Operation System Level Virtualization: An Efficent Alternative to Hypervisors

Franz Kuntke

Otto-von-Guericke University Magdeburg, Germany

Abstract. Virtualization is a technology that allows the sharing of resources of a physical computer between multiple operating systems. The concept of Virtual Machines was introduced in the 1960's by IBM. One revolutionary step was done by the FreeBSD project in 2000 with introducing FreeBSD jails. With this work, the concept of *operating system level virtualization* as one of two big categories of virtualization technologies was boosted in the BSD community and later became also in the GNU/Linux world more and more popular. Nowadays, companies which act in the field of cloud computing use such approaches for sharing server capabilities in their own company and selling computational power to customers in a highly dynamic way.
In this paper, we present details of *operating system level virtualization* and go into detail for selected implementations.

1 Introduction

Virtualization allows the sharing of resources of a physical computer between multiple operating systems (OS), often called guest OS. Especially cloud computing providers use virtualization extensively to control and manage the resources for different applications. According to Xavier et. al. [12] virtualization can be considered as a foundation of cloud computing.

With the rise of cloud computing in the last decade, studies were published, presenting how traditional virtual machines (VMs) compare to native execution [5, 6]. The results show that a VM has up to 6 times higher application response time compared to native execution. The existence of this performance loss through virtualization leads to approaching this problem by optimization of the software and hardware [13, 8], to reduce the loss in performance.

In the last years, operating system level virtualization (OSLV) became an interesting alternative to VMs, especially in the cloud, due to a much lower overhead [11]. Comparisions between implementations of both traditional VMs and OSLV with specific domain benchmarks were published [3, 7, 11, 12], showing the benefits and disadvantages of both technologies in specific domains.

The contribution of this paper is the following:

– Give a brief introduction to virtualization in general, including advantages and disadvantages.
– Presenting details of OSLV, including popular approaches.

2 Background

Traditional deployment of server software involves the installation and configuration of multiple applications. This process is often very time consuming. Due to possible conflicting requirements of multiple applications the administration of a system needs understanding of how the applications and the OS work. This can lead to higher costs for the system administration than the costs for the software itself [3].

Typical used OSs in the area of Web technolgies, like GNU/Linux, implement neither the *principle of least privilege* nor the *least common mechanism* in a strong manner. Both aspects were introduced by Saltzer et. al. [10]. The former aspect describes that every process and user of a system should operate using the least set of privileges needed to complete the job. The latter aspect describes that every mechanism, that allows sharing components between multiple applications can be a potential information path between users and must be designed with care to be sure it does not compromise security. The stronger the implementation of both aspects in a server environment is, the stronger the security of the server applications is.

Virtualization allows a more economic server administration and can help to improve the overall security of server applications. In the following we will discuss some advantages and disadvantages of virtualization in the area of Web services.

2.1 Advantages of virtualization

Easy distribution of services

Some Web services are required to be available in a large area of the world. A provider of such a service wants to have the ability to distribute this service easily on many places inside the target area, to keep a low latency, resulting in more satisfied customers. Therefore what a provider really wants are *ready to use packages* which can be deployed easily on a physical machine in an additional computer center. These packages must contain every needed file of the software to get the service running. This is the great strength of virtualization: If a complete service is inside a virtualized system, this is already such a package. Usually, just some configuration files needs adaption after deploying a virtualized system on a new host. However the benefit of this advantage depends on the software project structure. If there is just one little piece of software needed to run the whole service, there is no need for packaging the software base. But often a Web service is a jigsaw puzzle, consisting of many pieces, which are sometimes hard to fit together. If you can easily start with a nearly finished jigsaw puzzle the working time and the probability of making mistakes is hardly reduced.

Encapsulating of services

Another positive aspect is the improvement of the security by taking account of the previously mentioned aspects *principle of least privilege* and *least common mechanism*. Virtualization provides the opportunity to encapsulate distinguishable services from each other. If services are encapsulated an attacker is limited to the compromised service. This could avoid a significant data loss, compared to an execution of all needed services on the same OS in the same user space.

