Arrakis: The Operating System as Control Plane

SIMON PETER AND THOMAS ANDERSON

Simon Peter is a post-doctoral research associate at the University of Washington, where his research focus is on operating systems and networks. Simon holds a Ph.D. in Computer Science from ETH Zurich, Switzerland, and is a founding member of the Barrelfish multi-core operating system research project. Simon has worked on many OS-related topics, including multi-core scheduling, distributed file systems, and distributed tracing, and contributes to various OS projects, including the Linux kernel, the GRUB2 boot loader, and the Debian distribution. simpeter@cs.washington.edu

Thomas Anderson is the Robert E. Dinning Professor of Computer Science and Engineering at the University of Washington. Professor Anderson is an ACM Fellow, and he has won the IEEE Koji Kobayashi Computers and Communications Award, the ACM SIGOPS Mark Weiser Award, the IEEE Communications Society William R. Bennett Prize, the NSF Presidential Faculty Fellowship, and the Alfred P. Sloan Research Fellowship. tom@cs.washington.edu

The recent trend toward hardware virtualization enables a new approach to the design of operating systems: instead of the operating system mediating access to the hardware, applications run directly on top of virtualized I/O devices, where the OS kernel provides only control plane services. This new division of labor is transparent to the application developer, but allows applications to offer better performance, security, and extensibility than was previously possible. After explaining the need for such an operating system design, we discuss the hardware and software challenges to realizing it and propose an implementation—Arrakis.

Consider a Web application, where one part executes within a Web service and another runs on the machine of an end user. On the service side it is important for operations to happen as efficiently as possible. Short response times are important to keeping users happy with the provided service, and if the application is executing in the cloud, the operator pays for the resources consumed. Users, on the other end, want to be as safe as possible from potentially buggy or malicious code that is now downloaded simply when they go to a Web page.

Unfortunately, today's operating systems are not designed to handle either of these cases efficiently. On the server side, the Web application might be created using multiple components, such as a MySQL database, an Apache Web server, and a Python language runtime, executing on top of an operating system like Linux. Figure 1 shows such an architecture. For every packet we handle on the network or database entry we read from the disk, we must invoke the Linux kernel and go through the various mechanisms it provides. This involves checking access permissions on system calls, data copies between user and kernel space, synchronization delays in shared OS services, and queues in device drivers to facilitate hardware multiplexing. Furthermore, hardware is typically virtualized in the cloud, and virtualization often requires another layer of multiplexing using another set of device drivers in the virtual machine monitor (VMM). Only after that is the I/O operation forwarded to the real hardware. As I/O performance keeps accelerating at a faster pace than single-core CPU speeds, this kind of interposition skews the I/O bottleneck to the operating system, which is mediating each application I/O operation in order to safely multiplex the hardware.

On the end-user side, we want fine-grained sandboxes to protect us from potentially harmful surprises from remote code of untrusted vendors, such as bugs and security holes. Systems such as Native Client (NaCl [6]) go to great lengths to provide a secure execution environment, while allowing the use of shared browser services, like the JavaScript runtime. Their task would be much simpler with the right level of hardware and OS support.

Driven by the commercial importance of cloud computing, hardware vendors have started to offer devices with virtualization features that can bypass the virtual machine monitor for many common guest OS operations. Among these are CPU virtualization, which has been around for several years, and I/O virtualization, which has entered the market recently. For example, an IOMMU makes device programming from a guest operating system safe, while Single-Root I/O Virtualization (SR-IOV) allows devices to do their own multiplexing and virtualization. Which hardware features are needed to improve the performance of our Web application beyond just bypassing the VMM?

