratio, rising by as much as 8.6% and 7.5%, respectively, at an OC ratio of 1. With an OC of 1.5, the latency for LXC and Docker reaches a maximum, whereas for the H-based platforms latency increases even further. Consequently, if latency (a crucial metric for many workloads) is particularly important, it is advisable not to use over-commit ratios above 1.5, even if there is more room for increased throughput. This illustrates the need to monitor the right performance metrics to determine the level of over-commitment that can be achieved with minimal impact on performance. Increasing the OC level also increases CPU and power usage as shown in Figures 13c and 13d.

Memory over-commitment is more challenging than CPU over-commitment as it requires careful analysis of the memory needs of all instances. It is generally outside the scope of the virtualization management framework and is therefore left for future work.



(a) Throughput. (b) Latency.

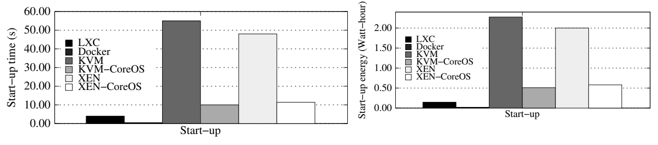
(c) CPU utilization. (d) power usage.

**Figure 13: Impact of CPU over-commit on performance and resource usage.**

## 4.6 Start-up latency and density

Start-up latency and density—the number of instances that can be supported per physical machine are two important characteristics of highly flexible resource management systems. A low start-up latency and the ability to support many instances per PM makes it possible to achieve better placement, dynamic scaling, consolidation, and/or recovery in case of failures.

Containers inherently have lower start-up latencies due to their lightweight nature while VMs are considered to take longer to start. To highlight the difference in start-up performance and show how technological advancement is changing the start-up cost, particularly for VMs, we performed experiments on the four virtualization platforms. For H-based systems, booting times were tested both using an Ubuntu 14.0 OS image image with a basic configurations and CoreOS—a container Linux image designed for minimal operational overhead. For both H- and OS-based systems, provisioning time is taken without considering benchmark start-up time. Figure 14 summarizes the start-up times and energy usage values for each platform. Docker offered the fastest provisioning time (0.5s), followed by LXC (4s) (shown in Figure 14a). While Docker needs only bring up the container process, LXC must start system-level processes like systemd, dhclient and sshd, XEN and KVM have booting times of 48s and 55s, respectively when using the Ubuntu image, with the corresponding high energy usage (shown in Figure 14b). In general, the H-based systems must boot a separate full OS with its own device drivers, daemons, and so on. But with CoreOS the booting times for KVM and XEN are reduced to 10s and 11.4s, respectively. This is an interesting direction for H-based systems which commonly are considered to start-up slowly.



**(a) Time.** **(b) Energy.**

**Figure 14: Start-up time and energy usage.**

**Summary**: OS-based platforms have lower start-up latencies (achieving average improvements of 91% and 15% over the "default" image of H-based systems in boot-time and power usage, respectively). Optimized measurements, such as the use of CoreOS, have significantly reduced VM start-up time (avg. 10.7s), making H-based systems better suited for rapid deployment.

Once in steady-state, our density test measured the impact of increasing numbers of instances on a shared physical machine. The test involved launching guest instances one by one and monitoring their resource usage. Memory costs constitute the biggest difference in overhead between containers and VMs [27] when it comes to consolidation density, hence we focus on this resource. We evaluated memory usage by summing the Resident set size (RSS) [23] values of each process of the instance held in RAM. The rest of the occupied memory exists in the swap space or file system. RSS values were easily extracted from CGroup pseudo-files. KVM was the only H-based system included in this test because it uses CGroup for resource allocation. KVM was evaluated with and without the Kernel Same Page Merging (KSM) feature. KSM works by removing duplicate copies and merging identical memory pages from multiple guests into a single memory region.

We created VMs with 1 vCPU and 1 GB memory. Figure 15 shows the average results of running 1-20 simultaneous instances. The results show that KVM without KSM (KVM-WKSM) used in average 213 MB memory per VM, 1.9x more than KVM with KSM (KVM-KSM). The LXC system container used 6 MB of RAM, while the memory usage of Docker container was mainly 1 MB, 6x and 109x lower than the corresponding values for LXC and KVM-KSM, respectively. RSS values reflect the total of the shared libraries used by the process, even though a shared library is only loaded into memory once. Hence, the actual memory footprint difference can be even larger between H-based and OS-based systems.

