Hype and Virtue

Timothy Roscoe[*] Kevin Elphinstone[†] Gernot Heiser[†‡]
National ICT Australia

Abstract

In this paper, we question whether hypervisors are really acting as a disruptive force in OS research, instead arguing that they have so far changed very little at a technical level. Essentially, we have retained the conventional Unix-like OS interface and added a new ABI based on PC hardware which is highly unsuitable for most purposes. Despite commercial excitement, focus on hypervisor design may be leading OS research astray. However, adopting a different approach to virtualization and recognizing its value to academic research holds the prospect of opening up kernel research to new directions.

1 Introduction

Are hypervisors really a disruptive technology? Both the IT industry and the academic OS research community have devoted much attention recently to virtualization, in particular the development of hypervisors for commodity hardware, and commodity hardware support for them.

Virtualization has been touted in popular articles as a disruptive technology, and indeed as “the new foundation for system software” [6]. A recent spirited debate has centered on the claim that the hypervisor-based approach to system software fixes most of the perceived flaws of microkernels while retaining their apparent advantages [13, 14].

While the importance of the current wave of virtualization technology seems clear from a commercial standpoint, in this paper we critically examine whether hypervisors represent an equally disruptive factor for the OS research community. One might ask, to coin a phrase, “are virtual machine monitors OS research done right?”

We argue that this is not the case at present, and that most current research based around virtualization is not very different (if at all) from the kinds of problems the community has always worked on. However, we *do* feel that virtualization presents truly interesting directions for academic research (as opposed to product development or business models), both as an enabler for new ideas, and as a source of a new class of problem. We lay out some of these directions towards the end of this paper.

A challenge with any technological development which creates intense interest simultaneously in both academia and venture capital circles is separating the long-term scientific and engineering questions traditionally relegated to academic research from the short-term issues closely tied to particular business models, contexts, and in some cases individual companies.

This is an unashamedly academic paper, deliberately bracketing short-term commercial pressures to concentrate instead on longer-term research questions in OS design. We do not wish to devalue short-term research strongly embedded in current products and markets, but we emphasize that it is not our concern here.

In the next section we compare hypervisors to other kernels from a long-term research perspective (rather than focusing on their short-term applications) and in Section 3 critique the new system interface offered by hypervisors. In Section 4 we identify an approach to building and using virtualization technology to move academic OS research along by freeing it from some of the business-oriented constraints that have dogged the field for some time. In Section 5 we outline a few possibilities that this view of virtualization opens up, and conclude in Section 6.

2 Much Ado

Current VM-related research falls into two areas: building better hypervisors, and novel applications for them.

Our target in this paper is the former, though first we remark in passing that a number of novel applications for hypervisors are either (admittedly useful) tools for writing existing operating systems, or ingenious workarounds for the deficiencies of the guest OS – the ideas are important, but a well-designed OS interface would make their implementation much easier, and they don’t investigate what a radically new operating system design might achieve.

What the VMM is providing here is a means to get the work done without changing an existing guest OS, perhaps because such a job is beyond the capacity of a single PhD student or does not fit into a time frame dictated by upcoming publishing deadlines. These are important practical considerations, but we should also explore the

[*] Now at ETH Zürich, Switzerland
[†] Also at the University of New South Wales, Sydney, Australia
[‡] Also with Open Kernel Labs

long-term question of whether the combination of modified VMM and legacy operating system is a better solution than simply building a better OS in the first place.

Rather than treating these application ideas as simply neat VMM tricks, we should take them as new *requirements* for OS design and implementation. Resorting to a hypervisor to implement replay debugging or sophisticated security mechanisms, for instance, is a tacit admission that the current guest OS of choice cannot be practically evolved to support this functionality.

In the rest of this section, we critically examine the canonical tasks of a kernel – resource sharing, protection, abstraction of hardware, and communication – and try to establish what is genuinely different in a hypervisor versus a conventional kernel.

