Virtualization and Implications for Kernel Design

Matthew Prashker
April 2023

I. Introduction

Virtualization in computer systems, at a high level, refers to emulating some physical resource in software and exposing interfaces to this emulated resource. This notion is very familiar from the domain of ordinary operating systems - indeed, the operating system itself can be viewed as virtualizing memory and the cpu, among other resources, so that processes running on top of it have a clean interface to these resources and the illusion that they have exclusive access to these resources. In the design of operating systems, it is natural to ask why the virtualization must be at the level of processes running on an underlying kernel. This survey will explore another level of the stack to provide virtualization at, namely between the hardware and the kernel itself - the piece of software responsible for providing this level of virtualization is referred to as a *hypervisor*.

Strong motivation to introduce hypervisors comes from the modern *cloud* environment. A basic unit of abstraction provided by modern cloud services is a virtual machine - an entirely software based construct designed to emulate a physical machine. If many of these virtual machine instances are to be run on a single physical server, there must exist mechanisms to intelligently multiplex them. For instance, much like how an operating system provides some level of performance isolation between processes, in order to create a robust cloud infrastructure, it needs to be the case that the performance of an application is not affected by the performance of another application running on the same hardware. Other services provided by a hypervisor directly translate to a more robust cloud environment.

Having introduced this extra layer of abstraction, this survey will then examine how the design of kernels running on top of a hypervisor may differ from the design of kernels meant to run directly on the hardware. Another motivation for introducing an extra layer of abstraction between the operating system and the hardware is to reduce complexity. Indeed, the hypervisor can be made much more lightweight than a typical kernel, and this complexity can be pushed up to the level of the guest operating system. This survey will be organized as follows. In section II we will examine techniques and performance results of real hypervisor systems spanning different design choices. In section III we will examine how

II. Approaches to Virtualization

This section will explore approaches to virtualize hardware in order to multiplex between many different guest operating systems, which are in turn multiplexing between running many different processes, all on a single machine. There are two high-level approaches to designing a hypervisors (the software responsible for the multiplexing), referred to as *type 1* and *type 2* hypervisors - the difference depending on whether the hypervisor itself runs on a host OS on top of the hardware.

We will explore three implementations of hypervisors, the Xen hypervisor of type 1 in sub-section II-A, and VMWare's Workstation and ESX Server, of type 2 and 1, respectively, in section II-B. Regardless of the type of hypervisor, there are some common *desiderata* that motivate the design decisions of hypervisors, and we explicitly list them here for reference.

Hypervisor Desiderata A - **Performance Isolation** - A crucial service hypervisors must provide is to isolate the performace between different virtual machine's it is running - that is, the performance of one VM should not be affected by the resources being consumed by another VM. Although a typical operating system does provide some level of isolation between applications, it is often leaky - i.e. the memory usage or number of io ops made by one application can affect the performance of other applications. A common way this manifests itself it through so called *cache pressure* - if two running processes are touching different parts of memory, then after every context switch, the process will start with a cold cache.

Hypervisor Desiderata B - **Safety Isolation** - The memory of one virtual machine should be completely isolated from the memory of another virtual machine. This is very similar to the role a typical OS plays in providing each process with its own virtual address space. Furthermore, one virtual machine should not be able to use any system resources on a behalf of another. This latter property can only be achieved by having the hypervisor isolate control of resources, such as the disk and device drivers, from all guest OS's.

Hypervisor Desiderata C - **Near Native Performance** - The extra level of indirection provided by hypervisors should not increase the latency of applications running on them compared to the same application running on a standard OS. Similarly, applications should not need to consume more resources simply because they are running on a hypervisor.

