Are Virtual Machine Monitors Microkernels Done Right?

Steven Hand, Andrew Warfield, Keir Fraser,
Evangelos Kotsovinos, Dan Magenheimer†
University of Cambridge Computer Laboratory
† HP Labs, Fort Collins, USA

1 Introduction

At the last HotOS, Mendel Rosenblum gave an 'outrageous' opinion that the academic obsession with microkernels during the past two decades produced many publications but little impact. He argued that virtual machine monitors (VMMs) had had considerably more practical uptake, despite—or perhaps due to—being principally developed by industry.

In this paper, we investigate this claim in light of our experiences in developing the Xen [1] virtual machine monitor. We argue that modern VMMs present a practical platform which allows the development and deployment of innovative systems research: in essence, VMMs are microkernels done right.

We first compare and contrast the architectural purity of microkernels with the pragmatic design of VMMs. In Section 3, we discuss several technical characteristics of microkernels that have proven, in our experience, to be incompatible with effective VMM design.

Rob Pike has irreverently suggested that "systems software research is irrelevant", implying that academic systems research has negligible impact outside the university. In Section 4, we claim that VMMs provide a platform on which innovative systems research ideas can be developed and deployed. We believe that providing a common framework for hosting novel systems will increase the penetration and relevance of systems research.

2 Motivation and μHistory

Microkernels and virtual machine monitors are both well explored areas of operating systems research dating back more than twenty years. Both areas have focused on a refactoring of systems into isolated components that communicate across well-defined, typically narrow interfaces. Despite considerable structural similarities, the two research areas are remarkable in their differences: Microkernels received considerable attention from academic researchers through the eighties and nineties, while VMM research has largely been the bailiwick of industrial research.

2.1 Microkernels: Noble Idealism

The most prolific academic microkernel ever developed was probably Mach [2]. A major research project at CMU, Mach's beginnings were in the Rochester Intelligent Gateway (RIG) [3] followed by the Accent kernel [4]. The key motivation to all of these systems was that the OS be "communication oriented"; that they have rigid, message-based interfaces between system components. Many of the abstractions used in Mach and later systems appeared initially in the RIG, including that of the *port*. However, the communications orientation of these systems originally intended to allow the distribution of system components across a set of dissimilar physical hosts.

The term "microkernel" was coined in response to the predominant monolithic kernels at the time. Microkernel advocates claimed that a smaller OS core would be easier to maintain, validate, and port to new architectures. A common theme throughout much of the microkernel work is that microkernels were *architecturally better* than monolithic kernels; from a research perspective they certainly are, as it is considerably easier to work on a single system component if that component is not entangled with other code.

Mach is hardly unique as an example of innovative microkernel projects. In the heyday of microkernels, many interesting systems were constructed including Chorus [5], Amoeba [6], and L3/L4 [7, 8]. Several of these evolved to show that microkernels, which were often criticized for poor performance, could match and even outperform commercial unix variants.

2.2 VMMs: Rough Pragmatism

Early work on Virtual Machines (VMs) [9, 10] was motivated by the need to improve hardware utilisation by facilitating the secure time-sharing of machines. Typically, VMs in IBM's model are identical "copies" of the underlying hardware where each instance runs its own operating system. Multiple VMs can be created and managed via interfaces exported by the VMM, a component running on the physical hardware.

As virtual machines may be owned by multiple, competing users, strong resource isolation mechanisms are required in the VMM. Another important facility provided by VMMs is that of *sharing* the hardware: securely multiplexing several virtual machines over a single set of physical resources.

The use of a VMM presents an additional layer of indirection between the hardware and the user, and it is necessary that this does not result in a noticeable performance degradation. For that reason, a significant amount of research effort in VMMs has been directed towards maintaining a low *performance overhead*, with considerable success [1].

Although VMM architectures differ in the degree of modification required to the guest operating systems they host, these modifications typically range from very small to none at all. Xen and Denali [11] host slightly modified "guest OSes" for improved system performance while VMware[1] provides full hardware virtualization so that no guestOS changes are needed.

An important characteristic of most VMMs is their ability to support the execution of *out-of-the-box applications*; users can run code that is executable on their regular desktop machines.

Because of the above properties of allowing users to securely share hardware on machines at a low performance cost, improving machine utilization, and not requiring modifications to the applications, VMMs have always presented a very appealing platform for practical deployment.