**Summary:** Containers provide a smaller memory footprint (avg. 31x smaller) than KVM, even when the later is used with KSM. KSM enables a more efficient use of available memory for certain workload types (up to 2x improvement) and can potentially be used to achieve a high level of memory overcommitment.

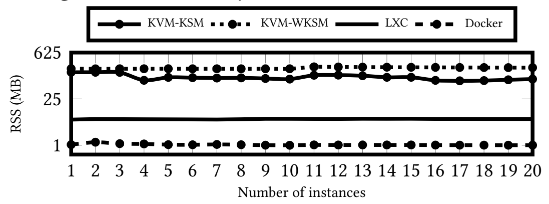


**Figure 15: Memory footprints for KVM, LXC and Docker.**

## 4.7 Power efficiency considerations

As the evaluated virtualization techniques differ in resource usage, we performed additional experiments to illustrate the impact of resource usage (CPU, memory, network, and disk) on power consumption to further guide data center operators.

**Resource and power usage relationship:** The processor is considered as one of the largest power consumers in modern servers [36].

To evaluate this and the dynamic power contribution by CPU, memory, network activity, and disks, we ran workloads that stress each resource (Lookbusy, Memtester, Netperf and FIO respectively) at varying intensities and monitored their power consumption. Figure 16 shows the utilization of each resource normalized against the peak usage and the respective power usage over time. The workloads were generated so as to first utilize the CPU followed by the memory, the network, and finally the disk. The corresponding power usage values are shown at the top of the figure, where it is shown that CPU utilization has larger impact on power usage than memory, network, and disk. Although the contributions of individual resources to system power usage is generally dependent on the workload and hardware architecture, the CPU accounted for the lion's share - 31% of the server's total power usage. Resource allocation systems should consider differences in resource power usage to reduce power usage and datacenter energy efficiency.

**Impact of CPU frequency scaling on memory bandwidth:** Dynamic voltage and frequency scaling (DVFS) is a commonly-used power-management technique that reduces the processor's clock frequency to reduce the energy usage for a computation, particularly for memory-bound workloads [45]. The motivation is often that a system's main memory bandwidth is unaffected by the reduced clock frequency but power usage is reduced appreciably.

To validate this common assumption, we performed experiments using STREAM with different CPU frequencies on each run. Figure 17 shows the correlation between CPU frequency and main memory bandwidth with power usage. Figure 17a, shows the memory bandwidth normalized against the peak CPU frequency. As the CPU frequency was scaled down, the memory bandwidth also fell, showing the dependency of the latter on the former: reducing

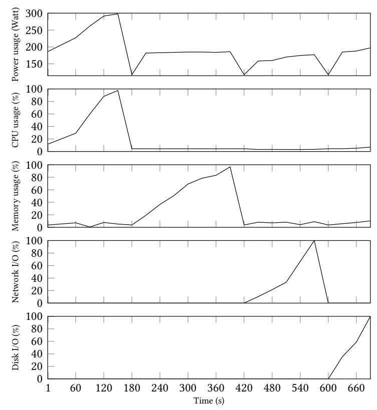
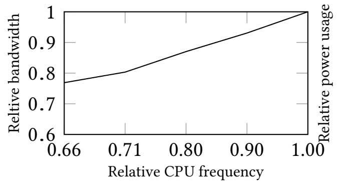


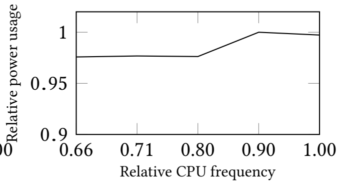
Figure 16: Impact of CPU, memory, network, and disk resources in power usage of a server.

the CPU frequency from the highest value to the lowest reduced the memory bandwidth by 23%. The power usage at different CPU frequencies is shown in Figure 17b. Similar behaviour has been observed on different micro-architectures [42].

**Summary**: Our analysis reveals the dependency of memory performance on CPU frequency. These results are likely to be useful in performance and power modeling, and for application developers aiming to optimize their applications.



(a) Performance.



(b) Power usage.

Figure 17: Performance of a memory-bound application and power usage for varying CPU frequency.