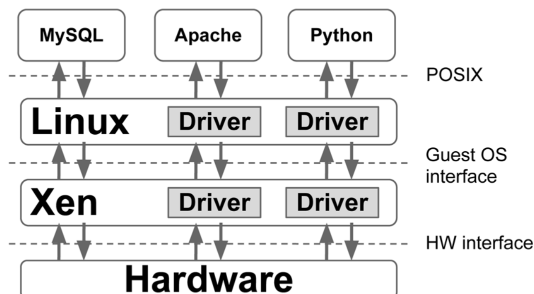

Figure 1: Application I/O paths for a virtualized Web service.

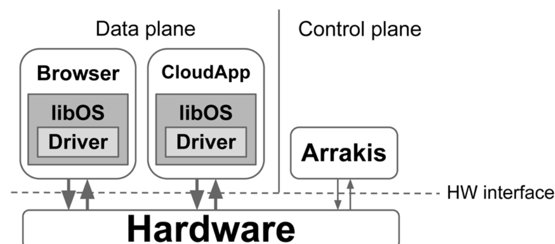

Figure 2: Arrakis I/O architecture

Hardware Support for User-Level Operating Systems

An inspiration for this work is the recent development of virtualizable network interfaces, such as the Intel X520 10 Gigabit Ethernet controller. These interfaces provide a separate pool of packet buffer descriptors for each virtual machine. The network interface demultiplexes incoming packets and delivers them into the appropriate virtual memory location based on the buffer descriptors set up by the guest operating system. Of course, the VMM still specifies which guest VMs are assigned to which virtual network device. Once the setup is done, however, the data path never touches the VMM. We would like to be able to demultiplex packets directly to applications, based on IP addresses and port numbers. For this to work, the network device needs to be more sophisticated, but Moore's Law favors hardware complexity that delivers better application performance, so such features are likely to be added in the future.

Entering the market now are hard disk controllers that allow hard disk partitions to be imported directly as virtual disks to guest operating systems. What we need is something more: the ability to give any application direct access to its own virtual disk blocks from user space. Unlike a fixed disk partition, applications could request the kernel to extend or shrink their allocation, as they are able to do for main memory today. The disk device maps the virtual disk block number to the physical location. Flash wear leveling and bad block remapping already support this type of virtualization. As with the network interface, the disk hardware would then read and write disk data directly to application memory.

An interesting research question we are investigating is whether we can efficiently simulate this model on top of existing hardware. The idea is to create a large number of disk partitions, which are then allocated as needed to different applications. Application data is spread across different partitions, but the application library synthesizes these partitions into a logical whole seen by the higher level code.

Power management can also be virtualized [4]. At the application level, knowing which devices need to be powered on and which can be put into low-power mode is easier. Applications are likely to know more about their present and future usage of a device, and therefore are capable of smarter power management than a device driver running within a traditional kernel.

Finally, Intel now supports multiple levels of (multi-level) page translation (Extended Page Tables [5]). The intent of this is to support direct read-write access by a guest operating system to its own page tables, without needing to trap into the hypervisor to reflect every change into the host kernel shadow page table seen by hardware. While useful for operating system virtualization, page translation hardware can also be used for a raft of application-level services, such as transparent, incremental checkpointing, external paging, user-level page allocation, and so forth.

Arrakis: The Operating System Is the Control Plane

What is required on the software side to allow applications direct hardware I/O? Ideally, we would like a world in which the operating system kernel is solely responsible for setting up and controlling data channels to hardware devices and memory. The hardware delivers data and enforces resource and protection boundaries on its own. Applications receive the full power of the unmediated hardware. To make this possible, we partition the operating system into a data plane and a control plane. This is in analogy to network routing, where the router OS is responsible for setting up data flows through the router that can occur without any software mediation.

Figure 2 shows this division in the Arrakis operating system. In Arrakis, the operating system (control plane) is only responsible for setting up hardware data channels and providing an interface to applications to allow them to request and relinquish access to I/O hardware, CPUs, and memory. Applications are able to operate directly on the unmediated hardware (data plane).