2.2 Disadvantages of virtualization

Performance loss

One main disadvantage of virtualization is the loss of performance. Every software inside a host OS is managed by a scheduler, which also applies for virtualization software. This results in a computational overhead. The extent of overhead depends on the specific virtualization technology and implementation.

Need for configuration

To improve the performance of a virtualized service, the configuration of the used virtualization software must be tweaked, as outlined in the work of Soltesz et. al. [11]. This requires an understanding of how the applications inside the guest OS work. Finding the optimal configuration can be a very time consuming process.

2.3 Overview of virtualization technologies

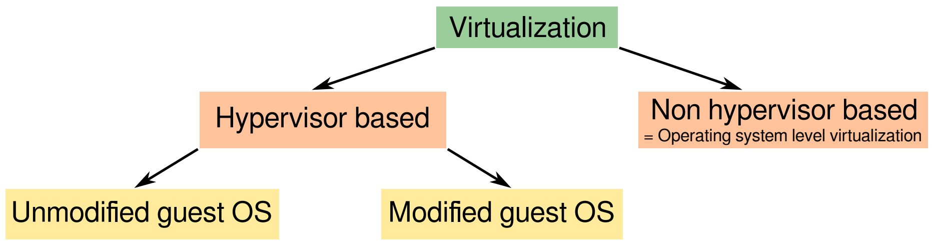

Fig. 1. Categorization of virtualization technologies

As seen in Fig. 1, virtualization technologies fall roughly into two categories: Hypervisor based virtualization and non-hypervisor based virtualization/OSLV.

The hypervisor is the central component of a hypervisor based virtualization. This component assigns parts of the host OS resources, like CPU cores, memory, and network adapter to the guest OS. Therefore it forwards and filters hardware accesses between the guest OS and the host OS. Hypervisor based virtualization is divided into two subcategories, which are different in the way the hypervisor works. This technology is already described, e.g. in the survey paper of Sahoo et. al. [9]. In contrast to hypervisor based virtualization stands OSLV, examined in the following chapter.

3 Operating System Level Virtualization

OSLV covers self-contained execution environments. They contain their own, isolated CPU, memory, block I/O, and network resources, managed by an container engine and share the kernel of the host OS. The result looks like a traditional VM, but without a hypervisor and without the need for a guest OS itself - as seen in Fig. 2. Because a virtualized instance is called container, this category of virtualization is also called container based virtualization.

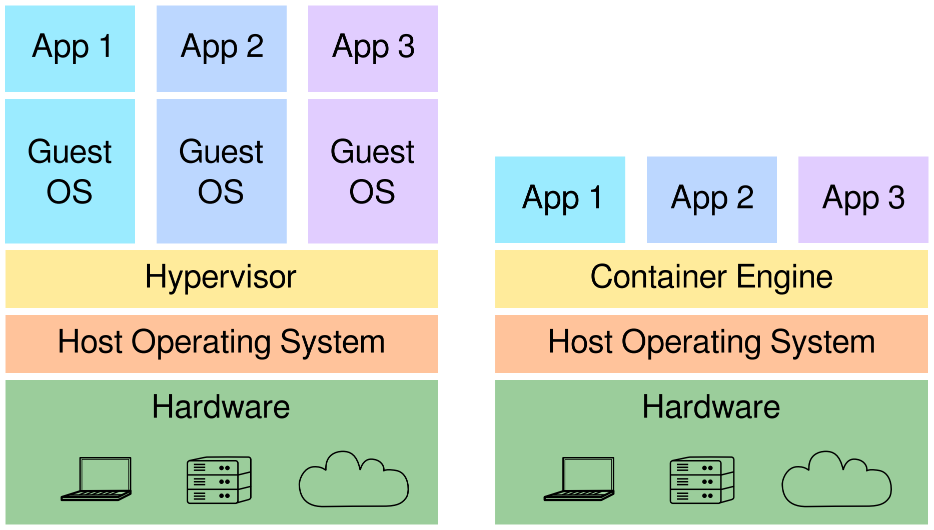

Fig. 2. Schematic comparison of hypervisor based virtualization (left) with OSLV (right)

