Arrakis: A Case for the End of the Empire

Simon Peter Thomas Anderson

Department of Computer Science & Engineering, University of Washington

Abstract

In this paper, we argue that recent device hardware trends enable a new approach to the design of operating systems: instead of the operating system mediating access to hardware, applications run directly on top of virtualized I/O devices, where the kernel provides only control plane services. This new division of labor is transparent to the user, except that applications are able to offer more robust extensibility, security and performance than was previously possible. We discuss some of the hardware and software challenges to realizing this vision.

1 Introduction

Operating system researchers have long struggled with the tension between the different design aims of application developers and those of the operating system. The key features of an operating system—sandboxed execution of application code, resource allocation between applications, and virtualization of limited physical resources—all seem to require applications to operate at one level removed from the hardware. The result is predictable if lamentable: application developers have long railed that the operating system should "get out of the way", with few robust, secure solutions available to address the problem.

Twenty years ago, many operating systems researchers promoted a nano-kernel design pattern to try to address this tension [2, 7, 12, 25]. In this model, the operating system retains its sandboxing role and allocates resources to applications, but as much as possible, from that point forward the application is in complete control over how it uses its resources. User-level virtual memory pagers in Mach [21], scheduler activations for multiprocessor management [3], and Exokernel disk management [16] all took this approach. The key idea in all of these systems is that the operating system remains free to change its allocation decisions, as long as it notifies the application. However, the nano-kernel movement largely failed in practice. Instead, modern operating systems provide hooks for applications to request pinned physical resources, which the OS then promises to respect. This far simpler model achieves most of the performance benefits of a nano-kernel design, especially for dedicated server environments.

In this paper, we argue that recent application and hardware trends merit a fresh look at the application-operating system interface. We have named our project Arrakis, as it presages the end of the operating system kernel as the singular source of operating system control.

On the application side, increasingly applications are becoming miniature operating systems – not just wanting to be able to do their own resource optimization, as in a nano-kernel, but also their own sandboxing, resource delegation, and virtualization. An example is a web browser wanting to protect itself against untrusted scripts and extensions (e.g., NaCl [29]). Web servers likewise need to sandbox to combat security attacks; complex applications desire transactional semantics for their data storage; and so on.

At the same time, I/O devices have become increasingly sophisticated, taking on more and more of traditional operating system functions. For example, some modern network interface cards can demultiplex incoming packets directly into the target application waiting on the port [1, 17]; modern disks and flash memory devices embed a virtual to physical block translation layer for enhanced device reliability. With increasing I/O performance (e.g., 100 Gb/s Ethernet, microsecond I/O latency on solid state disks), unmediated hardware access becomes ever more important.

The trend towards increasingly sophisticated I/O devices is further powered by the commercial importance of virtual machines. Systems increasingly provide hardware support for direct execution of a guest virtual machine without mediation by the host kernel. For example, Intel supports a nested paging structure, enabling the guest operating system to directly manipulate its own page tables, without trapping to the kernel to reflect these changes in the system-level page tables [20].

We take a further step: can we remove the operating system kernel from normal application execution? An apt analogy is that the kernel becomes a network *router*. The operating system on a hardware router sets up data transfers to occur without any software mediation. Policy control in the router software sets up the hardware to prevent transfers that would violate security constraints.

Similarly, in our vision the operating system performs only control plane, and no data plane, operations: it sets up the application, and interacts with it in the rare case where resources need to be reallocated or name conflicts need to be resolved, but otherwise gets completely out of the way. The application gets the full power of the unmediated hardware, through an application-specific library linked into the application address space, and can interact with it directly for its fast-path I/O activities (the data plane).

This would be relatively easy if applications were complete silos – we could just run each application in its own lightweight virtual machine [28], 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.

The rest of this paper discusses the challenges needed to realize this vision: in hardware, operating system kernel design, and in the application library.

2 Hardware Considerations

An inspiration for this work is the recent development of virtualizable network interfaces [15]. High performance network access requires pipelining: instead of programming one I/O operation at a time, modern network interfaces have a queue of buffer descriptors, specifying where in memory to put/get each incoming and outgoing packet (and even where to separately put/get packet headers and packet data). As long as the operating system keeps both queues full, the hardware is able to operate completely asynchronously from the kernel, achieving full line rate even if the kernel or application is busy elsewhere.

Virtualizable network interfaces take this idea one step further, and provide a separate queue of buffer descriptors for each application. The network interface demultiplexes incoming packets on the address or port, and delivers the packet into the appropriate *virtual memory* location based on the buffer descriptors set up by the application. Of course, the kernel still specifies which addresses and ports are assigned to which virtual network interface, so there is no security vulnerability. Once the setup is done, however, the data path never touches the kernel: packets are read and written directly into and out of the (virtually addressed) buffers specified by the application. For this to work, the network device needs to be more sophisticated, but Moore's Law favors hardware complexity that delivers better application performance.

