The nonkernel: A Kernel Designed for the Cloud

Muli Ben-Yehuda[1] Omer Peleg[1] Orna Agmon Ben-Yehuda[1] Igor Smolyar[1,2]
Dan Tsafrir[1]
[1] *Technion—Israel Institute of Technology*
[2] *Open University of Israel*
{muli,omer,ladypine,igors,dan}@cs.technion.ac.il

Abstract

Infrastructure-as-a-Service (IaaS) cloud computing is causing a fundamental shift in the way computing resources are bought, sold, and used. We foresee a future whereby every CPU cycle, every memory word, and every byte of network bandwidth in the cloud would have a constantly changing market-driven price. We argue that, in such an environment, the underlying resources should be exposed directly to applications without kernel or hypervisor involvement. We propose the *nonkernel*, an architecture for operating system kernel construction designed for such cloud computing platforms. A nonkernel uses modern architectural support for machine virtualization to securely provide unprivileged user programs with pervasive access to the underlying resources. We motivate the need for the nonkernel, we contrast it against its predecessor the exokernel, and we outline how one could go about building a nonkernel operating system.

1 The Cloud is Different

Infrastructure-as-a-Service (IaaS) cloud computing, where clients rent virtual machines from providers for short durations of time, is running more and more of the world's computing workloads. Despite representing a fundamentally new way of buying, selling, and using computing resources, nearly all virtual machines running in IaaS clouds run the same legacy operating systems that previously ran on traditional bare-metal stand-alone servers; that have been designed for the hardware available twenty and thirty years ago; and that assume that all system resources are always at their disposal. We argue that this is neither efficient nor sustainable and that the system software stack must adapt to the new and fundamentally different run-time platform posed by IaaS cloud computing.

We begin by contrasting IaaS clouds with traditional servers. We ask the following questions: Who owns resources? What is the economic model? At what granularity are resources acquired and released? And what architectural support does the platform provide?

Resource ownership and control. On a traditional server, the operating system assumes that it owns and controls all resources. For example, it may use all available memory for internal caches, assuming there is no better use for it; on the other hand, in case of memory pressure it swaps pages out, assuming it cannot get more physical memory. In an IaaS cloud, the operating system (running in a virtual machine) unwittingly shares a physical server with other operating systems running in other virtual machines, each of which assumes it has full ownership and control over all resources assigned to it. Resource clashes inevitably arise, leading to performance variability and security breaches.

Economic model. In the cloud, the guest operating system's owner and the hypervisor's owner are separate selfish economic entities. Due to economic pressure, resources are overcommitted and constantly change hands. In the Resource-as-a-Service (RaaS) cloud, into which the IaaS clouds are gradually turning, those ownership decisions are made on an economic basis [1], reflecting the operating systems'

owners valuation of the different resources at the time. Thus, in the cloud, each resource has a time-dependent associated cost—this can already be observed in, e.g., CloudSigma's (http://cloudsigma.com/) burst pricing—and the operating system must relinquish resources at a moment's notice when their prices rise beyond some application-specific threshold; conversely, the operating system must purchase the resources its applications need when they need them.

Resource granularity. In a traditional server, the operating system manages entire resources: all CPUs, all RAM, all available devices. In the cloud, the kernel acquires and releases resources on an increasingly finer granularity [1], with a goal of acquiring and releasing a few milliseconds of CPU cycles, a single page of RAM, a few Mb/s of network bandwidth. Although current cloud computing platforms operate at a coarser granularity, the trend toward fine-granularity is evident from our earlier work [1].

Architectural support. The operating systems running on traditional servers strive to support both the ancient and the modern at the same time. Linux, for example, only recently dropped support for the 27-year-old original Intel 386. Modern x86 cloud servers have extensive support for machine virtualization at the CPU, MMU, chipset, and I/O device level. We contend that any new kernel designed for running on cloud servers should eschew legacy platforms and take full advantage of this architectural support for machine virtualization.

2 Designing a Cloud Kernel

Given the fundamental differences between a traditional server and an IaaS cloud, we now ask: what requirements should we impose on a kernel designed for the cloud?

The first requirement is to allow applications to **optimize for cost**. On a traditional server, costs are fixed and applications only optimize for "useful work". Useful work might be measured in run-time performance, e.g., in cache hits per second. In the cloud, where any work carried out requires renting resources and every resource has a momentary price-tag associated with it [1], applications would still like to optimize for "useful work"—more useful work is always better—but now they would also like to optimize for cost. Why pay the cloud provider more when you could pay less for the same amount of useful work? Thus the cloud kernel should enable applications to bi-objective optimize for both useful work and cost.