Sharing the kernel between host OS and guest OS is the essential point of OSLV. There is no need for a complete guest OS, so a container can contain as little as one single process. But sharing the OS kernel forces the applications inside the container to be compatible to the host OS kernel. This can be a drawback for some scenarios, e.g. running continuous integration build nodes of different OS types on one single physical machine. In the Web scenario this is not a big drawback: Many cloud computing frameworks and services, e.g. Hadoop and Amazon EC2 require Linux or recommend using a specific GNU/Linux distribution. Because all available GNU/Linux distributions are based on the same kernel - only differing in specific versions and configurations - you can run virtually all other GNU/Linux distributions in a container on a Linux based host OS. Due to the fact that Linux is the de facto standard in the world of Web, OSLV is a considerable technology in this area. In the following we take a closer look at popular OSLV implementations.

3.1 FreeBSD Jails

FreeBSD jails is the implementation of OSLV in the FreeBSD kernel. It was introduced in 2000 by Kamp and Watson [4]. The implementation of jails relies on restricting access within the jail environment to a well defined subset of the overall host environment. This includes limiting interaction between processes, file access, and network resources. Unlike common chroot, which restricts processes to a particular view of the filesystem, the FreeBSD jails mechanism restricts what a process can do in relation to the rest of the system.

The jail() system call creates an own complete FreeBSD installation. This includes copies of all relevant system binaries, data files, and its own /etc directory. This ensures that all jails are independent and reduces the chance of interference between jails.

With FreeBSD jails come some restrictions [4], for example the following actions are prohibited:

- Modifying the running kernel by direct access and loading kernel modules is prohibited.
- Accessing raw, divert, or routing sockets is prohibited.
- Modifying kernel runtime parameters, such as most sysctl settings, is prohibited.
- Changing securelevel-related file flags is prohibited.
- Accessing network resources not associated with the jail is prohibited.
- Mounting and unmounting file systems is prohibited.
- Creating device nodes is prohibited.
- Modifying any of the network configuration, interfaces, addresses, and routing table is prohibited.

Most of these restrictions are typical for container based virtualization, but the last three mentioned are specific limitations of FreeBSD jails.
Despite this restrictions, FreeBSD jails is a pioneered OSLV developed by the

FreeBSD project. After the release it was used as a state-of-the-art solution for container based virtualization on FreeBSD and allowed providing virtual private server services in an efficient way already in the early 2000s.

3.2 Linux Containers

Linux containers are based on two Linux kernel features: Namespaces and CGroups. With this two techniques every Linux kernel (since version 2.6.14) is able to start a container process. Compared to competitive solutions for GNU/Linux, like Docker, OpenVZ and Linux-VServer, there is no need for installing additional software, besides the required Linux kernel.

The clone() system call allows creating separate instances of previously global namespaces. Following namespaces are provided by the Linux kernel: filesystem, PID, network, user, interprocess communication (IPC), and hostname. For illustration: Each filesystem namespace is based on an own root directory and mount table, similar to a chroot environment, but more powerful. Namespaces itself can be used in different ways, but one common approach is to create an isolated container, which has no access to resources outside the container. A processes running in a container has no idea, that it is not inside a normal GNU/Linux distribution although they use the same kernel of the host OS, with other processes in different namespaces. One feature of Linux container is to allow nested virtualization. So a hierarchically virtualization structure is possible to create, which is not possible with the majority of hypervisor based virtualization technologies.

Linux containers can be divided into two types:

- **System container**:
  - Behaves like a full OS, with an init system, daemons, etc.
- **Application container**
  - Runs just an application

Both types can be useful in different circumstances. An application container does not create memory overhead by a redundant process management, but does not introduce a strict isolated environment as a system container. Communication between containers or between a container and the host OS - which is in fact just a parent namespace - is as fast as normal Linux IPC.