Sharing and Protection

Sharing is about controlling how multiple clients, in this case virtual machines, use the resources of the hardware. Protection is about ensuring that these clients do not unduly interfere with each other – by accessing each others' data, or affecting each others' performance by acquiring resources that system policy has not allocated to them.

We should ask whether approaches to sharing and protection in the new generation of hypervisors are in any sense novel. It is true that CPU resource allocation has been given a new context by the possibility of selling resources (packaged as VMs) in the form of "utility computing". However, almost all the solutions to this problem (mostly in the form of hypervisor scheduling algorithms) are quite old, lifted from the now-moribund field of multimedia systems (e.g. [8,17,23]). Data protection is achieved via per-VM MMU state – hardware access aside, essentially the same as address-space protection in a conventional OS, even if it is abstracted differently.

Communication

Communication between VMM clients is addressed very differently. Rather than borrowing solutions from monolithic and micro-kernels, hypervisor designs appear to define the problem away.

The argument [13] goes as follows: since the communication principals are complete operating systems in themselves, they are largely self-contained, much like library OSes over exokernels like Aegis [11] and Nemesis [17]. Consequently the VMM has little need to support the equivalent of fast IPC in microkernels, since the system as a whole has little need for shared servers. Furthermore, the kernels inside VMs are expecting to communicate via their own networking stack (since they assume they have a machine to themselves), and hence an emulated Ethernet should suffice for the small amount of local inter-VM communication they do need.

However, it seems that fast IPC performance is indeed important for hypervisors after all. A trend in hypervisor design is moving drivers into guest OS domains used as "driver servers", mirroring microkernels.

For example, the early design of Xen [1] resembled an exokernel, with low-level multiplexing of resources between relatively self-contained domains. Recently, Xen has morphed into an architecture where drivers run in separate domains [12] which function like driver servers in microkernels [18]. There are compelling reasons to do this, though they're not particularly technical. In particular, Xen no longer needs device drivers since it can use those written for another kernel by running them in copies of that OS.

Hypervisors hence now perform a lot of IPC in the normal execution of guest OSes. Fortunately, a wealth of research in this area seems directly applicable, both for synchronous IPC (useful for short, bounded-execution-time calls such as driver invocations), e.g. [19], and asynchronous messages (suitable for data transport), e.g. [2].

Abstraction

The key area where hypervisors differ from traditional kernels is abstraction: how resources are presented to the client. Rather than a view based on processes, threads, and address spaces, hypervisors aim at an abstraction which resembles the hardware enough to run guest OS kernels written for the physical machine.

Compatibility aside, this abstraction has a clear advantage: it can be made to correspond to a user-level application. Ironically, both traditional microkernels and monolithic systems lack an explicit kernel representation of a complete application. For example, Unix applications often span multiple processes (or even process groups), while multiple applications can also share use of a single process (as with the X server).

A VM, on the other hand, can both contain and isolate an application, by bundling the guest OS with the application – as one industry figure has put it, "operating systems are the new middleware" [5]. We believe this is highly significant, even if it has occurred in hypervisors almost by accident. Software is now being sold on this basis in the form of "virtual appliances".

Aside from providing backward compatibility with existing operating systems, which from this paper's perspective is commercially important but of little research interest, this new interface is the salient feature of hypervisors: it offers an explicit abstraction of an application, to which one can apply security policies, resource allocation, etc.

However, as researchers thinking critically we must ask: while this new interface might be a better abstraction, is it the best or most appropriate one?

3 Virtual Hardware as an API

Having looked at what might be different about hypervisor design from an OS research standpoint, we now look at the design of the interface that hypervisors provide to their clients: VMs, and ultimately applications.

The interface provided by hypervisors was not designed with applications in mind at all. It is based on hardware, with modifications (paravirtualization) to address the worst performance problems. Its use for virtual appliances came after hypervisors had been around for some time, motivated by quite different reasoning (such as making parallel programming tractable [4]).

In fact, we argue that a paravirtualizing hypervisor is a bad choice for an OS interface, for a number of reasons.