A. Xen and Type 1 Hypervisors

The Xen system, presented in [1], was one of the first systems to multiplex guest OSs. It is an example of a *type 1 hypervisor*, which runs directly on the hardware without a hosted OS. Xen takes the process level virtualization of a typical operating system and adds an extra layer of abstraction by multiplexing the operating system itself. Usually, in order for application to interface with resources from the underlying hardware, they trap into the operating system by making system calls. Analogously, guest operating systems interface with the underlying hardware by trapping into the hypervisor by making so-called *hypercalls*. Instead of having the hypervisor expose an interface identical to the underlying hardware to the guest operating systems, Xen takes an approach called paravirtualization. Here, the idea is that the source code of the guest operating system may have to be changed slightly in order to interface with the hypervisor, but this allows the hypervisor to more seamlessly interface with multiple guest OSes. The rest of this section will be divided into two parts - in II-A1 we will look at the techniques Xen (and other type 1 hypervisors) use to multiplex resources among guest OSs, and in II-A2 we will examine the performance results of Xen compared to native OS performance.

1) Techniques: One of the most important design decisions of a hypervisor from the point of view of performance is how the hypervisor communicates with each guest OS. Xen facilitates this communication through an asynchronous IO ring The ring itself is a piece of shared memory between each guest OS and Xen, and both the OS and Xen maintain two pointers, a producer and consumer pointer, into the ring. Data itself is not kept in the ring, but rather handles to the data, similar to unix file descriptors, are kept in the ring. The guest OS then enqueues tasks, such as reading from a file or sending a packet over the network, into the queue, and Xen dequeues tasks and executes them. We note that this approach is very similar to io-uring, a recent addition to the linux kernel which allows for asynchronous, in particular non-blocking, IO requests from user space applications to the kernel.

A naive approach to virtualize memory would be to allow the hypervisor to allocate pages to each running OS, and the OS can manage the pages between different processes as if it was the only OS running on the machine. However, this does not provide satisfactory isolation between the different OSes and therefore violates *desiderata B*. Instead, the hypervisor (or VMM) maintains its own page table for the different OSes, mapping virtual page numbers from the point of view of each OS to the page numbers allocated to that VM from the point of view of the hypervisor - this hypervisor level page table is often referred to as a *shadow page table*. Like an ordinary page table in a traditional operating systems, each page table entry is associated with some privileges. This extra level of indirection of course leads to extra overhead - every time the page table of a process running on some domain on a hypervisor needs to change its page tables, it not only needs to consult the OS, but it also needs to consult the underlying hypervisor. This tradeoff between Near Native Performance (*desiderata C*) and Safety Isolation (*desiderata B*) is ubiquitous in the design of hypervisors. In order to get closer to near native performance, some hardware vendors have added support to their chips for exactly this purpose - i.e. Intel has Extended Page Tables which allow translation between host virtual address and guest virtual address.

We will explore how the hypervisor is able to virtualize I/O. Note that one advantage of this approach is that the underlying disk hardware may be upgraded, without changing any of the application code. Indeed, the hypervisor can simply dynamically translate io requests made by the application code into requests for the new hardware. Simarly, for networking code, the hypervisor can now play the role of a virtual switch by rerouting packets to the appropriate virtual machine running on top of it.

2) Performance: A shocking result of [1] is that adding this extra layer of software indirection between the kernel and the hardware does not too dramatically decrease performance of user space applications. This is suprising in light of the extra indirection required for an application to execute a system call. To do so, the user level application must first trap into the kernel, but before the kernel can execute privileged instructions, it must inform the hypervisor, which has permission to call the privleged OS trap handler. When the trap handler is finished it must again consult the hypervisor before returning to the user-mode level. The tradeoff is ultimately one between performance of the overall system and security in the form of isolation between different guest OS's running the on the hypervisor. Furthermore, VMWare workstation, to be explored in II-B, had 74 percent of the throughput, reflecting the even greater level of indirection present in type 2 hypervisors.

To benchmark the overhead, [1] added modifications to the Linux Kernel to create XenLinux so that it was capable of running on top of the Xen hypervisor. They then ran an instance of PostgreSQL and conducted an OLTP benchmark, which required many synchronous disk IO operations. The database running on the XenLinux os had 95 percent of the throughput, as measured in transactions per second, as the database running on linux on bare metal. It would be interesting to see a comparison on a benchmark using an OLAP workload - XenLinux might even perform closer to linux on bare metal as the reads and writes would be more sequential so their may be less context switching between user - kernel - hypervisor modes.