## 4.8 Summary of Results

Our results for single instance tests show that all of the studied platforms impose negligible CPU usage overhead. OS-based platforms are more efficient in terms of memory performance, followed by KVM and then XEN. LXC outperformed the other platforms in terms of disk I/O, while Docker suffered from a relatively high performance penalty when used with the default AUFS file storage but was more efficient with Docker volumes. KVM exhibited the worst performance for sequential disk I/O, even when using the faster para-virtualized driver, virtIO. Also for network I/O, KVM provided the worst performance, whereas LXC and Docker (using the host's networking capabilities) operated at native speeds. Docker's default Bridge-based network introduces more overhead but provides better isolation. Our multi-instance experiments confirmed the observations from the single instance tests and focused on the extent of overhead due to co-location. With respect to resource and power usage overhead, KVM exhibited the highest usage with respect to both disk and network I/O, followed by XEN. This was attributed to increased CPU utilization (and hence power usage) by the VMs , DOM0, and/or the hypervisor.

While VMs offer better isolation and protection against noisy neighbors, containers can be adversely affected to the extent that malicious instances could cause the whole OS to fail. CPU can by fully isolated e.g., through the use of CPU pinning. The rest show poor performance isolation due to imperfect resource partitioning. In CPU over-commit scenarios, H-based platforms perform very similarly to OS-based platforms; in both cases, the effectiveness of over-subscription depends heavily on the performance metrics of interest, the number of instances, and the types of applications that are being run. Our start-up time and density tests show that OS-based systems are more efficient and also that VMs are becoming more efficient as technology advances.

## 5 RELATED WORK

Many works have evaluated and compared the different aspects of H-based and/or OS-based virtualization platforms. Chen et al. [30] studied the resource utilization overhead introduced by virtualization layers by conducting experiments to characterize the relationship between the resource utilizations of virtual machines (VMs) and the virtualization layer in the Xen virtualization environment. Tafa et al. [46] compared the performance of hypervisors in terms of CPU consumption, memory utilization, total migration time, and downtime. Xavier et al. [49] conducted experiments to compare the performance of container-based systems alone. Containers have also been analyzed in high performance computing environments [26, 41, 50]. Felter et al. [35] conducted an experiment in which KVM was compared to Docker by stressing CPU, memory, networking and storage resources. The authors assumed that the two platforms provide similar performance to other platforms of the same type. Soltesz et al. [44] compared performance and relative scalability of the Linux-VServer environment to the XEN environment. Morabito et al. [40] compared the performance of hypervisor based virtualization and containers. Xavier et al. [50] compared the performance isolation of Xen to OS-based systems including Linux VServer, OpenVZ and Linux Containers (LXC). Sharma et al. [43] evaluated the effects of performance interference and overcommitment on the LXC and the KVM platforms.

All these works provide partial insight but lack comparative analysis encompassing performance, resource usage, and power consumption overheads, as well as isolation, over-commitment, start-up and density for multiple H-based and OS-based virtualization platforms using a diverse set of workloads. Hence, this paper provides an updated, thorough, and formal evaluation of different virtualization techniques to bridges gaps in previous evaluations and guide resource allocation decision.

## 6 CONCLUSION

H-based (hypervisor) and OS-based (container) virtualization are both used extensively in cloud data centers. Virtualization solutions need to be regularly reevaluated to better understand the trade-off provided by technological advances. This paper presents a thorough investigation of four virtualization platforms that are widely used. The analysis focuses on the most important cloud resource management dimensions, namely performance, isolation, over-commitment, efficiency of power and resource usage, provisioning times, and density to understand the current state of the technology. Our study is relevant to infrastructure providers seeking to improve resource, power usage, and/or facilitate deployment, to developers seeking to select the best solution for their needs, and to scholars to illustrate how these technologies evolved over time.

Our results show that no single system provide optimal results with respect to every criterion considered in this work, but that there are trade-offs. The higher density of OS-based virtualization makes it possible to minimize the total number of servers needed to run a given set of applications, making the system more energy efficient. However, this comes at the cost of reduced isolation and greater cross-platform challenges. Consequently, OS-based virtualization is appropriate in cases where multiple copies of specific applications are to be run. This is particularly convenient for cloud providers that run their own services and/or for applications that lack strict performance requirements (e.g. batch workloads). If the aim is to run multiple applications on servers (i.e. to operate a multi-tenancy environment) and/or to have a wide variety of OS, H-based virtualization is preferred. This is especially important when security is a major priority because VMs provide more robust isolation from untrusted co-located VMs.

