Comparing Virtual Machines and Linux Containers

Sébastien Vaucher
Université de Neuchâtel
Neuchâtel, Switzerland
sebastien.vaucher@unine.ch

***Abstract*—When a single server needs to be split between multiple customers, the solution that seems the most straightforward is virtual machines. In recent years, a new technology imposed itself as an alternative: lightweight containers. Containers enable isolation of different processes on a single computer. Each container acts as if it were alone on the system. While virtual machines use full virtualization and create a complete virtual computer, containers use standard features of the kernel.**

As such, containers provide better performance, at the expense of some rigidity compatibility-wise. This rigidity is compensated by convenient tools that permit the creation of an entire ecosystem of containerized applications.

I. Introduction

The advent of the Internet has brought a lot of changes in the way people accomplish certain tasks. As more and more people gain access to the Internet, more and more computing power needs to be deployed server-side. The number of requests performed on a given Internet service varies greatly depending on the time of day and a number of other factors [1]. Traditionally, this meant that an application had to be hosted on an infrastructure that could handle peak demand, but had unused processing capacity the rest of the time. The costs associated with running a server are mostly fixed over time, therefore hosting a service with fluctuating demand on a dedicated infrastructure is not very cost-effective.

A solution to this problem is to share the infrastructure between different services with different demand patterns. This way, a high-demand period of an application can compensate a coincident low-demand period of another application. This solution has been implemented in very large proportions by Cloud hosting providers. Their business is to host a large number of physical servers that host a larger number of virtualized applications [2].

System virtual machines are commonly used to create the illusion that a dedicated machine exists. They emulate a complete virtual computer along with virtual peripheral devices. The virtualization process eats up some CPU cycles, slowing down the execution of the program running inside the virtual machine [3]. Furthermore, the program running inside the virtual machine happens to be a complete operating system. This added layer will also execute overhead operations on the CPU. A user-supplied application running in a system virtual machine will consequently run at a slower pace compared to the same application running on a physical computer.

This observation led to the development of a new type of virtualization: application containers. In a container, the overhead is reduced to a minimum. Most of the software and hardware components are shared directly by the host operating system and the guest applications. Isolation between applications is provided using features of the host operating system [4].

In this research paper, we will compare system virtual machines with application containers. More specifically, we will choose KVM [5] and Docker [6] as representatives of each category.

This research will be conducted in two parts. The first part will result in a theoretical comparison between virtual machines and containers. It will be based on existing literature. In the second part, we will experiment both systems on a physical computer and collect data relative to energy efficiency.

II. System virtual machines

The goal of a virtual machine manager (abbreviated as VMM or called a hypervisor) is to abstract the hardware of a computer, thus creating a virtual computer. A physical computer hosting a VMM is called a host, while the program running inside a virtual machine is called a guest. The interface that is provided to the guest is identical (or nearly identical in certain cases) to that of the host. All guests and the host are mutually independent from each other [7].

Contemporary hypervisors have been commercialized since the 1970s, starting on the IBM VM370 mainframe. Since then, a lot of effort has been geared towards the reduction of overhead induced by the hypervisor. In fact, most of the slowness attributed to virtual machine execution comes from the distinction between kernel-mode and user-mode. Only the host kernel is allowed to run privileged instructions on the physical CPU. However, the guest program needs to execute kernel-mode instructions on its virtual CPU. A technique that is used is called *trap-and-emulate*; when the guest tries to execute a kernel instruction while in user-mode, an error occurs, causing a trap to the hypervisor. The VMM will intercept the trap and modify the instruction to permit execution in user-mode. This process takes a lot more CPU instructions than the initial instruction would have taken [7].

A number of more efficient approaches have since been invented. Most of them consists in additions at the hardware level. For instance, Intel has added a number of extensions to the x86 architecture in their processors [8]. Their goal is to reduce the amount of emulation done at the software level [7].

VMM implementations can be sorted in 3 categories [7]:

Type 0 Hardware-based hypervisors, commonly found in mainframes.

Type 1 Hypervisors implemented as operating systems, or tightly integrated in a general-purpose operating system.

Type 2 Hypervisors running on top of a standard operating system.

Type 0 hypervisors are only found in specific applications. They can only virtualize certain guests due to hardware constraints. Their speed is quite good, but their rigidity is too big a penalty to permit large-scale deployments [7].

Type 1 hypervisors are operating systems whose only purpose is to provide a VMM. They have full access to physical CPUs, they can therefore run in kernel-mode. The performance of type 1 VMMs is good, and they provide the largest set of features among all types. There exist a special type of type 1 hypervisors: VMMs that are tightly integrated in the core of a general-purpose operating system, such as Microsoft Hyper-V or KVM. They usually provide a smaller set of features than regular type 1 hypervisors, but the integration in the kernel means that they are allowed to execute in privileged mode, with all associated implications [7].

Type 2 hypervisors are regular programs that execute as a normal process on a regular operating system. The operating system does not know that this specific process is in fact a VMM. Type 2 hypervisors have poorer performance in general than their type 0 and 1 counterparts. Their major advantage is the simplicity with which they can be installed and used by a non-specialist [7].

Paravirtualization is a technique that allows the hypervisor to present an interface that is not completely identical to the host interface. The guest needs to be modified in order to run in such a VMM. This disparity permits higher performance because some instruction translation happens directly at the guest kernel level [7]. While paravirtualization can be used to simplify CPU virtualization, it is also used to simplify virtual I/O devices implementations. Virtio [9] is a standard which aims to provide a generic interface of virtual I/O devices under Linux.