A second class of experiments conducted in [1] aimed to determine the overhead of running multiple VMs on a single hypervisor versus running many processes on a single OS. The benchmark used, *SPEC WEB99*, involved running a web-server which responds to multiple HTTP requests as well as writing logs of these requests to disk. Thus the benchmark tests both network performance and file system performance, as well as how efficiently the system can schedule tasks, giving a better overall signal of performance as opposed to repeatedly stressing just one aspect of the system. The experiment was conducted with $1, 2, 4, 8,$ and 16 instances of the web server. The suprising result was that as the number of web-servers increased, the performance of the servers on the Xen VMs, as measured in requests served per second, approached the performance of the servers running on bare Linux. We note that each web server on the hypervisor was run in its own Xen VM instance, while the web servers on bare Linux were run as concurrent processes.

B. VMWare Workstation and ESX Server

This section will examine two hypervisors, both created by VMWare, of type 2 and 1, to examine some of the trade offs and performance characteristics of the two approaches. Both of these hypervisors were designed to specifically emulate the x86 instruction set.

1) Workstation and Type 2 Hypervisors: Recall that a type 2 hypervisor runs on top of an actual OS, referred to as the host OS, and multiplexes other OS's on top of its, referred to as guest OS's. One motivation for VMWare to create the type 2 hypervisor Workstation in [5] was that emulating the complete x86 instruction set in software, the architecture VMWare was targeting, would introduce new substantial complexity to the entire hardware-software stack. Indeed, x86 is a CISC architecture and is notoriously complex. Furthermore, it is not possible to directly emulate the entire instruction set from a less privileged. One potential advantage of a type 2 hypervisor, from the point of view of application development, is that application code can run unmodified, as the hypervisor can pass through system calls made by the application to the underlying kernel. However, because the hypervisor is itself running on an OS, there are more layers of abstraction in the entire stack than would be present on a type 1 hypervisor, which will make achieving *desiderata C* more challenging.

The architecture of VMWare workstation described in [5], referred to as a *hosted architecture*, revolves around three modules called the $VMDriver$, the $VMApp$, and the VMM. The $VMDriver$ is loaded into the host OS and is responsible for handling certain privileged instructions delegated to it by the VMM - it can be thought of as a special type of device driver. When the VMM needs to execute a privileged instruction such as performing IO or receives a hardware interrupt, it performs a *world switch* to the host world and gives control back to the host OS, which actually performs the requested IO. It is important to note that although the VMM runs in kernel mode, it does not run in the context of the host OS, and is the module responsible for actually performing the multiplexing of resources a hypervisor provides - in particular, there is one VMM instance for each VM.

One benefit of the *hosted architecture* approach is that it allows the overall hypervisor to be much more hardware independent than a type 1 hypervisor because it is able to directly leverage the host OS to perform any hardware dependent instructions. However, this comes at the expense of the VMM having less control over the hardware resources because it is being treated just as another user level application from the point of view of the host OS.

To analyze the performance of *Workstation*, [5] examined the latency of sending network packets through a VM running on it. To better appreciate the performance statistics, we will walk through the stack of how a guest OS on a VM sends a packet through a virtual network card (NIC). To emulate a virtual NIC, there is a special device driver inside the host OS called the VMNet driver. When the guest OS wants to send a packet, the VMM will delegate this instruction to the $VMDriver$, which will in turn delegate this instruction to the $VMApp$ running in the context of the host OS. The $VMApp$ will then make the appropriate system call to the $VMNet$ driver which then passes the packet to the physical NIC. These extra layers of indirection will of course add extra overhead to the throughput of sending packets. Notably, there is an extra memory copy from the VMM world to the Host World, and performing a world switch from the VMM into the host OS is more expensive than a typical trap of a process into the kernel.