The Linux control groups (cgroups) feature is used to group processes and manage their summed up resource consumption. This is often used to limit the memory and/or CPU usage of a container. An unsolved problem of container resource management is that applications inside a container are not aware of their resource limits. For example, a process can see all CPUs of the system, even in the case it is just allowed to use a subset of them. This can have a big impact, if an application tries to tune itself automatically, based on the visible system specification.

3.3 Docker

Docker is one of many tools for managing Linux containers, besides LXC, systemd-nspawn, lmctfy, and Warden. It was introduced in 2013 by dotCloud, Inc (now Docker, Inc). The goal was to provide an easy to use interface for deployment of applications inside OSLV containers.

One advantage of Docker over its competitors is layered filesystem images, usually implemented by AUFS (Another UnionFS). AUFS provides a layered stack of filesystems and allows the reuse of these layers between containers. This reduces space usage and simplifies filesystem management. One single guest OS image can be used as a basis for many containers while allowing each container to have its own overlay of modified files. As examined by Felter et. al. [3], Docker container images require less disk space and I/O than equivalent VM disk images. This leads to faster deployment in the cloud computing area since images usually have to be copied over the network to local disk before the virtualized system can start.

With the first releases Docker was based on Linux containers. Since version 0.9, released in 2014, Docker has an own library: libcontainer. This allows Docker a more direct communication with Linux kernels and potential adjustment to more host OSs, besides GNU/Linux. One convenient aspect of docker is the integration of Docker Hub, a public storage for Docker container images. Just by calling *docker pull* a complete image can be downloaded. The available image spectrum reaches from GNU/Linux distributions, like Debian, over server components, like nginx, to complete ready to use Web applications, like Wordpress. Due the high feature set and ease to use of Docker, it has rapidly become the standard management tool and image format for OSLV. The deployment of Docker containers is possible by many cloud infrastructure tools, including Amazon Web Services, Microsoft Azure, Google Cloud Platform, OpenStack, and VMware vSphere Integrated Containers.
In June 2016, Microsoft announced that Docker could be used natively on Windows 10 with Hyper-V Containers, to build, ship and run containers utilizing the Windows Server 2016 Technical Preview 5 Nano Server container OS image [1].

4 Comparison

As stated in the introduction, some comparisons between implementations of both hypervisor based virtualization and OSLV were published [3, 11, 12].
The work of Felter et. al. [3], released in 2015 is the most current comparison. This work compares Kernel Virtual Machine (KVM) as a hypervisor based virtualization with Docker as an OSLV. The authors compared the performance of both implementations under scenarios where one or more hardware resources are fully utilized, including PXZ, Linpack, Stream and RandomAccess.

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

Table 1. Results for PXZ, Linpack, Stream, and RandomAccess. Each data point is the arithmetic mean of ten runs. Departure from native execution is show within parentheses "()", the standard deviation within square brackets "[]". From Felter et. al. [3]

Table 1 presents averaged results of ten runs of each scenario.
PXZ is a parallel lossless data compression utility, implementing the LZMA algorithm. As most compression algorithms the speed depends mostly on the CPU. As expected, native and Docker performance are very similar while KVM is slower (-22%).
Linpack solves a dense system of linear equations using an algorithm that carries out LU factorization with partial pivoting. The used Linpack binary is highly adaptive and optimizes itself based on both the available floating point resources, as well as the cache topology of the system. By default, KVM does not expose topology information to guest VMs. This results in the bad results (-17%) of KVM-untuned. Tuning KVM to expose the underlying CPU and cache topology increases performance nearly to par with native execution. Docker is as expected as fast as native execution.
The RandomAccess benchmark is designed to stress random memory performance. It initializes a large section of memory as a working set. Random 8-byte words in this memory section are read, modified and written back. RandomAccess typifies the behavior of workloads with large data sets and minimal computation such as working with in-memory databases. This benchmarks creates the same overhead on both virtualized and non-virtualized systems.
The Stream benchmark is a program that measures sustainable memory bandwidth while performing simple operations on vectors. Performance is therefore limited by memory bandwidth. The benchmark has four components: Copy, Scale, Add and Triad (sorted by ascending computational intensity). Native Linux execution, Docker and KVM are almost identical, due to the fact that memory prefetching algorithms are performed mostly on the underlying hardware.