A. KVM

The Kernel-based Virtual Machine is a hypervisor implemented as a Linux kernel module [5]. It is integrated in the mainline kernel since Linux 2.6.20[1]. Because the virtual machine manager runs in privileged mode inside the kernel, it is considered as a Type 1 hypervisor [7]. KVM is based on full virtualization, which means that the guest operating system can run completely unmodified [10]. As a result, the majority of today's operating systems are supported [11].

KVM runs on x86 processors with virtualization extensions enabled. Processor extensions differ between Intel and AMD, fortunately KVM abstracts them and can use whichever [12].

KVM itself only provides an interface through `/dev/kvm`. For most users, this turns out to be too much of a burden to use. For this reason, third-party higher-level APIs have been developed in parallel. One of them is *libvirt* [13]. It provides an API that is compatible with different hypervisors, including KVM. On top of *libvirt*, one can use a graphical user interface like *virt-manager* (as seen in Figure 1) [14]. With a GUI, operating virtual machines running in KVM becomes as simple as with any other GUI-based hypervisor.

[1]Released in February 2007

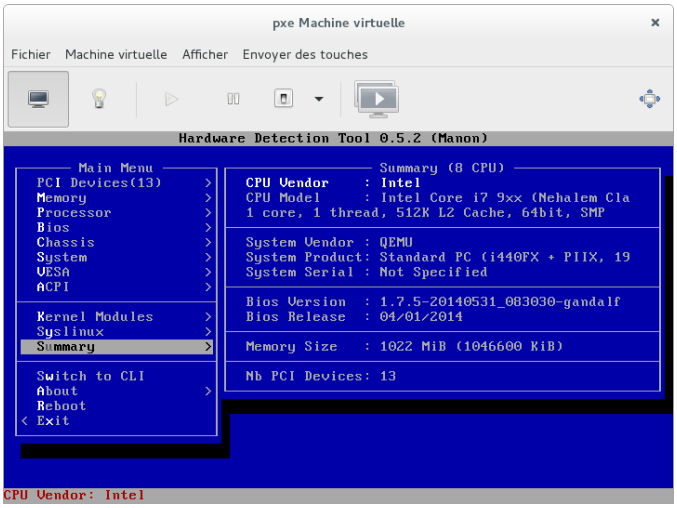


Figure 1. Hardware detection tool running in KVM through *virt-manager*

An advantage of KVM is that it supports live migration. Live migration is the ability to move a virtual machine from one physical machine to another with only a negligible pause in the execution [12]. Using this feature, physical resources can be better allocated throughout a cluster of machines. For example, a virtual machine that suddenly starts to use more resources can be moved to a "less crowded" server to free up resources for other guests, without any perceivable downtime.

III. Containers

Application containment is a relatively new technique of operating-system-level virtualization to run multiple programs (*containers*) in isolation on a single computer. Instead of virtualizing full virtual computers like hypervisors do, container-based isolation is done directly at the kernel level. Guest processes run directly on the host kernel, and thus need to be compatible with it.

Operating-system-level virtualization has existed for some time. An early implementation is the *chroot* operation introduced in Version 7 AT&T UNIX[2]. It can be used to confine a process within a specific directory of the file system by presenting a different file system root to the process [15].

More recent techniques apply the isolation to more than just the file system. The *jail* mechanism introduced in FreeBSD 4.0[3] is capable of isolating processes on a considerable number of aspects [16]. It can be considered as a container implementation.

In Linux, container systems are newer, because the required kernel features appeared later. They mostly depend on two

[2]Released in 1979

[3]Released in March 2000

features offered by the kernel: *cgroups* and *namespaces*. Control Groups provide a hierarchical grouping of processes in order to limit and/or prioritize resource utilization [17]. Namespaces can isolate processes by presenting them with an isolated instance of global resources [18]. While the two features above are probably the most important to implement a container system on Linux, other kernel features can be used to further harden the isolation between processes. For example, the Linux Containers (LXC) system uses six different features of the kernel to ensure that each guest is isolated from the rest of the system and cannot damage the host or other guests [19].

In traditional hypervisor-based virtualization approaches, the guest is often a full operating system. With containers, the guest can be as limited as a simple program. A container can also host a full operating system, with the limitation of sharing the same kernel as the host. This is called a system container, while a container running a single application is called an application container. Running a program directly inside the container has the advantage of removing the overhead created by having a second OS executing on top of the host OS [20].

With most container systems, it is possible to ship an image of an application as a single file. The image contains the application along with all its dependencies. It can be used to create a new container, effectively providing fast deployment. This mechanism is put to use by Platform as a Service (PaaS) providers or by development teams to test an application at different stages (e.g. development, testing, production).

A. Docker

Docker is a cross-platform container system implementation. It was created at dotCloud, a PaaS provider, for internal purposes [21]. Docker has since been open-sourced and can be used by anyone. In the documentation, Docker is summarized as follows:

> Docker is an open platform for developing, shipping, and running applications. With Docker you can separate your applications from your infrastructure and treat your infrastructure like a managed application [22].