Some of the shortcomings of each platform can be addressed using existing techniques. It is important to understand the capabilities and techniques available for a given platform as well as the characteristics of workloads to optimize systems. For example, it may be possible to alleviate the security issues associated with containers by extending existing security policies (e.g., anti-colocation constraints) rather than completely redesigning them and also to reduce the overhead of VMs by optimizing start-up performance and memory footprint. Overhead could be reduced by employing techniques such as CPU pinning and scheduling techniques, by sharing memory pages (KSM), by selecting appropriate image formats, by modifying storage allocations and/or network drivers. Another way to address shortfalls is to combine the best characteristics of multiple platforms into a single architecture. Hybrid systems formed in this way offer promising solutions that combine the isolation and compatibility benefits of H-based systems with the easy provisioning and deployment speed of OS-based systems.

In the near future, we hope to expand this work by including an analysis of hybrid solutions that nest containers in VMs. We also aim to extend the comparison to include unikernels, which offer greater security and efficiency than traditional operating systems. We are also hoping that our work inspires developers to further push the envelop of what is technically achievable and further reduce the gap between the two types of virtualization technologies.

## REFERENCES

[1] Dec 2016. KVM [online]. (Dec 2016). https://www.linux-kvm.org/page/Main_Page.
[2] Feb 2016. STREAM Benchmark [online]. (Feb 2016). http://www.cs.virginia.edu/stream/.
[3] Feb 2017. Bonnie++ [online]. (Feb 2017). http://www.coker.com.au/bonnie++/.
[4] Jan 2015. Optimizing memory bandwidth on stream triad [online]. (Jan 2015). https://software.intel.com/en-us/articles/optimizing-memory-bandwidth-on-stream-triad.
[5] Jan 2017. Google containers [online]. (Jan 2017). https://cloudplatform.googleblog.com/2014/06/an-update-on-container-support-on-google-cloud-platform.html.
[6] June 2015. Mesos [online]. (June 2015). http://mesos.apache.org.
[7] June 2016. Window container [online]. (June 2016). https://docs.microsoft.com/en-us/virtualization/windowscontainers/index.
[8] March 2014. YARN [online]. (March 2014). https://www.ibm.com/developerworks/library/bd-yarn-intro/.
[9] March 2015. Bootchart [online]. (March 2015). http://www.bootchart.org.
[10] March 2016. Fio Benchmark [online]. (March 2016). https://github.com/axboe/fio.
[11] March 2017. Blogbench [online]. (March 2017). https://www.pureftpd.org/project/blogbench.
[12] March 2017. Isolation Benchmark Suite (IBS) [online]. (March 2017). http://web2.clarkson.edu/class/cs644/isolation.
[13] March 2017. MallocBomb [online]. (March 2017). http://web2.clarkson.edu/class/cs644/isolation/download.html.
[14] May 2015. Sysbench [online]. (May 2015). https://www.howtoforge.com/how-to-benchmark-your-system-cpu-file-io-mysql-with-sysbench.
[15] Nov 2015. Openstack [online]. (Nov 2015). https://www.openstack.org/.
[16] Nov 2016. Docker [online]. (Nov 2016). https://www.docker.com/.
[17] Nov 2016. Linux VServer [online]. (Nov 2016). http://linux-vserver.org/Welcome_to_Linux-VServer.org.
[18] Nov 2016. Lookbusy:A synthetic load generator [online]. (Nov 2016). http://www.devin.com/lookbusy/.
[19] Nov 2016. LXC [online]. (Nov 2016). https://linuxcontainers.org/.
[20] Nov 2016. Memtester:Memory-stresser benchmark [online]. (Nov 2016). https://linux.die.net/man/8/memtester.
[21] Nov 2016. OpenVZ [online]. (Nov 2016). https://openvz.org/Main_Page.
[22] Nov 2016. Rocket Container [online]. (Nov 2016). https://coreos.com/rkt.
[23] Nov 2016. RSS:Resident set size [online]. (Nov 2016). http://www.unixlore.net/articles/quick-easy-way-monitor-process-memory-usage.html.
[24] Nov 2016. Type1vsType2 [online]. (Nov 2016). http://searchservervirtualization.techtargt.com/news/2240034817/KVM-reignites-Type-1-vs-Type-2-hypervisor-debate.