The second requirement is to **expose physical resources**. On a traditional server, the kernel serves multiple roles: it abstracts and multiplexes the physical hardware, it serves as a library of useful functionality (e.g., file systems, network stacks), and it isolates applications from one another while letting them share resources. This comes at a price: applications must access their resources through the kernel, incurring run-time overhead and its associated costs; the kernel manages their resources in a one-size-fits-all manner; and the functionality the kernel provides, "good enough" for many applications, is far from optimal for any specific application.

In the cloud, where costs of resources constantly change, the kernel should **get out of the way** and let applications manage their resources directly. This has several important advantages: first, applications can decide when and how much of each resource to use depending on its momentary price-tag. This enables applications to trade off cost with useful work, or to trade off the use of a momentarily expensive resource with a momentarily cheap one according to the trade-offs that their algorithms are capable of making. For example, when memory is expensive, one application might use less memory but more bandwidth while another might use less memory but more CPU cycles. Second, applications know best how to use the resources they have. An application knows best what paging policy is best for it, or whether it wants a NIC driver that is optimized for throughput or for latency, or whether it needs a small or large routing table. The kernel, which has to serve all applications, cannot be optimal for any one application. Exposing physical resources directly to application means that nearly all of the functionality of traditional kernels can be moved to application-level, where applications can then specialize it to suit their specific needs.

The third requirement is to **isolate applications**. In the cloud, the kernel can rely on the underlying hardware for many aspects of safe sharing and isolation it previously had to take care of, and thus reduce costs and increase resource utilization. For example, using an IOMMU, the kernel can give each application direct and secure access to its own I/O device "instance" (an SR-IOV Virtual Function (VF) [10,18] or a paravirtual I/O device [20]) instead of multi-

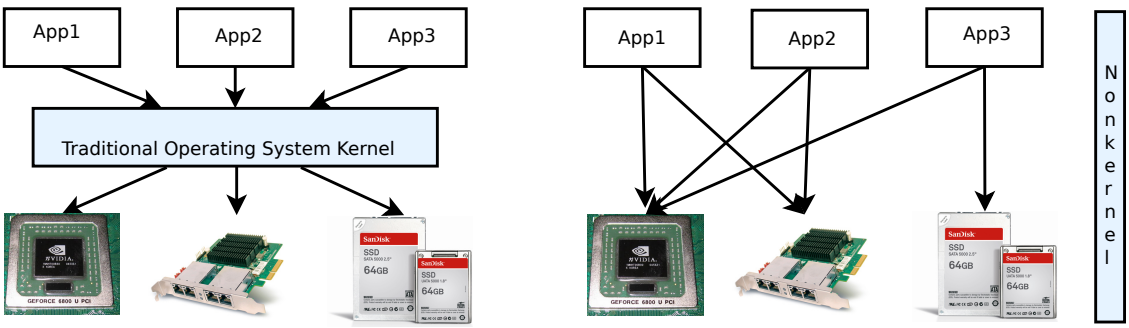

Figure 1: Traditional kernel structure compared with the nonkernel. The nonkernel grants applications safe direct access to their I/O devices.

plexing in software few I/O devices between many different applications.

3 The Nonkernel

We propose the *nonkernel*, a new approach for kernel construction designed for cloud computing platforms (Fig. 1). The nonkernel is a hybrid kernel/hypervisor designed to satisfy all three functional requirements mentioned in the previous section: it allows bi-objective optimization of both useful work and cost; it exposes resources and their costs directly to applications; and it isolates applications from one another.

The nonkernel can run in one of two modes: either as the bare-metal "hypervisor" (left side of Fig. 2) or as the operating system kernel of a virtual machine running on top of a legacy cloud hypervisor (right side of Fig. 2). Since the nonkernel uses hardware-assisted virtualization to run its applications each in its own virtual machine (in either mode), it assumes any underlying hypervisor supports nested virtualization [5].

The nonkernel takes advantage of hardware-assisted virtualization technology to achieve the functional requirements. It exposes and lets applications manipulate their CPU state directly, by running every application in a virtual machine. It lets applications manipulate their page tables directly using the architectural support for MMU virtualization. Most importantly, it lets applications bypass the kernel and access their I/O devices directly using the chipset and I/O device support for machine virtualization.

Bypassing the kernel has several implications. First, the kernel itself is minimized, since it no longer contains fine-grained scheduling or fine-grained memory management, file systems, storage device drivers, TCP/IP stack, network device drivers or any other device drivers. Instead, all resource-related code is provided to applications as libraries they can choose to link with. The libraries enable the application owners to control their level of optimization. Convenience-oriented owners will use a default library, or maybe benchmark several alternative libraries and choose among them. However, interested application owners can optimize the application for both useful work and cost by adjusting existing libraries and even by implementing specialized versions of "interesting" libraries.