The main selling point of Docker is that it allows to separate applications from the underlying layers, from the operating system to the hardware. The key high-level components are the containers, images and registries [22]. An image is a read-only template of an application meant to be run in Docker. When it gets loaded in the Docker daemon, an additional read-write layer will be added on top of the image. The UnionFS filesystem is used to provide the copy-on-write mechanism [22], [23]. The image, the read-write layer, along with some metadata and configuration data is what constitutes a container [24].

An image can be built as a layer on top of another image, called the parent image. Stacking multiple images to create a final, current base image is similar to how version control systems work (many commits stacked form the current state of a repository). Figure 2 shows an example of a container built on Debian, with emacs and Apache images added on top.

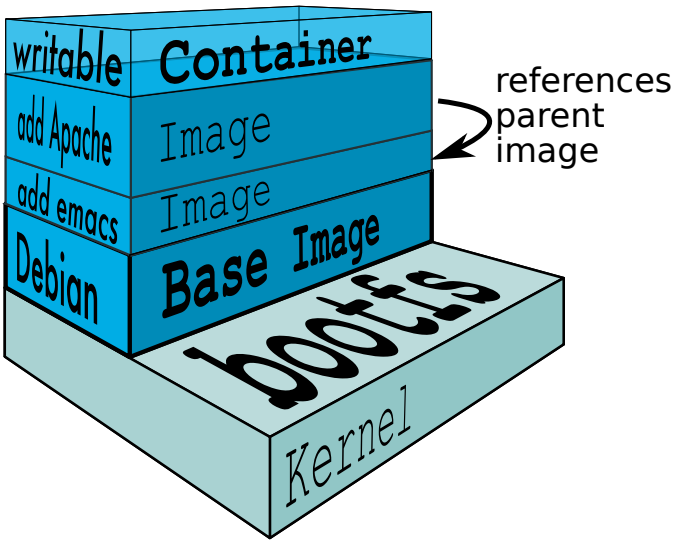


Figure 2. Stacking of multiple images and a read-write layer, forming a Docker container[5].

Docker has historically been developed for Linux. It uses Linux-specific technologies like *cgroups* and *namespaces*, as described in section III. Docker used to depend on LXC to access required container-related Linux features. Since version 0.9[6], Docker can operate using its own *libcontainer* implementation. In April 2015, a demonstration of the Docker engine running on Windows Server was shown at the Microsoft Build conference [25]. Docker will therefore support Windows hosts in the near future (when Windows Server 2016 gets released).

Using Docker simplifies the developers' and system administrators' jobs. The configuration resides in the Docker container and stays the same regardless of the hosting infrastructure. The environment used in development and in production being the same, the famous sentence "it works on my machine" becomes less frequently used. Also, configuring the platform only needs to be done once. Each Docker container being independent, it becomes possible to work on different projects that may not be compatible with each other, due to dependency problems for example (dependency hell) [21].

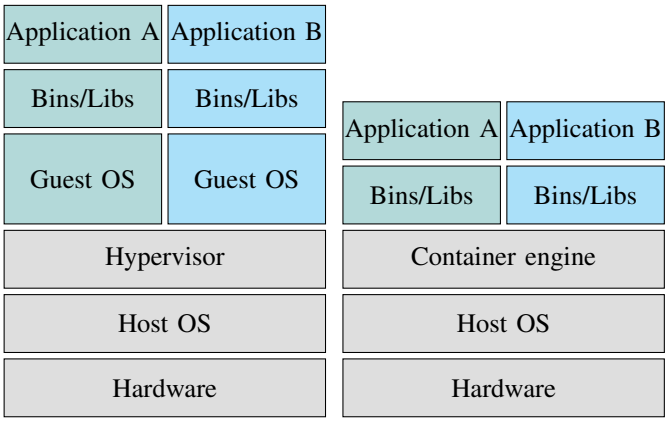
IV. Reasons to virtualize

Nowadays, even if our only goal is to serve a handful of services on a physical machine, we still have a problem that can be summarized as "dependency hell"[7] [26]. As an example, let's take a server hosting two applications written in the same language, using the same libraries. The only difference between them being that they depend on different incompatible versions of the same library. Our options in solving this problem can exist at the application level — the

[5]Source of the illustration: https://github.com/docker/docker/blob/master/docs/sources/terms/images/docker-filesystems.svg

[6]Released in April 2014

[7]Also referred to as "DLL Hell" in the Microsoft world.



(a) Virtual machine layers (b) Container layers

Figure 3. Comparison[9] of the layers involved when running an application in a virtual machine vs. in a container.

Python *virtualenv* [27] package is a good example — which is convenient, but such a solution may not exist. In this case, we must resort to implement the separation of the applications at a higher level. Usually, the solution consists in running two separate copies of the operating system on different machines. These machines may be separate physical machines, or more wisely, virtual machines.

Using system virtual machines to solve "dependency hell" means that two almost identical operating systems will run in parallel on the same machine. Application containers claim a lesser overhead, so they should be better tailored for this use case.

Another reason to use virtualization is to have more flexibility over the computing capacity running an application. For instance, it is possible to variate the amount of system memory (RAM) without pausing the application. Virtual machines and containers both provide settings to enforce quotas on system resources utilization. Some pieces of virtualization software (KVM for instance) are capable of migrating a virtual machine from computer to computer with only a negligible pause in the execution [12]. These features are relevant to cloud hosting providers who can finely optimize the utilization of their hardware.