The experiment [5] performed involved sustained transmission of TCP packets. The experiment was set up so that the same bytes were used in each packet payload, which avoids paging and disk I/O related performance impacts. On a typical networking stack of a single OS running directly on the hardware, this would be an I/O bound task. However, because of the amount of overhead involved in switching between VMM and host world, this task actually wound up being CPU bound. On average, sending a single packet to approximately $31\mu s$. Of that time, 77 percent of it was spent inside of the VMM, and approximately $9\mu s$ was spent context switching back and forth between the Host and VMM worlds (some of this latter chunk was spent in the host world, and other in the VMM). Of the time spent in the VMM, a vast majority $(17.5\mu s)$ was spend inside the VMDriver. This overhead is quite serious, as passing through the VMM must be done on top of using the network stack of the host operating system, which would be the bulk of the work without an intervening hypervisor. The paper [5] does unfortunately not give direct comparisons with the latency of sending packets in this latter setup. Although the *hosted architecture* approach to hypervisors has a number of advantages, such as hardware independence (running a VM on a new machine can take advantage of device drivers, for example, in the host OS), these results show that it is quite far from achieving the near native performance of *Desiderata C*.

C. ESX Server and Type 1 Hypervisors

After the introduction of VMWare workstation, WMWare later released their ESX server, which aims to emulate an x86 cpu directly - i.e. acts as a type 1 hypervisor. Unlike the hosted architecture of *Workstation*, there was no longer any host operating system that the VMM could rely on to execute certain privileged instructions - part of what made this possible was increasing hardware support from vendors for virtualization. As noted in [6], new hardware capabilities also allowed for improved performance - a large reason why ESX server VM's can approach near native speeds is that the VMM module takes advantage of dynamic binary translation, which requires the VMM module to configure the hardware in certain custom ways, such as adding support for shadow page tables. The basic architecture of the ESX consists of several VMMs, each running on top of the ESX hypervisor, which itself is running directly on the hardware. The VMM, which aims to provide an emulated x86 environment, uses much of the same code as the VMM in *workstation*, with the hypervisor now playing the role of the host OS.

The novel techniques introduced in [7] involve how to efficiently share memory between VMs. The techniques used to virtualize other system aspects such as I/O are very similar to Xen - because the hypervisor itself manages the physical device drivers, there is no need to perform world switches in this hypervisor, which caused most of the overhead in the virtualization of I/O by workstation. Thus, in the rest of II-C, we will examine the techniques ESX server uses to manage memory and how this affects overall performance.

1) Techniques and Performance: The basic data structures used by ESX server to virtualize memory, similar to Xen, is maintaining for each VM a mapping from physical page numbers to machine page numbers. Recall that physical page numbers are the page numbers the VM believes are physical but are in fact virtual, while the machine page numbers refer to real physical pages on the machine. For optimization purposes, the ESX server also maintains its own *shadow page tables* which contain virtual to machine address translations. This improves performance because the TLB will now contain mapping from virtual addresses to machine addresses instead of physical addresses, which avoids the need for an extra translation from physical to machine address on TLB hits.

What is novel about how ESX server virtualizes memory is its method for memory reclamation. The basic design is that the ESX server will maintain a *balloon* module within the guest OS in each VM, which will act similar to a device driver within that OS. The ESX server maintains an open channel with this balloon, allowing for efficient and synchronous communication. The balloon driver communicates each physical page number directly to the ESX server, which then translates this physical page number into a machine page number, and updates its own data structures. This design allows for much more informed page-replacement policies. Indeed, the guest OS itself is most informed about what pages to swap out from its from virtual disk, as opposed to having the hypervisor itself decide what pages to reclaim what memory is sparse.

ESX server also introduces a novel page-sharing technique between different VMs by efficiently recognizing globally identical pages. It does this by maintaining hashes of the contents of a page to identify duplicates, and employing copy-on-write semantics for shared pages. Unlike other common techniques such as transparent page sharing, this technique requires no modifications to the guest OS and is entirely implemented in the hypervisor. A common scenario where this technique can be used to save physical memory is that all VMs can now share a single all zero-byte page.

To measure the performance of ballooning, [7] ran a file server benchmark on a single linux VM running on an ESX hypervisor. The benchmarks were run with the VM configured to have main memory of various sizes S, and with a VM with a large fixed memory size (256)Gb scaled down by ballooning to S. The results were the ballooned VMs had slightly higher throughput than the corresponding non-ballooned VM, despite having a large main-memory size. To measure the performance of the memory sharing technique, [7] ran many linux VMS on an ESX hypervisor, and varied the number of VMs. A SPEC95 (floating point calculations) benchmark was run. As the number of VMs went from 1 to 10, approximately 67 percent of the VM memory was shared.