Previous research has combined microkernel and VM concepts to provide recursive VMs running on a microkernel-based OS [12]. User-mode Linux [13] achieves software-level virtualization by running a VMM as an application inside a host Linux system. Additionally, several research systems do not fall cleanly into either the VMM or microkernel camps; for example both the Exokernel [14] and Nemesis [15] systems provide low-level interfaces and resource protection above a small trusted kernel, but without the fine-grained modularization of microkernels or the OS-granularity multiplexing of VMMs.

[1] http://www.vmware.com

3 Architectural Lessons

While both microkernels and VMMs share rich histories of innovation, it is increasingly obvious that VMMs have achieved predominance in modern systems. In Section 4 we will revisit how many of the goals of microkernels remain relevant today. We first discuss some technical characteristics that consumed the research efforts of the microkernel community, but which have proven in our experience to be inconsequential in the development of modern VMMs.

We note that VMMs and microkernels bear a great deal of architectural similarity. The Denali team has re-titled their VMM μDenali in reference to its explicit restructuring as a microkernel, while there has recently been an effort to develop VMM functionality on top of the L4 microkernel. In this section, however, we focus on what we perceive to be the important differences between the two approaches.

3.1 Avoid Liability Inversion

One of the fundamental properties of microkernels is the division of a system into isolated user-space components. While the resulting kernel is smaller, this functional reduction relaxes the dependability boundaries within the system: applications must depend on other user-level components in order to run. More importantly, the microkernel itself depends on application level components, such as pagers, to make forward progress.

External pagers are an excellent example of this phenomenon: the failure conditions associated with them are one of the earliest and most recurrent problems discussed in microkernel-related literature [16]. Relegating a critical system-wide component to user-space, the kernel can be left waiting on the pager to evict a page before it can proceed. Various inelegant timeout and fallback mechanisms were required to avoid deadlock. By depending on arbitrary user-level components in order to continue execution, the kernel abdicates its liability for system liveness. We refer to this as *liability inversion*.

One of the principal design guidelines in Xen has been to avoid exactly these situations. Xen's memory management system, for instance, has no notion of paging whatsoever; rather it strictly partitions memory between VMs and allows limited facilities for sharing. VMs are themselves responsible for any paging within these allocations. The point here is perhaps a subtle one: decisions such as this are engineered to ensure that VM failure is

isolated and cannot degrade the stability of the system as a whole.

Consider, as a counterexample to external paging, the storage virtual machine used in Parallax [17]. In this case a storage VM is used to serve block storage to a collection of client VMs. A crash in the storage server could compromise the function of its clients, but not of the system as a whole: in particular, Xen itself does not depend on the correctness of the storage VM to function. Moreover, the dependency between the storage VM and its clients is *explicit*: the isolation between dependent VMs can be increased by separating the storage VM into multiple instances. This is essentially just the traditional trade-off between isolation and sharing which is observed in the design of any system.

3.2 Make IPC Performance Irrelevant

IPC performance is arguably the most revered hallmark of microkernel research. As message-based communication between system components is crucial to the operation of any microkernel, the literature is saturated with papers measuring IPC performance, improving IPC performance, and even questioning the relevance of IPC performance. However in our experience fast IPC is not a critical design concern in the construction of high-performance VMMs.

There are a number of reasons why we can avoid relying on fast, typically synchronous, IPC mechanisms. First, since VMMs hold isolation to be a key goal, IPC between virtual machines is considerably less common in general. This is a natural consequence of the fact that VMM design considers entire operating systems to be the unit of scheduling and protection: hence synchronization and protected control transfer are only necessary when two virtual machines wish to explicitly communicate.

Secondly, we have determined that a clear separation between *control* and *data* path operations allows us to optimize for the common case. In particular, we observe that by explicitly setting up communication channels, we can perform potentially expensive permission and safety checks at initialization time and then elide validation during more frequent data path operations. This decoupling furthermore allows higher-level communcation mechanisms great freedom in how they are implemented.

A particular example of this is seen in the implementation of *control*- and *device-channels* within Xen. Both of these are built upon a simple asynchronous unidirectional event mechanism which is the only communications primitive provided by Xen. However by combining pairs of events with shared memory, we can build both synchronous IPC for control operations and asynchronous producer-consumer rings for bulk, batched, data transfer. Even these latter allow considerable flexibility in use: by determining how often notifications are generated or waited upon, one can explicitly trade-off throughput and latency.