V. Comparisons between virtual machines and containers

In this section, we first compare virtual machines and containers from a theoretical point of view (subsection V-A). Then, in subsection V-B, we analyze the respective performance of both systems by summarizing results taken from the existing literature.

A. Comparison of intrinsic characteristics

The role of both virtual machines and container engines is essentially the same: providing the illusion that a given physical machine can run multiple machines. These machines have to be isolated from each other and from the host. The difference between the two systems is how they achieve isolation. As shown in Figure 3, virtual machines execute on top of a hypervisor, while containers execute on the host operating system through an engine. As explained in section II, a hypervisor virtualizes all the hardware of a normal computer. A container engine integrates itself into the kernel of the host OS where it can differentiate executions of containerized processes and user-level processes.

[9]Inspired from the figure shown at https://www.docker.com/whatisdocker/

Table I
Features supported in KVM and Docker

Feature	KVM	Docker
Host OS $\neq$ Guest OS	✓	✗
Application packaging	✓	✓
Centralized images repository	✗	✓
Live migration	✓	(✓)
Resource sharing	✓	✓

The second major difference highlighted on Figure 3 is what is executed on each system. A full operating system is executed on a hypervisor, while a container can contain as little as our application files.

Both systems allow packaging of an application with its dependencies in a single file. A wildly adopted standard for virtual machines is the Open Virtualization Format [30]. It defines a file format that can accommodate all the necessary data to describe a virtual machine, including the disks content. A user willing to deploy the VM can just import the file into his favorite virtual machine manager and the corresponding guest will be created. In Docker, this mechanism is integrated at its core. A Docker image contains everything that is needed for a given containerized application to execute. While OVF is universal and compatible across different hypervisor implementations, there are no universal image format that suits different container engines. Packaged virtual machines are in principle bigger than container images because they always contain an entire operating system.

An important component of Docker is the Docker Hub registry. It provides ready-to-use images of common operating systems and applications. Thanks to this, downloading an image is as simple as invoking `docker pull` from the command line. KVM does not have an equivalent functionality. If the user wants a specific operating system to run in a virtual machine, he either has to install it himself, or search for a ready-to-use OVF archive on the Internet.

Live migration is a feature that makes seamlessly moving a guest between hosts possible. The guest state (memory, ...) is copied from its original host to the new host iteratively while still executing. When the copy is finished, the guest is frozen on the original host and the execution resumes on the new host. This technique minimizes the downtime involved when migrating a guest. KVM natively supports this feature [12]. In Docker, live migration is not (yet) natively supported. However, there exists a third party project called CRIU which

Table II
RESULTS OF VARIOUS EXPERIMENTS

	noploop [28] exec. time (ms)	PXZ [20] (MB/s)	Linpack [20] (GFLOPS)	STREAM [20] $\overline{C}$ (GB/s)	netperf [28] TCP latency (µs)	netperf [28] UDP latency (µs)
Native	2.391	76.2	290.8	43.48	48.21	47.18
KVM	2.397 (+0.25 %)	62.2 (−18.4 %)	284.2 (−2.27 %)	42.53 (−2.18 %)	91.57 (+89.9 %)	86.99 (+84.4 %)
Docker	2.393 (+0.08 %)	73.5 (−3.54 %)	290.9 (+0.03 %)	43.40 (−0.18 %)	59.79 (+24.0 %)	53.69 (+13.8 %)

	nuttcp [20] $\overline{C}$ (Gbit/s)	fio [20] sequential r/w (MB/s)	fio [20] random mixed (IO/s)	Ubuntu w/ MySQL [29] image size (MB)	Boot time [29] (s)
Native	9.34	778.98	104553	n/a	n/a
KVM	9.28 (−0.64 %)	771.09 (−1.01 %)	50737 (−51.5 %)	1080	19.4
Docker	9.32 (−0.21 %)	779.50 (+0.07 %)	104518 (−0.03 %)	381.5 (−64.7 %)	3.85 (−80.2 %)

enables live migration in Docker [31].

Up to now, we have only covered cases where we want the host to be isolated as much as possible from its guests. However, there are certain use cases where we need to share resources between the host and guests, for example to transfer files. In KVM, it is possible to mount a folder from the host in the virtual machine using VirtFS [32]. The guest operating system needs to support the *virtio* driver. It is included in the Linux kernel. For Windows virtual machines, it has to be installed separately. The Fedora project provides signed *virtio* drivers for Windows [33]. In Docker, it is possible to directly mount the guest filesystem on the host. This is called a data volume in Docker's documentation [34].

A summary of which features are supported in either system is presented in Table I. The differences between systems are mostly due to the use cases for which they were designed. KVM is designed to host virtual machines that should mimic bare machines behavior as much as possible. Docker design is tailored to developers and system administrators. Its goal is to obtain predictable deployment of applications.

As far as security is concerned, a quick search though the U.S. National Vulnerability Database [35] revealed that critical privilege escalation flaws were found in both Docker and KVM. As both systems are rooted in the Linux kernel, such vulnerabilities may enable an attacker to execute arbitrary code with system privileges. Therefore, a system administrator has to be wary that both systems are not perfect. A malicious program may still damage the host operating system, even when running inside a container or virtual machine. The most paranoid users can still execute the hypervisor inside a container. If a malicious program succeeds in exiting the virtual machine, it would still be inside the container.