[25] Sep 2017. Kubernetes 1.8 [online]. (Sep 2017). https://www.mirantis.com/blog/expect-kubernetes-1-8/.
[26] T. Adufu, J. Choi, and Y. Kim. 2015. Is container-based technology a winner for high performance scientific applications?. In *17th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. 507–510.
[27] K. Agarwal, B. Jain, and D. E. Porter. 2015. Containing the hype. In *The 6th Asia-Pacific Workshop on Systems*. 8.
[28] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. 2003. Xen and the Art of Virtualization. *SIGOPS Oper. Syst. Rev.* 37, 5 (Oct. 2003), 164–177.
[29] D. Bernstein. 2014. Containers and cloud: From LXC to Docker to kubernetes. *IEEE Cloud Computing* 1, 3 (2014), 81–84.
[30] L. Chen, S. Patel, H. Shen, and Z. Zhou. 2015. Profiling and understanding virtualization overhead in cloud. In *ICPP*. 31–40.
[31] L. Cherkasova, D. Gupta, and A. Vahdat. 2007. Comparison of the three CPU schedulers in Xen. *SIGMETRICS Perform. Eval. Rev.* 35, 2 (2007), 42–51.
[32] C. Delimitrou and C. Kozyrakis. 2013. ibench: Quantifying interference for datacenter applications. In *Workload Characterization (IISWC)*. 23–33.
[33] C. Delimitrou and C. Kozyrakis. 2014. Quasar: resource-efficient and QoS-aware cluster management. In *ACM SIGPLAN Notices*, Vol. 49. 127–144.
[34] Z. J. Estrada, Z. Stephens, C. Pham, Z. Kalbarczyk, and R. K. Iyer. 2014. A performance evaluation of sequence alignment software in virtualized environments. In *CCGrid*. 730–737.
[35] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. 2015. An updated performance comparison of virtual machines and linux containers. In *ISPASS*. 171–172.
[36] Y. Gao, H. Guan, Z. Qi, B. Wang, and L. Liu. 2013. Quality of service aware power management for virtualized data centers. *JSA* 59, 4 (2013), 245–259.
[37] C. D. Graziano. 2011. A performance analysis of Xen and KVM hypervisors for hosting the Xen Worlds Project. (2011).
[38] R. Jones et al. 1996. NetPerf: a network performance benchmark. *Information Networks Division, Hewlett-Packard Company* (1996).
[39] P.H. Kamp and R. NM. Watson. 2000. Jails: Confining the omnipotent root. In *The 2nd International SANE Conference*, Vol. 43. 116.
[40] R. Morabito, J. Kjällman, and M. Komu. 2015. Hypervisors vs. lightweight virtualization: a performance comparison. In *IC2E*. 386–393.
[41] C. Ruiz, E. Jeanvoine, and L. Nussbaum. 2015. Performance evaluation of containers for HPC. In *VHPC*. 12.
[42] R. Schöne, D. Hackenberg, and D. Molka. 2012. Memory Performance at Reduced CPU Clock Speeds: An Analysis of Current x86_64 Processors. *HotPower* 12.
[43] P. Sharma, L. Chaufournier, P. J Shenoy, and YC Tay. 2016. Containers and Virtual Machines at Scale: A Comparative Study. In *Middleware*. 1–1.
[44] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. 2007. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review*, Vol. 41. 275–287.
[45] V. Spiliopoulos, S. Kaxiras, and G. Keramidas. 2011. Green governors: A framework for continuously adaptive DVFS. In *GCC*. 1–8.
[46] I. Tafa, E. Zanaj, E. Kajo, A. Bejleri, and A. Xhuvani. 2011. The comparison of virtual machine migration performance between xen-hvm, xen-pv, open-vz, kvm-fv, kvm-pv. *IJCSMS Int. J. Comput. Sci.: Manag. Stud* 11, 2 (2011), 65–75.
[47] S.K. Tesfatsion, L. Tomás, and J. Tordsson. 2017. OptiBook: Optimal resource booking for energy-efficient datacenters. In *IWQoS*.
[48] S.K. Tesfatsion, E. Wadbro, and J. Tordsson. 2016. Autonomic Resource Management for Optimized Power and Performance in Multi-tenant Clouds. In *ICAC*.
[49] M. G. Xavier, M. V. Neves, and C. A. F. De Rose. 2014. A performance comparison of container-based virtualization systems for mapreduce clusters. In *PDP*.
[50] M. G Xavier, M. V. Neves, F. D Rossi, T. C Ferreto, T. Lange, and C. AF De Rose. 2013. Performance evaluation of container-based virtualization for high performance computing environments. In *PDP*. IEEE, 233–240.
[51] J. Zhi. 2015. Literature Survey on Virtual Machine Performance Isolation. (2015).