The difference between approaches to communication between isolated components is a very interesting example of the idealism versus pragmatism dichotomy described in the previous section. Microkernel designers view systems as sets of components that interact over IPC-, and potentially RPC-, based interfaces: they consider these interactions as procedure calls, in which the entire system is a collection of well-isolated components. VMM designers do not assume anywhere near the same degree of coherency within their systems: where VMs do communicate, they may not only be written in separate programming languages, but may also be running completely different operating systems. A consequence of this is that communications within VMMs typically looks like interactions with devices: a simple asynchronous control path combined with fixed-format transparent bulk data transfer.

3.3 Treat the OS as a Component

The final important difference between VMMs and microkernels is that of the *granularity of componentization*. By positioning themselves as a response to monolithic kernels, microkernels focused on dividing the functional units of an OS into discrete parts. A practical problem faced by microkernel developers is that which faces any new OS effort: by changing the API visible to applications, an OS forfeits the complete set of software available to existing systems. As such, most microkernel projects were left spending considerable effort to implement emulation interface layers for existing OSes.

VMMs differ significantly here in that their *a priori* intention is to support existing operating systems. For example, out-of-the-box code, compiled to be executable on a range of existing OSes, can be run on a guest operating system on top of Xen. This reduces the cost of entry for users and applications, makes virtualization attractive and practical for a wider community, and addresses two of the main problems of microkernel systems — the difficulty in attracting a substantial user base, and the challenge in keeping microkernel operating systems up to date with the feature sets of existing OSes.

By supporting existing OSes, VMMs need only justify the potential performance overheads they incur in order to be an attractive option. As shown in [1] and independently verified in [18], the overhead imposed by Xen is very small.

Secondly, VMMs appeal to developers because they present a *familiar development environment*. Using existing OSes as fundamental blocks of componentization allows developers to continue using the same tool set that they have on their existing system, freeing them to concentrate on more important issues.

The Parallax storage system [17], mentioned earlier, is an example of the sort of componentization that VMMs allow: The storage VM is a set of daemons running on Linux in an isolated virtual machine. The system can be used by any OS that runs on Xen because it provides the same block interface that Xen's existing block virtualization uses. Parallax provides an extension to an OS function, an ability touted by microkernels, but does it in a familiar development environment, using existing OS drivers, and providing support in turn for a range of client OSes. Moreover, the implementation is independent from both Xen and client OS code: provided that the block interface remains common, the OS extension itself does not depend on the source of the client OSes or the VMM.

Similar benefits accrue for the developers of the VMM itself: for example, Xen makes extensive use of existing tools for network routing, disk management, and configuration as part of the control software running in the privileged management VM.

The size of components — i.e., guest OSes — running on a VMM can be adjusted, depending on the functionality required from them. One example is *ttylinux*, a minimalistic Linux distribution, providing multi-tasking, multi-user, and networking capabilities within less than 4 megabytes of operating system size. It is also easy to build a simple single-threaded 'library OS' which enables the use of extremely lightweight components when desired for security or performance reasons.

4 The future for VMMs

Having illustrated what we feel are the key differences between microkernel and VMM design, we now consider how VMMs may be used to realize many of the research benefits achieved by the microkernel community. These include narrow interfaces between system components providing easy **extensibility** of device and OS functionality, a small code base that can guarantee **security** more easily than monolithic kernels, and strong isolation providing opportunities for improved **manageability**.

Narrow interfaces between system components are crucial in facilitating extensibility. The clean IPC interfaces provided by microkernels allowed researchers the ability to focus on specific system components without becoming entangled in unrelated code. Similarly, the narrow interfaces present in Xen allow devices and OSes to be easily extended. Xen's device architecture has allowed device drivers to be isolated in a separate VM for dependability [19], and permitted low-level interfaces to be extended without necessitating modification of the target OS or VMM [20]. Indeed, it seems very likely that the exploration of how services and management will be structured in a multi-OS VMM system will continue to present many exciting research opportunities.

A further advantage of narrow interfaces, coupled with a minimal privileged kernel, is the tractability of achieving a high degree of confidence in the security of a system. This has been explored in the microkernel community by projects such as Flask [21] and EROS [22]. Several groups have expressed interest in developing these ideas for Xen, using concepts from projects such as the Flask-derived SELinux.

A final avenue of innovation realized recently by VMMs has been to explore less performance-centric aspects of systems development. As with the examples above, VMMs are a promising platform because these so-called 'ilities' can be developed and applied to existing systems. For example, live OS migration [23] allows a running OS to be relocated to a new physical host, empowering administrators to better manage physical resources. The ability to 'rewind' a VM's state has been used for intrusion detection [24], debugging [25] and administration [26].