Implementation complexity

A feature of the VMM-based approach is *less coding*, since to support an existing application, one simply runs the OS it expects underneath.

However, this simplicity comes at the cost of *more code*. There's now a lot of machinery on the path between the application and the (real) hardware, not directly contributing to the application's functionality (or duplicated in the stack).

The resultant bloat comes with its own security problems, since the system's trusted computing base increases in size, and its correctness becomes even harder to ascertain. Whereas previously an administrator had to be concerned with vulnerabilities in the underling OS compromising the application, now he or she needs to be worried about the application's guest OS, the VMM itself, device drivers in driver domains, and the guest OSes supporting these drivers: all of these components must now be kept up-to-date, and vulnerabilities in any of these components can jeopardize the application.

Interface complexity

It is sometimes claimed that the VMM approach leads to better system design because the VM interface is simple and low-level, leading to a more policy-free and verifiable system as a whole. This argument sounds appealing – after all, the systems research community we have always valued "elegance" and "simplicity" in our designs. Unfortunately, on close inspection there is absolutely no evidence for this.

Part of the trouble is that we can't say what we mean by "simple" or "elegant" designs. There are no agreed-upon metrics or definitions. This problem has been eloquently pointed out recently in networking [22].

A good case can be made that the low-level interface provided by hypervisors is more complex and harder to specify than typical OS ABIs (or abstract virtual machines like Java's VM or Microsoft CIL), and is hardly a move closer to formally-specified interfaces. PC-based virtual machines vary widely in supported instruction set extensions, available (virtual) hardware, physical memory layout, MMU functionality, interrupt delivery semantics, etc.

Furthermore, we know of no attempts to even informally specify this interface. At best, paravirtualization interfaces such as [24] attempt to capture the *differences* between the VM's ABI and that of the (unspecified) real hardware.

In contrast, interfaces to operating systems (both microkernels and monolithic systems) and language-based virtual machines have a long history of careful documentation [16, 20], standardization [9, 15], and more recently, formal specification [10].

An anonymous reviewer of an earlier version of this paper suggested there might be something "almost magical" about the PC hardware ABI responsible for its recent success – an intriguing notion worthy of further study.

We conjecture that there are some features of the PC ABI (for example, an upcall-based interface in the form of interrupts) which are well-suited to supporting complex applications, while other aspects (for example, the ia32 MMU design) present serious obstacles to development. Evidence for this comes in the form of which parts of the interface have been redesigned by implementers of paravirtualizing VMMs. The challenge is to take what we can from this ABI, but design a better one.

Performance and scalability

Full virtualization, considered purely as an OS ABI, results in remarkable performance and scalability penalties compared to more conventional kernel interfaces. Part of this is due to the semantic bottleneck of the hardware-like interface – it's hard for a guest OS to express complex requests efficiently across this interface, but part of the problem is the overhead of running a complete copy of an OS in each VM.

For example, Xen scales remarkably well considering its aims, but in no way compares to vanilla Linux in terms of the number of application processes runnable at a time, due almost entirely to memory overhead. Even when modified to support copy-on-write images of mostly-identical VMs as in the Potemkin project [25], they still come out as much more heavyweight than processes.

Paravirtualization addresses these problems, but only slightly: paravirtualization starts from "pure" virtualization and backs off the minimum necessary for roughly correct execution and adequate performance. This works, but only up to a point, since the structure of the OS inside the VM still constrains the paravirtualized interface: the extra functionality typically appears as device drivers, for example.

In defence of hypervisors, we might say that the lack of scalability is a deliberate and considered consequence of providing performance isolation (through memory partitioning, etc.) between virtual machines. This argument, of course, does not address our point here, namely that a VM is a poor interface to an OS kernel. Moreover, it also fails to acknowledge two additional points that we feel are critical: firstly, we have no good metrics for measuring isolation, and secondly, it is not clear how effective VMMs can be at performance isolation in the presence of cache contention, even on a uniprocessor. We return to these below.

