H-0330 (HAI1801-001) January 3, 2018
Computer Science

IBM Research Report

A Comparison of Virtualization Technologies for Use in Cloud Data Centers

Joel Nider
IBM Research Division
Haifa Research Laboratory
Mt. Carmel 31905
Haifa, Israel

A Comparison of Virtualization Technologies for Use in Cloud Data Centers

Joel Nider
IBM Research Haifa
joeln@il.ibm.com

abstract - Cloud data center operators are constantly facing the challenge of making their clouds run more efficiently. Various forms of virtualization have been employed over the years to squeeze out as much performance as possible from the data center - a very expensive resource. As is often the case when new technologies are introduced, a certain amount of confusion ensues as developers and consumers alike try to figure out what are the benefits and what are the costs associated with moving to a new technology such as containers or unikernels. This paper is intended to provide some basic facts regarding the design and usage of the various technologies to outline their relative strengths and weaknesses.

INTRODUCTION

Virtualization is the underpinning technology that makes any cloud data center successful. Virtualization technologies are constantly improving as new techniques are being discovered and introduced. To properly compare the various technologies, it is important to understand why each one was developed, and what its goals are. We will start by briefly explaining what virtualization is, the different types and how they work. We will then compare two types (unikernels and containers) across several points to get a feel for what each one does best. The goal of isolating users and their programs has not changed over the years, while the implementation details have changed because of additional requirements from the users such as direct access to I/O devices and rapid deployment of virtual machines in a cloud environment. It is important to understand some of the history behind these technologies to understand what we have today, and where we are headed in the future.

HISTORY

Machine virtualization was first implemented on the IBM 370 mainframe as a means to allow several users to share the machine as if each was the only user present. The machine was very expensive, and it was important to maximize its use by allowing multiple programmers to use it concurrently. Therefore, one of the main goals of virtualization at that time was for the user to be completely unaware that the program was running on a virtual machine, rather than a physical machine (bare metal). The hypervisor (sometimes called a *virtual machine monitor* or *VMM*) is the piece of software required to allocate and manage the hardware resources provided by the host and present multiple guest environments (the *virtual machine* or *VM* for short). It is important to realize that at that time, there was no separation between operating system and application, so it was necessary for programs to access hardware directly. For that reason, it was important that the users were completely isolated from each other through the hypervisor.

Around the same time, the concept of a *process* was developed as an operating

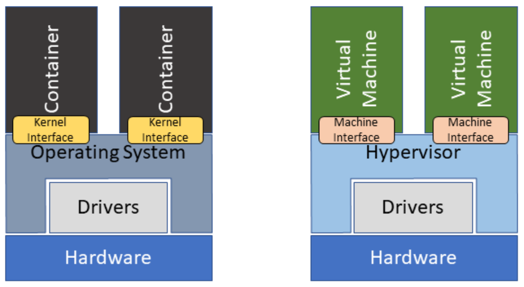

Figure 1 - Operating system virtualization (left) vs machine virtualization (right)

system abstraction of an executing context. In other words, programs needed isolation from each other, regardless of which user was running them, and this concept was supported by features in the hardware such as virtual memory and paged memory. Processes can be considered a form of virtualization, in that they allow the operating system to share the machine's limited resources among multiple programs simultaneously without these programs being aware of or actively taking part in the scheduling.

Machine Virtualization

At first, the hypervisor was implemented completely in software and relied on a set of tricks to work around the fact that the x86 hardware was never designed to support such behaviour. Sometimes these tricks were quite costly (thousands of machine cycles) and to improve performance, the operating system was purposefully modified to communicate with the hypervisor directly to avoid these costs. This is known as *paravirtualization*, and it means the guest software can no longer work in an identical manner when running on a bare metal machine since the guest operating system is now 'aware' of the hypervisor. Once chip vendors implemented extensions to their hardware that support virtualization, these tricks in the hypervisor and modifications to the guest operating systems were no longer needed. However, the concept of paravirtualization remains useful when applied to the problem of improving I/O performance.

Operating System Virtualization

In some cases, machine virtualization is not necessary, or not possible. Operating system virtualization allows the machine to be shared in a different way - by allowing multiple instances of an operating environment to run simultaneously. Sharing resources at the operating system level is known by many names. One of the first such systems became known as BSD Jails [3], but there are many other similar implementations such as Cells [4] (on Android), LXC [5] (on Linux) and LPAR [6] (on AIX). This kind of virtualization does not use a hypervisor. Instead, it relies on the kernel (the protected part of the operating system) to provide some basic services (the kernel interface) that can be used to create different "personalities" of the operating system. What is important to note is that the kernel provides all of the services required by the system partition, which runs in user space. This is much less flexible than machine virtualization, which allows various operating systems to run concurrently, and is only constrained by the instruction set of the CPU and the available peripherals. In operating system virtualization, the partitions must understand the kernel's ABI (application binary interface) - in other words, be compiled to run on top of that particular kernel. In general, these partitions are implemented as a set of one or more processes which are restricted from accessing general system resources such as file systems, hardware devices and user accounts. Instead, they are given their own personalized view of the operating system.