5 Conclusion

Despite having dissimilar motivations and origins, microkernels and VMMs share many architectural commonalities. In this paper we have attempted to illustrate some of the technical separations between the two classes of system that, in our opinion, have favoured the success of VMMs in recent years. More importantly though, we posit that—despite the decline in microkernel research— modern VMMs, Xen in particular, are in fact a specific point in the microkernel design space; that VMMs are microkernels done right. In light of this opinion, we observe that many of the advantages realized through the structure of microkernel systems may be similarly developed above a VMM. Moreover, because VMMs run commodity operating systems and applications we claim that they present a valuable platform for innovative systems research to have impact outside the academic laboratory.

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 ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[2] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proc. Summer USENIX Conference*, June 1986.

[3] E. Ball, J. Feldman, J. Low, R. Rashid, and P. Rovner. RIG, Rochester's Intelligent Gateway: System overview. In *Proc. 2nd International Conference on Software Engineering*, page 132, 1976.

[4] R. Rashid and G. Robertson. Accent: A communication oriented network operating system kernel. In *Proc. 8th ACM Symposium on Operating Systems Principles (SOSP)*, pages 64–75, 1981.

[5] V. Abrossimov, M. Rozier, and M. Gien. Virtual memory management in chorus. In *Proc. European Workshop on Process in Distributed Operating Systems and Distributed Systems Management*, pages 45–59, 1990.

[6] S. Mullender, G. van Rossum, A. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba: A distributed operating system for the 1990s. *IEEE Computer*, 23(5):44–53, 1990.

[7] J. Liedtke. Improving IPC by kernel design. In *Proc. 14th ACM Symposium on Operating Systems Principles (SOSP)*, December 1993.

[8] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The Performance of μ-Kernel-Based Systems. In *Proc. 16th ACM Symposium on Operating Systems Principles (SOSP)*, October 1997.

[9] R. Adair, R. Bayles, L. Comeau, and R. Creasy. A virtual machine system for the 360/40. Technical Report 320-2007, IBM Corporation, Cambridge Scientific Center, May 1966.

[10] R. Goldberg. Architectural principles for virtual computer systems. PhD thesis, Harvard University, 1972.

[11] A. Whitaker, M. Shaw, and S. Gribble. Scale and performance in the Denali isolation kernel. In *Proc. 5th Symposium on Operating System Design and Implementation (OSDI)*, December 2002.

[12] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proc. 2nd Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–151, October 1996.

[13] J. Dike. User-mode Linux. In *Proc. 5th Annual Linux Showcase and Conference*, November 2001.

[14] D. Engler, F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proc. 15th ACM Symposium on Operating Systems Principles (SOSP)*, December 1995.

[15] 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. 14(7):1280–1297, September 1996.

[16] M. Young, A. Tevanian, R. F. Rashid, D. B. Golub, J. L. Eppinger, J. Chew, W. J. Bolosky, D. L. Black, and R. V. Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proc. 11th ACM Symposium on Operating Systems Principles (SOSP)*, 1987.

[17] A. Warfield, R. Ross, K. Fraser, C. Limpach, and S. Hand. Parallax: Managing storage for a million machines. In *Proc. 10th Workshop on Hot Topics in Operating Systems (HotOS X*.

[18] B. Clark, T. Deshane, E. Dow, S. Evanchik, M. Finlayson, J. Herne, and J. Matthews. Xen and the art of repeated research. In *Proc. USENIX Annual Technical Conference*, June 2004.

[19] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proc. ACM OASIS Workshop*, 2004.

[20] A. Warfield, K. Fraser, S. Hand, and T. Deegan. Facilitating the development of soft devices. In *Proc. USENIX Annual Technical Conference*, April 2005.

[21] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security architecture: System support for diverse security policies. In *Proc. Eighth USENIX Security Symposium*, August 1999.

[22] J. Shapiro, J. Smith, and D.Farber. EROS: a fast capability system. In *Symposium on Operating Systems Principles*, 1999.

[23] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.

[24] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, 2002.

[25] S. King, G. Dunlap, and P. Chen. Debugging operating systems with time-traveling virtual machines. In *Proc. USENIX Annual Technical Conference*, 2005.

[26] A. Whitaker, R. Cox, and S. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *Proc. 6th Symposium on Operating System Design and Implementation (OSDI)*, pages 77–90, December 2004.