As we can see, Docker as an OSLV is in those benchmarks comparable or superior to KVM as a hypervisor based virtualization. These findings appear to be consistent with the results of other comparisons [11, 12]. Especially I/O operations tends to be a bottleneck of hypervisor based virtualization, while it has virtually no impact on OSLV implementations.

5 Conclusions

Virtualization is a technology, that is part of many enterprise scenarios. Especially the cloud computing area requires such approaches for an economic and secure deployment of services. The much lower overhead compared to hypervisor based virtualization has led to a wide distribution of OSLV in this area.
A major drawback of OSLV is the need for compatibility between the virtualized software and the host OS kernel. Other drawbacks depends mainly on the specific implementation and reaches from a difficult setup to limitations of the container capabilities. For most Unix based systems exists sophisticated solutions, providing container based virtualization. One of the most convenient solutions is Docker, providing ready to use images for default cases. Since June 2016, Docker could be used natively on Windows 10, which allows comparing OSLV on different host OSs.
We expect a widely more distribution of OSLV besides cloud computing. Trends show that the application of OSLV is investigated in more areas include high performance computing and operating system design, like CoreOS [2].

References

[1] "Announcing Windows 10 Insider Preview Build 14361" [Internet]. https://blogs.windows.com/windowsexperience/2016/06/08/announcing-windows-10-insider-preview-build-14361/#6UOEcFZwlppBOIDM.99, Accessed: 2016-06-22

[2] "CoreOS" [Internet]. http://coreos.com, Accessed: 2016-06-30

[3] Felter, W., Ferreira, A., Rajamony, R., Rubio, J.: An updated performance comparison of virtual machines and linux containers. In: Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On. pp. 171–172. IEEE (2015)

[4] Kamp, P.H., Watson, R.N.: Jails: Confining the omnipotent root. In: Proceedings of the 2nd International SANE Conference. vol. 43, p. 116 (2000)

[5] Matthews, J.N., Hu, W., Hapuarachchi, M., Deshane, T., Dimatos, D., Hamilton, G., McCabe, M., Owens, J.: Quantifying the performance isolation properties of virtualization systems. In: Proceedings of the 2007 workshop on Experimental computer science. p. 6. ACM (2007)

[6] Padala, P., Zhu, X., Wang, Z., Singhal, S., Shin, K.G., et al.: Performance evaluation of virtualization technologies for server consolidation. HP Labs Tec. Report (2007)

[7] Rathore, M.S., Hidell, M., Sjödin, P.: Kvm vs. lxc: comparing performance and isolation of hardware-assisted virtual routers. American Journal of Networks and Communications 2(4), 88–96 (2013)

[8] Reed, E.R.: Virtual machine use and optimization of hardware configurations (Jan 20 2009), uS Patent 7,480,773

[9] Sahoo, J., Mohapatra, S., Lath, R.: Virtualization: A survey on concepts, taxonomy and associated security issues. In: Computer and Network Technology (ICCNT), 2010 Second International Conference on. pp. 222–226. IEEE (2010)

[10] Saltzer, J.H., Schroeder, M.D.: The protection of information in computer systems. Proceedings of the IEEE 63(9), 1278–1308 (1975)

[11] Soltesz, S., Pötzl, H., Fiuczynski, M.E., Bavier, A., Peterson, L.: Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In: ACM SIGOPS Operating Systems Review. vol. 41, pp. 275–287. ACM (2007)

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

[13] Zhong, A., Jin, H., Wu, S., Shi, X., Gen, W.: Optimizing xen hypervisor by using lock-aware scheduling. In: Cloud and Green Computing (CGC), 2012 Second International Conference on. pp. 31–38. IEEE (2012)