Discussion

It is ironic (and somewhat tragic) that after years of OS research we should look to the designers of PC hardware for guidance in formulating a kernel ABI. The new hypervisors have added to the traditional OS interface (POSIX or Win32) a second: paravirtualized PC hardware at the bottom. Neither is really new, nor are the designs of these interfaces terribly interesting from a future research perspective.

4 The opportunity

In OS research, we should use virtualization for what it's good for. By virtualizing a commodity OS over a low-level kernel, we gain support for legacy applications, and devices we don't want to write drivers for. Other than that, we should avoid these interfaces and move on. Virtualization presents us with an opportunity to return to basic research in system organization.

We should not conflate the facility of virtualizing hardware with the decision to multiplex the hardware at this level, even though current hypervisors do this. Instead, we could design a simple, clear interface to a low-level resource multiplexer in addition to the virtualization interface in current monolithic hypervisors like Xen. This would provide the benefits of a well-specified API to a low-level kernel, combined with the optional ability to run legacy OSes on top, rather than mandating the hardware-emulation interface of current hypervisors.

We must be clear that we are proposing is to write new operating systems which can virtualize existing ones as a means to gain compatibility with legacy applications and to reuse drivers for some hardware. We are *not* proposing new operating systems targetted at running inside virtual machine monitors (though we agree VMMs can be useful as hardware emulators in the early stages of kernel development). The latter is of dubious value in understanding how to effectively multiplex machine resources, the basic reason for an OS in the first place. The former allows us to concentrate on this fundamental research problem without the distractions of compatibility.

This could be done in two ways. A monolithic hypervisor like Xen could be extended so as to provide a more direct interface to the Xen kernel's facilities via hypercalls. Alternatively, a modular approach would split off hypervisor functionality into an optional hardware virtualization layer running as a library over a small resource multiplexer with a clean, well-specified interface. This latter component might be described without irony as a "microkernel".

Punting support for legacy applications and device drivers to a virtualized guest OS might raise performance concerns, but from a research perspect these issues are illusory. A research operating system project can demonstrate good native device performance by simply not virtualizing drivers for the devices that the researchers care about, and writing drivers for enough devices to demonstrate to the community that the design is sound. An analogous argument applies to applications: all legacy applications can be run with acceptable performance (if not, this calls into question much of the value of virtualization), but more importantly we can allow very different applications to emerge, and port applications to the new design where maximum performance is important.

All this allows the research community to finally escape the straitjacket of POSIX or Windows compatibility, both in drivers and applications. Rob Pike [21] has estimated that about 90-95% of Plan9's development effort was occupied with compatibility (excluding drivers). We have the opportunity to concentrate on OS design without further concern for backward compatibility. The result might be research kernels which are actually usable for real work, a rare sight these days.

5 Disruptive virtualization

Done right, virtualization removes the problems of driver support and legacy application compatibility, leaving open the questions of what the low-level kernel looks like and what its API is. Research kernels can experiment and *obtain practical experience* with a number of different approaches in this space.

Many systems research directions, and hypervisors are no exception, are characterized by what one assumes to be fixed (for example, the hardware and the processor architecture) and what can conversely be changed. Virtualization gives us the opportunity to redefine OS interfaces at any level above the physical hardware, allowing many areas of OS design to be rethought. Here we make a present a few of the possible avenues to investigate.

Kernel and API design

Many kernel interfaces are designed for single-threaded C programs using explicit memory management. These interfaces are in many respects a poor match for modern high-level programming languages.

For example, a well-known problem is how to integrate garbage collection with virtual memory: a copying collector can page in large amounts of memory from disk only to deallocate it. Exposing control of hardware page tables to the garbage collector can greatly improve matters, but is hard to achieve in a Unix-like system. The design of a suitable, shared interface to the MMU is a challenge.

There is also an opportunity to rethink API design (particularly with regard to I/O and concurrency) in the light of transactional memory, concurrent hardware, and high-level language constructs such as parallel combinators.