Direct hardware access may be made transparent to the application developer, as needed. We can link library operating systems into applications that can provide familiar abstractions and

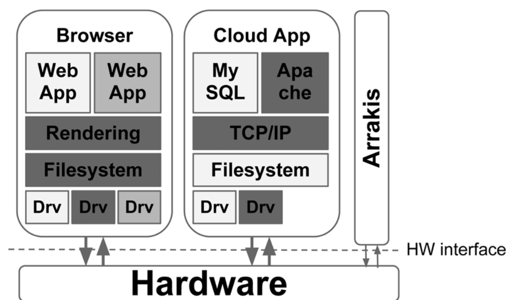

Figure 3: Example application containers containing a browser and a cloud application

mechanisms, such as the POSIX system call interface, thread scheduling, inter-processor communication, virtual memory management, file systems, and network stacks. These lightweight library operating systems execute within the same protection domain as the application.

The most important abstraction we are providing in Arrakis is that of an application container. An application container is a protection domain that provides a small interface to the Arrakis kernel to request the setup and tear down of unmediated channels to I/O hardware and memory, but otherwise provides the hardware itself. Figure 3 shows two such application containers. The Arrakis kernel is solely responsible for providing the mechanisms to allow allocating hardware resources to these containers, and, to allow applications to communicate, an interface to share memory, as well as a mechanism for directed context switches, akin to lightweight remote procedure calls (LRPC [1]), which facilitates low latency communication on a single core.

Use Cases

A number of applications can benefit from Arrakis' design, among them Web applications, databases, cloud computing, and high-performance computing (HPC) applications. We look back at Figure 3 and discuss two concrete examples of Web browsers and cloud applications within this section.

High-Performance Cloud Infrastructure

In Arrakis, we are able to execute the TCP/IP stack and network card device driver all within the same application and eliminate any system call protection boundary crossings, packet demultiplexing code, and kernel copy in/out operations that are typically required in a monolithic operating system. What's more, we can customize the network stack to better match the requirements of the Web server, down to the device driver. For example, the device driver could map packet buffers into the application in such a way that TCP/IP headers can be pre-fabricated and just mapped in front of the payload. The application can simply write the payload into the mapped buffer space. If packet checksumming is required, it can be offloaded to the network interface card.

A more complex cloud application may include a MySQL database server in addition to the Web server. The database is a fully trusted component of the cloud application; however, both MySQL and Apache ship within their own set of processes. Typically, these are connected via UNIX domain or TCP/IP sockets that need to be traversed for every request and the operating system has to be invoked for each traversal. This introduces overhead due to the required context switch, copy and access code operations, as well as OS code to ensure that data passed from one application to the other does not violate security. Avoiding these overheads can further reduce round-trip request latencies.

Arrakis allows us to run processes of both servers within the same protection domain. This eliminates most of the aforementioned overheads. Data can simply be remapped between applications, without sanity checks, and a context switch would not involve a journey through the operating system.

Application-Level Sandboxing

Web browsers have evolved into running a myriad of complex, untrusted Web applications that consist of native and managed code, such as HTML5 and JavaScript. These applications have access to low-level hardware and OS features, such as file systems and devices. Sandboxing this code is important to protect against security flaws and bugs that threaten system integrity.

In Arrakis, we are able to leverage hardware support for Extended Page Tables (EPT) to set up different protection domains within the browser. Each sandbox occupies a different protected address space within the browser's application container, with shared code and data mapped into all of its address spaces. This allows for a simple sandboxing implementation that, consequently, has a smaller attack surface.

Device drivers may be sandboxed as well using this approach. Furthermore, requesting channels to multiple virtual functions of the same hardware device from the kernel is possible. This allows us to replicate device drivers within the Web browser and run each replica within its own protection domain directly on these virtual functions multiplexed by the hardware. For example, we can request a virtual function per Web application and run the driver replica within that Web application. If a buggy device driver fails, only the Web application instance that triggered the failure will have to be restarted. The failure will not impact the rest of the browser environment or, worse, the rest of the system.