Now that we have covered the basics, let us try to compare these two contrasting approaches to sharing a machine's resources to better understand their relative strengths and weaknesses.

Machine Interface

Machine virtualization has proven its effectiveness in solving several common problems such as running multiple operating systems simultaneously on a single machine, allowing multiple untrusted users to share the resources of a single machine, and increasing the utilization of a machine's resources. What is important to note about machine virtualization is that there are two sides that must cooperate to ensure performance - the hypervisor on the host side, and the guest operating system inside the virtual machine.

It is important to emphasize that the hypervisor is responsible for the isolation between the guest and the host, as well as isolation between guests. The hypervisor defines a rigid interface by which the guest must communicate with the host, and may employ special features in the hardware to enforce the isolation. Such features generally provide protection to virtual machines from I/O hardware, and protection to I/O hardware from virtual machines. The original goal of the hypervisor was to provide a virtual machine environment that is indistinguishable from its physical counterpart, meaning the guest OS should not have to be aware that it is running in a virtual environment. However, virtual machines and hypervisors may end up doing a lot of unnecessary work if they are not aware of each other's capabilities and requirements. By carefully looking at the interface between the two components, sometimes we can recognize more efficient ways of doing things which can gain huge performance improvements, but at the cost of breaking compatibility. This is a tradeoff that each cloud operator must consider carefully.

Unikernels

If we were to design an operating system that was to run solely as a guest (i.e. only ever inside a VM) we could make many assumptions about the generality that the operating system must provide. By imposing constraints such as the number of processes that must run concurrently and I/O interfaces that are available, it becomes possible to design a system that is more efficient than a general purpose one which cannot make those assumptions. This is exactly the thought process that the unikernel approach advocates. Some examples of unikernels are OSv [1] and MirageOS [2].

Unikernels are built using a library OS, which means only the operating system components which are required to support the application are included in the final image. This reduces the image size, as unneeded components are excluded from the system. It also potentially reduces attack surface, because there are fewer components that could be compromised during an attack, making unikernels more secure than full VMs (at least in theory).

Performance

Linux containers such as Docker have recently started to gain popularity as a form of "lightweight virtualization", implying that they don't incur the same performance and resource overhead associated with machine virtualization. While that is generally true, it is important to understand why, and what is lost in return. If we compare the performance of a given application running on a bare metal machine (no virtualization) to the same application running inside a VM or a container, the bare metal version will almost invariably be faster. CPU instructions for both VMs and containers run directly on the physical CPU, and memory accesses both use the physical memory through the virtual memory subsystem of the processor. However, there are some important differences that cause virtualized applications to run slower such as how the virtual memory hardware works for virtual machines.

VIRTUAL MEMORY

Hypervisors expect a virtual machine to come with its own operating system, and manage its own virtual memory. But at the end of the day, the memory needs to be managed by the hypervisor, because the hypervisor is responsible for allocating the machine memory among the multiple guests, and ensuring the protection mechanisms are in place. That means every time a guest accesses a virtual address, it must be translated twice before getting a physical memory address that the CPU can use.

Since containers run as host processes, they benefit from the same virtual memory addressing scheme as other host processes, so each memory address is only translated once per access. Additional hardware inside the processor makes these lookups very fast, and caching (in both cases) often eliminates lookups completely, but on average, there are more memory translations performed in a VM than in a container.

STARTUP TIME

It is commonly believed that containers start faster than virtual machines, which is why they should be preferred for short running jobs. This follows from the thinking that starting a single (or small set) of processes in a running system is faster than starting up an entire operating system, including several processes. However, a recent study from IBM Research [7] suggests that this is not always true. The study shows that the overhead is actually an artifact of the hypervisor implementation, and virtual machines can actually start faster than containers by throwing away certain features in the hypervisor that won't be used, or by reimplementing them in a different way. This leads to a loss of generality - the hypervisor would no longer be suitable for running all kinds of virtual machines, but does provide an impressive performance boost which is critical for certain workloads such as serverless computing. When we consider that unikernels do not run generic workloads - that is, the application of the unikernel VM is known when it is built - we could start the hypervisor to support only a subset of features, thus reducing startup time.

In addition, the unikernel itself has much less work to do than a general purpose operating system when starting. Based on the assumption that the unikernel is running in a virtual environment, it has fewer devices and services that must be initialized, which also contribute to the faster startup time.