B. Comparison of performances

In previous subsection V-A, we established what are the differences between KVM and Docker from an operational point of view. In this section, we analyze existing papers to gather data about the performance of KVM vs. the performance of Docker. Docker evolves very rapidly, it is therefore important to use results that are as up-to-date as possible.

The papers from which the results come from are the following: the first one is a recent and thorough paper comparing Docker and KVM performances. It comes from IBM Research and was published in July 2014 [20]. A more recent paper from Ericsson Research is also used; it only exists in pre-print version for the moment [28]. Some more data were taken from experiments conducted by Boden Rusell who worked for IBM at the time (May 2014) [29]. The data that we are analyzing is represented in Table II.

The first expected comparison that we can draw is the difference in size for a similar image. According to [29], the same image containing MySQL on top of Ubuntu weighs 1080 MB in KVM, and only 381.5 MB in Docker. That is a staggering 64.7 % reduction in size. The difference can be explained by the removal of components that are unnecessary in a containerized environment. The virtual machine image contains everything needed to run the operating system, including peripheral drivers, the kernel, etc. The Docker image does not need any of that content, as it has access to it directly through the host kernel. The size of the same configuration for a physical machine has not been evaluated, but it is expected to be very close or even equal to the KVM image.

The next criteria is CPU performance. Raw single-core performance is tested in [28] by using the noploop utility, which, as its name suggests, executes NOP operations in a loop. We see that simple operations that do nothing but use the CPU are not affected at all on any system. This proves that executions of user-level instructions are not disrupted by the hypervisor or the container engine.

The Linpack benchmark, on the other end, is more representative of real computations, as it is floating-point intensive. The results in the table show that Docker executes at the same speed as code running on bare-metal. To analyze KVM's results, we must highlight an important detail: the experimenter used a non-uniform memory access (NUMA) machine. By default, KVM does not expose the physical topology to its guests. In that case, the result was a 17 % drop in performance compared to the native one. Pinning each virtual CPU to a physical CPU greatly improves the overall performance of the benchmark. The performance drop is now only 2.3 % [20].

The last processor benchmark consists in running PXZ, a compression program based on the LZMA algorithm, which exploits parallelism. In this test, Docker is still very efficient, losing only 3.5 % compared to native performance. KVM is far

worse, even with the optimization (vCPU pinning) activated. It is 18 % slower, and even 22 % without vCPU pinning. The experimenter estimates that this is due to "extra TLB pressure of nested paging" [20], a TLB (translation look-aside buffer) being a very small cache inside the CPU that maps page numbers with frame numbers [7].

STREAM is a benchmark measuring memory (RAM) throughput. We observe that neither Docker nor KVM suffer much in this area. The authors also performed a random access test and noticed negligible performance loss of 1 to 2 % [20]. Virtualization has only very limited influence on memory access performance.

Network throughput and latency are very important in Cloud environments, where virtualization is of common use. For this test, KVM was set to use the *virtio* paravirtualized network card. Throughput results come from [20]. Unfortunately, their Microsoft Excel sheet does not include latency results for Docker. Therefore, the latency measurements come from [28]. All systems managed to use the 10 Gbit/s link to the maximum permitted by the network card. Latency suffers from the multiple layers involved to send or receive an Ethernet frame in virtualized environments. While the numbers show an almost doubled latency in KVM's case, the increase is only $\approx 83\,\mu s$ in average. This is almost three orders of magnitude smaller than typical latencies on the Internet, that range in dozens of milliseconds. The only applications on which this could have an impact are those with real-time requirements, like life-critical applications.

Disk performance was assessed with the *fio* benchmark. A fast SSD connected via Fibre Channel was used for the test [20]. Docker was set to use data volumes, in order to bypass the UnionFS layer. The results are similar to the network in some aspects. First, sequential read/write performance of large files is unaffected by virtualization, especially in Docker's case. The limiting factor is the number of operations per second that can be done. This is important in work scenarios where a lot of small files need to be processed. Docker manages to fully utilize the disk's IO operations per second. KVM is severely impacted because every operation needs to go through the QEMU emulator. Moreover, the experimenter noted increased CPU usage in KVM when doing input/output operations compared to the other systems [20].

An interesting real-life metric was collected by B. Russell: the time needed to boot the system [29]. When running in KVM, it took 19.4 s for the operating system to start. On Docker, it took only 3.9 s. This reduction is important for systems that start only when necessary. Free Dynos on Heroku or Gears on Openshift, two PaaS providers, employ this technique to reduce costs induced by free accounts. When trying to access a stopped application, a user has to wait for it to start again. Heroku already uses Linux containers (LXC) to minimize overhead [36].

VI. Experiments on power consumption

Power consumption and heat dissipation are two trending topics in computer science nowadays. They are often the limiting factor when designing a large cluster of computers. Virtualization is frequently used in these environments. The experiments that we conduct will compare how virtualization influences on energy consumption.

Table III
Configuration of the test machine

Machine type	Lenovo ThinkPad T420
CPU	Intel Core i7 2620M @ 2.70 GHz
Cores configuration	1 socket, 2 cores, 4 threads
RAM	8 GB DDR3 1333 MHz
Disk	Samsung 840 EVO SSD 750 GB
NIC	Intel 82579LM
Operating system	Debian GNU/Linux 8.1 ("jessie")

A. Methodology

In the experiments, we will launch the same workload on bare metal, KVM and Docker. All tests will run on a laptop, a type of computer that should be energy efficient. The characteristics of the test machine are reported in Table III. The workload that we use is the *raytrace* benchmark contained in the *PARSEC* benchmark suite [37]. It is a highly parallel process that demands a lot of processing power and requires good memory bandwidth.