Lightweight Sharing

Providing Arrakis would be relatively easy if applications were complete silos—we could just run each application in its own lightweight virtual machine and be done. Our interest is also in providing the same lightweight sharing between applications as in a traditional operating system, so the user sees one file

system, not many partitions, and applications are able to share code and data segments between different processes. How might this be done?

In Arrakis, an application can directly read and write its file and directory data to disk, without kernel mediation. File layout and recovery semantics are up to the application; for example, a Web browser cache might use a write-anywhere format, since losing several seconds of data is not important, while others might use traditional write-ahead logging. In the common case, most files are used only by the applications that wrote them; however, we still need to be able to support transparent access by other applications and system utilities, such as system-wide keyword search and file backup. How do we design OS services that efficiently allow the same sharing among multiple applications as that offered by operating systems that mediate each I/O operation?

To achieve this, the format of files and directories is independent of name look up. In Arrakis, we insert a level of indirection, akin to NFS vnodes. When a file name look up reaches an application-specific directory or file, the kernel returns a capability associated with the application handling storage of the corresponding file. The capability is used to access the file's contents, by invoking a file memory mapping interface that is provided by the storage handling application's library operating system. This allows us to share files safely and efficiently among untrusted applications.

Related Work

The security/performance tradeoffs of monolithic operating system designs have been of concern several times in the past. Particularly relevant are Exokernel [3] and the SPIN operating system [2].

Exokernel tried to eliminate operating system abstractions, and thus allowed applications to implement their own. Applications can link library operating systems that contain the abstractions that fit best with an application's operation. Note that it was not possible to set up several protection domains within a library operating system and thus sandboxing was equally difficult as in today's operating systems. Furthermore, to be able to safely multiplex a single hardware device to multiple library operating systems, Exokernel had to resort to the use of domain-specific languages that had to be uploaded into the kernel for proper disk and network multiplexing.

SPIN allowed uploading application-specific extensions into the operating system kernel. This way, applications could access the hardware and OS services more directly and gain a speedup. To make this safe and protect the rest of the system from potentially buggy or malicious extensions that were executing in supervisor mode, SPIN required the use of a type safe programming language (Modula-3) for extension development. This allowed for an extension to be checked against all its accesses before executing it within the OS kernel, but required the implementation of all extensions within this language.

Conclusion

Now is the time to take a fresh look at the division of labor between the operating system, applications, and hardware. Recent hardware trends are enabling applications to become miniature operating systems, with direct I/O and virtual memory access, while safety and resource boundaries are enforced by the hardware.

We propose a division of the operating system into a control plane and a data plane that allows applications direct access to the hardware in the common case. Applications can provide their own storage, network, process, and memory management without mediation by the operating system kernel.

We are in the early stages of developing the Arrakis operating system. Our Web site, http://arrakis.cs.washington.edu/, provides further information and development status updates.

References

[1] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy, "Lightweight Remote Procedure Call," *Proceedings of the 12th ACM Symposium on Operating Systems Principles,* December 1989, pp. 102-113.

[2] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, "Extensibility, Safety and Performance in the SPIN Operating System," *Proceedings of the 15th ACM Symposium on Operating Systems Principles,* December 1995, pp. 267-284.

[3] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr., "Exokernel: An Operating System Architecture for Application-Level Resource Management," *Proceedings of the 15th ACM Symposium on Operating Systems Principles,* December 1995, pp. 251-266.

[4] R. Nathuji and K. Schwan, "Virtualpower: Coordinated Power Management in Virtualized Enterprise Systems," *Proceedings of the 21st ACM Symposium on Operating Systems Principles,* October 2007, pp. 265-278.

[5] Intel 64 and IA-32 Architectures Software Developer's Manual, August 2012.

[6] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A Sandbox for Portable, Untrusted X86 Native Code," *Communications of the ACM,* vol. 53, no. 1, January 2010, pp. 91-99.