It is hard or impossible to attack these problems effectively either above a virtual hardware layer, or inside a hypervisor that supports such a layer, since the virtual hardware interface gets in the way. A more appropriate use of virtualization is enable a radical redesign of the lowest layer in the system while preserving compatibility for legacy devices and applications.

A final open question is whether a modular microkernel-based virtualization design can be as efficient at virtualization as a monolithic hypervisor. Note that situation is different from traditional microkernel issues, since it's more about vertical communication between the guest OS and microkernel, rather than horizontal IPC between processes (which is much the same in both cases).

Implementation techniques

A related problem is building an assured kernel, that is, one where we are reasonably certain the interface and implementation conform to a set of well-specified behaviors. It is famously hard to formally specify the behavior of existing OS APIs, let alone verify that a C or C++-based implementation results in the specified behavior.

However, it is hard to either gain traction with a new language in the context of an existing kernel written in C, or conversely validate a language by building a new kernel with little chance of deployment. The approach in Section 4 allows kernel design and language research to proceed together and result in a deployable OS.

A number of groups are working on better language support for systems programming. For example, Ivy [3] aims at evolving C with safe and checkable extensions.

The `seL4` project is investigating an alternative approach: build a prototype model of a kernel in Haskell [7,10]. The model can be combined with a machine simulator to execute real application binaries under simulation to gain experience with the API design, while the use of a high-level language facilitates clean implementation and rapid design iteration. A machine-checkable formal specification in higher-order logic can also be extracted from the model and used to verify API properties. The final design can then be executed on real hardware via a port of the Haskell runtime or translated semi-automatically into a low-level systems programming language for a high-performance implementation.

Applications

A principal justification to perform research into different designs of OS (rather than improvements to Unix or Windows) is to enable new user-visible functionality. A large portion of this is new applications. Given the opportunity to radically rethink kernel and API design, it is natural to ask what applications might be enabled by such research.

This is a difficult question, since "killer apps" tend to emerge unexpectedly from new enabling technologies, and the current state of the art often frames thinking about new applications. There are classes of applications (for example, rich QoS-based multimedia applications) which motivated considerable OS-related research in the past, but which arguably failed because of the difficulty of integrating the techniques with existing mainstream systems.

At the very least, virtualizing a legacy guest OS and redefining the underlying kernel interfaces does strictly increase the space of feasible applications.

The need for metrics

As mentioned above, there are no good OS benchmarks for isolation or VMM scalability. In other areas of computer systems research (such as databases and filing systems) benchmarks have demonstrated benefit in evaluating solutions and comparing approaches.

Benchmarks are problematic in kernel design, in part because it is hard to devise metrics which are not tied to a particular interface, and an important aspect of what we do as OS researchers is devise better interfaces. But without objective measures of how well isolation kernels (VMMs, monolithic kernels, or raw microkernels) do their job, we cannot make meaningful comparisons.

More significantly, in the absence of well-designed benchmarks it will be hard to gauge whether performance isolation is feasible at all without hardware support. Anecdotal evidence suggests that contention for cache lines between processes can have a serious effect on application progress. As multiprocessors become the norm, this effect will become more pronounced and contention for main memory will become important as well. Any argument that VMMs (or any other isolation technology) is an improvement on the previous state of the art must be set in this context.

Another benefit of benchmarks and metrics is less quantitative: it prompts discussion of which performance dimensions are actually important. A wider question is, what kinds of metrics are suitable for research kernels, whether monolithic hypervisors or microkernel-based?

In the absence of metrics, OS research often appeals to notions of simplicity. But as we have seen, these are highly ambiguous: from one perspective, a paravirtualized hardware interface represents a new "narrow waist" in system software, but from another is maddeningly complex, ill-specified, and poorly designed for programmers.

6 Conclusion

This paper has argued that OS researchers these days face a choice. On the one hand, we can investigate better ways to implement existing interfaces, i.e. further tweaks to Unix, better hypervisor implementation, and better paravirtualization techniques in the guest OS. This is short-term, immediately applicable, commercially relevant, but cannot be described as disruptive, since it leaves most things completely unchanged.