III. Kernel Architectures

A great consequence of adding a hypervisor as an extra layer of abstraction between the hardware and the kernel is a re imagining of the design and role of a typical kernel. Indeed, as noted in [3], the virtualization of the hardware creates the illusion that the hardware scales dynamically, for instance by adding more virtual cpus and memory, or by creating new instances of virtual machines on top of a hypervisor. This illusion of infinitely scalable compute and memory resources lends itself to a more modular design of operating systems. Indeed, before the introduction of hypervisors, trying to create modular and application specific kernels ran into the issue of having to support a wide range of hardware, which led to the dominance of monolithic kernels. Hypervisors fix this challenge by providing a clean abstraction for modular operating systems to be built against. This section will explore some of these newly unlocked kernel designs, and their implications for computing in an illusory, unconstrained-resource environment, such as the cloud.

A. Exokernels

1) Design and Motivation: In a similar spirit to the idea of a hypervisor, the paper [2] introduced the idea of an exokernel - a kernel design which provides a thinner layer of abstraction on top of the hardware for applications to run than is normally provided by a monolithic kernel. This allows applications to implement their own functionality to manage resources in a more specialized manner. Applications often do this by employing modular *library operating systems* - custom software which implements various aspects of a typical OS, such as page-table strategies and networking protocols. The idea is that applications will link against only those library operating systems which they need, and the exokernel will provide a minimal amount of abstraction of hardware resources that these libraries can run against. This is in contrast to the situation for monolithic kernels, where applications typically link against a large library like libc, and implementations of various OS functionalities are forced upon them. Furthermore, such a redesign gives the opportunity for various performance advantages, because various pieces of kernel functionality can now run in user space.

As motivation for the soundness of this design, consider the ubiquitous principle in computer system design - the end-to-end principle - which purports that application logic should be pushed as close as possible to the application layer itself. This principle is commonly applied to computer networking, for example, where lower levels of the networking stack should not need to understand application level logic. The exokernel follows this principle, in that the core operating system itself should not be responsible for knowing about application level logic. Instead, the exokernel should just provide secure bindings to hardware resources, and applications can choose custom library operating systems to use on a specialized basis.

A representative example of how an application may benefit from such a thinner layer of abstraction comes from Database Management Systems (DBMS). Such systems often have a buffer pool of pages for the database, similar to a linux page cache. However, the database has more information about what pages to keep in cache, say those relevant for a given query that is touching a certain database table, than the OS does. Similarly, the DBMS will be able to make better informed decisions about what pages to prefetch into memory. Thus, although it is relatively straightforward for a DBMS to use a native OS system call like mmap to manage the transfer of pages from disk to memory and back, the eviction policy will be far from optimal. Indeed, the performance harm of using the OS provided mmap mechanism in DBMS's is explored in detail in [4].

The rest of this section will examine the implementations and performance results of the systems *Aegis* and *ExOs*, an exokernel and a library operating system, respectively, built in [2].

2) Performance: The *Aegis* exokernel targets the MIPS instruction set, and was designed to run the *ExOs* library operating systems, which user level applications link against. The authors of [2] compare the performance of various kernel operations against the monolithic, Unix-based, operating system *Ultrix*.

Part of the reason *Aegis* exhibits superior performance is that, as with all exokernels, various kernel functionalities such as managing virtual memory and interprocess communication are implemented at the user level. For example, the time for two processes to communicate with each other though a pipe on ExOs was around $30\mu s$, while for Ultrix it took around an order of magnitude slower at $300\mu s$.

A potential performance bottleneck for exokernels involves how they handle hardware exceptions. Typically exception handlers run in kernel mode, while in exokernels, exception handling is done in user space, so the exokernel must propagate the exception up into user space, which may degrade performance. However, the results from *Aegis* suggest that this may not lead to too much overhead. Indeed, the experiments in [2] show that *Aegis* takes on average $2\mu s$ to dispatch exceptions to user space, which may or may not be a bottleneck depending on the overall latency requirements of a task.