I/O

When an operating system performs I/O on behalf of an application in a bare metal system (such as reading or writing files to a disk or sending packets on the network), it does so by calling functions in a driver which manipulates the necessary hardware device (such as a disk or NIC). The driver does so by reading and writing small pieces of information through registers which tells the hardware what to do. Registers are a very convenient way to talk to hardware, and is relatively efficient. When the operating system wishes to perform I/O inside a VM, it must also talk to a device. However, the device is generally not hardware, but rather a device that is emulated by the hypervisor. This is how the hypervisor helps trick the guest operating system into thinking it is running on bare metal. That means the guest operating system must read and write the same registers that it would use on a real device. The problem is that we don't have a real device - we are emulating it in software! Here, the concept of paravirtualization comes in handy. If we relax the constraint that the guest doesn't know it is actually running in a virtual machine, we can provide a new, much more efficient interface called paravirtual

I/O. This requires a new driver for each guest operating system for each kind of virtual device. Some examples of this kind of interface are VMXNet [8] on VMWare, and virtio [9] on KVM.

In a unikernel, the software layers to perform I/O are much simpler than in a general purpose VM. Since there is only a single process, file systems and driver code don't need to deal with the same level of complexity, and have a simplified model for locking and concurrency. This can provide a huge boost when performing I/O operations. Even larger performance gains can be realized by breaking the standard sockets API. While this requires some modifications to application programs, in many cases the benefits can outweigh the effort.

Containers don't have the problem of having to perform all I/O through a hypervisor. Instead, I/O is performed through system calls to the kernel, rather than through a hypervisor. However, since containers run in a general-purpose operating system (which is necessarily multi-process), I/O must be vetted through a complicated I/O subsystem, which can hurt performance. Also, I/O performance can be potentially limited by the kernel for fairness through mechanisms such as cgroups.

APPLICATION DEPLOYMENT

Companies such as Docker market the concept of containers as a simple deployment model for DevOps applications, where release cycles are short, and continuous deployment is encouraged. In such a work environment, it is important to have precise control over the deployment environment where applications may be dependent on hundreds of different packages and components, and the ability to quickly change any particular component is critical when working on a production system.

For unikernels, there are deployment systems which accomplish the same goal, but in a slightly different manner. For example, the Capstan [10] system enables packaging any application with the OSv unikernel to be deployed as a virtual machine.

CONCLUSION

Both containers and unikernels are modern methods for effectively deploying your application to the cloud. They both offer easy deployment models, good performance and security features. Selecting which method will work best for a particular application depends on understanding the demands of that application, and understanding the underlying virtualization technologies to be able to make an educated decision.

ACKNOWLEDGEMENTS

This project has received funding from the *European Union's Horizon 2020 research and innovation programme* under grant agreement No 645402.

BIBLIOGRAPHY

[1] Linux Foundation, "OSv: The Open Source Cloud Operating System That is Not Linux," The Linux Foundation, 14 11 2013. [Online]. Available: https://www.linuxfoundation.org/blog/osv-the-open-source-cloud-operating-system-that-is-not-linux/.

[2] MirageOS, "A programming framework for building type-safe, modular systems," MirageOS, 2017. [Online]. Available: https://mirage.io/.

[3] P.-H. Kamp and R. Watson, "Building Systems to Be Shared, Securely," in *Building Systems to Be Shared, Securely*, New York, NY, USA, 2004.

[4] C. Dall, J. Andrus, A. Van't Hof, O. Laadan and J. Nieh, "The Design, Implementation, and Evaluation of Cells: A Virtual

Smartphone Architecture," in *ACM Transactions on Computer Systems*, New York, NY, USA, 2012.

[5] C. C. C. B. N. SA, "Linux Containers," LXC, 2017. [Online]. Available: https://linuxcontainers.org/lxc/introduction/.

[6] B. Buros, "An Introduction to LPAR," IBM Systems Magazine, 10 2002. [Online]. Available: http://ibmsystemsmag.com/aix/administrator/lpar/an-introduction-to-lpar/.

[7] R. Koller and D. Williams, "Will Serverless End the Dominance of Linux in the Cloud?," in *HotOS '17*, Whistler, BC, Canada, 2017.

[8] VMware, "Performance Evaluation of VMXNET3 Virtual Network Device," VMware Inc., 08 2009. [Online]. Available: https://www.vmware.com/techpapers/2009/performance-evaluation-of-vmxnet3-virtual-network-10065.html.

[9] R. Russell, "Virtio: Towards a De-facto Standard for Virtual I/O Devices," *SIGOPS Oper. Syst. Rev.*, vol. 42, pp. 95--103, 2008.

[10] OSv, "Rapid VM builds - Capstan," Cloudius, 2017. [Online]. Available: http://osv.io/capstan/.

[11] M. Rosenblum, "Vmware's virtual platform: A virtual machine monitor for commodity pcs," in *Hot Chips 11*, 1999.

[12] IBM Knowledge Center, "IBM Workload Partitions for AIX," IBM, 2015. [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/ssw_aix_72/com.ibm.aix.wpar/wpar-kickoff.htm.