We tried to have as much homogeneity as possible throughout the different systems. They are all based on the Debian GNU/Linux 8.1 operating system. To reduce the auxiliary power consumption, the wireless network card was shut down, the battery removed and all non-essential peripherals unconnected. The desktop manager (*gdm3*) was stopped and the screen powered off. All invocations were performed from a different machine via SSH. The only remaining auxiliary device is the Bluetooth controller, which is necessary to receive data from the power meter. The power meter is a PowerSpy 2 manufactured by Alciom. It was plugged directly into a 230 V wall socket on one side, and to a 90 W 20 V Lenovo charger on the other side. The *powerspy.py* script [38] was used to collect the data once per second.

For each system, the benchmark was ran using the following command:

```
date +%s; ./bin/parsecmgmt -a run -p
 ↪  raytrace -i native -n 4; date +%s
```

The date command executes before and after the benchmark run. Its output is a UNIX timestamp used to properly extract the interesting portion of the power meter log. The -i native option tells the benchmark to use the largest data set for the test. The -n 4 flag instructs the benchmark to use all 4 threads of the CPU.

The benchmark was run 3 times on each platform. In between tests of different platforms, the laptop was powered off during 10 minutes to allow it to cool down. Before each individual execution, we collected data while the system was idle. The final data used for the analysis is always the average of all executions, apart for graphs where we used the median execution.

Table IV
POWER CONSUMPTION TEST RESULTS

Context	Idle		Executing benchmark			
Measurement	$\overline{U}$ (V_{RMS})	$\overline{P}$ (W_{RMS})	Execution time (s)	$\overline{P}$ (W_{RMS})	E (J)	E (Wh)
Native	235.49	12.56	108.0	43.38	4801.2	1.33
KVM	234.45	15.57 (+24.0 %)	114.3 (+5.86 %)	46.08 (+6.21 %)	5375.9 (+11.97 %)	1.49
Docker	234.40	15.48 (+23.25 %)	108.3 (+0.31 %)	47.63 (+9.78 %)	5271.0 (+9.78 %)	1.46

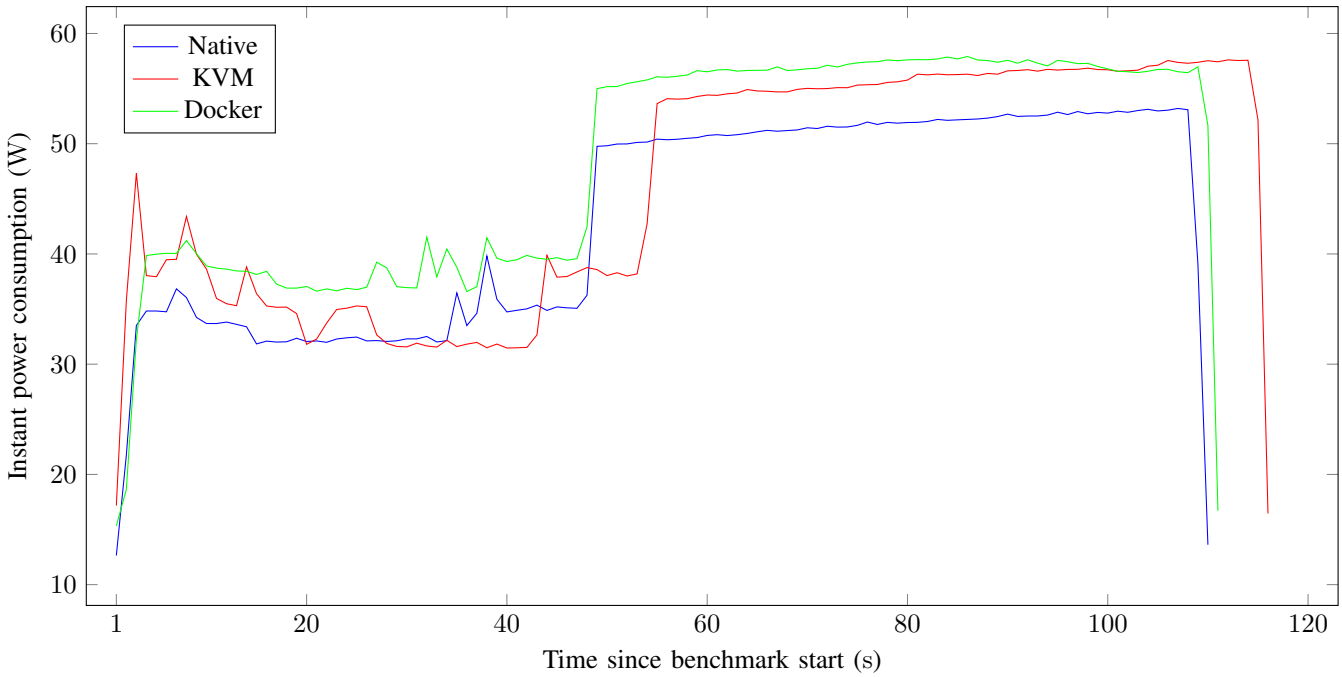


Figure 4. Power consumption during benchmark execution

B. Results

The summarized results in Table IV are divided in two contexts: idle and executing. Idle is when the system is ready but does nothing special. Executing is when the system actively executes the benchmark.