At the end of the day, someone has to manage resources—but it is the application and application alone that knows how to best manage resources for its purposes. Since all resource-related code runs in application context, application writers can co-design the application logic and the way it uses available resources. Application writers can optimize their application to take advantage of the changing costs of resources. This would be hard if not impossible when there is only a single I/O stack as in a traditional operating system kernel. For interested applications, this can provide a big improvement in cost-per-useful-work. For applications that do not care about costs, it is no worse than letting the kernel manage resources.

So far we have described what the nonkernel does *not* do. What does the nonkernel do? It is in charge of (1) machine booting (2) resource acquisition and (3) application isolation.

The nonkernel contains the first instructions executed when the (physical or virtual) machine boots. It boots the system to a state in which the user can launch new applications.

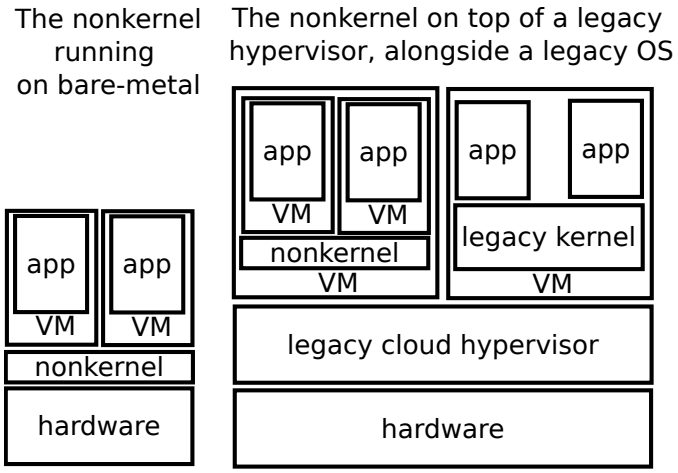

Figure 2: The nonkernel can run on bare metal or virtualized.

The nonkernel maintains a global view of available resources and their current prices. It exposes this information to applications and provides mechanisms by which they can acquire, pay for, and release these resources. It is in charge of resource arbitration—deciding which application gets which resource when there is contention—and of the initial allocation of enough CPU and memory to allow applications to start running and purchase more resources.

The nonkernel isolates applications while also permitting them to communicate directly. This is the result of containing each application in its own virtual machine. On x86, VMs access I/O devices using PIO and MMIO instructions, which the nonkernel isolates using architectural support [5, 24]; when the I/O device performs Direct Memory Access (DMA) into the application's memory, the IOMMU (programmed by the nonkernel) validates and translates this access [6]; and when the I/O device raises an interrupt, the interrupt is delivered directly to the application [10]. For efficient communication between applications, the nonkernel provides an IPC mechanism with no kernel involvement other than for setup and tear-down.

There are two ways one could go about building a nonkernel: based on an existing operating system kernel/hypervisor and implemented from scratch. To turn the Linux kernel, for example, into a nonkernel, Linux could run applications in virtual machines using a mechanism such as Dune [4] and provide these applications with direct access to their I/O devices using direct device assignment [24]. However, this would still be a Linux kernel, containing hundreds of thousands of lines of the core Linux kernel code, which would constrain the design space and dictate certain decisions. Implementing a nonkernel from scratch would allow a wider and deeper investigation of the design space.

4 Nonkernel vs. Exokernel

One way of thinking of the nonkernel is as an exokernel designed for today and future computing clouds. The original exokernel [9] broke ground in systems research. However, as originally designed, it was impossible to implement it as secure and complete as needed for today's public clouds. Lacking hardware support, the original exokernel implementation relied on downloading user code into the kernel, which exposed the kernel to arbitrary malicious code.

One way presented by the exokernel prototype to overcome this limitation was to give the user a context-specific language, similar to user-specified rule sets for kernel packet filters. While this solution might prevent the user from executing arbitrary code, it still allowed her to monopolize packets destined to other processes; thus the original exokernel's security relied on trusting processes, which is infeasible in a public cloud. Furthermore, context specific languages are abstractions created by the kernel, the very thing the exokernel model is meant to prevent.