Something similar is happening with disks, but in a more limited way. A hardware RAID controller can be set up to provide a guest operating system direct, unmediated, access to a disk partition [18]; the host kernel only enables access for the partition pre-configured for that guest.

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. Most importantly, the application would have direct access to the full power of the disk hardware: an explicit asynchronous request queue, the ability to insert barriers between adjacent operations, and the ability to specify which blocks represent discarded data. For example, this would enable an application to implement an efficient write-ahead logging or copy-on-write system for its own data, something not possible on top of a standard file system interface. Of course, we also need to allow files written by one application to be read by another; we discuss this issue in the next section.

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 into a logical whole seen by the higher level code.

Other devices can also be virtualized. For example, we could provide efficient interprocessor interrupts between instances of the same application running on different cores; today, interrupts are mediated by the kernel, but for no essential reason. In fact, for good parallel performance, it would be useful for an application to be able to control which of its cores is to receive each type of interrupt; the hardware provides this ability to the kernel, but it is not exported today.

Likewise, power management can be virtualized [19]. At the application level, it is easier to know which devices need to be powered on, and which can be put into low-power mode. 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.

It is beyond the scope of this paper to discuss the issues involved in providing direct, unmediated access to the portion of the display under control of the application, and so we do not discuss it further.

Finally, we observe that Intel now supports multiple levels of (multi-level) page translation (Extended Page Tables [20]). 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 kernel to reflect every change into the host kernel, shadow page table seen by hardware. The Dune [5] project recently showed how to provide this capability to application libraries, to allow applications to manipulate their own virtual memory space, without mediation by the kernel. This can support secure sandboxing, where an application can set up a restricted execution environment for untrusted code, so that it can only touch certain memory locations. But 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.

For portability, the operating system will need to correctly handle the case where hardware virtualization does not exist, or when the number of contexts is smaller than the number of applications needing direct access. While we expect this case to be rare due to hardware trends, it can be handled in much the same way as a software router works today—by emulating the hardware data transfer in software.

3 Operating System Considerations

Given the above trends in hardware support, how does that change how we should build operating systems? To explore this, we sketch the design of Arrakis, a next-generation nano-kernel designed to take advantage of the hardware trends we have outlined. A number of goals guide the design of Arrakis:

Customizability. We allow applications maximum freedom to develop their own OS services and abstractions. Cloud, web and desktop applications have for some time provided their own custom versions of many major OS abstractions and services, such as protection, process management, and memory management. Often this has required extensive development expense. For example, NaCl has to play many tricks to realize multiple protection domains within a web browser. Arrakis supports customization of OS functionality as a first-class principle, by allowing these services to be implemented within applications using native hardware support.

Safety. Data, including that of an OS service, is shared among application containers only when necessary. Each application is confined along with the necessary OS services, such as file systems and device drivers, in its own protection domain by default. This provides for a small trusted computing base (TCB).

User transparency. From the user perspective, the behavior of the system is unchanged: files can be saved, directories listed, applications can crash and be restarted, and so forth. Arrakis gives applications additional capabilities, which they can in turn use to provide better security, performance, and reliability to users.

Arrakis provides the following abstractions to realize these design goals:

Semantic names. In Arrakis, an application can directly read and write its file data to disk, and even directories, 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.

To achieve this, the format of files and directories is independent of name lookup. We insert a level of indirection, akin to NFS vnodes [24] or names in the Semantic File System [14]. When a file name lookup reaches an application-specific directory or file, the kernel routes an upcall to an application-specific library which fills in the requested data. If the file is intended to be exported to others, e.g., a PDF file, the application can write the file in the traditional way through the kernel read/write interface.

Application containers. In place of traditional process management, application resources are allocated in containers, similar to virtual machines. The kernel assigns each container an allocation of hardware resources and provides an interface for the application library to request and release additional resources, such as cores, memory, and disk space. The abstraction is that of the hardware; the setup of individual protection domains, address translation and the ability to create new containers is provided within a container.

Application containers are similar to resource containers [4] and can be used in the same way. By coupling OS services in containers, resource limits can be enforced through the entire software stack, but with minimum implementation overhead. A special root container that receives all host resources at system startup is responsible for managing the creation of new application containers

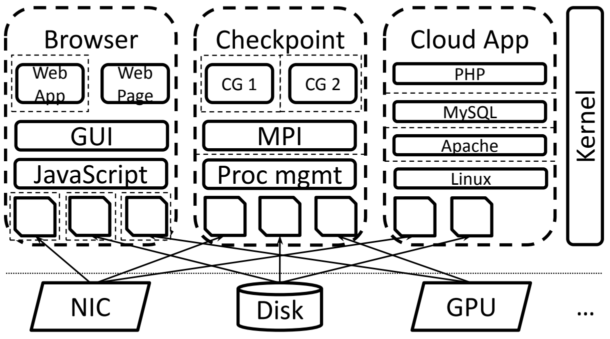

Figure 1: Arrakis architecture, showing three example application containers (rounded dashed boxes) and their components, including device drivers (unlabeled boxes). Thin dashed lines and boxes separate protected address spaces.

by relinquishing some of its resources to them.