In the idle scenario, we see that KVM and Docker use 23 % more power than in the native case. This could cause issues for a hosting company that has many servers idling most of the time. The voltage is within 2 % of the nominal 230 V distributed in Switzerland.

When executing the intensive benchmark, the difference in average power consumed gets smaller than when idling. The execution time of the benchmark is consistent to what we saw in the performance comparison. Docker is as fast as the native execution, and KVM is 5.9 % slower. KVM consumes 6.2 % more power in average compared to the native execution. As KVM is slower and uses more power, it means that it uses more energy in total: +12 %. With Docker, we see an interesting trend: the execution time is the same as the native case, but with 9.8 % more power and energy consumption. In fact the increase in power and energy is exactly the same. Therefore, we can say that Docker is as fast as bare metal, but needs more energy to achieve the same result.

On the graph in Figure 4, we see that the benchmark operates in two phases. In the beginning, there is a preparation phase which uses less resources. Then, the real computations begin, maximizing the utilization of all cores. During this second phase, the power consumption increases slightly over time. Our hypothesis is that the temperature increases, and so the fans must rotate faster to dissipate the heat.

The observations are quite surprising. While the results for KVM were predictable, those of Docker are interesting. The graph shows clearly that they are similar to the native case, but translated upwards. The cause of this increased consumption is unknown. Further research is required to pinpoint why this happens. Also, we need to keep in mind that the experiments were conducted on a single machine, which was not server-grade.

VII. CONCLUSION

Linux containers, and especially Docker, are trending topics in the field of virtualization. With the information gathered in this paper, we can easily understand why more and more developers and system administrators are interested in this technology.

The Docker ecosystem simplifies certain tasks related to application development. One of its key strength is predictable deployments. Thanks to it, we know that an application that works on one computer A will work as expected on another computer B. The Docker centralized registry provides ready-to-use images that accelerate the start of a project. Compared to hypervisors, container systems provide improved performances in all aspects, as long as it is properly configured. The UnionFS layer slows down disk operations, in order to provide additional operational flexibility.

We could question why would someone use traditional virtual machine managers in an era where container systems seem to perform better in terms of performance and operational flexibility. Hypervisors are not dead yet. They still outshine containers when full hardware virtualization is needed, for instance to operate a guest operating system that is not the same as the host's. Also, hypervisors exist since a long time. Extensive research has been conducted on them, they are a proven technology. Legacy applications can also warrant the usage of hypervisors, as transitioning to containers might prove to be unfeasible due to technical or cost factors.

As far as energy consumption is concerned, virtualization necessarily implies more electrical usage. Both systems need at least $10\,\%$ more energy. We also found out that Docker can achieve the same performance as on bare metal, but it needs more power to do so. KVM is both slower and more power hungry than the native case. The difference in energy usage between KVM and Docker is quite small ($2\,\%$).

The technology behind containers evolves very rapidly. Docker is only 2 years old [21], and yet there is a lot of interest regarding it. Who knows what the Docker ecosystem will be like in one year. If so many people are interested, it means that it is a technology that was needed, and that will spare resources and time for some companies.

Acknowledgment

I would like to thank Mascha Kurpicz who supervised my work on this project. I also thank Dr. Hugues Mercier for the flawless organization of the R&D Workshop, along with Prof. Jacques Savoy, Dr. Anita Sobe and Verónica Estrada.

References

[1] "The zettabyte era: Trends and analysis", Cisco, White Paper, 10 Jun. 2014. [Online]. Available: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/VNI_Hyperconnectivity_WP.html.

[2] P. Mell and T. Grance, "The NIST definition of cloud computing", National Institute of Standards and Technology, Special Publication 800-145, Sep. 2011. [Online]. Available: http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf.

[3] R. P. Goldberg, "Survey of virtual machine research", *IEEE Computer*, vol. 7, no. 6, pp. 34–45, Jun. 1974. DOI: 10.1109/MC.1974.6323581.

[4] S. Soltesz *et al.*, "Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors", in *ACM SIGOPS Operating Systems Review*, ACM, vol. 41, 2007, pp. 275–287. DOI: 10.1145/1272996.1273025.

[5] (14 May 2015). Kernel based virtual machine, [Online]. Available: http://www.linux-kvm.org/ (visited on 25/05/2015).

[6] (2015). Docker, [Online]. Available: https://www.docker.com/ (visited on 09/04/2015).

[7] A. Silberschatz, P. B. Galvin and G. Gagne, *Operating System Concepts*, 9th ed. Wiley, 2013, ch. 8.5, 16, pp. 366–377, 711–740, ISBN: 978-1-118-06333-0.

[8] M. Righini, "Enabling Intel virtualization technology features and benefits", Intel Corporation, White paper, 2010. [Online]. Available: https://www-ssl.intel.com/content/dam/www/public/us/en/documents/white-papers/virtualization-enabling-intel-virtualization-technology-features-and-benefits-paper.pdf (visited on 20/04/2015).

[9] R. Russell, "Virtio: Towards a de-facto standard for virtual I/O devices", *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 95–103, 2008. DOI: 10.1145/1400097.1400108.