Lastly, an experiment was done to test the performance of the virtual memory implementations in both systems - programs multiplying 150×150 matrices was run on both Aegis and Ultrix. Because both systems were running on the same cpu, this experiment tests the latency of accessing memory in a semi-sequential pattern. The difference in the time taken was negligible, taking around $5\mu s$ depending on the exact cpu, suggesting that there is not too much overhead in application level virtual memory management.

B. Unikernels

1) Motivation: Each Virtual Machine is usually specialized to run one type of application - i.e. a Database Management System of a Web Server. Similar to Exokernels, Unikernels aim to replace the services provided by the operating system with library operating which application then statically link against. Note that this has the potential to greatly reduce the size of the binaries of applications - i.e. any program currently running in a unix like environment must link against the entire libc library. However, as opposed to exokernels, UniKernels are specialized to run a single application type. In the unikernel approach, applications would only link against the library operating systems which they actually use. This is especially important in cloud environments, where saving resources such as memory usage directly resulting in cost savings.

Configuration of applications now become programmable. Each running application, i.e. a database or a web-server, is now treated as a library, and can be configured programmatically. This has a number of advantages. Instead of configuring text files and writing shells scripts, the configuration can take advantage of properties of properties of the underlying language and compiler, such as type safety and static analysis, which reduces the potential for configuration bugs and can also possibly enable dynamic configuration changes to the application at runtime, which would not be available in a typical configuration workflow involving editing text files and shell scripts before the application starts running.

2) Implementation and Performance: The idea in [3] is to bundle together application code and a runtime at compile time, and to deploy this bundled virtual machine image directly on a hypervisor. The authors of [3] built the *Mirage* runtime in OCaml, a statically typed language, and used a Xen-like hypervisor to deploy their unikernel VMs. In particular, there is no notion of a userspace in a Mirage unikernel, which can lead to performance improvements such as reducing the amount of copying for IO.

To examine the network performance of the Mirage runtime, the authors of [3] built a DNS server unikernel. The Mirage unikernel outperformed by a factor of around 2 other DNS servers running on Linux. Aside from raw performance, the experiments suggests certain ways in which the unikernel approach may prove to be superior in a cloud envionrment to typical VMs running on a hypervisor or the exokernel approach. Firstly, the size of the final deployed unikernel binaries were much smaller than the size of the corresponding VM images. For example, the DNS unikernel was 183kb in size, compared to the 462Mb DNS server running on a Linux VM - reduing binary size translates to monetary savings on the cloud and allows for quicker deployments. Secondly, as an interesting consequence of bundling together the runtime with the application code, booting a unikernel VM is very fast compared to booting an OS image. In particular, it is fast enough to boot and respond to network traffic in real time, which adds an extra level of granularity at which VM resources can be deployed and stripped down in a cloud environment.

IV. Conclusion

Virtualization is playing an importantly important role in modern computing as computation and storage continues to be outsourced to cloud providers. The techniques and systems analyzed in this survey demonstrate that it is possible to depoloy safe and performant systems in these virtualized compute environments. This opens up new possibilities for more modular kernel designs, as opposed to standard monolithic kernels. The Unikernel approach in particular is a promising method to deploy bundled, performant, and light weight applications to the cloud, especially in resource constrained environments, and should prove to be a powerful technique in the future.

References

[1] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 164–177, 2003.

[2] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 251–266, 1995.

[3] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Steven Hand, Jon Crowcroft, Thomas Gazagnaire, and Steven Smith. Unikernels: Library operating systems for the cloud. *SIGARCH Computer Architecture News*, 41(1):461–472, 2013.

[4] Crott Andrew Pavlo Andrew, Leis Victor. Are you sure you want to use MMAP in your database management system? *IEEE Data Eng. Bull.*, 37(4):3–7, 2014.

[5] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *2001 USENIX Annual Technical Conference (USENIX ATC 01)*, Boston, MA, June 2001. USENIX Association.

[6] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems Design and Implementation*. Prentice Hall, 3rd edition edition, 2006.

[7] Carl A. Waldspurger. Memory resource management in vmware esx server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 181–194, 2002.