Application memory. Similar to VM guest physical memory, application memory is an intermediate memory abstraction, between virtual memory and physical memory. It is realized via Extended Page Tables, which provide two levels of hardware memory translation. Application memory may be shared. The Arrakis kernel maintains a list of shared memory regions, associated with the container that owns the region. A container can restrict which other containers are permitted to share one of its memory regions.

Directed context switches. To facilitate low-latency communication among applications, Arrakis provides an asynchronous notification facility between applications. Akin to LRPC [6], an application container can request that a particular, specified container should be switched to immediately, in order to react to an event, such as a shared memory page being updated.

4 Use Cases

A number of applications can benefit from Arrakis' design, among them are web applications, cloud computing, and high-performance computing (HPC) applications. We show a few examples of these applications along with the architecture of Arrakis in Figure 1 and discuss their use cases within this section.

4.1 Sandboxing in Web Browsers

Web browsers have become platforms that run a myriad of complex, untrusted web applications that consist of native and managed code, such as HTML5 or JavaScript. Via libraries, applications have access to low-level OS and hardware features, like file systems and devices [10, 22]. Sandboxing this code is important to protect system integrity.

Neither threads nor processes are adequate OS abstractions to represent browser sandboxes, as they need strong protection, but they also need to share common browser services, such as the programming language run-time. Sandboxing solutions, such as NaCl go to great lengths to provide a secure, portable execution environment, but their task would be much simpler with the right level of hardware and OS support.

Arrakis enables web application sandboxes to be implemented within application containers. Each sandbox occupies a different protected address space within the web browser application container, with shared code and data mapped into all of its address spaces. This model affords a much simpler sandboxing implementation that, consequently, has a smaller attack surface. In the example in Figure 1, a web application runs in its own protection domain alongside a regular web page and the browser's normal mechanisms, like the graphical user interface and JavaScript JIT compiler. Downloaded device drivers operate within their own protection domains directly on devices multiplexed and protected by the hardware. If a buggy device driver fails, only the application instance using that driver instance will have to be restarted. The failure will not impact the rest of the browser environment or, worse, the operating system.

4.2 Application Checkpointing

High-performance computing applications have long needed robust checkpointing to recover from failures in long-running computations. However, while implementations for distributed rollback exist [11], rolling back to a previously stored checkpoint is complicated when OS state, such as session, socket and file system state, is involved. Thus, many implementations of checkpointing in HPC severely limit the types of system calls that can be made by an application under checkpointing [23].

In Arrakis, we can create checkpoint containers that contain all processes involved in the computation, along with the OS and run-time code necessary to orchestrate them. In Figure 1, two processes together carry out a conjugate gradient (CG) computation and are orchestrated by the MPI run-time, using UNIX-style process management. The container uses a trusted GPU device driver to allow the use of accelerated GPGPU algorithms that do not need to be sheltered in their own address space and can run together with the other OS services for better performance.

The state of this container can simply be saved and rolled back as a unit, including operating system services and device driver state. Hardware device state might be maintained along with the container. This rollback capability is similar to that found in some kernel-level device drivers (e.g., to restore devices from low power states).

Migration of a checkpoint to a different machine is also possible. This is difficult on traditional operating systems, when the OS version is different between the different machines. Arrakis allows us to simply migrate application containers that contain the OS services they rely upon.

4.3 Cloud Resource Sharing and Safety

Cloud infrastructure providers have struggled with software bloat for a long time. Even small, simple cloud applications use whole software stacks, such as LAMP (Linux, Apache, MySQL, PHP—cf. Figure 1), and running such applications has significant memory and load time overheads, even if the application itself might only run for a short amount of time. It is thus important to share joint resources among cloud applications, even if programs do not explicitly share data. Unfortunately, the virtual machine model sequesters application resources.

In Arrakis, software stacks, such as LAMP, could be packaged akin to dynamic link libraries. They would be loaded once on demand and then mapped copy-on-write into cloud applications that use them, significantly reducing memory and load-time overhead. We want to use techniques similar to those used in Disco [8] to share common program code and data structures.

Similar to the web application use case, each cloud application can run its own copy of the necessary device drivers. Driver failures would thus only impact the application with the failing driver. This feature also allows for highly optimized network and disk I/O, as applications have direct network and disk access.

Further, our design has enhanced security features, such as safer logging [9], which we get since application containers are fully virtualized environments.