[10] M. J. Scheepers, "Virtualization and containerization of application infrastructure: A comparison", *21st Twente Student Conference on IT*, vol. 21, 23 Jun. 2014.

[11] (23 Jan. 2015). Guest support status, KVM, [Online]. Available: http://www.linux-kvm.org/page/Guest_Support_Status (visited on 25/05/2015).

[12] A. Kivity *et al.*, "Kvm: The Linux virtual machine monitor", in *Proceedings of the Linux Symposium*, vol. 1, 2007, pp. 225–230.

[13] (2015). Libvirt: The virtualization API, [Online]. Available: https://libvirt.org/index.html (visited on 25/05/2015).

[14] (2013). Virtual machine manager, [Online]. Available: https://virt-manager.org/ (visited on 25/05/2015).

[15] *chroot(2), OpenBSD manual*, OpenBSD. [Online]. Available: http://www.openbsd.org/cgi-bin/man.cgi/OpenBSD-current/man2/chroot.2 (visited on 21/04/2015).

[16] *jail(8), FreeBSD system manager's manual*, FreeBSD, 4 Aug. 2014. [Online]. Available: https://www.freebsd.org/cgi/man.cgi?query=jail&sektion=8 (visited on 21/04/2015).

[17] P. Menage, P. Jackson and C. Lameter, *cgroups*, Dec. 2014. [Online]. Available: https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt (visited on 25/05/2015).

[18] *namespaces(7), Linux programmer's manual*, 21 Sep. 2014. [Online]. Available: http://man7.org/linux/man-pages/man7/namespaces.7.html (visited on 25/05/2015).

[19] S. Graber, *LXC 1.0: Security features*, 1 Jan. 2014. [Online]. Available: https://www.stgraber.org/2014/01/01/lxc-1-0-security-features/ (visited on 21/04/2015).

[20] W. Felter *et al.*, "An updated performance comparison of virtual machines and Linux containers", IBM Research Report, Jul. 2014.

[21] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment", *Linux Journal*, vol. 2014, no. 239, p. 2, Mar. 2014.

[22] (2015). Understanding Docker. version 1.5, [Online]. Available: https://docs.docker.com/introduction/understanding-docker/ (visited on 13/04/2015).

[23] C. Anderson, "Docker, Software engineering", *IEEE Software*, no. 3, pp. 102–105, May/June 2015. DOI: 10.1109/MS.2015.62.

[24] (21 Apr. 2015). Definitions of a container, Docker, [Online]. Available: https://docs.docker.com/terms/container/ (visited on 26/05/2015).

[25] J. Barbier, *Microsoft demonstrates Docker in build 2015 keynote address*, 29 Apr. 2015. [Online]. Available: https://blog.docker.com/2015/04/microsoft-demonstrates-docker-in-build-2015-keynote-address/ (visited on 30/05/2015).

[26] S. Ayukov, *Shared libraries in Linux: Growing pains or fundamental problem?*, 16 May 1999. [Online]. Available: http://www.ayukov.com/essays/linuxdll.html.

[27] (7 Apr. 2015). Virtual Python environment builder, [Online]. Available: https://pypi.python.org/pypi/virtualenv (visited on 09/04/2015).

[28] R. Morabito, J. Kjallman and M. Komu, "Hypervisors vs. lightweight virtualization: A performance comparison", in *2015 IEEE International Conference on Cloud Engineering (IC2E)*, Mar. 2015, pp. 386–393. DOI: 10.1109/IC2E.2015.74.

[29] B. Russell, *KVM and Docker LXC benchmarking with OpenStack*, 1 May 2014. [Online]. Available: http://bodenr.blogspot.com/2014/05/kvm-and-docker-lxc-benchmarking-with.html (visited on 01/06/2015).

[30] L. Lamers and H. Shah, "Open virtualization format specification", Distributed Management Task Force, DMTF Standard, version 2.1.0, 12 Dec. 2013. [Online]. Available: http://www.dmtf.org/sites/default/files/standards/documents/DSP0243_2.1.0.pdf.

[31] (28 Apr. 2015). Checkpoint/restore in userspace, [Online]. Available: http://criu.org/Main_Page (visited on 31/05/2015).

[32] V. Jujjuri *et al.*, "VirtFS – a virtualization aware file system passthrough", in *Ottawa Linux Symposium (OLS)*, Citeseer, 2010, pp. 109–120.

[33] (22 May 2015). Windows virtio drivers, Fedora Project, [Online]. Available: https://fedoraproject.org/wiki/Windows_Virtio_Drivers (visited on 01/06/2015).

[34] (21 Apr. 2015). Managing data in containers. version 1.6, Docker, [Online]. Available: https://docs.docker.com/userguide/dockervolumes/ (visited on 01/06/2015).

[35] (2015). National vulnerability database, National Institude of Standards and Technology, [Online]. Available: https://nvd.nist.gov (visited on 02/06/2015).

[36] (7 May 2015). Dynos and the dyno manager, Heroku, [Online]. Available: https://devcenter.heroku.com/articles/dynos (visited on 01/06/2015).

[37] C. Bienia, "Benchmarking modern multiprocessors", PhD thesis, Princeton University, Jan. 2011.

[38] P. Marlier, *Powerspy.py*, GitHub repository, 12 Mar. 2015. [Online]. Available: https://github.com/patrickmarlier/powerspy.py (visited on 06/06/2015).