Rethinking the exokernel in light of modern hardware virtualization support allows the nonkernel to achieve two goals. First, it enables the nonkernel to securely assign resources to untrusted processes without adding any software abstractions, thus fulfilling

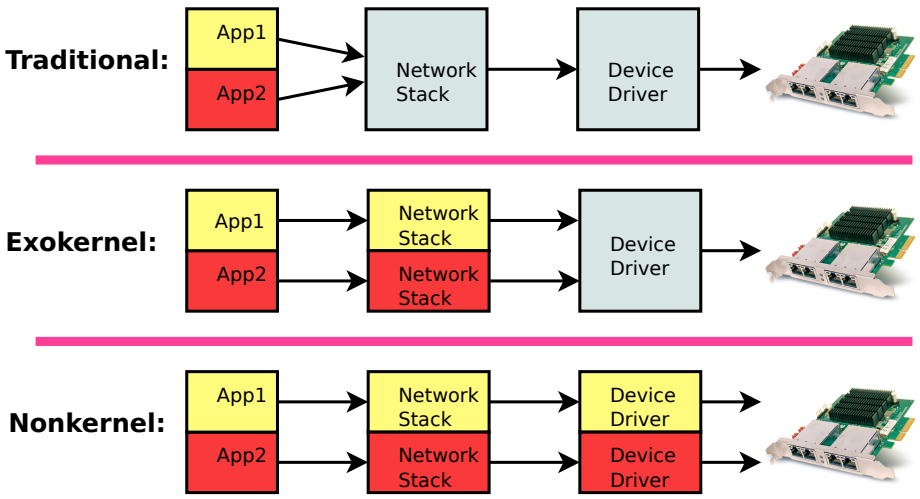

Figure 3: The I/O critical path in different kernels

the original goal of the exokernel model. Second, it allows the I/O critical path to be carried out without any kernel involvement (Fig. 3), removing the last redundant domain switching and giving the user maximum control over her software stack and the way it uses available resources.

5 Discussion: Pros and Cons

The nonkernel has several advantages when compared with traditional kernels and exokernels. The first advantage is its performance. Traditional userspace-I/O systems (e.g., [8,21,23]) show that performance can be gained by limiting kernel-induced overhead. Among other things, better performance can be achieved by refraining from data copy between kernel data structures and user-supplied buffers, by avoiding the overhead of system calls, and by directly accessing device control structures. Nonkernel applications benefit from these performance boosts as they bypass the kernel completely on their I/O paths. Although nonkernel applications run in virtual machines, we have recently shown that virtual machines can achieve bare-metal performance even under the most demanding I/O intensive workloads [10]; thus we do not expect this pervasive use of machine virtualization to cause any performance degradation.

Other benefits of the nonkernel include reduced driver complexity, since drivers now run completely in userspace, each driver instance serving a single application; easier debugging, development and verification of drivers and I/O stacks, for the same reason; simpler and easier to verify trusted-computing-base in the form of the nonkernel itself [12]; and hopefully a more secure system overall, for the same reason. The nonkernel economic model can also be useful for systems where operating power is a concern, by letting applications tune their resource requirements to the current thermal envelope limits.

The main disadvantage of the nonkernel approach is that it forsakes legacy architectures and legacy applications. The nonkernel is designed for modern hardware—in some cases, for hardware that is pre-production—and thus will simply not run on older machines. Likewise, the nonkernel is not designed to run legacy applications; realizing its benefits to the fullest extent requires some level of cooperation and effort from the application developer. We believe that the cloud represents such a large shift in computing platform that breaking away from legacy is no longer unthinkable. Nonetheless, a nonkernel can support to some extent legacy applications, either through emulation libraries that provide a subset of POSIX or Win32 semantics or by running a full "library OS" inside each nonkernel application context [19].

6 Related work

The nonkernel design draws inspiration from several ideas in operating system and hypervisor con-

struction. In addition to the original exokernel, the nonkernel's design also borrows from past work on userspace I/O (e.g., [7, 8, 21, 23]), virtual machine device assignment (e.g., [14,15,24]), multi-core aware and extensible operating systems (e.g., [3,13]), and library operating systems (e.g., [2,19,22]). The nonkernel shares the underlying philosophy of specializing applications for the cloud with Mirage [16,17] and the underlying philosophy of a minimal hypervisor with NoHype [11].

7 Conclusions

The cloud is a different kind of run-time platform, which poses new challenges but also provides an opportunity to rethink how we build system software. We propose the nonkernel, a new kind of kernel where applications access their resources directly and securely and respond to changing resource costs at a fine granularity. We are building a nonkernel called *nom* to experiment with and evaluate the ideas brought forward in this paper.

8 Acknowledgements

The research leading to the results presented in this paper is partially supported by the Israeli Ministry of Science and Technology.