5 Related Work

By supporting direct application access to virtualized I/O hardware, Arrakis provides very fast and flexible data plane operations. Although fast I/O was not a primary goal of earlier nano-kernel designs, we leverage several ideas from that earlier work.

Exokernel [12] moves almost all operating system functions to user-level, allowing applications to fine-tune their own implementations of common OS services. An Exokernel, however, is still responsible for multiplexing hardware devices. To support application customization of the I/O stack, it resorts to loading device-specific virtual machine code into the OS kernel.

Spin [7] takes the opposite approach and embeds application functionality together with OS functionality as trusted code in the kernel, eliminating expensive protection boundary crossings. This trusted application code is implemented in a type-safe language, like Modula-3, for safety and therefore can be given direct access to low-level hardware functionality to enhance application performance without impairing system security.

In both cases, the lack of hardware support for device virtualization limits implementations to running application code in the kernel using a particular programming language or virtual machine abstraction.

Dune [5] allows Linux applications safe access to privileged CPU and virtual memory features. With Arrakis, we investigate the OS architecture and hardware support needed to virtualize both the CPU and I/O devices.

Microdrivers [13] and Microkernel driver architectures have moved device drivers to user space in order to reduce the trusted computing base and to tackle the problem of a driver crash taking down the entire system. Integrating device drivers into application containers in Arrakis has similar effects (only the application with the faulty driver is affected by a crash), but with other benefits: it allows several custom device driver versions to coexist within the same system and provides a performance opportunity when tightly coupling a trusted driver and application in the same protection domain.

The Nooks project [27] and shadow device drivers [26] try to minimize the effects of driver failure by isolating drivers within lightweight protection domains and providing fast fail-over mechanisms. This work augments our efforts and we plan to investigate whether its integration with Arrakis would simplify our implementation.

6 Conclusion

We believe that it is time to take a fresh look at nano-kernel operating systems. Recent hardware trends are leading towards direct application access to the raw capabilities of the underlying system, with security and resource boundaries enforced in hardware. This enables a separation between the control plane and data plane of an operating system: application storage, networking, and processor and memory management can all be done by the application itself, with no intervention by the operating system kernel in the common case.

References

[1] D. Abramson. Intel virtualization technology for directed I/O. *Intel Technology Journal*, 10(3):179–192, 2006.

[2] T. Anderson. The case for application-specific operating systems. In *Proceedings of the 3rd Work-*

shop on Workstation Operating Systems, pages 92–94, 1992.

[3] T. E. Anderson, B. N. Bershad, E. D. Lazoswka, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of threads. *ACM Transactions on Computer Systems*, 10:53–79, 1992.

[4] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 45–58, Feb. 1999.

[5] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, pages 335–348, Oct. 2012.

[6] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 102–113, Dec. 1989.

[7] 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. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, Dec. 1995.

[8] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15:412–447, 1997.

[9] P. M. Chen and B. D. Noble. When virtual is better than real. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, May 2001.

[10] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 339–354, Dec. 2008.

[11] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, Sept. 2002.

[12] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, Dec. 1995.

[13] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The design and implementation of microdrivers. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 168–178, 2008.

[14] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole, Jr. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 16–25, 1991.

[15] Intel Corporation. *Simplify VMware vSphere* 4 Networking with Intel Ethernet 10 Gigabit Server Adapters*, Feb. 2010. Technical White Paper.

[16] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on Exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 52–65, Oct. 1997.

[17] P. Kutch. PCI-SIG SR-IOV primer: An introduction to SR-IOV technology. *Intel application note*, 321211–002, Jan. 2011.

[18] LSI Corporation. *LSISAS2308 PCI Express to 8-Port 6Gb/s SAS/SATA Controller*, Feb. 2010. Product Brief.

[19] R. Nathuji and K. Schwan. Virtualpower: coordinated power management in virtualized enterprise systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 265–278, Oct. 2007.

[20] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Aug. 2012.

[21] R. Rashid, A. Tevanian, Jr., M. Young, D. Golub, R. Baron, D. Black, W. J. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computers*, C-37:896–908, 1988.

[22] D. Richardson and S. Gribble. Maverick: Providing web applications with safe and flexible access to local devices. In *Proceedings of the 2011 USENIX Conference on Web Application Development*, June 2011.

[23] E. Roman. A survey of checkpoint/restart implementations. Technical Report LBNL-54942, Lawrence Berkeley National Laboratory, July 2002.

[24] R. Sandberg. The Sun network file system: Design, implementation and experience. In *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, 1986.

[25] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 213–228, Nov. 1996.

[26] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, Dec. 2004.

[27] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Oct. 2003.

[28] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the Denali isolation kernel. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[29] 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*, 53(1):91–99, Jan. 2010.