On the other hand, we can recognize that virtualization techniques do not necessarily mandate a hardware-like VM interface or a (para)virtualized conventional OS above, even if this is all that current hypervisors support. As researchers thinking long-term, virtualization techniques might give us the freedom to look at alternatives to these two interfaces without having to give up existing application and hardware support.

This opens up a variety of avenues in OS research, including novel engineering methodologies, application for formal methods, better support for modern programming languages and processor architectures, and discussion of appropriate metrics for evaluating future systems. We have tried to suggest a few of these in this paper.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. 19th SOSP*, October 2003.

[2] R. Black, P. Barham, A. Donnelly, and N. Stratford. Protocol implementation in a vertically structured operating system. In *IEEE LCN'97*, pages 179–188. IEEE, November 1997.

[3] E. Brewer, J. Condit, B. McCloskey, and F. Zhou. Thirty years is long enough: Getting beyond C. In *Proc. HotOS-X*, June 2005.

[4] E. Bugnion, S. Devine, and M. Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. In *Proc. 16th SOSP*, 1997.

[5] S. Crosby. Personal communication, August 2006.

[6] S. Crosby and D. Brown. The virtualization reality. *ACM Queue*, 4(10), December 2006.

[7] P. Derrin, K. Elphinstone, G. Klein, D. Cock, and M. M. T. Chakravarty. Running the manual: An approach to high-assurance microkernel development. In *Proc. ACM SIGPLAN Workshop on Haskell*, September 2006.

[8] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proc. 17th SOSP*, 1999.

[9] ECMA International. *Common Language Infrastructure (CLI)*, 4th edition, June 2006. ECMA standard 335.

[10] K. Elphinstone, G. Klein, P. Derrin, T. Roscoe, and G. Heiser. Towards a practical, verified kernel. In *Proc. HotOS-XI*, May 2007.

[11] D. R. Engler, M. F. Kaashoek, and J. J. O'Toole. Exokernel: an operating system architecture for application-level resource management. In *Proc. 15th SOSP*, 1995.

[12] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proc. 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, October 2004.

[13] S. Hand, A. Warfield, K. Fraser, E. Kotsovinos, and D. Magenheimer. Are Virtual Machine Monitors Microkernels Done Right? In *Proc. HotOS-X*, June 2005.

[14] G. Heiser, V. Uhlig, and J. LeVasseur. Are virtual-machine monitors microkernels done right? *SIGOPS Oper. Syst. Rev.*, 40(1):95–99, 2006.

[15] IEEE. *Std 1003.1-1988 (POSIX)*, 1988.

[16] L4Ka Team, System Architecture Group, Dept. of Computer Science, Universität Karlsruhe. *L4 eXperimental Kernel Reference Manual, revision 5*, June 2004.

[17] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, September 1996.

[18] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proc. 6th Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, Dec. 2004.

[19] J. Liedtke. Improving IPC by kernel design. In *Proc. 14th SOSP*, 1993.

[20] T. Lindholm and F. Yellin. *The Java^{TM} Virtual Machine Specification*. Prentice-Hall PTR, 2nd edition, 1999.

[21] R. Pike. System Software Research is Irrelevant. `http://www.cs.bell-labs.com/cm/cs/who/rob/utah2000.pdf`, February 2000. Available December 2006.

[22] S. Ratnasamy. Capturing complexity in networked systems design: The case for improved metrics. In *Proc. 5th Workshop on Hot Topics in Networks (HotNets-V)*, November 2006.

[23] V. Sundaram, A. Chandra, P. Goyal, and P. Shenoy. Application performance in the QLinux multimedia operating system. In *Proc. 8th ACM Conference on Multimedia*, November 2000.

[24] VMware, Inc. `vmi_spec.txt`: Paravirtualization API Version 2.5. `http://www.vmware.com/pdf/vmi_specs.pdf`, February 2006.

[25] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. In *Proc. 20th SOSP*, 2005.