References

[1] Agmon Ben-Yehuda, O., Ben-Yehuda, M., Schuster, A., and Tsafrir, D. The Resource-as-a-Service (RaaS) cloud. In *HotCloud* (2012).

[2] Ammons, G., Silva, D. D., Krieger, O., Grove, D., Rosenburg, B., Wisniewski, R. W., Butrico, M., Kawachiya, K., and Hensbergen, E. V. Libra: A library operating system for a JVM in a virtualized execution environment. In *VEE* (2007).

[3] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., and Singhania, A. The multikernel: a new OS architecture for scalable multicore systems. In *SOSP* (2009).

[4] Belay, A., Bittau, A., Mashtizadeh, A., Terei, D., Mazires, D., and Kozyrakis, C. Dune: Safe user-level access to privileged CPU features. In *OSDI* (2012).

[5] Ben-Yehuda, M., Day, M. D., Dubitzky, Z., Factor, M., Har'El, N., Gordon, A., Liguori, A., Wasserman, O., and Yassour, B.-A. The Turtles project: Design and implementation of nested virtualization. In *OSDI* (2010).

[6] Ben-Yehuda, M., Mason, J., Krieger, O., Xenidis, J., Van Doorn, L., Mallick, A., Nakajima, J., and Wahlig, E. Utilizing IOMMUs for virtualization in Linux and Xen. In *OLS* (2006).

[7] Caulfield, A. M., Mollov, T. I., Eisner, L. A., De, A., Coburn, J., and Swanson, S. Providing safe, user space access to fast, solid state disks. In *ASPLOS* (2012).

[8] Chen, Y., Bilas, A., Damianakis, S. N., Dubnicki, C., and Li, K. UTLB: a mechanism for address translation on network interfaces. *SIGPLAN Not. 33* (October 1998).

[9] Engler, D. R., Kaashoek, M. F., and O'Toole Jr., J. Exokernel: an operating system architecture for application-level resource management. In *SOSP* (1995).

[10] Gordon, A., Amit, N., Har'El, N., Ben-Yehuda, M., Landau, A., Tsafrir, D., and Schuster, A. ELI: Bare-metal performance for I/O virtualization. In *ASPLOS* (2012).

[11] Keller, E., Szefer, J., Rexford, J., and Lee, R. B. Nohype: virtualized cloud infrastructure without the virtualization. In *ISCA* (New York, NY, USA, 2010), ACM.

[12] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. seL4: formal verification of an OS kernel. In *SOSP* (2009).

[13] Krieger, O., Auslander, M., Rosenburg, B., Wisniewski, R. W., Xenidis, J., Da Silva, D., Ostrowski, M., Appavoo, J., Butrico, M., Mergen, M., Waterland, A., and Uhlig, V. K42: building a complete operating system. In *EuroSys* (2006).

[14] LeVasseur, J., Uhlig, V., Stoess, J., and Götz, S. Unmodified device driver reuse and improved system dependability via virtual machines. In *OSDI* (2004).

[15] Liu, J., Huang, W., Abali, B., and Panda, D. K. High performance VMM-bypass I/O in virtual machines. In *USENIX ATC* (2006).

[16] Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., and Crowcroft, J. Unikernels: library operating systems for the cloud. In *ASPLOS* (2013).

[17] Madhavapeddy, A., Mortier, R., Sohan, R., Gazagnaire, T., Hand, S., Deegan, T., McAuley, D., and Crowcroft, J. Turning down the LAMP: software specialisation for the cloud. In *HotCloud* (2010).

[18] PCI SIG. Single root I/O virtualization and sharing 1.0 specification, 2007.

[19] Porter, D. E., Boyd-Wickizer, S., Howell, J., Olinsky, R., and Hunt, G. C. Rethinking the library OS from the top down. In *ASPLOS* (2011).

[20] Russell, R. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Oper. Sys. Rev. (OSR) 42*, 5 (2008), 95–103.

[21] Schaelicke, L., and Davis, A. L. Design Trade-Offs for User-Level I/O Architectures. *IEEE Trans. Comput. 55* (August 2006).

[22] Van Hensbergen, E. P.R.O.S.E.: partitioned reliable operating system environment. *SIGOPS Oper. Syst. Rev. 40*, 2 (Apr. 2006).

[23] von Eicken, T., Basu, A., Buch, V., and Vogels, W. U-Net: a user-level network interface for parallel and distributed computing. In *SOSP* (New York, NY, USA, 1995).

[24] Yassour, B.-A., Ben-Yehuda, M., and Wasserman, O. Direct device assignment for untrusted fully-virtualized virtual machines.